

---

# **Deep Learning for Molecules and Materials**

**Andrew D. White**

**Jun 27, 2022**



# CONTENTS

<b>I</b>	<b>A. Math Review</b>	<b>7</b>
<b>1</b>	<b>Tensors and Shapes</b>	<b>9</b>
1.1	Einstein Notation . . . . .	10
1.2	Tensor Operations . . . . .	11
1.3	Broadcasting . . . . .	13
1.4	Modifying Rank . . . . .	15
1.5	View vs Copy . . . . .	17
1.6	Chapter Summary . . . . .	17
1.7	Exercises . . . . .	17
1.8	Cited References . . . . .	18
<b>II</b>	<b>B. Machine Learning</b>	<b>19</b>
<b>2</b>	<b>Introduction to Machine Learning</b>	<b>21</b>
2.1	The Ingredients . . . . .	22
2.2	Supervised Learning . . . . .	22
2.3	Running This Notebook . . . . .	22
2.4	Unsupervised Learning . . . . .	30
2.5	Chapter Summary . . . . .	33
2.6	Exercises . . . . .	34
2.7	Cited References . . . . .	34
<b>3</b>	<b>Regression &amp; Model Assessment</b>	<b>35</b>
3.1	Running This Notebook . . . . .	35
3.2	Overfitting . . . . .	36
3.3	Exploring Effect of Feature Number . . . . .	39
3.4	Bias Variance Decomposition . . . . .	40
3.5	Regularization . . . . .	42
3.6	Strategies to Assess Models . . . . .	43
3.7	Computing Other Measures . . . . .	44
3.8	Training Data Distribution . . . . .	47
3.9	Chapter Summary . . . . .	49
3.10	Exercises . . . . .	49
3.11	Cited References . . . . .	50
<b>4</b>	<b>Classification</b>	<b>51</b>
4.1	Data . . . . .	52
4.2	Running This Notebook . . . . .	52
4.3	Molecular Descriptors . . . . .	53

4.4	Classification Models . . . . .	54
4.5	Classification Metrics . . . . .	57
4.6	Class Imbalance . . . . .	60
4.7	Overfitting . . . . .	63
4.8	Chapter Summary . . . . .	63
4.9	Exercises . . . . .	63
4.10	Cited References . . . . .	64
<b>5</b>	<b>Kernel Learning</b>	<b>65</b>
5.1	Solubility Example . . . . .	66
5.2	Running This Notebook . . . . .	66
5.3	Kernel Definition . . . . .	67
5.4	Model Definition . . . . .	67
5.5	Training . . . . .	68
5.6	Regularization . . . . .	69
5.7	Training Curves . . . . .	70
5.8	Exercises . . . . .	71
5.9	Chapter Summary . . . . .	72
5.10	Cited References . . . . .	72
<b>III</b>	<b>C. Deep Learning</b>	<b>73</b>
<b>6</b>	<b>Deep Learning Overview</b>	<b>75</b>
6.1	What is a neural network? . . . . .	76
6.2	Revisiting Solubility Model . . . . .	77
6.3	Running This Notebook . . . . .	77
6.4	Prepare Data for Keras . . . . .	78
6.5	Neural Network . . . . .	78
6.6	Exercises . . . . .	80
6.7	Chapter Summary . . . . .	80
6.8	Cited References . . . . .	81
<b>7</b>	<b>Standard Layers</b>	<b>83</b>
7.1	Hyperparameters . . . . .	84
7.2	Common Layers . . . . .	85
7.3	Running This Notebook . . . . .	87
7.4	Example . . . . .	87
7.5	Back propagation . . . . .	89
7.6	Regularization . . . . .	90
7.7	Residues . . . . .	91
7.8	Blocks . . . . .	92
7.9	Dropout Regularization Example . . . . .	92
7.10	L2 Weight Regularization Example . . . . .	94
7.11	Activation Functions . . . . .	95
7.12	Discussion . . . . .	95
7.13	Chapter Summary . . . . .	95
7.14	Cited References . . . . .	96
<b>8</b>	<b>Graph Neural Networks</b>	<b>97</b>
8.1	Representing a Graph . . . . .	97
8.2	Running This Notebook . . . . .	99
8.3	A Graph Neural Network . . . . .	100
8.4	Kipf & Welling GCN . . . . .	101
8.5	Solubility Example . . . . .	105



8.6	Message Passing Viewpoint . . . . .	108
8.7	Gated Graph Neural Network . . . . .	109
8.8	Pooling . . . . .	109
8.9	Readout Function . . . . .	110
8.10	Battaglia General Equations . . . . .	110
8.11	The SchNet Architecture . . . . .	112
8.12	SchNet Example: Predicting Space Groups . . . . .	114
8.13	Current Research Directions . . . . .	121
8.14	Relevant Videos . . . . .	122
8.15	Chapter Summary . . . . .	122
8.16	Cited References . . . . .	123
<b>9</b>	<b>Input Data &amp; Equivariances</b>	<b>125</b>
9.1	Equivariances . . . . .	126
9.2	Equivariances of Coordinates . . . . .	126
9.3	Constructing Equivariant Models . . . . .	127
9.4	Examples . . . . .	129
9.5	Running This Notebook . . . . .	129
9.6	Trajectory Alignment Example . . . . .	133
9.7	Distance Features . . . . .	136
9.8	Chapter Summary . . . . .	137
9.9	Cited References . . . . .	137
<b>10</b>	<b>Equivariant Neural Networks</b>	<b>139</b>
10.1	Do you need equivariance? . . . . .	140
10.2	Running This Notebook . . . . .	140
10.3	Group Theory . . . . .	141
10.4	Equivariance Definition . . . . .	146
10.5	G-Equivariant Convolution Layers . . . . .	147
10.6	Converting between Space and Group . . . . .	148
10.7	G-Equivariant Convolutions on Finite Groups . . . . .	151
10.8	G-Equivariant Convolutions with Translation . . . . .	153
10.9	Group Representation . . . . .	158
10.10	G-Equivariant Convolutions on Compact Groups . . . . .	161
10.11	SO(3) Equivariant Example . . . . .	162
10.12	Equivariant Neural Networks with Constraints . . . . .	165
10.13	Chapter Summary . . . . .	168
10.14	Relevant Videos . . . . .	168
10.15	Cited References . . . . .	168
<b>11</b>	<b>Modern Molecular NNs</b>	<b>169</b>
<b>12</b>	<b>Explaining Predictions</b>	<b>171</b>
12.1	What is an explanation? . . . . .	172
12.2	Feature Importance . . . . .	173
12.3	Running This Notebook . . . . .	176
12.4	Feature Importance Example . . . . .	177
12.5	What is feature importance for? . . . . .	183
12.6	Training Data Importance . . . . .	183
12.7	Surrogate Models . . . . .	183
12.8	Counterfactuals . . . . .	184
12.9	Specific Architectures Explanations . . . . .	186
12.10	Model Agnostic Molecular Counterfactual Explanations . . . . .	186
12.11	Running This Notebook . . . . .	186
12.12	Summary . . . . .	188

12.13 Cited References . . . . .	189
<b>13 Attention Layers</b>	<b>191</b>
13.1 Example . . . . .	192
13.2 Attention Mechanism Equation . . . . .	193
13.3 Attention Reduction . . . . .	193
13.4 Tensor-Dot . . . . .	193
13.5 Soft, Hard, and Temperature Attention . . . . .	194
13.6 Self-Attention . . . . .	194
13.7 Trainable Attention . . . . .	194
13.8 Multi-head Attention Block . . . . .	194
13.9 Running This Notebook . . . . .	195
13.10 Code Examples . . . . .	195
13.11 Attention in Graph Neural Networks . . . . .	197
13.12 Chapter Summary . . . . .	198
13.13 Cited References . . . . .	199
<b>14 Deep Learning on Sequences</b>	<b>201</b>
14.1 Converting Molecules into Text . . . . .	202
14.2 Running This Notebook . . . . .	203
14.3 Recurrent Neural Networks . . . . .	204
14.4 Masking & Padding . . . . .	205
14.5 RNN Solubility Example . . . . .	206
14.6 Transformers . . . . .	208
14.7 Transformer Example . . . . .	209
14.8 Using the Latent Space for Design . . . . .	210
14.9 Representing Materials as Text . . . . .	210
14.10 Applications . . . . .	210
14.11 Summary . . . . .	210
14.12 Cited References . . . . .	211
<b>15 Variational Autoencoder</b>	<b>213</b>
15.1 VAE Loss function . . . . .	214
15.2 Running This Notebook . . . . .	215
15.3 VAE for Discrete Data . . . . .	216
15.4 Training . . . . .	218
15.5 Re-balancing VAE Reconstruction and KL-Divergence . . . . .	221
15.6 Regression VAE . . . . .	223
15.7 Bead-Spring Polymer VAE . . . . .	226
15.8 Using VAE on a Trajectory . . . . .	229
15.9 Relevant Videos . . . . .	231
15.10 Chapter Summary . . . . .	231
15.11 Cited References . . . . .	232
<b>16 Normalizing Flows</b>	<b>233</b>
16.1 Flow Equation . . . . .	234
16.2 Bijections . . . . .	234
16.3 Training . . . . .	234
16.4 Common Bijections . . . . .	235
16.5 Running This Notebook . . . . .	235
16.6 Moon Example . . . . .	236
16.7 Relevant Videos . . . . .	239
16.8 Chapter Summary . . . . .	239
16.9 Cited References . . . . .	240

<b>IV D. Applications</b>	<b>241</b>
<b>17 Predicting DFT Energies with GNNs</b>	<b>243</b>
17.1 Label Description . . . . .	243
17.2 Data . . . . .	244
17.3 Running This Notebook . . . . .	244
17.4 Baseline Model . . . . .	245
17.5 Example GNN Model . . . . .	247
17.6 Relevant Videos About Modeling QM9 . . . . .	250
17.7 Cited References . . . . .	250
<b>18 Generative RNN in Browser</b>	<b>251</b>
18.1 Running This Notebook . . . . .	251
18.2 Approach 1: Token Sampling . . . . .	251
18.3 Approach 2: Token RNN . . . . .	254
18.4 Model Deployment . . . . .	257
18.5 Cited References . . . . .	259
<b>V E. Contributed Chapters</b>	<b>261</b>
<b>19 Hyperparameter Tuning</b>	<b>263</b>
19.1 Training Hyperparameters . . . . .	263
19.2 Model Design-Related Hyperparameters . . . . .	266
19.3 Hyperparameter Optimization . . . . .	267
19.4 Running This Notebook . . . . .	269
19.5 Hyperparameter Tuning . . . . .	269
19.6 Discussion . . . . .	273
19.7 Cited References . . . . .	274
<b>VI F. Appendix</b>	<b>275</b>
<b>20 Style Guide</b>	<b>277</b>
20.1 Plots . . . . .	277
20.2 Citations . . . . .	277
20.3 Links to Other Resources . . . . .	277
20.4 Tips and Warnings . . . . .	277
20.5 Right-column Notes . . . . .	277
<b>21 Changelog</b>	<b>279</b>
21.1 Version 0.5.1 (2022-06-20) . . . . .	279
21.2 Version 0.5.0 (2022-06-17) . . . . .	279
21.3 Version 0.4.3 (2022-05-31) . . . . .	279
21.4 Version 0.4.2 (2022-05-30) . . . . .	279
21.5 Version 0.4.1 (2022-05-29) . . . . .	280
21.6 Version 0.3 (2022-05-28) . . . . .	280
21.7 Version 0.2 (2022-05-27) . . . . .	280
21.8 Version 0.1 (2022-05-26) . . . . .	280
<b>Bibliography</b>	<b>281</b>





Deep learning is becoming a standard tool in chemistry and materials science. Deep learning is specifically about connecting some input data (features) and output data (labels) with a neural network function. Neural networks are differentiable and able to approximate any function. The classic example is connecting a molecule's structure and function. A recent [example](#) is dramatically accelerating quantum calculations to the point that you can achieve DFT level accuracy with a neural network. What makes deep learning especially relevant is that it's a powerful tool for approximating previously intractable functions **and** its ability to generate new data.

In this book, we will view deep learning as a set of tools that allows us to create models that were previously infeasible with classical machine learning. What sets deep learning apart from classic machine learning is feature engineering. Much of the data-driven work in the past required decisions about what features are important and how to compute them from molecules. These are called descriptors. Deep learning is typically trained end-to-end, meaning decisions about which features are important are no longer relevant and we can work directly with molecular structures.

Another reason deep learning is a standard method is its mature tools. Previously, training and using models in machine learning was tedious because it required deriving and implementing new equations for each model. Deep learning has removed this need and model changes can be done nearly effortlessly. Deep learning is not a new paradigm of science or a replacement for a chemist. It's a tool that is mature and now ready for application in molecules and materials.

## Target Audience

The target audience of this book is students with a programming and chemistry background that are interested in building competency in deep learning. For example, PhD students or advanced undergraduates in chemistry or materials science with some Python programming skills will benefit from this book. Sections A and B provide a pedagogical introduction to the principles of machine learning, but only covering topics necessary for deep learning. For example, topics like decision trees and SVMs are not covered because they are not critical to understanding deep learning. Section C covers deep learning principles and details on specific architectures, like the important [Graph Neural Networks](#) and [Variational Autoencoder](#). Other chapters, like [Deep Learning on Sequences](#), give a survey-level overview of a much larger area targeted towards chemistry and materials science. Finally Section D gives more complex examples on authentic deep learning problems from chemistry and materials science. Each section states at the top the required background knowledge, but Python programming ability is assumed throughout. You can find a chemistry-specific introduction to Python at the Molecular Sciences Software Institute [resources page](#).


## Framework Choice

Deep learning is always a little tied up in the implementation details – it's hard to grasp without seeing code. Thus, framework choice can be a part of the learning process. This book assumes familiarity with Python and `numpy` and we use exclusively Python. For the deep learning framework, we use `Jax`, `Tensorflow`, `Keras`, and `scikit-learn`.

for different purposes. `Jax` is easy to learn because it's essentially `numpy` with automatic differentiation and GPU/TPU-acceleration. In this book, we use `Jax` when it's important to understand the implementation details and connect the equations to the code. `Keras` is a high-level framework that has many common deep learning features implemented. It is used when we would like to work with more complex models and I'm trying to show a more complete model. Of course, you can use `Jax` for complete models and show detailed implementations in `Keras`. This is just my reasoning for the choice of framework. `scikit-learn` is an ML package and thus we'll see in the early chapters on ML. Finally, `Tensorflow` is the underlying library of `Keras` so if we want to implement new layers in `Keras` we do it through `Tensorflow`. `TensorflowProbability` is an extension to `Tensorflow` that supports random variables and probability distributions used in our generative models. The most important framework left out of this book is `PyTorch`, which has recently taken the lead to be the most popular framework in deep learning research (not necessarily industry). Ultimately, this book presents the equations and implementation details so that you will learn concepts that are independent of the framework. You should thus be able to quickly pick up `PyTorch`, `MXNet`, or whatever the next new framework might be.

One of the most common mistakes I see from students is that they try to learn deep learning via web searching questions and reading documentation. *This is a terrible way to learn deep learning.* There is quite a bit of information out there, but you will end up with a distorted and framework-specific understanding of deep learning. Remember, a high-ranking search result may be relevant and popular, but that doesn't mean it will help you learn. More importantly, learning deep learning through blogs and Stack overflow makes it so hard to grasp the mathematics and intuition. Web searching and hacking together code is definitely a part of deep learning (for better or worse), but you should do this once you have a firm grasp of the math and details of the model you want to implement.

## Interactivity

On each chapter, you'll see the  button on the top. This launches the chapter as an interactive Google Colab. Each chapter also includes notes on the packages that may need to be installed. If you have problems with install, the complete current list of packages for the textbook is available [here](#).

When using interactivity, many of the chapter will benefit from using a graphics processing unit (GPU). GPUs are what makes deep learning fast enough to be practical on large dataset. This is possible in Google Colab, but may require additional steps if running this locally. Check the documentation of the package you're using (e.g., `Jax`, `PyTorch`, `Tensorflow`) to find out how to use a GPU locally. I have carefully constructed the examples to be small enough though to run on a normal CPU in a laptop though, so the GPU is optional.

## Table of Contents

- A. Math Review
  - *Tensors and Shapes*
- B. Machine Learning
  - *Introduction to Machine Learning*
  - *Regression & Model Assessment*
  - *Classification*
  - *Kernel Learning*
- C. Deep Learning
  - *Deep Learning Overview*
  - *Standard Layers*

- *Graph Neural Networks*
- *Input Data & Equivariances*
- *Equivariant Neural Networks*
- *Modern Molecular NNs*
- *Explaining Predictions*
- *Attention Layers*
- *Deep Learning on Sequences*
- *Variational Autoencoder*
- *Normalizing Flows*
- D. Applications
  - *Predicting DFT Energies with GNNs*
  - *Generative RNN in Browser*
- E. Contributed Chapters
  - *Hyperparameter Tuning*
- F. Appendix
  - *Style Guide*
  - *Changelog*

## Contributors

Thank you to contributors for offering suggestions, identifying errors, and helping improve this book! Twitter handles, if available

## Contributed Chapter

1. Mehrad Ansari (@MehradAnsari)

## Contributed Content to Chapter

1. Geemi Wellawatte (@GWellawatte)

## Substantial Feedback on Content

1. Lily Wang (@lilyminium)
2. Marc Finzi (@m\_finzi)
3. Kevin Jablonka (@kmjablonka)
4. Elana Simon

5. Cathrine Bergh (@cathrinebergh)

### Code Fixes, Math Fixes, Language Fixes

1. Oion Akif
2. Heta Gandhi (@gandhi\_heta)
3. Mattias Hartveit
4. Andreas Krämer
5. Mehrad Ansari (@MehradAnsari)
6. Ritsuya Niwayama
7. Varsha Jain
8. Simon Duerr
9. Julia Westermayr (@JWestermayr)
10. Ernest Awoonor-Williams
11. Joshua Schrier (@joshuaschrier)
12. Marin Bukov
13. Arun Pa Thiagarajan (@arunppsg)
14. Ankur Parmar
15. Erik Thiede (@erik\_der\_elch)

### Citation

Please cite as

White, Andrew D. *Deep Learning for Molecules and Materials*. 2021.

```
@book{whiteDeep2021,  
  title={Deep Learning for Molecules and Materials},  
  author={White, Andrew D},  
  url={https://dmol.pub},  
  year={2021}  
}
```

### Image Credit

A picture by @alinnnaaaa of art by Tomás Saraceno.

### Funding Support



Research reported in this work was supported by the National Institute of General Medical Sciences of the National Institutes of Health under award number R35GM137966. This material is based upon work supported by the National Science Foundation under Grant No. 1764415.

### **License (CC BY-NC 3.0)**

Creative Commons Legal Code

Attribution-NonCommercial 3.0 Unported

See complete description of license at <https://creativecommons.org/licenses/by-nc/3.0/> or at repo <https://github.com/whitead/dmol-book>



## **Part I**

### **A. Math Review**



## TENSORS AND SHAPES

This textbook will draw upon linear algebra, vector calculus, and probability theory. Each chapter states the expected background in an `Audience & Objectives` admonition (see example below). This math review fills in details missing from those typical classes: working with tensors. If you would like a chemistry-focused background on those topics, you can read the online book *Mathematical Methods for Molecule Sciences* by John Straub[Str20].

Tensors are the generalization of vectors (rank 1) and matrices (rank 2) to arbitrary **rank**. Rank can be defined as the number of indices required to get individual elements of a tensor. A matrix requires two indices (row, column), and is thus a rank 2 tensor. We may say in normal conversation that a matrix is a “two-dimensional object” because it has rows and columns, but this is ambiguous because the row could be 6 dimensions and the columns could be 1 dimension. Always use the word rank to distinguish vectors, matrices, and higher-order tensors. The components that make up rank are called **axes** (plural of **axis**). The **dimension** is how many elements are in a particular axis. The **shape** of a tensor combines all of these. A shape is a tuple (ordered list of numbers) whose length is the rank and elements are the dimension of each axis.

---

### Audience & Objectives

You should have a background in linear algebra and basic Python programming to read this chapter. After reading, you should be able to

- Define a tensor and specify one in Python
- Modify tensor rank, shape, and axes
- Use Einstein notation to define equations/expressions of tensors
- Understand and use broadcasting rules to work with tensors

---

### Rank

**Tensor rank** and **matrix rank** are two different concepts. Matrix rank is the number of linearly independent columns and has nothing to do with tensor rank. Some authors may use *order* to refer to tensor rank to distinguish the two terms.

Let's practice our new vocabulary. A Euclidean vector  $(x, y, z)$  is a rank 1 tensor whose 0th axis is dimension 3. Its shape is  $(3)$ . Beautiful. A 5 row, 4 column matrix is now called a rank 2 tensor whose axes are dimension 5 and 4. Its shape is  $(5, 4)$ . The scalar (real number) 3.2 is a rank 0 tensor whose shape is  $()$ .

TensorFlow has a [nice visual guide to tensors](#).

---

**Note:** Array and tensor are synonyms. Array is the preferred word in numpy and often used when describing tensors in Python. Tensor is the mathematic equivalent.

---

## 1.1 Einstein Notation

Einstein notation is the way tensor operations can be written out. We'll be using a simplified version, based on the `einsum` function available in many numerical libraries (`np.einsum`, `tf.einsum`, `jnp.einsum`). It's relatively simple. Each tensor is written as a lower case variable with explicit indices, like  $a_{ijk}$  for a rank 3 tensor. The reason the variable name is written in lower case is because if you fill in the indices  $a_{023}$ , you get a scalar. A variable without an index,  $b$ , is a scalar. There is one rule for this notation: if an index doesn't appear on both sides of the equation, it is summed over on the one side in which it appears. Einstein notation requires both sides of the equation to be written-out, so that its clear what the input/output shapes of the operation are.

**Warning:** The concept of Tensors from physics involves a more complex picture of connecting algebraic sets of objects, typically vectors in a space. Here we just treat tensors as a synonym for multidimensional array. Be aware that looking-up tensor on wikipedia will bring you to the physics picture.

Here are some examples of writing tensor operations in Einstein notation.

### Total Sum

Sum all elements of a rank 4 tensor. In Einstein notation this is:

$$a_{ijkl} = b \quad (1.1)$$

in normal mathematic notation, this would be

$$\sum_i \sum_j \sum_k \sum_l a_{ijkl} = b \quad (1.2)$$

There even is a framework-independent Einstein notation library that enables you to use this notation across multiple frameworks for neural network layers. It is called [einops](#)

### Sum Specific Axis

Sum over last axis

$$a_{ijkl} = b_{ijk} \quad (1.3)$$

In normal notation:

$$\sum_l a_{ijkl} = b_{ijk} \quad (1.4)$$

### Dot Product

In Einstein notation:

$$a_i b_i = c \quad (1.5)$$

In normal notation:

$$\sum_i a_i b_i = c \quad (1.6)$$

Notice that  $a_i$  and  $b_i$  must have the same dimension in their 0th axis in order for the sum in the dot product to be valid. This makes sense, since to compute a dot product the vectors must be the same dimension. In general, if two tensors share the same index ( $b_{ij}$ ,  $a_{ik}$ ), then that axis must be the same dimension.

Can you write the following out in Einstein notation?

### Matrix Multiplication

The matrix product of 2 tensors, where each tensor is rank 2.

---

**Answer**

$$a_{ij}b_{jk} = c_{ik} \quad (1.7)$$


---

### Matrix Vector Product

Apply matrix  $a$  to column vector  $b$  by multiplication.  $\vec{Ab}$  in linear algebra notation.

---

**Answer**

$$a_{ij}b_j = c_i \quad (1.8)$$


---

### Matrix Transpose

Swap the values in a matrix to make it a transpose.

---

**Answer**

$$a_{ij} = t_{ji} \quad (1.9)$$


---

## 1.2 Tensor Operations

### Why Tensors?

Tensors are the main building block of modern deep learning. Nearly all variables in equations are actually tensors. Being able to understand how shape affects them is the key to understanding how algorithms work.

Although you can specify operations in Einstein notation, it is typically not expressive enough. How would you write this operation: sum the last axis of a tensor? Without knowing the rank, you do not know how many indices you should indicate in the expression. Maybe like this?

$$a_{i_0, i_1, \dots, i_N} = a_{i_0, i_1, \dots, i_{N-1}} \quad (1.10)$$

Well, that's good but what if your operation has two arguments: which axis to sum and the tensor. That would also be clumsy to write. Einstein notation is useful and we'll see it more, but we need to think about **tensor operations** as analogies to functions. Tensor operations take in 1 or more tensors and output 1 or more tensors and the output shape depends on the input shape.

One of the difficult things about tensors is understanding how shape is treated in equations. For example, consider this equation:

$$g = \exp(a - b)^2 \quad (1.11)$$

Seems like a reasonable enough equation. But what if  $a$  is rank 3 and  $b$  is rank 1? Is  $g$  rank 1 or 3 then? Actually, this is taken from a real example where  $g$  was rank 4. You subtract each element of  $b$  from each element of  $a$ . You could write this in Einstein notation:

$$g_{ijkl} = \exp(a_{ijk} - b_l)^2 \quad (1.12)$$

except this function should work on arbitrary ranked  $a$  and always output  $g$  being the rank of  $a + 1$ . Typically, the best way to express this is explicitly stating how rank and shape are treated.

### 1.2.1 Reduction Operations

Reduction operations reduce the rank of an input tensor. `np.sum(a, axis=0)` is an example. The `axis` argument means that we're summing over the 0th axis so that it will be removed. If `a` is a rank 1 vector, this would leave us with a scalar. If `a` is a matrix, this would remove the rows so that only columns are left over. That means we would be left with *column sums*. You can also specify a tuple of axes to be removed, which will be done in that order `np.sum(a, axis=(0,1))`.

In addition to `np.sum`, there are `np.minimum`, `np.maximum`, `np.any` (logical or), and more. Let's see some examples

```
import numpy as np

a_shape = (4, 3, 2)
a_len = 4 * 3 * 2

a = np.arange(a_len).reshape(a_shape)
print(a.shape)
print(a)
```

```
(4, 3, 2)
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
 [10 11]]

 [[12 13]
  [14 15]
 [16 17]]

 [[18 19]
  [20 21]
 [22 23]]]
```

Try to guess the shape of the output tensors using `a` in the below code based on what you've learned.

```
b = np.sum(a, axis=0)
print(b.shape)
print(b)
```

```
c = np.any(a > 4, axis=1)
print(c.shape)
print(c)
```

```
d = np.product(a, axis=(2, 1))
print(d.shape)
print(d)
```



## 1.2.2 Element Operations

Default operations in Python, like  $+$   $-$   $*$   $/$   $^$ , are also tensor operations. They preserve shape so that the output shape is the same as the inputs'. The input tensors must have the same shape or be able to become the same shape through broadcasting, which is defined in the next section.

```
a.shape
```

```
(4, 3, 2)
```

```
b = np.ones((4, 3, 2))
b.shape
```

```
(4, 3, 2)
```

```
c = a * b
c.shape
```

```
(4, 3, 2)
```

## 1.3 Broadcasting

One of the difficulties with the elementary operations is that they require the input tensors to have the same shape. For example, you cannot multiply a scalar (rank 0) and a vector (rank 1). Of course, if you're familiar with `numpy` this is common. It is done with **broadcasting** comes in. Broadcasting increases the rank of one of the input tensors to be compatible with another. Broadcasting works at the last axis and works its way forward. Let's see an example

### Broadcasting order

Broadcasting starts at the last axis and goes forward because getting an element at the last axis gives a scalar (rank 0) no matter what rank. This makes it possible to copy to fill up axis to align shapes.

$$A + B \tag{1.13}$$

### Input A

Rank 2, shape is (2, 3)

```
A:
 4  3  2
-1  2  4
```

### Input B

Rank 1, shape is (3), a vector:

```
B: 3  0  1
```

Now let's see how the broadcasting works. Broadcasting starts by lining up the shapes from the end of the tensors

**Step 1: align on last axis**

```

tensor      shape
A:          2  3
B:          3
broadcasted B: .  .

```

### Step 2: process last axis

Now broadcasting looks at the last axis (axis 1) and if one tensor has axis dimension 1, its value is copied to match the others. In our case, they agree.

```

tensor      shape
A:          2  3
B:          3
broadcasted B: .  3

```

### Step 3: process next axis

Now we examine the next axis, axis 0. B has no axis there, because its rank is too low. Broadcasting will insert a new axis by (i) inserting a new axis with dimension 1 and (ii) copying the value at this new axis until its dimension matches.

#### Step 3i:

Add new axis of dimension 1. This is like making  $B$  have 1 row and 3 columns:

```

B:
 3  0  1

```

#### Step 3ii:

Now we copy the values of this axis until its dimension matches  $A$ 's axis 0 dimension. We're basically copying  $b_{0j}$  to  $b_{1j}$ .

```

B:
 3  0  1
 3  0  1

```

### Final

```

tensor      shape
A:          2  3
B:          3
broadcasted B: 2  3

```

Now, we compute the result by addition elementwise.

```

A + B
 4 + 3  3 + 0  2 + 1  =  7  3  3
-1 + 3  2 + 0  4 + 1    2  2  5

```

Let's see some more examples, but only looking at the input/output shape

A Shape	B Shape	Output Shape
(4,2)	(4,1)	(4,2)
(4,2)	(2,)	(4,2)
(16,1,3)	(4,3)	(16,4,3)
(16,3,3)	(4,1)	Error

Try some for yourself!

A Shape	B Shape	Output Shape
(7,4,3)	(1,)	?
(16, 16, 3)	(3,)	?
(2,4,5)	(5,4,1)	?
(1,4)	(16,)	?

Answer

A Shape	B Shape	Output Shape
(7,4,3)	(1,)	(7,4,3)
(16, 16, 3)	(3,)	(16,16,3)
(2,4,5)	(5,4,1)	Error
(1,4)	(16,)	Error

### 1.3.1 Suggested Reading for Broadcasting

You can read more about broadcasting in the [numpy tutorial](#) or the [Python Data Science Handbook](#).

## 1.4 Modifying Rank

### newaxis

`newaxis` slices like `a[np.newaxis]` are possible in tensorflow, jax, and numpy. In PyTorch there is `unsqueeze`. You can also use `reshape` and ignore `newaxis`

The last example we saw brings up an interesting question: what if we want to add a (1,4) and (16,) to end up with a (4,16) tensor? We could insert a new axis at the end of *B* to make its shape (16, 1). This can be done using the `np.newaxis` syntax:

```
a = np.ones((1, 4))
b = np.ones(
    16,
)
result = a + b[:, np.newaxis]
result.shape
```

```
(16, 4)
```

Just as `newaxis` can increase rank, we can decrease rank. One way is to just slice, like `a[0]`. A more general way is to `np.squeeze` which removes any axes that are dimension 1 without needing to know the specific axes that are dimension 1.

```
a = np.ones((1, 32, 4, 1))
print("before squeeze:", a.shape)
a = np.squeeze(a)
print("after squeeze:", a.shape)
```

```
before squeeze: (1, 32, 4, 1)
after squeeze: (32, 4)
```

It turns out that `np.newaxis` and `tf.newaxis` are actually defined as `None`. Some programmers will exploit this to save some keystrokes and use `None` instead:

```
a = np.ones((1, 4))
b = np.ones(
    16,
)
result = a + b[:, None]
result.shape
```

```
(16, 4)
```

I recommend against this because it can be a bit confusing and it's really not saving that many keystrokes.

### 1.4.1 Reshaping

The most general way of changing rank and shape is through `np.reshape`. This allows you to reshape a tensor, as long as the number of elements remains the same. You could make a (4, 2) into an (8,). You could make a (4, 3) into a (1, 4, 3, 1). Thus it can accomplish the two tasks done by `np.squeeze` and `np.newaxis`.

There is one special syntax element to shaping: A `-1` dimension. `-1` can appear once in a reshape command and means to have the computer figure out what goes there. We know the number of elements doesn't change in a reshape, so the computer can infer what goes in the dimension marked as `-1`. Let's see some examples.

```
a = np.arange(32)
new_a = np.reshape(a, (4, 8))
new_a.shape
```

```
(4, 8)
```

```
new_a = np.reshape(a, (4, -1))
new_a.shape
```

```
(4, 8)
```

```
new_a = np.reshape(a, (1, 2, 2, -1))
new_a.shape
```

```
(1, 2, 2, 8)
```

## 1.4.2 Rank Slicing

Hopefully you're familiar with slicing in numpy/Python. Review at the [Python Tutorial](#) and the [numpy tutorial](#) for a refresher if you need it. **Rank Slicing** is just my terminology for slicing without knowing the rank of a tensor. Use the `...` (ellipsis) keyword. This allows you to account for unknown rank when slicing. Examples:

- Access last axis: `a[..., :]`
- Access last 2 axes: `a[..., :, :]`
- Add new axis to end `a[..., np.newaxis]`
- Add new axis to beginning `a[np.newaxis, ...]`

Let's see if we can put together our skills to implement the equation example from above,

$$g = \exp(a - b)^2 \quad (1.14)$$

for arbitrary rank  $a$ . Recall  $b$  is a rank 1 tensor and we want  $g$  to be the rank of  $a + 1$ .

```
def eq(a, b):
    return np.exp((a[..., np.newaxis] - b) ** 2)

b = np.ones(4)
a1 = np.ones((4, 3))
a2 = np.ones((4, 3, 2, 1))

g1 = eq(a1, b)
print("input a1:", a1.shape, "output:", g1.shape)

g2 = eq(a2, b)
print("input a2:", a2.shape, "output:", g2.shape)
```

```
input a1: (4, 3) output: (4, 3, 4)
input a2: (4, 3, 2, 1) output: (4, 3, 2, 1, 4)
```

## 1.5 View vs Copy

In most machine learning frameworks, it is actually not possible to modify an array because element assignment interferes with automatic differentiation. So this distinction of a view vs a copy is irrelevant.

Most slicing and reshaping operations produce a **view** of the original array. That means no copy operation is done. This is default behavior in all frameworks to reduce required memory – you can slice as much as you want without increasing memory use. This typically has no consequences for how we program; it is more of an optimization detail. However, if you modify elements in a view, you will also modify the original array from which the view was constructed. Sometimes this can be unexpected. You should not rely on this behavior though, because in `numpy` a copy may be returned for certain `np.reshape` and `slicing commands`. Thus, I recommend being aware that views may be returned as an optimization, but not assume that is always the case. If you actually want a copy you should explicitly create a copy, like with `np.copy`.

## 1.6 Chapter Summary

- Tensors are the building blocks of machine learning. A tensor has a rank and shape that specifies how many elements it has and how they are arranged. An axis describes each element in the shape.
- A euclidean vector is a rank 1 tensor with shape (3). It has 1 axis of dimension 3. A matrix is a rank 2 tensor. It has two axes.
- Equations that describe operating on 1 or more tensors can be written using Einstein notation. Einstein notation uses indices to indicate the shape of tensors, how things are summed, and which axes must match up.
- There are operations that reduce ranks of tensors, like `sum` or `mean`.
- Broadcasting is an automatic tool in programming languages that modifies shapes of tensors with different shapes to be compatible with operations like addition or division.
- Tensors can be reshaped or have rank modified by `newaxis`, `reshape`, and `squeeze`. These are not standardized among the various numeric libraries in Python.

## 1.7 Exercises

### 1.7.1 Einstein notation

Write out the following in Einstein notation:

1. Product of two matrices
2. Trace of a matrix
3. Outer product of two Euclidean vectors
4. **A** is a rank 3 tensor whose last axis is dimension 3 and contains Euclidean vectors. **B** is Euclidean vector. Compute the dot product of each of the vectors in **A** with **B**. So if **A** is shape (11, 7, 3), it contains  $11 \times 7$  vectors and the output should be shape (11,7). **B** is shape (3)

### 1.7.2 Reductions

Answer the following with Python code with reductions. Write your code to be as general as possible – being able to take arbitrary rank tensors unless it is specified that something is a vector.

1. Normalize a vector so that the sum of its elements is 1. Note the rank of the vector should be unchanged.
2. Normalize the last axis of a tensor
3. Compute the mean squared error between two tensors
4. Compute the mean squared error between the last axis of tensor **A** and vector  $\vec{b}$

### 1.7.3 Broadcasting and Shapes

1. Consider two vectors  $\vec{a}$  and  $\vec{b}$ . Using reshaping and broadcasting alone, write python code to compute their outer product.
2. Why is the code `a.reshape((-1, 3, -1))` invalid?
3. You have a tensor of unknown rank **A** and would like to subtract both 3.5 and 2.5 from every element, giving two outputs for every input. Your output will be a new tensor **B** with rank  $\text{rank}(\mathbf{A}) + 1$ . The last axis of **B** should be dimension 2. Here is the example:

```
a = np.array([10])
f(a)
# prints [[6.5, 7.5]]

b = np.array([[5, 3, 0], [0, 2, 6]])
f(b)
# [[[ 1.5  2.5]
#   [-0.5  0.5]
#   [-3.5 -2.5]]
#
#  [[-3.5 -2.5]
#   [-1.5 -0.5]
#   [ 2.5  3.5]]]
```

## 1.8 Cited References





## **Part II**

### **B. Machine Learning**



## INTRODUCTION TO MACHINE LEARNING

There are probably a thousand articles called *introduction to machine learning*. Rather than rewrite this, I will instead introduce the main ideas focused on a chemistry example. Here are some introductory sources, and please do recommend new ones to me:

1. The book I first read in grad school about machine learning by Ethem Alpaydin[Alp20]
2. Nils Nillson's online book [Introductory Machine Learning](#)
3. Two reviews of machine learning in materials[FZJS21, Bal19]
4. A review of machine learning in computational chemistry[GomezBAG20]
5. A review of machine learning in metals[NDJ+18]

I hope you learn from these sources about how machine learning is a method of modeling data, typically with predictive functions. Machine learning includes many techniques, but here we will focus on only those necessary to transition into deep learning. For example, random forests, support vector machines, and nearest neighbor are widely-used machine learning techniques that are effective but not covered here.

---

### Audience & Objectives

This chapter is intended for novices of machine learning with familiarity of chemistry and python. It is recommended that you look over one of the above recommended introductory articles. This specific article assumes a very small amount of knowledge of the `pandas` library (loading and selecting a column), awareness of `rdkit` (how we draw molecules), and that we store/retrieve molecules as `SMILES` [Wei88]. After reading this, you should be able to:

- Define features, labels
  - Distinguish between supervised and unsupervised learning
  - Understand what a loss function is and how it can be minimized with gradient descent
  - Understand what model is and its connection to features and labels
  - Be able to cluster data and describe what it tells us about data
-

### 2.1 The Ingredients

Machine learning is about constructing models by fitting them to data. Firstly, definitions:

#### Features

set of  $N$  vectors  $\{\vec{x}_i\}$  of dimension  $D$ . Can be reals, integers, etc.

#### Labels

set of  $N$  integers or reals  $\{y_i\}$ .  $y_i$  is usually a scalar

#### Labeled Data

set of  $N$  tuples  $\{(\vec{x}_i, y_i)\}$

#### Unlabeled Data

set of  $N$  features  $\{\vec{x}_i\}$  that may have unknown  $y$  labels

#### Model

A function  $f(\vec{x})$  that takes a given feature vector in and returns a predicted  $\hat{y}$

#### Predictions

$\hat{y}$ , our predicted output for a given input  $\vec{x}$ .

### 2.2 Supervised Learning

Our first task will be **supervised learning**. Supervised learning means predicting  $y$  from  $\vec{x}$  with a model trained on data. It is *supervised* because we tell the algorithm what the labels are in our dataset. Another method we'll explore is **unsupervised learning** where we do not tell the algorithm the labels. We'll see this supervised/unsupervised distinction can be more subtle later on, but this is a great definition for now.

To see an example, we will use a dataset called AqSolDB[SKE19] that is about 10,000 unique compounds with measured solubility in water (label). The dataset also includes molecular properties (features) that we can use for machine learning. The solubility measurement is solubility of the compound in water in units of log molarity.

### 2.3 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

### 2.3.1 Load Data

Download the data and load it into a **Pandas** data frame. The hidden cells below sets-up our imports and/or install necessary packages.

```
# soldata = pd.read_csv('https://dataverse.harvard.edu/api/access/datafile/3407241?
↳format=original&gbrecs=true')
# had to rehost because dataverse isn't reliable
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
↳csv"
)
soldata.head()
```

	ID	Name \
0	A-3	N,N,N-trimethyloctadecan-1-aminium bromide
1	A-4	Benzo[cd]indol-2(1H)-one
2	A-5	4-chlorobenzaldehyde
3	A-8	zinc bis[2-hydroxy-3,5-bis(1-phenylethyl)benzo...
4	A-9	4-({4-[bis(oxiran-2-ylmethyl)amino]phenyl}meth...

	InChI \
0	InChI=1S/C21H46N.BrH/c1-5-6-7-8-9-10-11-12-13-...
1	InChI=1S/C11H7NO/c13-11-8-5-1-3-7-4-2-6-9(12-1...
2	InChI=1S/C7H5ClO/c8-7-3-1-6(5-9)2-4-7/h1-5H
3	InChI=1S/2C23H22O3.Zn/c2*1-15(17-9-5-3-6-10-17...
4	InChI=1S/C25H30N2O4/c1-5-20(26(10-22-14-28-22)...

	InChIKey \
0	SZEMGTQCPRXEG-UHFFFAOYSA-M
1	GPYLCFQEKPUWLD-UHFFFAOYSA-N
2	AVPYQKSLEYISFPO-UHFFFAOYSA-N
3	XTUPUYCJWKHGSW-UHFFFAOYSA-L
4	FAUAZXVRLVIARB-UHFFFAOYSA-N

	SMILES	Solubility	SD \
0	[Br-].CCCCCCCCCCCCCCCC[N+](C)(C)C	-3.616127	0.0
1	O=C1Nc2cccc3cccc1c23	-3.254767	0.0
2	Clc1ccc(C=O)cc1	-2.177078	0.0
3	[Zn++].CC(c1cccc1)c2cc(C(C)c3cccc3)c(O)c(c2)...	-3.924409	0.0
4	C1OC1CN(CC2CO2)c3ccc(Cc4ccc(cc4)N(CC5CO5)CC6CO...	-4.662065	0.0

	Ocurrences	Group	MolWt	...	NumRotatableBonds	NumValenceElectrons \
0	1	G1	392.510	...	17.0	142.0
1	1	G1	169.183	...	0.0	62.0
2	1	G1	140.569	...	1.0	46.0
3	1	G1	756.226	...	10.0	264.0
4	1	G1	422.525	...	12.0	164.0

	NumAromaticRings	NumSaturatedRings	NumAliphaticRings	RingCount	TPSA \
0	0.0	0.0	0.0	0.0	0.00
1	2.0	0.0	1.0	3.0	29.10
2	1.0	0.0	0.0	1.0	17.07
3	6.0	0.0	0.0	6.0	120.72
4	2.0	4.0	4.0	6.0	56.60

	LabuteASA	BalabanJ	BertzCT
0			
1			
2			
3			
4			

(continues on next page)

(continued from previous page)

```
0 158.520601 0.000000e+00 210.377334
1 75.183563 2.582996e+00 511.229248
2 58.261134 3.009782e+00 202.661065
3 323.755434 2.322963e-07 1964.648666
4 183.183268 1.084427e+00 769.899934
```

```
[5 rows x 26 columns]
```

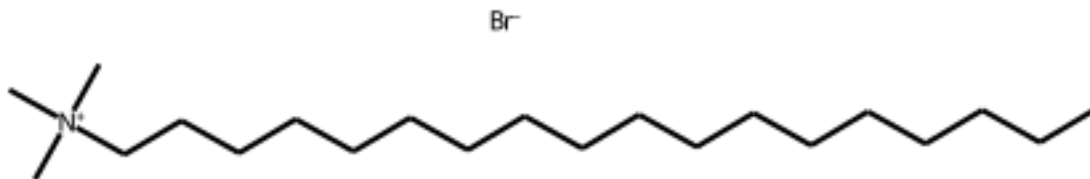
## 2.3.2 Data Exploration

### EDA

If doing EDA as a way to choose features, you should do the train/test/(valid) split prior to EDA to avoid contaminating model selection with test data.

We can see that there are a number of features like molecular weight, rotatable bonds, valence electrons, etc. And of course, there is the label **solubility**. One of the first things we should always do is get familiar with our data in a process that is sometimes called **exploratory data analysis** (EDA). Let's start by examining a few specific examples to get a sense of the range of labels/data.

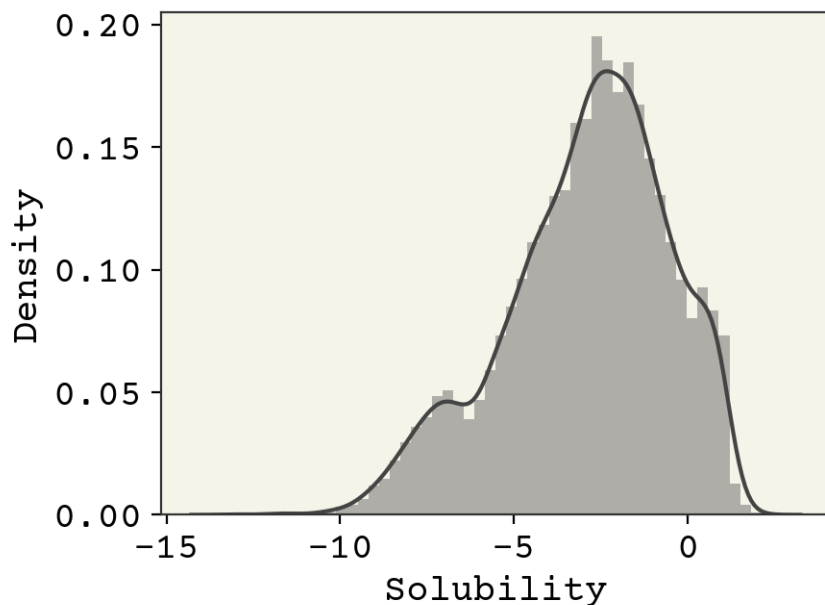
```
# plot one molecule
mol = rdkit.Chem.MolFromInchi(soldata.InChI[0])
mol
```



This is first molecule in the dataset rendered using `rdkit`.

Let's now look at the extreme values to get a sense of the **range** of solubility data and the molecules that make it. First, we'll histogram (using `seaborn.distplot`) the solubility which tells us about the shape of its probability distribution and the extreme values.

```
sns.distplot(soldata.Solubility)
plt.show()
```



Above we can see the histogram of the solubility with kernel density estimate overlaid. The histogram shows that the solubility varies from about -13 to 2.5 and is not normally distributed.

```
# get 3 lowest and 3 highest solubilities
soldata_sorted = soldata.sort_values("Solubility")
extremes = pd.concat([soldata_sorted[:3], soldata_sorted[-3:]])

# We need to have a list of strings for legends
legend_text = [
    f"{x.ID}: solubility = {x.Solubility:.2f}" for x in extremes.itertuples()
]

# now plot them on a grid
extreme_mols = [rdkit.Chem.MolFromInchi(inchi) for inchi in extremes.InChI]
rdkit.Chem.Draw.MolsToGridImage(
    extreme_mols, molsPerRow=3, subImgSize=(250, 250), legends=legend_text
)
```

<IPython.core.display.SVG object>

The figure of extreme molecules shows highly-chlorinated compounds have the lowest solubility and ionic compounds have higher solubility. Is A-2918 an **outlier**, a mistake? Also, is  $\text{NH}_3$  really comparable to these organic compounds? These are the kind of questions that you should consider *before* doing any modeling.

### Outliers

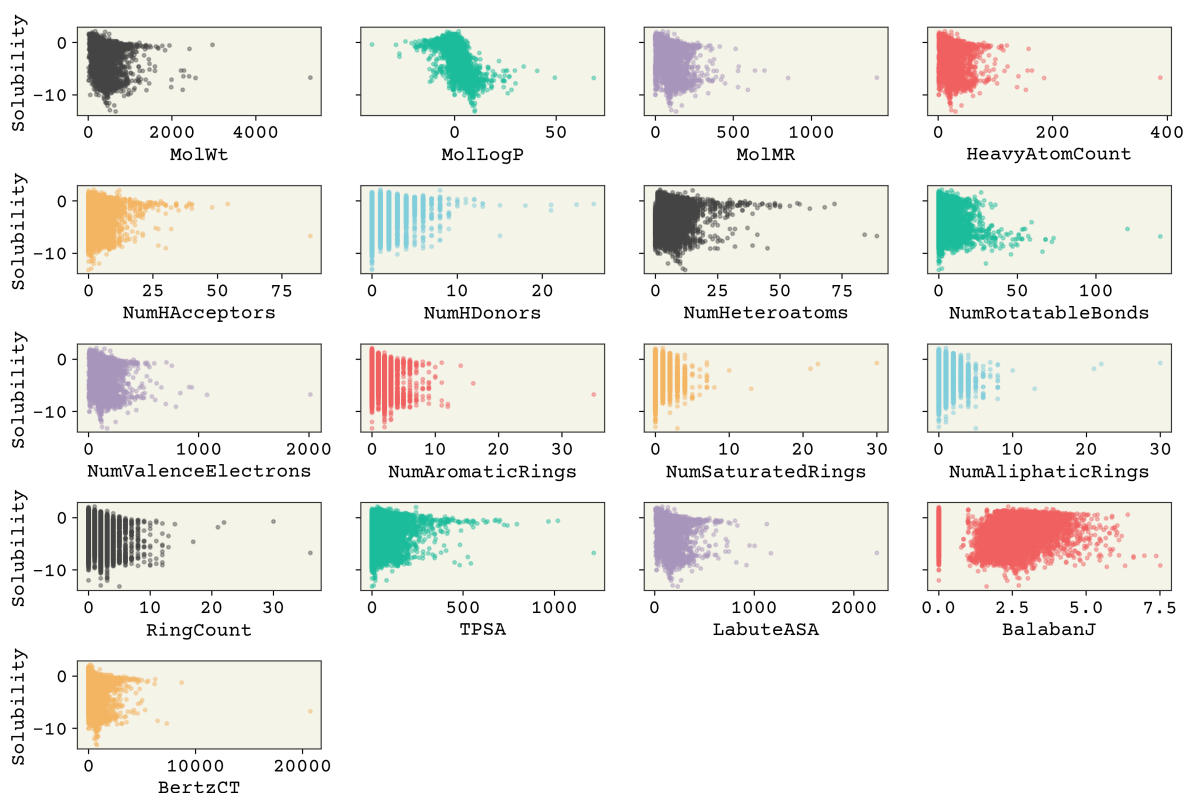
Outliers are extreme values that fall outside of your normal data distribution. They can be mistakes or be from a different distribution (e.g., metals instead of organic molecules). Outliers can have a strong effect on model training.

### 2.3.3 Feature Correlation

Now let's examine the features and see how correlated they are with solubility. Note that there are a few columns unrelated to features or solubility: SD (standard deviation), Occurrences (how often the molecule occurred in the constituent databases), and Group (where the data came from).

```
features_start_at = list(soldata.columns).index("MolWt")
feature_names = soldata.columns[features_start_at:]

fig, axs = plt.subplots(nrows=5, ncols=4, sharey=True, figsize=(12, 8), dpi=300)
axs = axs.flatten()  # so we don't have to slice by row and column
for i, n in enumerate(feature_names):
    ax = axs[i]
    ax.scatter(
        soldata[n], soldata.Solubility, s=6, alpha=0.4, color=f"C{i}"
    )  # add some color
    if i % 4 == 0:
        ax.set_ylabel("Solubility")
    ax.set_xlabel(n)
# hide empty subplots
for i in range(len(feature_names), len(axs)):
    fig.delaxes(axs[i])
plt.tight_layout()
plt.show()
```



It's interesting that molecular weight or hydrogen bond numbers seem to have little correlation, at least from this plot. MolLogP, which is a calculated descriptor related to solubility, does correlate well. You can also see that some of these features have low **variance**, meaning the value of the feature changes little or not at all for many data points (e.g., "NumHDonors").



### 2.3.4 Linear Model

Let's begin with one of the simplest approaches — a linear model. This is our first type of supervised learning and is rarely used due to something we'll see — the difficult choice of features.

#### Autodiff

**Autodiff** is a computer program tool that can compute analytical gradients with respect to two variables in a program.

Our model will be defined by this equation:

$$y = \vec{w} \cdot \vec{x} + b \quad (2.1)$$

which is defined for a single data point. The shape of a single feature vector,  $\vec{x}$ , is 17 in our case (for the 17 features).  $\vec{w}$  is a vector of adjustable parameters of length 17 and  $b$  is an adjustable scalar (called **bias**).

We'll implement this model using a library called `jax` that is very similar to `numpy` except it can compute analytical gradients easily via `autodiff`.

```
def linear_model(x, w, b):
    return jnp.dot(x, w) + b

# test it out
x = np.array([1, 0, 2.5])
w = np.array([0.2, -0.5, 0.4])
b = 4.3

linear_model(x, w, b)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
↳and rerun for more info.)
```

```
DeviceArray(5.5, dtype=float32)
```

#### Loss

A loss is a function which takes in a model prediction  $\hat{y}$ , labels  $y$ , and computes a scalar representing how poor the fit is. Our goal is to minimize loss.

Now comes the critical question: *How do we find the adjustable parameters  $\vec{w}$  and  $b$ ?* The classic solution for linear regression is computing the adjustable parameters directly with a pseudo-inverse,  $\vec{w} = (X^T X)^{-1} X^T \vec{y}$ . You can read more about [this here](#). We'll use an **iterative** approach that mirrors what we'll do in deep learning. This is not the correct approach for linear regression, but it'll be useful for us to get used to the iterative approach since we'll see it so often in deep learning.

To iteratively find our adjustable parameters, we will pick a **loss** function and minimize with **gradients**. Let's define these quantities and compute our loss with some initial random  $w$  and  $b$ .

```
# convert data into features, labels
features = soldata.loc[:, feature_names].values
labels = soldata.Solubility.values
```

(continues on next page)

(continued from previous page)

```

feature_dim = features.shape[1]

# initialize our paramaters
w = np.random.normal(size=feature_dim)
b = 0.0

# define loss
def loss(y, labels):
    return jnp.mean((y - labels) ** 2)

# test it out
y = linear_model(features, w, b)
loss(y, labels)

```

```
DeviceArray(3265882.2, dtype=float32)
```

Wow! Our loss is terrible, especially considering that solubilities are between -13 and 2. But, that's right since we just guessed our initial parameters.

### 2.3.5 Gradient Descent

We will now try to reduce loss by using information about how it changes with respect to the adjustable parameters. If we write our loss as:

$$L = \frac{1}{N} \sum_i^N [y_i - f(\vec{x}_i, \vec{w}, b)]^2 \quad (2.2)$$

This loss is called **mean squared error**, often abbreviated MSE. We can compute our loss gradients with respect to the adjustable parameters:

#### **jax.grad**

`jax.grad` computes an analytical derivative of a Python function. It takes two arguments: the function and which args to take the derivative of. For example, consider  $f(x, y, z)$ , then `jax.grad(f, (1, 2))` gives  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ . Note too that  $x$  may be a tensor.

$$\frac{\partial L}{\partial w_i}, \frac{\partial L}{\partial b} \quad (2.3)$$

where  $w_i$  is the  $i$ th element of the weight vector  $\vec{w}$ . We can reduce the loss by taking a step in the direction of its negative gradient:

$$(w_i, b') = \left( w_i - \eta \frac{\partial L}{\partial w_i}, b - \eta \frac{\partial L}{\partial b} \right) \quad (2.4)$$

where  $\eta$  is **learning rate**, which an adjustable but not trained parameter (an example of a **hyperparameter**) which we just guess to be  $1 \times 10^{-6}$  in this example. Typically, it's chosen to be some power of 10 that is at most 0.1. Values higher than that cause stability problems. Let's try this procedure, which is called **gradient descent**.

```

# compute gradients
def loss_wrapper(w, b, data):
    features = data[0]
    labels = data[1]

```

(continues on next page)

(continued from previous page)

```
y = linear_model(features, w, b)
return loss(y, labels)

loss_grad = jax.grad(loss_wrapper, (0, 1))

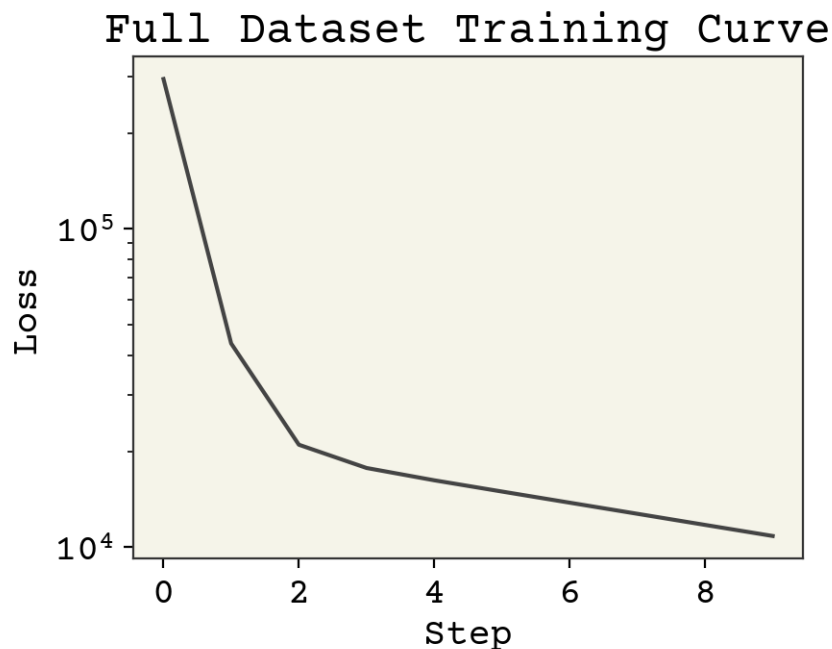
# test it out
loss_grad(w, b, (features, labels))
```

```
(DeviceArray([1.1313606e+06, 7.3055479e+03, 2.8128903e+05, 7.4776180e+04,
              1.5807716e+04, 4.1010732e+03, 2.2745816e+04, 1.7236283e+04,
              3.9615741e+05, 5.2891074e+03, 1.0585280e+03, 1.8357089e+03,
              7.1248174e+03, 2.7012794e+05, 4.6752903e+05, 5.4541211e+03,
              2.5596618e+06], dtype=float32),
 DeviceArray(2671.3784, dtype=float32, weak_type=True))
```

We've computed the gradient. Now we'll minimize it over a few steps.

```
loss_progress = []
eta = 1e-6
data = (features, labels)
for i in range(10):
    grad = loss_grad(w, b, data)
    w -= eta * grad[0]
    b -= eta * grad[1]
    loss_progress.append(loss_wrapper(w, b, data))
plt.plot(loss_progress)

plt.xlabel("Step")
plt.yscale("log")
plt.ylabel("Loss")
plt.title("Full Dataset Training Curve")
plt.show()
```



### 2.3.6 Training Curve

The figure above is called a **training curve**. We'll see these frequently in this book and they show us if the loss is decreasing, indicating the model is learning. Training curves are also called **learning curves**. The x-axis may be example number, total iterations through dataset (called epochs), or some other measure of amount of data used for training the model.

### 2.3.7 Batching

#### batch

A batch is a subset of your data of size *batch size*. Batch size is usually as a power of 2 (e.g., 16, 128). Having random batches of data is how gradient descent becomes stochastic gradient descent.

This is making good progress. But let's try to speed things up with a small change. We'll use **batching**, which is how training is actually done in machine learning. The small change is that rather than using all data at once, we only take a small **batch** of data. Batching provides two benefits: it reduces the amount of time to compute an update to our parameters, and it makes the training process random. The randomness makes it possible to escape local minima that might stop training progress. This addition of batching makes our algorithm **stochastic** and thus we call this procedure **stochastic gradient descent** (SGD). SGD, and variations of it, are the most common methods of training in deep learning.

```
# initialize our paramaters
# to be fair to previous method
w = np.random.normal(size=feature_dim)
b = 0.0

loss_progress = []
eta = 1e-6
```

(continues on next page)

(continued from previous page)

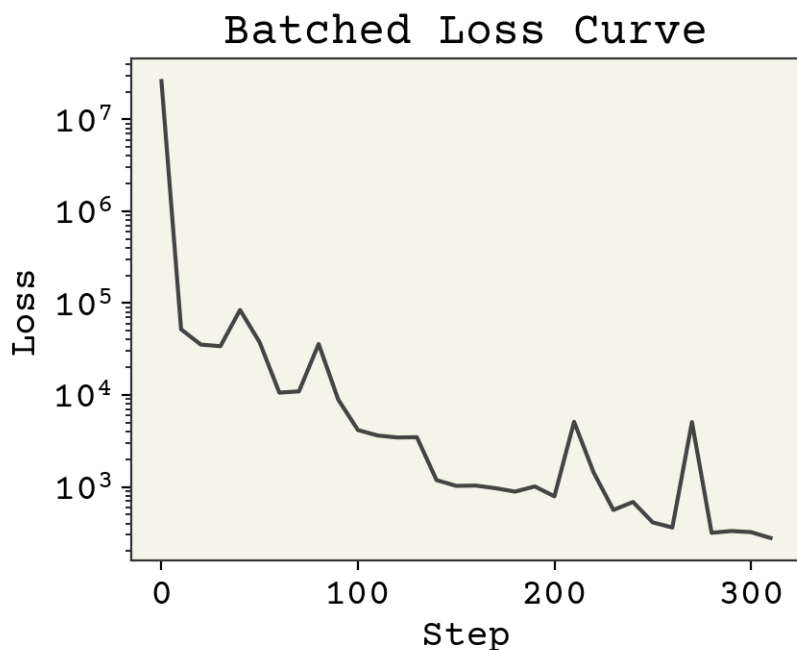
```

batch_size = 32
N = len(labels) # number of data points
data = (features, labels)
# compute how much data fits nicely into a batch
# and drop extra data
new_N = len(labels) // batch_size * batch_size

# the -1 means that numpy will compute
# what that dimension should be
batched_features = features[:new_N].reshape((-1, batch_size, feature_dim))
batched_labels = labels[:new_N].reshape((-1, batch_size))
# to make it random, we'll iterate over the batches randomly
indices = np.arange(new_N // batch_size)
np.random.shuffle(indices)
for i in indices:
    # choose a random set of
    # indices to slice our data
    grad = loss_grad(w, b, (batched_features[i], batched_labels[i]))
    w -= eta * grad[0]
    b -= eta * grad[1]
    # we still compute loss on whole dataset, but not every step
    if i % 10 == 0:
        loss_progress.append(loss_wrapper(w, b, data))

plt.plot(np.arange(len(loss_progress)) * 10, loss_progress)
plt.xlabel("Step")
plt.yscale("log")
plt.ylabel("Loss")
plt.title("Batched Loss Curve")
plt.show()

```



There are three changes to note:

1. The loss is lower than without batching

2. There are more steps, even though we iterated over our dataset once instead of 10 times
3. The loss doesn't always go down

The reason the loss is lower is because we're able to take more steps even though we only see each data point once. That's because we update at each batch, giving more updates per iteration over the dataset. Specifically if  $B$  is batch size, there are  $N/B$  updates for every 1 update in the original gradient descent. The reason the loss doesn't always go down is that each time we evaluate it, it's on a different set of data. Some molecules are harder to predict than others. Also, each step we take in minimizing loss may not be correct because we only updated our parameters based on one batch. Assuming our batches are mixed though, we will always improve in expectation (on average).

### 2.3.8 Standardize features

It seems we cannot get past a certain loss. If you examine the gradients you'll see some of them are very large and some are very small. Each of the features have different magnitudes. For example, molecular weights are large numbers. The number of rings in a molecule is a small number. Each of these must use the same learning rate,  $\eta$ , and that is ok for some but too small for others. If we increase  $\eta$ , our training procedure will explode because of these larger feature gradients. A standard trick we can do is make all the features have the same magnitude, using the equation for standardization you might see in your statistics textbook:

$$x_{ij} = \frac{x_{ij} - \bar{x}_j}{\sigma_{x_j}} \quad (2.5)$$

where  $\bar{x}_j$  is column mean and  $\sigma_{x_j}$  is column standard deviation. To be careful about contaminating training data with test data – leaking information between train and test data – we should only use training data in computing the mean and standard deviation. We want our test data to approximate how we'll use our model on unseen data, so we cannot know what these unseen features means/standard deviations might be and thus cannot use them at training time for standardization.

```
fstd = np.std(features, axis=0)
fmean = np.mean(features, axis=0)
std_features = (features - fmean) / fstd
```

```
# initialize our paramaters
# since we're changing the features
w = np.random.normal(scale=0.1, size=feature_dim)
b = 0.0

loss_progress = []
eta = 1e-2
batch_size = 32
N = len(labels) # number of data points
data = (std_features, labels)
# compute how much data fits nicely into a batch
# and drop extra data
new_N = len(labels) // batch_size * batch_size
num_epochs = 3

# the -1 means that numpy will compute
# what that dimension should be
batched_features = std_features[:new_N].reshape((-1, batch_size, feature_dim))
batched_labels = labels[:new_N].reshape((-1, batch_size))
indices = np.arange(new_N // batch_size)

# iterate through the dataset 3 times
for epoch in range(num_epochs):
```

(continues on next page)

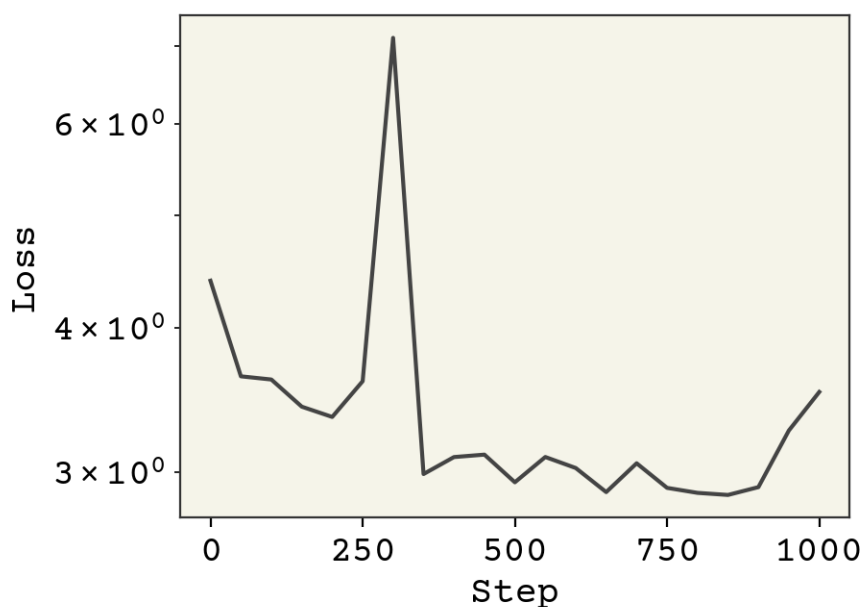
(continued from previous page)

```

# to make it random, we'll iterate over the batches randomly
np.random.shuffle(indices)
for i in indices:
    # choose a random set of
    # indices to slice our data
    grad = loss_grad(w, b, (batched_features[i], batched_labels[i]))
    w -= eta * grad[0]
    b -= eta * grad[1]
    # we still compute loss on whole dataset, but not every step
    if i % 50 == 0:
        loss_progress.append(loss_wrapper(w, b, data))

plt.plot(np.arange(len(loss_progress)) * 50, loss_progress)
plt.xlabel("Step")
plt.yscale("log")
plt.ylabel("Loss")
plt.show()

```



Notice we safely increased our learning rate to 0.01, which is possible because all the features are of similar magnitude. We also could keep training, since we're gaining improvements.

### 2.3.9 Analyzing Model Performance

This is a large topic that we'll explore more, but the first thing we typically examine in supervised learning is a **parity plot**, which shows our predictions vs. our label prediction. What's nice about this plot is that it works no matter what the dimensions of the features are. A perfect fit would fall onto the line at  $y = \hat{y}$

```

predicted_labels = linear_model(std_features, w, b)

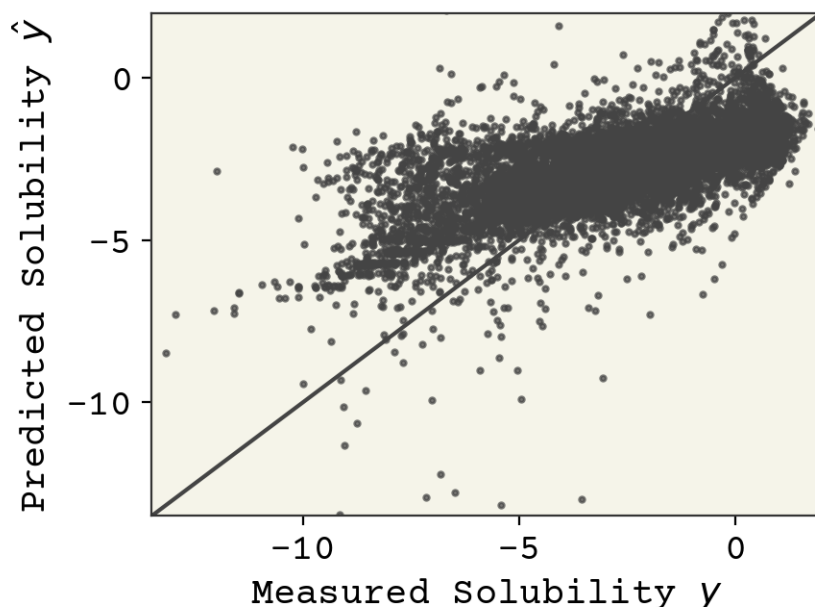
plt.plot([-100, 100], [-100, 100])
plt.scatter(labels, predicted_labels, s=4, alpha=0.7)
plt.xlabel("Measured Solubility  $y$ ")

```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Predicted Solubility  $\hat{y}$ ")
plt.xlim(-13.5, 2)
plt.ylim(-13.5, 2)
plt.show()
```



Final model assessment can be done with loss, but typically other metrics are also used. In regression, a **correlation coefficient** is typically reported in addition to loss. In our example, this is computed as

```
# slice correlation between predict/labels
# from correlation matrix
np.corrcoef(labels, predicted_labels)[0, 1]
```

```
0.6475304402750959
```

A correlation coefficient of 0.65 is OK, but not great.

## 2.4 Unsupervised Learning

In unsupervised learning, the goal is to predict  $\hat{y}$  *without* labels. This seems like an impossible task. How do we judge success? Typically, unsupervised learning can be broken into three categories:

### Clustering

Here we assume  $\{y_i\}$  is a class variable and try to partition our features into these classes. In clustering we are simultaneously learning the definition of the classes (called clusters) and which cluster each feature should be assigned to.

#### Class

In machine learning, a class is a type of label like `dog` or `cat`. Formally, we have a set of possible labels (e.g., all animals) and each feature vector has one (hard) or a probability distribution of classes (soft).



## Finding Signal

$x$  is assumed to be made of two components: noise and signal ( $y$ ). We try to separate the signal  $y$  from  $x$  and discard noise. Highly-related with **representation learning**, which we'll see later.

## Generative

Generative methods are methods where we try to learn  $P(\vec{x})$  so that we can sample new values of  $\vec{x}$ . It is analogous to  $y$  being probability and we're trying to estimate it. We'll see these more later.

### 2.4.1 Clustering

Clustering is historically one of the most well-known and still popular machine learning methods. It's always popular because it can provide new insight from data. Clustering gives class labels where none existed and thus can help find patterns in data. This is also a reason that it has become less popular in chemistry (and most fields): there is no right or wrong answer. Two people doing clustering independently will often arrive at different answers. Nevertheless, it should be a tool you know and can be a good exploration strategy.

#### cluster labels

Clustering comes in many variants and some blur what exactly  $y_i$  is. For example, in some clustering methods  $y_i$  can include no assignment or  $y_i$  is not a single class, but a tree of classes.

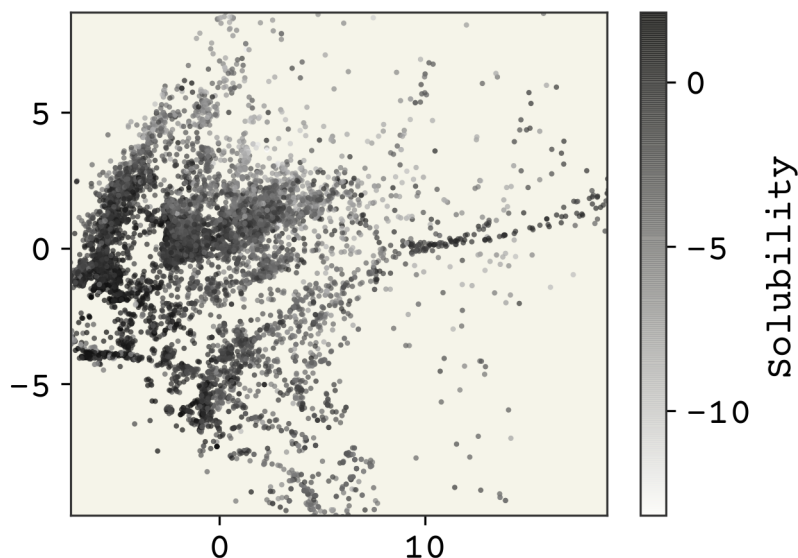
We'll look at the classic clustering method: k-means. Wikipedia has a [great article](#) on this classic algorithm, so I won't try to repeat that. To make our clustering actually visible, we'll start by projecting our features into 2 dimensions. This will be covered in representation learning, so don't worry about these steps.

```
# get down to 2 dimensions for easy visuals
embedding = sklearn.manifold.Isomap(n_components=2)
# only fit to every 25th point to make it fast
embedding.fit(std_features[:, ::25, :])
reduced_features = embedding.transform(std_features)
```

We're going to zoom into the middle 99th percentile of the data since some of the points are extremely far away (though that is interesting!).

```
xlow, xhi = np.quantile(reduced_features, [0.005, 0.995], axis=0)

plt.figure(dpi=300)
plt.scatter(
    reduced_features[:, 0],
    reduced_features[:, 1],
    s=4,
    alpha=0.7,
    c=labels,
    edgecolors="none",
)
plt.xlim(xlow[0], xhi[0])
plt.ylim(xlow[1], xhi[1])
cb = plt.colorbar()
cb.set_label("Solubility")
plt.show()
```



### Dimensionality Reduction

Reducing  $\vec{x}$ , your feature vectors to a low dimensional space. The classic example is PCA, which is a linear operator. However, most prefer nonlinear methods like [t-SNE](#).

The dimensionality reduction has made our features only 2 dimensions. We can see some structure, especially with the solubility as the coloring. Note in these kind of plots, where we have reduced dimensions in some way, we do not label the axes because they are arbitrary.

Now we cluster. The main challenge in clustering is deciding how many clusters there should be. There are a number of methods out there, but they basically come down to intuition. You, as the chemist, should use some knowledge outside of the data to intuit what is the cluster number. Sounds unscientific? Yeah, that's why clustering is hard.

```
# cluster - using whole features
kmeans = sklearn.cluster.KMeans(n_clusters=4, random_state=0)
kmeans.fit(std_features)
```

Very simple procedure! Now we'll visualize by coloring our data by the class assigned.

```
plt.figure(dpi=300)
point_colors = [f"C{i}" for i in kmeans.labels_]
plt.scatter(
    reduced_features[:, 0],
    reduced_features[:, 1],
    s=4,
    alpha=0.7,
    c=point_colors,
    edgecolors="none",
)
# make legend
legend_elements = [
    plt.matplotlib.patches.Patch(
        facecolor=f"C{i}", edgecolor="none", label=f"Class {i}"
    )
]
```

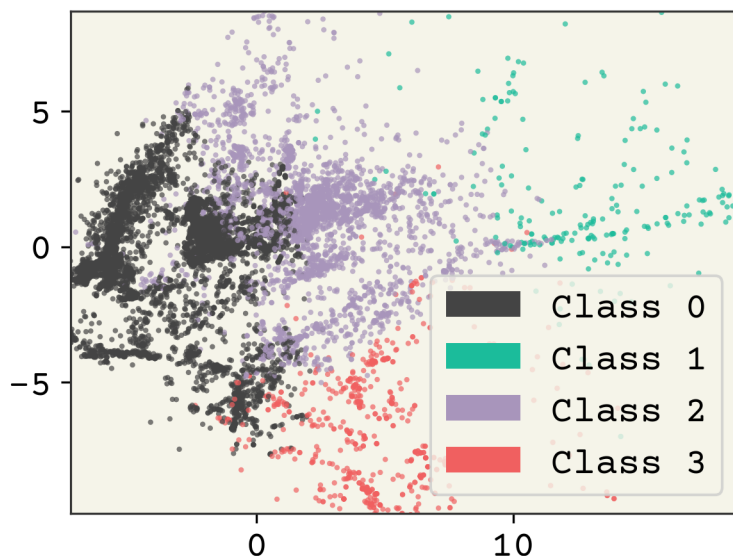
(continues on next page)

(continued from previous page)

```

    for i in range(4)
]
plt.legend(handles=legend_elements)
plt.xlim(xlow[0], xhi[0])
plt.ylim(xlow[1], xhi[1])
plt.show()

```



## 2.4.2 Choosing Cluster Number

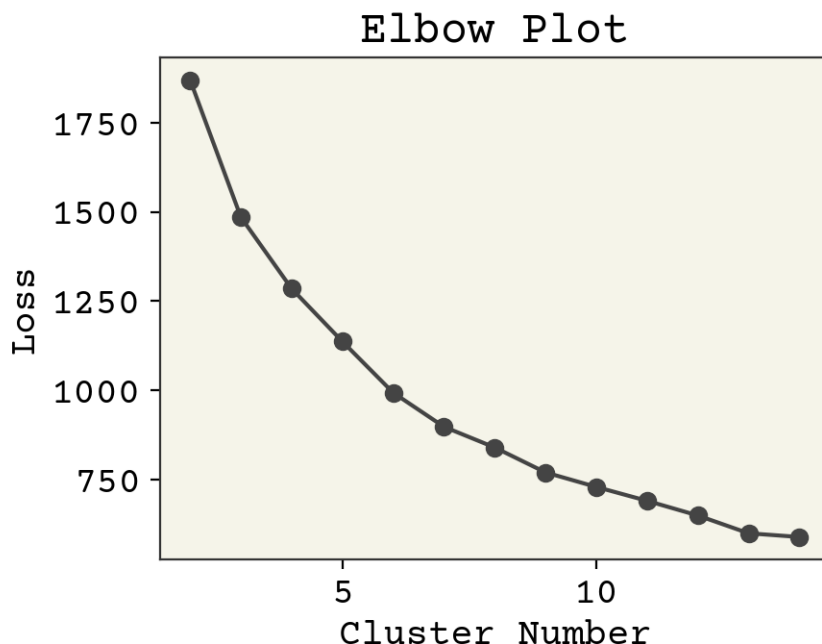
How do we know we had the correct number? Intuition. There is one tool we can use to help us, called an **elbow plot**. The k-means clusters can be used to compute the mean squared distance from cluster center, basically a version of loss function. However, if we treat cluster number as a trainable parameter we'd find the best fit at the cluster number being equal to number of data points. Not helpful! However, we can see when the slope of this loss becomes approximately constant and assume that those extra clusters are adding no new insight. Let's plot the loss and see what happens. Note we'll be using a subsample of the dataset to save time.

```

# make an elbow plot
loss = []
cn = range(2, 15)
for i in cn:
    kmeans = sklearn.cluster.KMeans(n_clusters=i, random_state=0)
    # use every 50th point
    kmeans.fit(std_features[:, :50])
    # we get score -> opposite of loss
    # so take -
    loss.append(-kmeans.score(std_features[:, :50]))

plt.plot(cn, loss, "o-")
plt.xlabel("Cluster Number")
plt.ylabel("Loss")
plt.title("Elbow Plot")
plt.show()

```



Where is the transition? If I squint, maybe at 6? 3? 4? 7? Let's choose 4 because it sounds nice and is plausible based on the data. The last task is to get some insight into what the clusters actually are. We can extract the most centered data points (closest to cluster center) and consider them to be representative of the cluster.

```
# cluster - using whole features
kmeans = sklearn.cluster.KMeans(n_clusters=4, random_state=0)
kmeans.fit(std_features)

cluster_center_idx = []
for c in kmeans.cluster_centers_:
    # find point closest
    i = np.argmin(np.sum((std_features - c) ** 2, axis=1))
    cluster_center_idx.append(i)
cluster_centers = soldata.iloc[cluster_center_idx, :]

legend_text = [f"Class {i}" for i in range(4)]

# now plot them on a grid
cluster_mols = [rdkit.Chem.MolFromInchi(inchi) for inchi in cluster_centers.InChI]
rdkit.Chem.Draw.MolsToGridImage(
    cluster_mols, molsPerRow=2, subImgSize=(400, 400), legends=legend_text
)
```

<IPython.core.display.SVG object>

So what exactly are these classes? Unclear. We intentionally did not reveal solubility (unsupervised learning) so there is not necessarily any connection with solubility. These classes are more a result of which features were chosen for the dataset. You could make a hypothesis, like class 1 is all negatively charged or class 0 is aliphatic, and investigate. Ultimately though there is no *best* clustering and often unsupervised learning is more about finding insight or patterns and not about producing a highly-accurate model.

The elbow plot method is one of many approaches to selecting cluster number [PDN05]. I prefer it because it's quite clear that you are using intuition. More sophisticated methods sort-of conceal the fact that there is no right or wrong answer in

clustering.

---

**Note:** This process does not result in a function that predicts solubility. We might try to gain insight about predicting solubility with our predicted classes, but that is not the goal of clustering.

---

## 2.5 Chapter Summary

- Supervised machine learning is building models that can predict labels  $y$  from input features  $\vec{x}$ .
- Data can be labeled or unlabeled.
- Models are trained by minimizing loss with stochastic gradient descent.
- Unsupervised learning is building models that can find patterns in data.
- Clustering is unsupervised learning where the model groups the data points into clusters

## 2.6 Exercises

### 2.6.1 Data

1. Using numpy reductions `np.amin`, `np.std`, etc. (not pandas!), compute the mean, min, max, and standard deviation for each feature across all data points.
2. Use rdkit to draw the 2 highest molecular weight molecules. Note they look strange.

### 2.6.2 Linear Models

1. Prove that a nonlinear model like  $y = \vec{w}_1 \cdot \sin(\vec{x}) + \vec{w}_2 \cdot \vec{x} + b$  could be represented as a linear model.
2. Write out the linear model equation in Einstein notation in batched form. **Batched form** means we explicitly have an index indicating batch. For example, the labels will be  $y_{bi}$  where  $b$  indicates the batch of data and  $i$  indicates the individual data point.

### 2.6.3 Minimizing Loss

1. We standardized the features, but not the labels. Would standardizing the labels affect our choice of learning rate? Prove your answer.
2. Implement a loss that is mean absolute error, instead of mean squared error. Compute its gradient using `jax`.
3. Using the standardized features, show what effect batch size has on training. Use batch sizes of 1, 8, 32, 256, 1024. Make sure you re-initialize your weights in between each run. Plot the log-loss for each batch size on the same plot. Describe your results.

### 2.6.4 Clustering

1. We say that clustering is a type of unsupervised learning and that it predicts the labels. What exactly are the predicted labels in clustering? Write down what the predicted labels might look like for a few data points.
2. In clustering, we predict labels from features. You can still cluster if you have labels, by just pretending they are features. Give two reasons why it would not be a good idea to do clustering in this manner, where we treat the labels as features and try to predict new labels that represent class.
3. On the isomap plot (reduced dimension plot), color the points by which group they fall in (G1, G2, etc.). Is there any relationship between this and the clustering?

## 2.7 Cited References

## REGRESSION & MODEL ASSESSMENT

Regression is supervised learning with continuous (or sometimes discrete) labels. You are given labeled data consisting of features and labels  $\{\vec{x}_i, y_i\}$ . The goal is to find a function that describes their relationship,  $\hat{f}(\vec{x}) = \hat{y}$ . A more formal discussion of the concepts discussed here can be found in Chapter 3 of Bishop's Pattern Recognition and Machine Learning[Bis06].

---

### Audience & Objectives

This lecture introduces some probability theory, especially expectations. You can get a refresher of [probability of random variables](#) and/or [expectations](#). Recall an expectation is  $E[x] = \sum P(x)x$  and variance is  $E[(x - E[x])^2]$ . We also use and discuss [linear regression techniques](#). After completing this chapter, you should be able to:

- Perform multi-dimensional regression with a loss function
  - Understand how to and why we batch
  - Understand splitting of data
  - Reason about model bias and model variance
  - Assess model fit and generalization error
- 

### 3.1 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

As usual, the code below sets-up our imports.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import jax.numpy as jnp
from jax.example_libraries import optimizers
import jax
import dmol
```

```
# soldata = pd.read_csv('https://dataverse.harvard.edu/api/access/datafile/3407241?
    ↪format=original&gbrecs=true')
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
    ↪csv"
)
features_start_at = list(soldata.columns).index("MolWt")
feature_names = soldata.columns[features_start_at:]
```

## 3.2 Overfitting

We'll be working again with the AqSolDB[SKE19] dataset. It has about 10,000 unique compounds with measured solubility in water (label) and 17 molecular descriptors (features). We need to create a better assessment of our supervised ML models. The goal of our ML model is to predict solubility of new unseen molecules. Therefore, to assess we should test on unseen molecules. We will split our data into two subsets: **training data** and **testing data**. Typically this is done with an 80%/20%, so that you train on 80% of your data and test on the remaining 20%. In our case, we'll just do 50%/50% because we have plenty of data and thus do not need to take 80% for training. We'll be using a subset, 50 molecules chosen randomly, rather than the whole dataset. So we'll have 50 training molecules and 50 testing molecules.

Let's begin by seeing what effect the split of train/test has on our linear model introduced in the previous chapter.

```
# Get 50 points and split into train/test
sample = soldata.sample(50, replace=False)
train = sample[:25]
test = sample[25:]

# standardize the features using only train
test[feature_names] -= train[feature_names].mean()
test[feature_names] /= train[feature_names].std()
train[feature_names] -= train[feature_names].mean()
train[feature_names] /= train[feature_names].std()

# convert from pandas dataframe to numpy arrays
x = train[feature_names].values
y = train["Solubility"].values
test_x = test[feature_names].values
test_y = test["Solubility"].values
```

We will again use a linear model,  $\hat{y} = \vec{w}\vec{x} + b$ . One change we'll make is using the `@jit` decorator from `jax`. This decorator will tell `jax` to inspect our function, simplify it, and compile it to run quickly on a GPU (if available) or CPU. The rest of our work is the same as the previous chapter. We begin with defining our loss, which is mean squared error (MSE) again.

A decorator is a Python-specific syntax that modifies how a function behaves. It is indicated with the `@` symbol. Examples include caching results, compiling the function, running it in parallel, and timing its execution.

```
# define our loss function
@jax.jit
def loss(w, b, x, y):
    return jnp.mean((y - jnp.dot(x, w) - b) ** 2)
```

(continues on next page)



(continued from previous page)

```

loss_grad = jax.grad(loss, (0, 1))
w = np.random.normal(size=x.shape[1])
b = 0.0
loss_grad(w, b, x, y)

```

```

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)

```

```

(DeviceArray([-2.8831835 ,  1.3794075 , -3.0262635 , -3.1956701 ,
             -4.007525  , -0.83476734, -3.0994835 , -4.0455275 ,
             -3.675129  ,  2.5466921 , -3.11543   , -4.171584  ,
             -1.5834932 , -3.3554041 , -3.1797354 ,  0.86207145,
             -2.0010393 ], dtype=float32),
 DeviceArray(5.7721677, dtype=float32, weak_type=True))

```

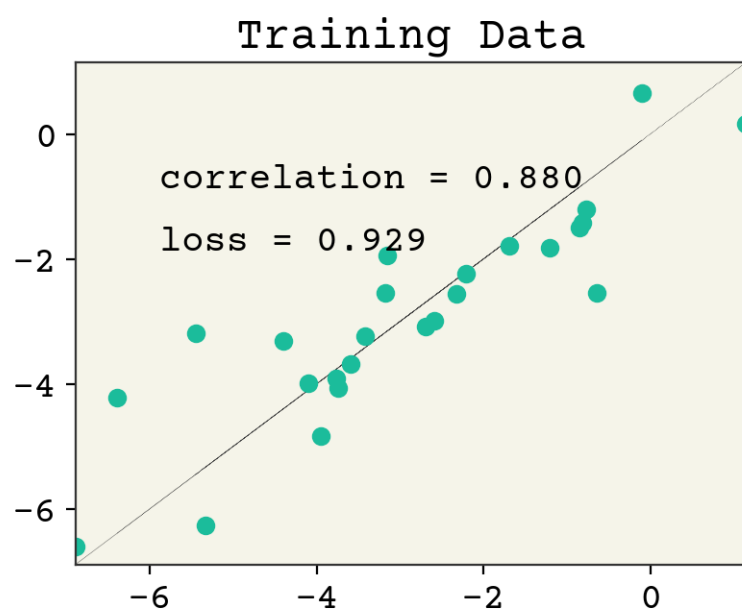
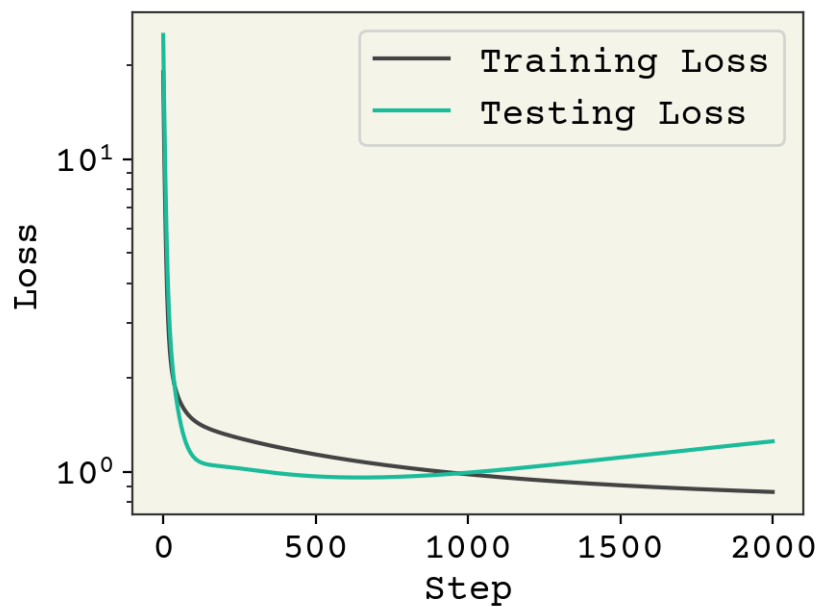
Now we will train our model, again using gradient descent. This time we will not batch, since our training data only has 25 points. Can you see what the learning rate is? Why is it so different from the last chapter when we used the whole dataset?

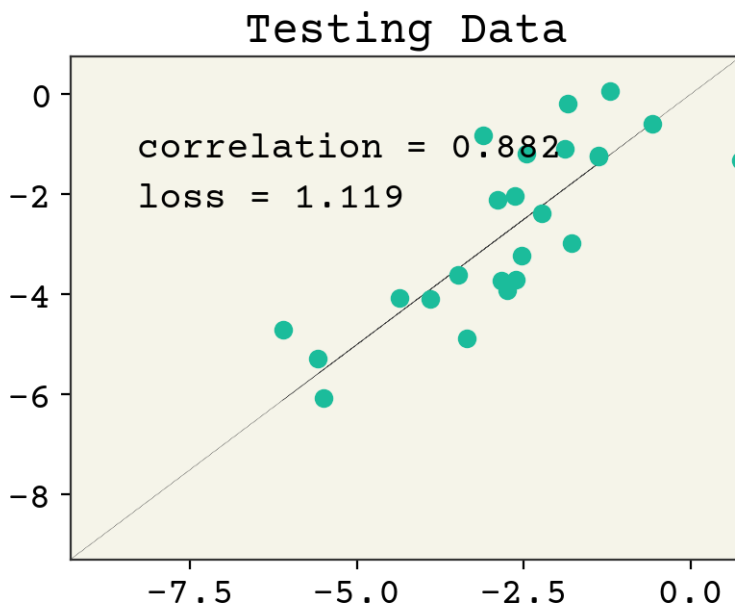
```

loss_progress = []
test_loss_progress = []
eta = 0.05
for i in range(2000):
    grad = loss_grad(w, b, x, y)
    w -= eta * grad[0]
    b -= eta * grad[1]
    loss_progress.append(loss(w, b, x, y))
    test_loss_progress.append(loss(w, b, test_x, test_y))
plt.plot(loss_progress, label="Training Loss")
plt.plot(test_loss_progress, label="Testing Loss")

plt.xlabel("Step")
plt.yscale("log")
plt.legend()
plt.ylabel("Loss")
plt.show()

```





We've plotted above the loss on our training data and testing data. The loss on training goes down after each step, as we would expect for gradient descent. However, the testing loss goes down and then starts to go back up. This is called **overfitting**. This is one of the key challenges in ML and we'll often be discussing it.

Overfitting is a result of training for too many steps or with too many parameters, resulting in our model learning the **noise** in the training data. The noise is specific for the training data and when computing loss on the test data there is poor performance.

To understand this, let's first define noise. Assume that there is a "perfect" function  $f(\vec{x})$  that can compute labels from features. Our model is an estimate  $\hat{f}(\vec{x})$  of that function. Even  $f(\vec{x})$  will not reproduce the data exactly because our features do not capture everything that goes into solubility and/or there is error in the solubility measurements themselves. Mathematically,

$$y = f(\vec{x}) + \epsilon \quad (3.1)$$

where  $\epsilon$  is a random number with mean 0 and unknown standard deviation  $\sigma$ .  $\epsilon$  is the noise. When fitting our function,  $\hat{f}(\vec{x})$ , the noise is fixed because our labels  $y$  are fixed. That means we can accidentally learn to approximate the sum of  $f(\vec{x})$  and the noise  $\epsilon_i$  instead of only capturing  $f(\vec{x})$ . The noise is random and uncorrelated with solubility. When we move to our testing dataset, this noise changes because we have new data and our model's effort to reproduce noise is useless because the new data has new noise. This leads to worse performance.

Overfitting arises when three things happen: you have noise, you have extra features or some part of your features are not correlated with the labels, and your training has converged (your model fit is at the global minimum). This last one is what we saw above. Our model wasn't overfit after about 100 steps (the training and testing loss were both decreasing), but then they starting going in opposite directions. Let's see how these things interplay to lead to overfitting in an example where we can exactly control the features and noise.

### 3.2.1 Overfitting with Synthetic Data

We'll explore overfitting in a synthetic example. Our real function we're trying to learn will be:

$$f(x) = x^3 - x^2 + x - 1 \quad (3.2)$$

which we can rewrite as a linear model:

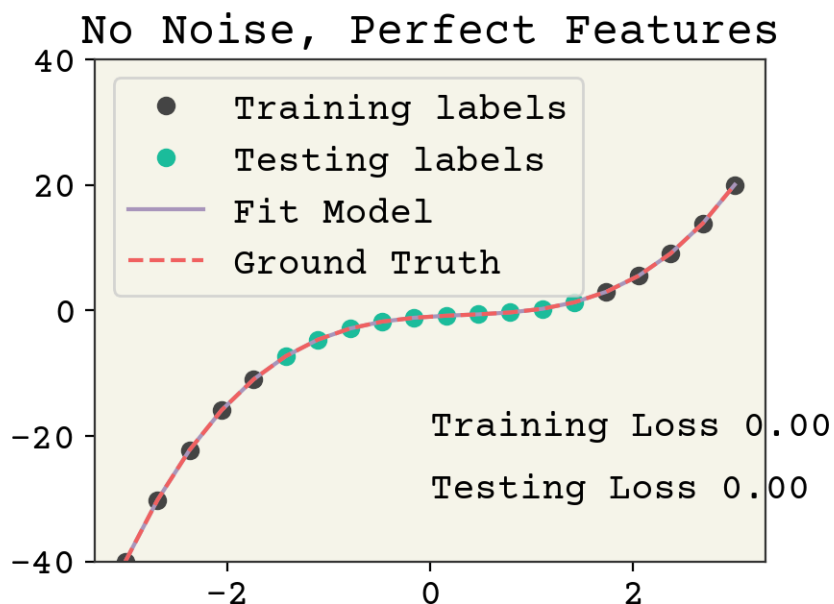
$$f(\vec{x}) = \vec{w} \cdot \vec{x} = [1, -1, 1, -1] \cdot [x^3, x^2, x, 1] \quad (3.3)$$

where our features are  $[x^3, x^2, x, 1]$ . To do our split, we'll take the positive points as training data and the negative as testing data. To avoid the issue of convergence, we will use least squares to fit these models instead of gradient descent.

Let's establish a benchmark. How well can a model do without noise? We'll use 10 training data points and 10 testing data points. We'll put our testing data in the center of the polynomial.

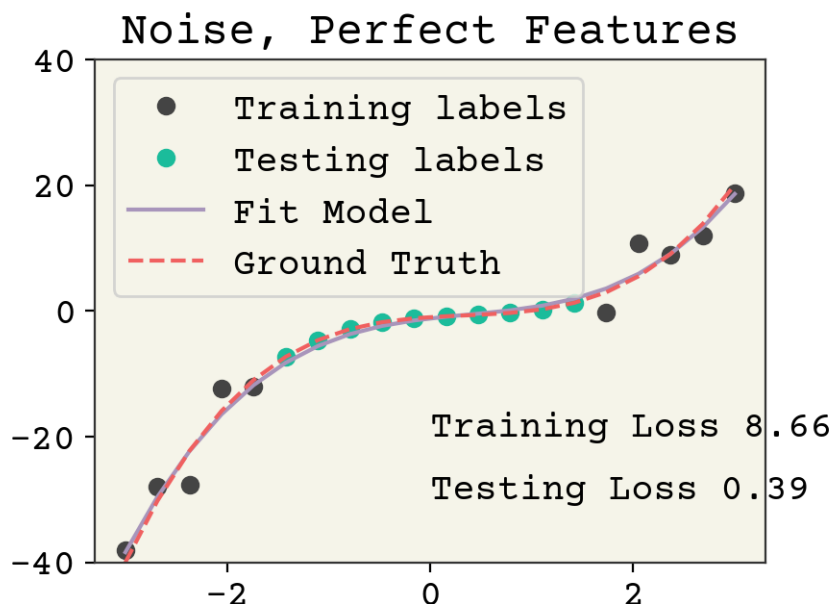
Expand the Python cells below to see how this is implemented.

```
# generate data from polynomial
N = 20
syn_x = np.linspace(-3, 3, N)
# create feature matrix
syn_features = np.vstack([syn_x**3, syn_x**2, syn_x, np.ones_like(syn_x)]).T
syn_labels = syn_x**3 - syn_x**2 + syn_x - 1
```



There is no overfitting and the regression is quite accurate without noise. Now we'll add noise to both the training labels.

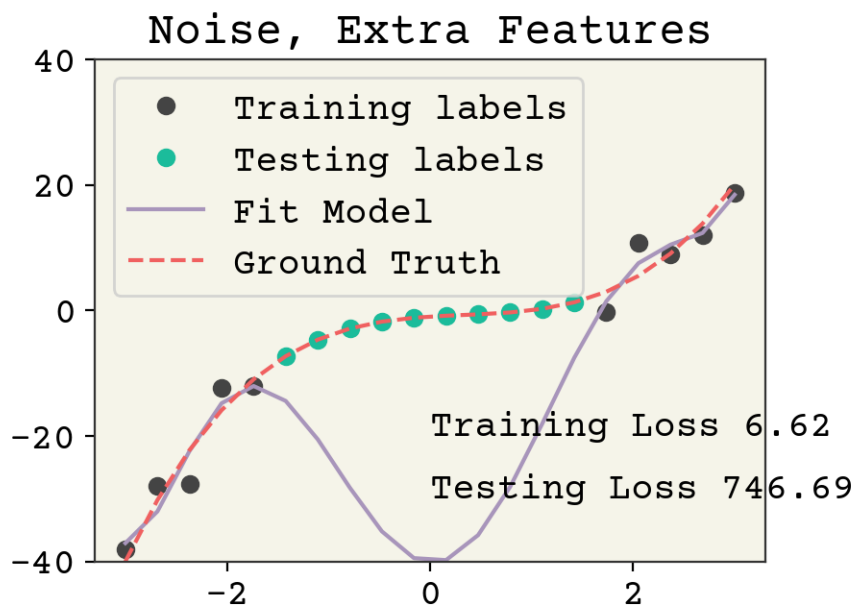
```
train_y = train_y + np.random.normal(scale=5, size=train_y.shape)
```



Adding noise reduces the accuracy on the training data. The testing labels have no noise and the model is not overfit, so the accuracy is good for the testing loss.

Now we'll try adding redundant features. Our new features will be  $[x^6, x^5, x^4, x^3, x^2, x, 1]$ . Still less than our data point number but not all features are necessary to fit the labels.

```
syn_features = np.vstack([syn_x**i for i in range(7)]).T
```

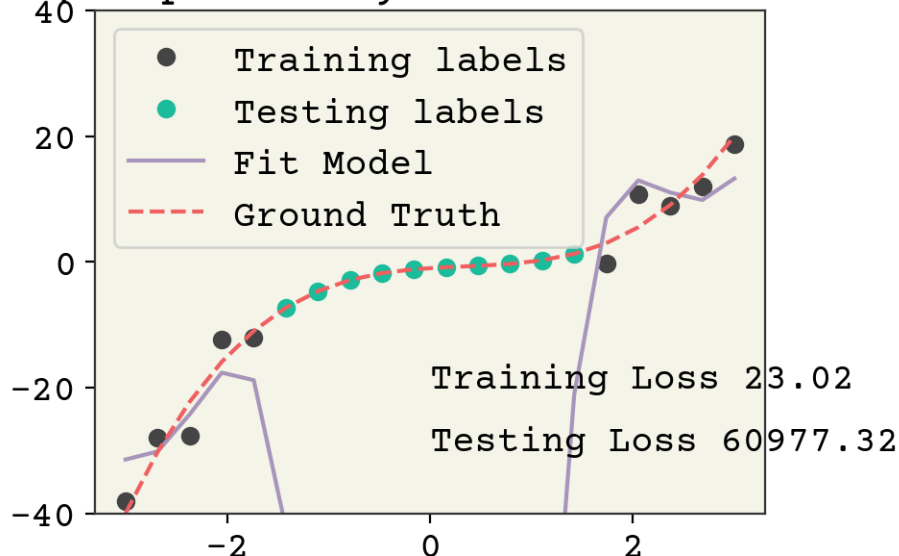


This is an overfit model. The training loss went down (note the noise was the same in the previous two examples), but at the expense of a large decrease in testing loss. This wasn't possible in the previous example because over-fitting to noise wasn't feasible when each feature was necessary to capture the correlation with the labels.

Let's see an example where the feature number is the same but they aren't perfectly correlated with labels, meaning we cannot match the labels even if there was no noise.

```
syn_features = np.vstack([
    syn_x**2, syn_x, np.exp(-(syn_x**2)), np.cos(syn_x), np.ones_like(syn_x)]
).T
```

## Noise, Imperfectly Correlated Features



It's arguable if this is overfitting. Yes, the testing loss is high but it could be argued it's more to do with the poor feature choice. In any case, even though our parameter number is less than the clear cut case above, there is still left over variance in our features which can be devoted to fitting noise.

Would there overfitting with fewer features that are perfectly correlated with labels?

### Answer

Yes, because we can use the left over variance in our features to fit noise.

## 3.2.2 Overfitting Conclusion

- Overfitting is inevitable in real data because we cannot avoid noise and rarely have the perfect features.
- Overfitting can be assessed by splitting our data into a train and test split, which mimics how we would use the model (i.e., on unseen data).
- Overfitting is especially affected by having too many features or features that don't correlate well with the labels.
- We can identify overfitting from a loss curve which shows the testing loss rising while training loss is decreasing.

### 3.3 Exploring Effect of Feature Number

We've seen that overfitting is sensitive to the number and choice of features. Feature selection is a critical decision in supervised learning. We'll return to the solubility dataset to discuss this. It has 17 molecular descriptors, but these are just a small fraction of the possible molecular descriptors that can be used. For example, there is a software called [Dragon](#) that can compute over 5,000 descriptors. You can also create linear combinations of descriptors and pass them through functions. Then there is the possibility of experimental data, data from molecular simulations, and from quantum calculations. There is essentially an unlimited number of possible molecular descriptors. We'll start this chapter by exploring what effect of number of features (dimension of features) has on the data.

**Descriptor** is chemistry and materials specific word for feature. It pre-dates the word features and comes from the field of “quantitative-structure activity relationship” (QSAR), which has a long history in drug design and molecular design.

We are now working with a real dataset, which means there is randomness from which features we choose, which training data we choose, and randomness in the labels themselves. In the results below, they are averaged over possible features and possible training data splits to deal with this. Thus the code is complex. You can see it on [the Github repository](#), but I've omitted it here for simplicity.

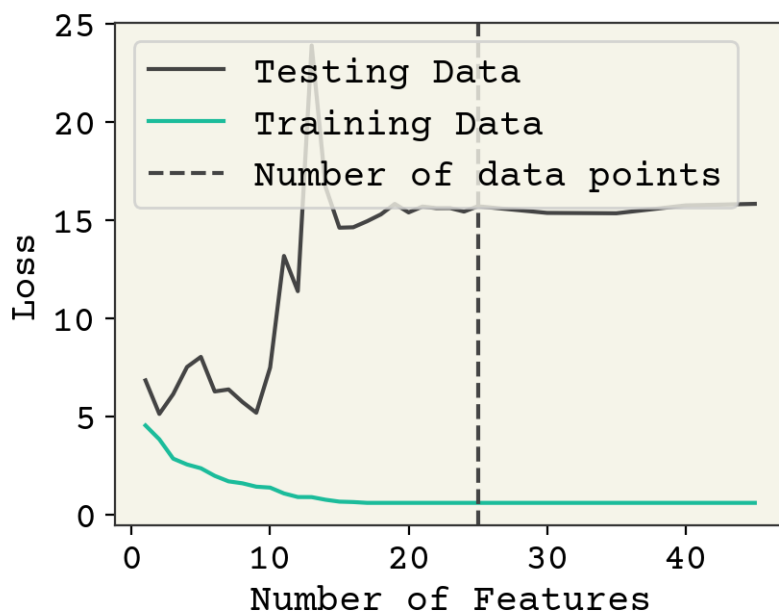


Fig. 3.1: Effect of feature number on 25 training data points averaged over 10 data samples/feature choices combinations. Notice there is not a significant change when the number of features crosses the number of data points.

Fig. 3.1 shows the effect of choosing different features on both the loss on training data and the loss on test data. There are three regimes in this plot. At 1-3 features, we are **underfit** meaning both the training and testing losses could be improved with more features or more training. In this case, it is because there are too few features. Until about 10 features, we see that adding new features slightly improves training data but doesn't help test data meaning we're probably slightly overfitting. Then at 10, there is a large increase as we move to the overfit regime. Finally at about 30 features, our model is no longer converging and training loss rises because it is too difficult to train the increasingly complex model. “Difficult” here is a relative term; you can easily train for more time on this simple model but this is meant as an example.

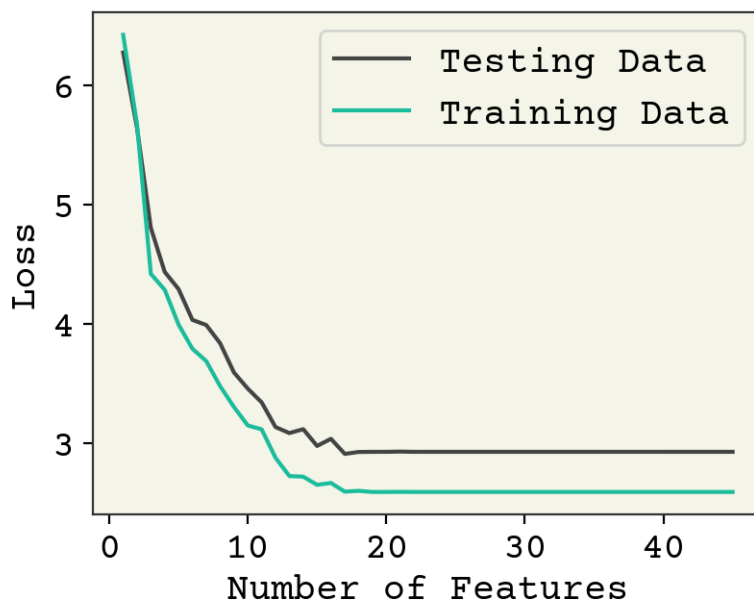


Fig. 3.2: Effect of feature number on 250 training data points averaged over 10 data samples/feature choices combinations.

Fig. 3.2 shows the same analysis but for 250 train and 250 test data. The accuracy on test data is better (about 1.9 vs 2.5). There is not much overfitting visible here. The model is clearly underfit until about 10 features and then each additional feature has little effect. Past 20 features, we again see an underfit because the model is not trained well. This could be fixed by adding more training steps.

Increasing feature numbers is useful up to a certain point. Although some methods are unstable when the number of features is exactly the same as the number of data points, there is reason overfitting begins at or near feature numbers equal to the number of data points. Overfitting can disappear at large feature numbers because of model size and complexity. Here there is also a risk of underfitting.

The risk of overfitting is lower as your dataset size increases. The reason for this is that the noise becomes smaller than the effect of labels on training as you increase data points. Recall from the Central Limit Theorem that reducing noise by a factor of 10 requires 100 times more data, so this is not as efficient as choosing better features. Thinking about these trade-offs, to double your feature number you should quadruple the number of data points to reduce the risk of overfitting. Thus there is a strong relationship between how complex your model can be, the achievable accuracy, the data required, and the noise in labels.

### 3.4 Bias Variance Decomposition

We will now try to be more systematic about this difference in model performance between training and testing data. Consider an unseen label  $y$  and our model  $\hat{f}(\vec{x})$ . Our error on the unseen label is:

$$E \left[ \left( y - \hat{f}(\vec{x}) \right)^2 \right] \quad (3.4)$$

What is the expectation over? For now, let's just assume the only source of randomness is in the noise from the label (recall  $y = f(\vec{x}) + \epsilon$ ). Then our expression becomes:

$$E \left[ \left( y - \hat{f}(\vec{x}) \right)^2 \right] = E \left[ y^2 \right] + E \left[ \hat{f}(\vec{x})^2 \right] - 2E \left[ y \hat{f}(\vec{x}) \right] \quad (3.5)$$



$$E \left[ (y - \hat{f}(\vec{x}))^2 \right] = E \left[ (f(\vec{x}) - \epsilon)^2 \right] + \hat{f}(\vec{x})^2 - 2E[(f(\vec{x}) - \epsilon)] \hat{f}(\vec{x}) \quad (3.6)$$

I have dropped the expectations over deterministic expression  $\hat{f}$ . You can continue this, again dropping any  $E[f(\vec{x})]$  terms and using the definition of  $\epsilon$ , a zero mean normal distribution with standard deviation  $\sigma$ . You will arrive at:

$$E \left[ (y - \hat{f}(\vec{x}))^2 \right] = (f(\vec{x}) - \hat{f}(\vec{x}))^2 + \sigma^2 \quad (3.7)$$

This expression means the best we can do on an unseen label is the noise of the label. This is very reasonable, and probably matches your intuition. The best you can do is match exactly the noise in the label when you have a perfect agreement between  $f(\vec{x})$  and  $\hat{f}(\vec{x})$ .

However, this analysis did not account for the fact our choice of training data is random. Things become more complex when we consider that our choice of training data is random. Return to Equation (3.4) and now replace  $\hat{f}(\vec{x})$  with  $\hat{f}(\vec{x}; \mathbf{D})$  where  $\mathbf{D}$  is a random variable indicating the random data sample. You can find a complete derivation on [Wikipedia](#). The key change is that  $(f(\vec{x}) - \hat{f}(\vec{x}; \mathbf{D}))^2$  is now a random variable. Equation (3.7) becomes:

$$E \left[ (y - \hat{f}(\vec{x}))^2 \right] = E \left[ f(\vec{x}) - \hat{f}(\vec{x}; \mathbf{D}) \right]^2 + E \left[ (E[\hat{f}(\vec{x}; \mathbf{D})] - \hat{f}(\vec{x}; \mathbf{D}))^2 \right] + \sigma^2 \quad (3.8)$$

This expression is the most important equation for understanding ML and deep learning training. The first term in this expression is called **bias** and captures how far away our model is from the correct function  $f(\vec{x})$ . This is the expected (average) loss we get given a random dataset evaluated on a new unseen data point. You may think this the most important quantity – expected difference between the true function and our model on a new data point. However, bias does not determine the expected error on an unseen data point alone, there other terms.

In Equation(3.8)  $\vec{x}$  is a fixed quantity, unlike what you may be used to in probability. The actual random variables are  $\epsilon$  (noise in label) and  $\mathbf{D}$  (our chosen training data).

The second term is surprising. It is called the **variance** and captures how much change at the unseen data point  $(\vec{x}, y)$  there is due to changes in the random variable  $\mathbf{D}$ . What is surprising is that the expected loss depends on the variance of the learned model. Think carefully about this. A model which is highly sensitive to which training data is chosen has a high expected error on test data. Furthermore, remember that this term **variance** is different than variance in a feature. It captures how the model value changes at a particular  $\vec{x}$  as a function of changing the training data.

**Note:** There are three sources of randomness in the expectation: the choice of test data, the label noise, and the choice of training data. However, once you pick the training data, the test data is fixed so we do not indicate or worry about this. A quantity like  $E[\hat{f}(\vec{x})]$  means splitting your data every possible way, fitting the models, then computing the value  $\hat{f}(\vec{x})$  on the unseen test  $\vec{x}$ . Then you take the average over the unseen test values. You can also skip the last step and leave  $E[\hat{f}(\vec{x})]$  as a function of  $\vec{x}$ , which is what is plotted in [Fig. 3.3](#) and [Fig. 3.4](#).

These three terms: noise, bias, and variance set the minimum value for test error. Noise is set by your data and not controllable. However, bias and variance are controllable. What does a high bias, low variance model look like? A 1D linear model is a good example. See [Fig. 3.3](#). It has one parameter so a sample of data points gives a consistent estimate. However, a 1D model cannot capture the true  $f(\vec{x})$  so it has a large average error (bias) at a given point. What does a low bias, high variance model look like? An overfit model like the one shown in [Fig. 3.4](#). It has extreme outliers on test data, but on average it actually has a low bias.

### The Tradeoff

The way to change bias and variance is through **model complexity**, which is feature number in our linear models. Increasing model complexity reduces bias and increases variance. There is an optimum for our polynomial example, shown

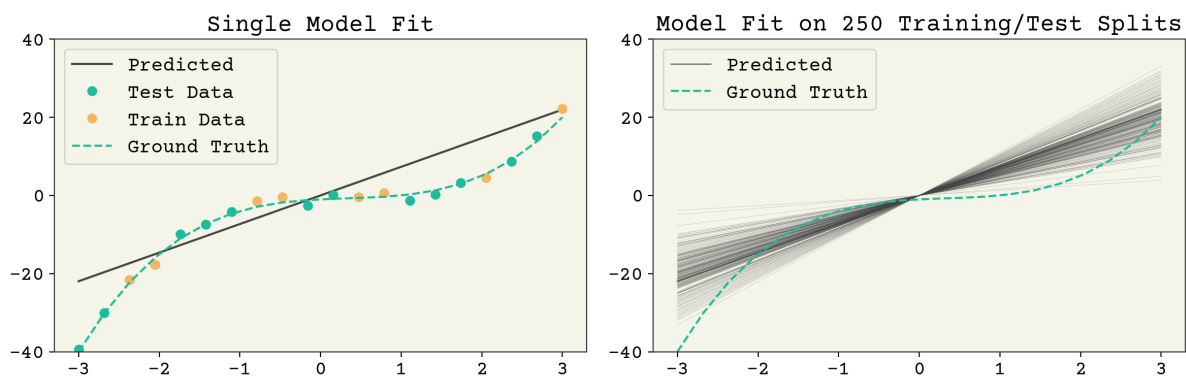


Fig. 3.3: A single feature fit to the polynomial model example above. The left panel shows a single train/test split and the resulting model fit. The right panel shows the result of many fits. The model variance is the variance across each of those model fits and the bias is the agreement of the average model. It can be seen that this model has low variance but poor average agreement (high bias).

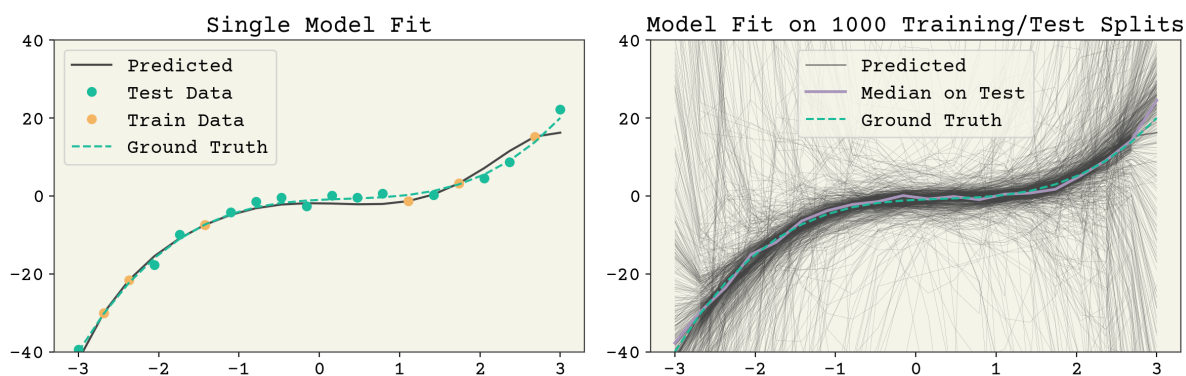


Fig. 3.4: A 7 feature fit to the polynomial model example above. The left panel shows a single train/test split and the resulting model fit. The right panel shows the result of many fits. The model variance is the variance across each of those model fits and the bias is the agreement of the average model. It can be seen that this model has high variance but good average agreement (low bias).

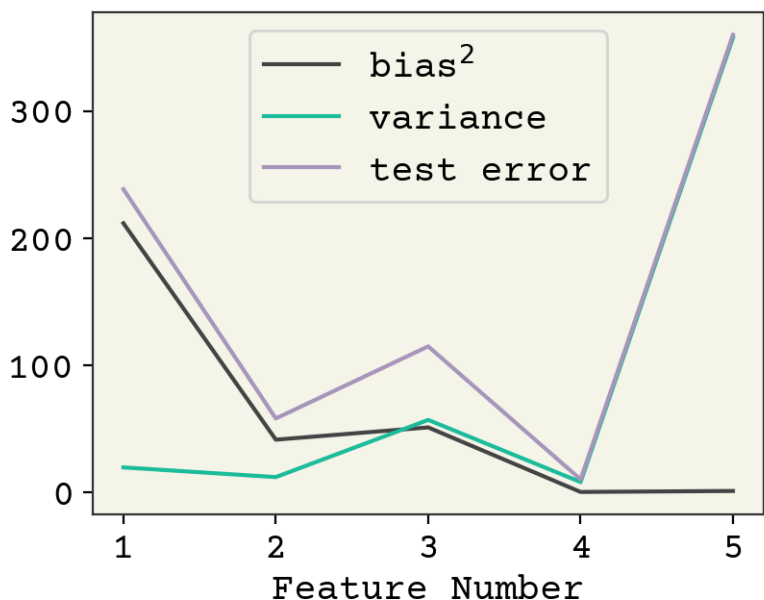


Fig. 3.5: The bias, variance, and fit on test values for the polynomial example averaged across 2,500 train/test splits. As the number of features increases, variance increases and bias decreases. There is a minimum at 4 features. The plot stops at 5 because the variance becomes very large beyond 5.

in Fig. 3.5. Indeed this is true of most ML models, although it can be difficult to cleanly increase model complexity and keep training converged. However, this is *not typically true in deep learning with neural networks*[NMB+18].

---

**Note:** The bias–variance tradeoff for model complexity is based on experience. The decomposition above does not prove a tradeoff, just that you can split these two terms. Intentionally underfitting, adding noise, and exchanging one feature for another are all ways to affect bias and variance without adjusting complexity. Also, sometimes you can just improve both with better models.

---

The bias–variance tradeoff is powerful for explaining the intuition we’ve learned from examples above. Large datasets reduce model variance, explaining why it is possible to increase model complexity to improve model accuracy only with larger datasets. Overfitting reduces bias at the cost of high variance. Not training long enough increases bias, but reduces variance as well since you can only move so far from your starting parameters.

## 3.5 Regularization

Adding features is a challenging way to exchange model bias and variance because it comes in discrete steps and some features are just better than others. A different way is to use a complex model (all features) but reduce variance through **regularization**. Regularization is the addition of an extra term to your loss function that captures some unwanted property about your model that you want to minimize.

### 3.5.1 L2 Regularization

You can add the bias  $b$  to the regularization term, but this should only be done if you have some prior belief that the bias should be 0 – like if it represents some physical quantity that should be minimized. Otherwise minimizing  $b$  has no effect on overfitting and so is not part of regularization.

Our first example is the magnitude of fit coefficients. The magnitude of the coefficients is  $\sum_k w_k^2$  where  $w_k$  the index of a single coefficient. We add this to our loss function:

$$L = \frac{1}{N} \sum_i^N [y_i - \hat{f}(\vec{x}_i, \vec{w}, b)]^2 + \lambda \sum_k w_k^2 \quad (3.9)$$

where  $\lambda$  is our strength of regularization. By changing  $\lambda$ , we control how large the magnitude of our parameters are and that directly reduces the variance. You can see the result in Fig. 3.6 for our polynomial example. Increasing the strength of regularization decreases variance at the cost of increasing model bias. Remember in deep learning there isn't as much of a tradeoff and often you just get a reduction in variance with no degradation in bias. Adding L2 regularization with a linear model has a specific name: **Ridge Regression**.

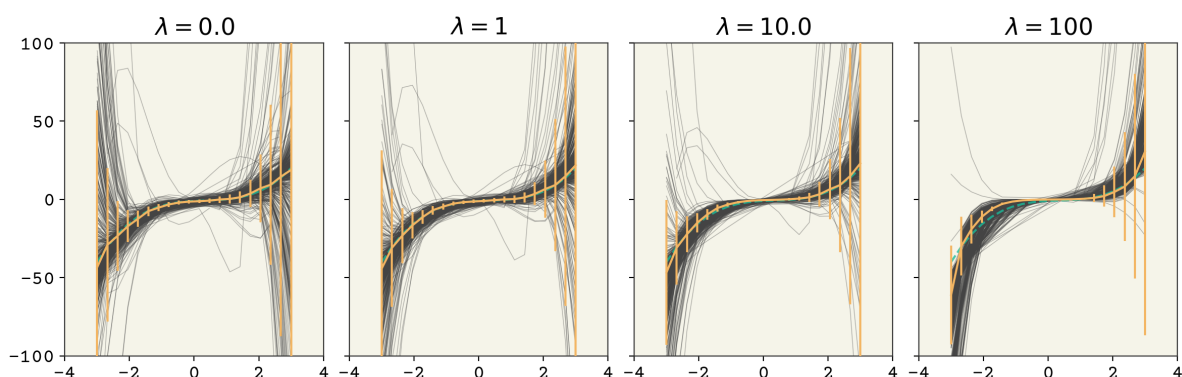


Fig. 3.6: A 7 feature fit to the polynomial model example above with increasing strength of regularization. The vertical bars indicate standard deviation of model at each point.

Why does this work? Look at the gradient of a particular weight of our new loss function:

$$\frac{\partial L}{\partial w_4} = \frac{2}{N} \sum_i^N [y_i - \hat{f}(\vec{x}_i, \vec{w}, b)] \frac{\partial \hat{f}(\vec{x}_i, \vec{w}, b)}{\partial w_4} + 2\lambda w_4 \quad (3.10)$$

where  $w_4$  is one of our weights. The first term on the right-hand side accounts for how  $w_4$  affects our accuracy, like usual. The second term is from the regularization. You can see that the gradient is just the value of weight times a constant. Let's contract the first term into a variable called  $g_{w_4}$  and look at how this new gradient affects our updates to  $w_4$ . Our gradient descent update of  $w_4$  becomes:

$$w'_4 = w_4 - \eta g_{w_4} - 2\eta \lambda w_4 \quad (3.11)$$

So our regularization pushes  $w'_4$  to always have a lower magnitude. If  $w'_4 = 2.5$ , the update will include a term of  $-2\eta \lambda 2.5$ , pushing our weight value closer to zero. This means our weights always are pushed towards zero. Of course the term coming from model error ( $g_{w_4}$ ) also has an effect so that we end up at a balance of lower magnitude weights and model error. We control that balance through the  $\lambda$  term.

The terms L1 and L2 come from the definition of norms. They indicate the coefficient used in the norm:  $(\sum_i x_i^p)^{1/p}$ , where  $p = 1$  for L1 and  $p = 2$  for L2. Others exist, like  $p = 0$  which counts dimension and  $p = \infty$  which takes the maximum element. The “L” comes from the word Lebesgue integral, via a confusing path.

### 3.5.2 L1 Regularization

L1 regularization changes our loss to be the following:

$$L = \frac{1}{N} \sum_i \left[ y_i - \hat{f}(\vec{x}_i, \vec{w}, b) \right]^2 + \lambda \sum_k |w_k| \quad (3.12)$$

It may appear at first that this is identical to L2. In fact, the L1 regularization has a powerful benefit: it induces sparsity. L2 just causes regression coefficients to be on average lower, but L1 forces some coefficients to be 0. This gives us a kind of “automatic” feature selection. This is called **Lasso Regression** when you combine L1 regularization with linear regression.

As far as choosing which regularization to use, I’ll quote Frank Harrell, a biostatistics professor at Vanderbilt:

Generally speaking if you want optimum prediction use L2. If you want parsimony at some sacrifice of predictive discrimination use L1. But note that the parsimony can be illusory, e.g., repeating the lasso process using the bootstrap [introduced below] will often reveal significant instability in the list of features “selected” especially when predictors are correlated with each other.

## 3.6 Strategies to Assess Models

We will now discuss more ways to assess model performance. These are more robust approaches to assess loss on testing data.

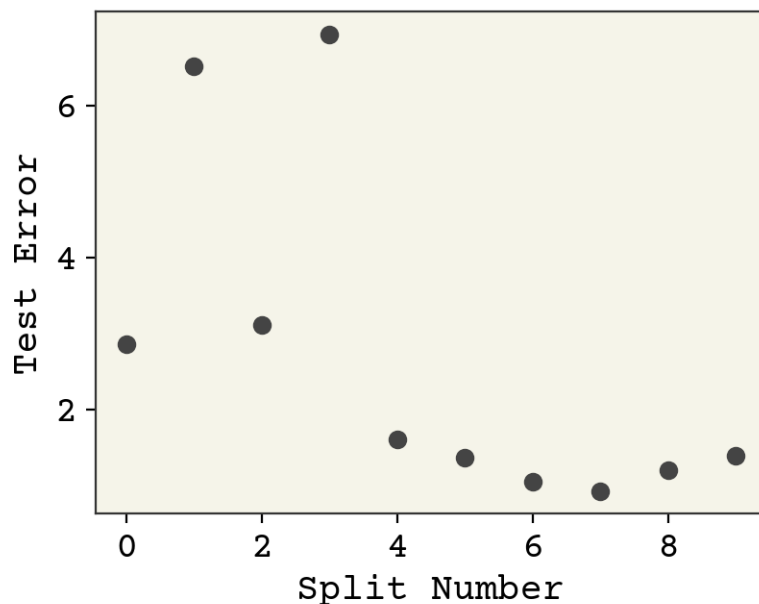
### 3.6.1 k-Fold Cross-Validation

The bias–variance decomposition shows that our testing error is sensitive to what training data has been chosen. The expected mean test error  $E \left[ (y - \hat{f}(\vec{x}))^2 \right]$  depends on the label noise **and** the way we split our data into training and testing data. Thus far, we’ve only gotten a single sample from this expectation by splitting. One way to better estimate the value on unseen data is to repeat the process of splitting data into training and testing multiple times. This is called **k-fold** cross-validation, where  $k$  is the number of times you repeat the process. k-fold cross-validation is useful because certain high-variance model choices can give different testing errors depending on the train/test split. k-fold also provides multiple samples so that you can estimate the **uncertainty** in testing error. As all things to do with model variance, the smaller the dataset the more important this is. Typically with very large datasets k-fold cross-validation is not done because label noise dominates and testing a model  $k$  times can be time-consuming.

k-fold cross-validation has a specific process for splitting testing and training data. What we did previously was split into a 50/50 split of training and testing. In k-fold, we split our data into  $k$  segments. Then we train on  $k-1$  segments and test on the last segment. You can do this  $k$ -ways. For example, with  $K = 3$  you would split your data into A, B, C. The first train/test split would be A, B for training and C for testing. Then B, C for training and A for testing. The last would be A, C for training and B for testing. Following this procedure means that your percentage split will be 90/10 for  $k = 10$  and 50/50 for  $k = 2$ . This has a disadvantage that the number of estimates for testing error depends on size of train/test split. For example, you cannot get 10 estimates for an 80/20 split. An 80/20 split means exactly 5-fold cross-validation. We’ll see other methods that relax this later on. The 80/20 is a typical rule that balances having enough data for good training and enough to robustly assess how well your model performs.

Let's now use k-fold cross-validation in two examples: our full dataset and a smaller 25 data point sample. Rather than using gradient descent here, we'll just use the pseudo-inverse to keep our code simple. The pseudo-inverse is the least-squares solution.

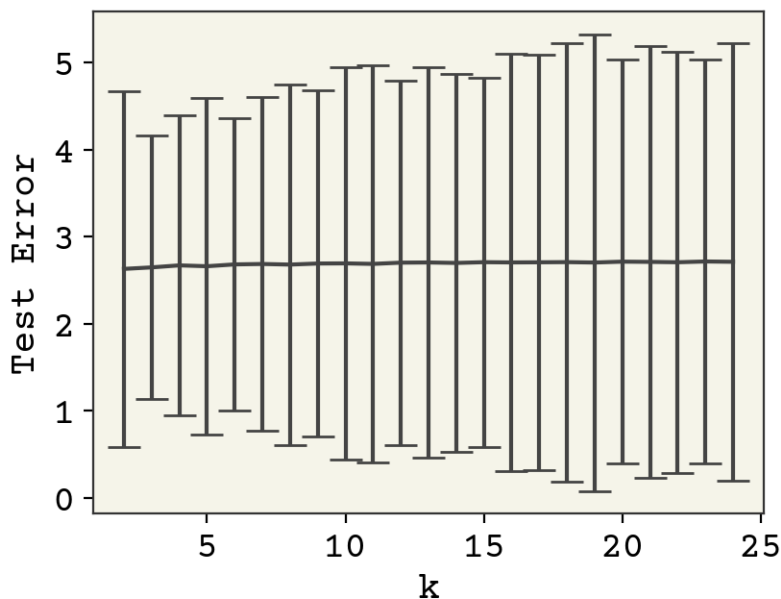
### 10-fold cross-validation of soldata



The final answer in this case is the average of these values:  $2.69 \pm 2.25$ . The advantage of the k-fold is that we can report standard deviation like this.

Now what effect does k have on the test error? Let's see how our choice of k matters

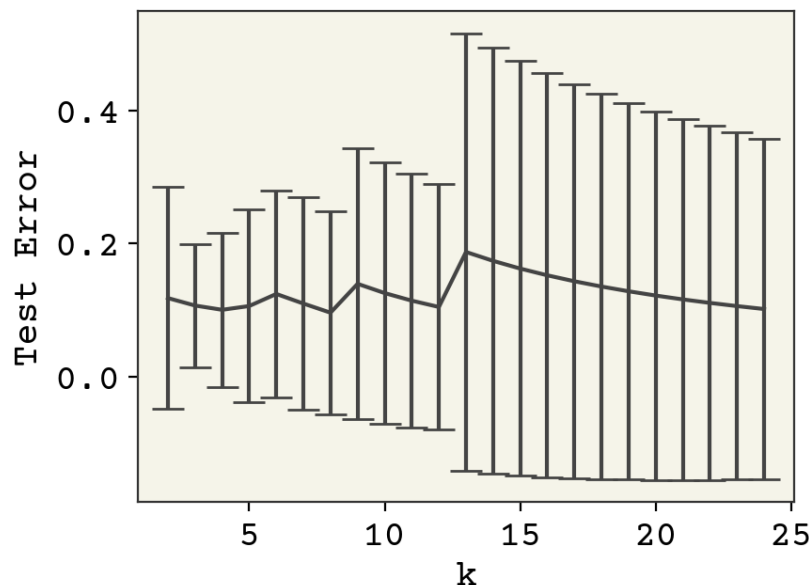
### k-fold cross-validation of soldata



As you can see, there is not much sensitivity to k. This is good, because k is mostly arbitrary. Larger k means more samples, but each test data is smaller so that these two effects should balance out.

Large datasets are not that sensitive because the training and testing splits are large. Let us examine what happens with  $N = 25$ , a realistic case in chemistry data. We'll just pick 25 data points at the beginning and not change that choice, mocking what would happen in a real example.

### k-fold cross-validation of soldata subsample



Our results are a little sensitive to the choice of  $k$ . Now why might test error decrease? Remember that a larger  $k$  means *more* data points for training. This did not matter above when we had 10,000 data points. Now it is very important, since we only have 25 data points. Thus larger  $k$  means more training data.

### 3.6.2 Leave-one-out CV

Larger  $k$  means more training data, so what is the largest it can be? Remember that  $k$  is the number segments in your data. So  $k = N$  is the max, where each data point is a segment. This is called **leave-one-out cross-validation (LOOCV)**. It creates  $N$  different models, one for each data point left out, and so is only used for small datasets. Thus the advantage of LOOCV is it maximizes training data, but maximizes the number of times the model needs to be trained.

## 3.7 Computing Other Measures

Using LOOCV and  $k$ -fold cross-validation, we're able to predict test error. This "test error" is specifically an expected error on an unseen data point. Now how do we actually treat a new data point? What will we report as the certainty in a new point? The test error? We'll call this point the **prediction point** and we'll try to estimate the quantiles of this point. Quantiles are the building blocks for confidence intervals. Recall that confidence intervals allow us to report our model prediction as  $4.3 \pm 0.2$ , for example.

Classically bootstrap resampling and **jackknife**, its predecessor, are used for estimating variance in model parameters (i.e., model variance). However, they are more commonly used in ML for predicting confidence intervals and/or test error for new points (also called generalization error).

### 3.7.1 Bootstrap Resampling

To estimate quantiles, we need to have a series of observations of predictions from the prediction point  $\hat{f}(\vec{x}')$ , where  $\vec{x}'$  is the prediction point. For example, we could do 5-fold cross-validation and have 5 estimates of  $\hat{f}_k(\vec{x}')$  and could estimate the quantiles using a t-statistic. Instead, we'll use a method called **bootstrap resampling** which removes the restriction that we can only use  $1 - 1/k$  of the training data. Bootstrap resampling is a general process for estimating uncertainty for empirical statistics without assuming a probability distribution (i.e., non-parametric). In bootstrap resampling, we create as many as desired new training datasets that are the same size as the original by sampling **with replacement** from the original dataset. That means our new dataset has fewer members than the original and makes up the difference with duplicates. Let's see an example. If your training dataset originally has data A, B, C, D, E, our bootstrap resampled training data is:

1. A, B, B, D, E
2. B, C, C, C, E
3. A, B, D, E, E
4. A, B, C, D, E
5. A, A, C, C, D

and so forth. The “with replacement” means that we allow repeats. This gives some variation to our training data. It also means we can generate  $2^N$  new datasets, which is practically as many as we want. Let's see now how we could use this to quantile the estimate for a prediction on a test point. We'll set  $N = 1000$  and do bootstrap resampling for 100 iterations.

```
# Create training data and 1 test point
N = 1000
# this line gets the data for our example
# it is not the bootstrap resampling
tmp = soldata.sample(N + 1, replace=False)
small_soldata = tmp.iloc[:N]
predict_point = tmp.iloc[-1]
```

```
samples = 100
predictions = []
for i in range(samples):
    # choose with replacement indices to make new dataset
    idx = np.random.choice(np.arange(N), size=N, replace=True)
    train = small_soldata.iloc[idx]
    x, y = train[feature_names].values, train["Solubility"].values
    # compute coefficients
    w, *_ = np.linalg.lstsq(x, y)
    # compute intercept (b)
    b = np.mean(y - np.dot(x, w))
    # compute test prediction
    predictions.append(np.dot(predict_point[feature_names].values, w) + b)
# compute quantiles (lower, median, upper)
qint = np.quantile(predictions, [0.025, 0.5, 0.975])
# compute avg distance from median to report +/-
print(
    f'prediction = {qint[1]:.2f} +/- {(qint[-1] - qint[0]) / 2:.2f}, label = {predict_
point["Solubility"]:.2f}'
)
```

```
prediction = -4.08 +/- 0.41, label = -3.45
```



The resulting prediction has confidence intervals, thanks to the bootstrap resampling. This approach has a few disadvantages though. The first is that we need to produce and keep 100 models, one for each bootstrap resample. Of course you could choose fewer, but you need to have enough for good statistics.

Another issue is that this process does not give a reportable test error. We could further split our data again and do k-fold cross-validation on this approach to get test error. However, this is a bit overly complex and then we'll be at a similar problem that we'll have k sets of 100 models and it's not obvious how to combine them. These prediction intervals also under-estimate the model bias, because it has no estimate of the test error. It only accounts for variation due to training data. Using the language above, it only accounts for model variance but not model bias.

Bootstrap resampling is still an excellent technique that is used often to estimate uncertainties. However, it is not a great choice for estimating model error on unseen datapoints.

### 3.7.2 Jackknife+

There is a method called Jackknife, which does not compute multiple predictions. It computes the residuals as mentioned above, but it trains one final model on all data. Since it requires you to compute all  $N$  models to get the residuals, it is preferred to just use Jackknife+ which is more robust.

An alternative approach that accounts for model variance like the bootstrap method and model bias like the k-fold cross-validation method is called **Jackknife+** [BCRT19]. Jackknife+ carries strong guarantees about accuracy of the confidence intervals generated, regardless of the underlying data or model. The change now is that we use LOOCV to create an ensemble of models (although you can subsample down if you do not want  $N$  of them) and also compute the models' test error on the withheld test data. The final quantile estimates incorporate the variance from the variety of models (model variance) and also each models' individual test error (model bias). Specifically, we compute:

$$R_i = |y_i - \hat{f}(\vec{x}_i; \mathbf{X} \setminus \vec{x}_i)| \quad (3.13)$$

where  $\mathbf{X} \setminus \vec{x}_i$  is the dataset to train the  $i$ th model and is the dataset excluding point  $(\vec{x}_i, y_i)$ ,  $\hat{f}(\vec{x}_i; \mathbf{X} \setminus \vec{x}_i)$  is the  $i$ th model evaluated on point  $\vec{x}_i$ , and  $R_i$  is the residual of model  $i$  computed by taking the difference between the label and prediction on point  $i$ .  $R_i$  encodes how good the  $i$ th model is. We then combine it with the predictions on our new test point  $(\vec{x}', y')$  to make our set for quantiling

$$q_1 = \{ \hat{f}(\vec{x}'; \mathbf{X} \setminus \vec{x}_i) - R_i \} \quad (3.14)$$

$$q_2 = \{ \hat{f}(\vec{x}'; \mathbf{X} \setminus \vec{x}_i) + R_i \} \quad (3.15)$$

where  $q$  means quantile. The first quantile  $q_1$ , with  $-R_i$ , is how low below the estimate from the  $i$ th model we could expect to see our prediction based on how the  $i$ th model did on its test point. The second set, with  $+R_i$ , is how high below the estimate from the  $i$ th model we could expect to see our prediction based on how the  $i$ th model did on its test point. To compute our final value, we take the median of  $\hat{f}(\vec{x}'; \mathbf{X} \setminus \vec{x}_i)$  and report the lower end of the interval as the 5% quantile of  $q_1$  and the top as the 95% quantile of  $q_2$ . You can see that this method combines the ensemble of prediction models given by bootstrap resampling with the error estimates from LOOCV. Let's see an example.

```
residuals = []
predictions = []
for i in range(N):
    # make train excluding test point
    # we just make a set and remove one element from it
    # and then convert back to list
    idx = list(set(range(N)) - set([i]))
    train = small_soldata.iloc[idx]
```

(continues on next page)

(continued from previous page)

```

x, y = train[feature_names].values, train["Solubility"].values
# compute coefficients
w, *_ = np.linalg.lstsq(x, y)
# compute intercept (b)
b = np.mean(y - np.dot(x, w))
# compute test prediction
predictions.append(np.dot(predict_point[feature_names].values, w) + b)
# now compute residual on withheld point
yhat = np.dot(small_soldata.iloc[idx][feature_names].values, w) + b
residuals.append(np.abs(yhat - small_soldata.iloc[idx]["Solubility"]))
# create our set of prediction - R_i and prediction + R_i
q1 = [p - ri for p, ri in zip(predictions, residuals)]
q2 = [p + ri for p, ri in zip(predictions, residuals)]
# compute quantiles (lower, median, upper)
qlow = np.quantile(q1, [0.05])[0]
qhigh = np.quantile(q2, [0.95])[0]
# compute avg distance from medianto report +/-
print(
    f'prediction = {np.median(predictions):.2f} +/- {(qlow - qhigh) / 2:.2f}, label = '
    + f'{predict_point["Solubility"]:.2f}'
)
print(f"Average test error = {np.median(residuals):.2f}")

```

```

prediction = -4.08 +/- -3.28, label = -3.45
Average test error = 1.05

```

The uncertainty is much higher, which is likely closer to reality. You can see that the residuals add about 1 solubility (model variance) unit and the variability in the data (label noise) adds about 2 solubility units. Jackknife+ should be the preferred method when you have small datasets (1-1000) and can train models quickly enough to be able to compute 1000 of them. You can also replace the exhaustive LOOCV with a random process, where you only do a few iterations (like 25) of LOOCV to avoid computing so many models.

### 3.8 Training Data Distribution

We have come a long ways now. We're able to compute test error, identify overfitting, understand model bias and variance, and predict uncertainty on unseen data points. One of the implied assumptions so far is that our splitting of data into training and testing data mimics what it will be like to predict on an unseen data point. More specifically, we assume that testing data comes from the same probability distribution as our training data. This is true when we're doing the splitting, but is often violated when we actually get new data to make predictions with.

There are specific categories for how we have left the training distribution. **Covariate shift** is when the distribution of features changes. Covariate is another word for features. An example might be that the molecular weights of your molecules are larger in your testing data. The relationship between features and labels,  $f(\vec{x})$  remains the same, but the distribution of features is different. **Label shift** means that the distribution of labels has changed. Perhaps our training data was all very soluble molecules but at test time, we're examining mostly insoluble molecules. Again, our fundamental relationship  $f(\vec{x})$  that we try to estimate with our model still holds.

#### Applicability Domain

Applicability domain is a term from cheminformatics describing avoiding covariate shift by trying to stay within the training data distribution.

There are two common reasons unseen data can be out of the training data distribution. The first is that you are extrapolating to new regions of chemical space. For example, you have training data of drug activities. You make a model that can predict activity. What do you do with the model? You obviously find the highest activity drug molecule. However, this molecule is likely to be unusual and not in your training data. If it was in your training data you would probably already be done – namely, you already synthesized and found a molecule with very high activity. Thus you will be pushing your model to regions outside of your training data. Another reason you can be out of training data is that the way you generated training data is different than how the model is used. For example, perhaps you trained on molecules that do not contain fluorine. Then you try your model on molecules that contain fluorine. Your features will be different than what you observed in training. The result of leaving your training data distribution is that your test error increases and the estimates you provide will be too low.

### 3.8.1 Leave One Class Out Cross-Validation

Thus understanding and assessing training data distribution is an important task. In general, standard models that minimize a loss are poor at predicting extreme values. We will approach this challenge later with specific methods like black-box function optimization. For now, be wary of using your models as tools to find extreme values. The second challenge, that you're leaving your training data due to how points are generated, can be assessed by computing a more realistic estimate of model error. Namely, your training data is typically gathered (generated) according to a different process than when your model is deployed at test time. This is generalization error, sometimes called **covariate shift**, and we sometimes wish to approximate its effect by simulating different training and testing distributions. This leads us to **leave one class out cross-validation** (LOCOCV).

In LOCOCV, we must first assign a class to each training data point. This is domain specific. It could be based on the molecule. You could use a clustering method. In our case, our solubility data actually is a combination of five other datasets so our data is already pre-classified based on who measured the solubility. We will now perform a kind of k-fold cross-validation, leaving one class out at a time and assessing model error. We'll compare this to k-fold cross-validation without classes.

```
# let's see what the groups (classes) are
unique_classes = soldata["Group"].unique()
print(unique_classes)
```

```
['G1' 'G3' 'G5' 'G4' 'G2']
```

```
# Leave one class out CV
N = len(soldata)
error = []
error_std = []
for c in unique_classes:
    # slice out segments
    test = soldata.loc[soldata["Group"] == c]
    train = soldata.loc[soldata["Group"] != c]
    test_x, test_y = test[feature_names].values, test["Solubility"].values
    x, y = train[feature_names].values, train["Solubility"].values
    # compute coefficients
    w, *_ = np.linalg.lstsq(x, y)
    # compute intercept (b)
    b = np.mean(y - np.dot(x, w))
    # compute test error
    k_error.append(np.mean((np.dot(test_x, w) + b - test_y) ** 2))
    error.append(np.mean(k_error))
    error_std.append(np.std(k_error, ddof=1))
print(f"test error = {np.mean(error):.2f}")
```

```
test error = 0.46
```

We computed above what the 5-fold cross-validation is for this data, 2.66. You can see the LOCOCV test error (0.46) is similar, which means our different data sources agree well. So perhaps on new unseen data we can expect similar (not so great) accuracy. There may be other ways to group this data into classes, like based on molecular weight or which atoms are contained in the molecule. It depends on what you believe to be important. Breaking it down into the constituent datasets, like we did above, is a reasonable approach because it captures how different research groups would measure solubility. It is not always obvious or possible to use LOCOCV, but it should be something you consider to assess out of training data distribution. You can read more about the issue of leaving training data distribution for materials in this recent article [SBG+20]. You can read more about model selection in general in this recent tutorial article [Ras18].

### 3.9 Chapter Summary

- Regression is supervised learning where the labels are real numbers. We only considered scalars
- To assess a regressed model, we split data into training and testing and only report error on testing data
- Overfitting causes a mismatch between training and testing error
- Overfitting can be understood via the bias-variance decomposition
- Increasing model complexity can improve fit (reduce bias), but increases model variance and thus test error
- Regularization is a strategy to decrease model variance. L2 is a good first choice
- More rigorous assessment of models can be done via k-fold cross-validation or Jackknife+ when the training data is small enough that we can train multiple models
- Much of our model assessments depends on the testing data being from the same distribution as the training data (similar values). This is often not true and can be measured with leave-one-class-out cross-validation.

### 3.10 Exercises

#### 3.10.1 Overfitting

1. What happens if we have redundant features but no noise? Is it possible to overfit?
2. We said that increasing dataset size reduces model variance. Show this by using k-fold cross-validation on a few different dataset sizes.

#### 3.10.2 Regularization

1. Implement L1 regularization on the solubility data with  $N = 35$  data points. Increase the strength until some feature coefficients ( $w_i$ ) go to zero. Which ones are they? Why do you think they go to zero first?
2. Repeat 1 with a few different sets of training data. Are your results consistent on which features disappear? Based on your results, do you think there is meaning to the features which go to zero?
3. Implement the L-infinity (supremum norm) regularization, which returns the absolute value of the maximum element only.

### 3.10.3 Model Assessment

1. Develop the best linear model for the complete solubility dataset and assess using your best judgment. Justify your choice of model and assessment.

## 3.11 Cited References



## CLASSIFICATION

Classification is supervised learning with categorical labels. You are given labeled data consisting of features and labels  $\{\vec{x}_i, \vec{y}_i\}$ , where  $\vec{y}_i$  is a vector of binary values indicating class membership. An example of  $\vec{y}_i$  that indicates membership of classes “soluble in THF”, “insoluble in water”, “soluble in chloroform” might be:

THF	water	chloroform
1	0	1

where we’ve indicated that the molecule is soluble in THF and chloroform but not water. As a vector, it is  $\vec{y} = (1, 0, 1)$ . This is the general format of classification and can be called **multi-label** classification because we are attaching three labels: THF soluble, water insoluble, chloroform soluble. This can be restricted so that each data point belongs to only one class – called **multi-class** classification. This might be like assigning visible color. A molecule can only be red or green or orange, but not multiple colors. Finally, you can only have one class and a label can only belong to the single class or does not belong to the single class. This is called **binary** classification and is the most common classification type. If you’re doing multi-label or multi-class classification, the shape of  $\vec{y}$  will be a vector length  $K$  where  $K$  indicates number of classes. In the case of binary classification, the label is a binary value of 1 or 0 where 1 means it is a member of the class. You can view this as there being two classes: a **positive** class ( $y = 1$ ) and **negative** class ( $y = 0$ ). For example, you could be predicting if a molecule will kill cells. If the molecule is in the positive class, it kills cells. If it is the negative class, it is inert and does not kill cells. Depending on your choice of model type, when you predict the labels ( $\hat{y}$ ), your model could predict probabilities.

---

### Audience & Objectives

This chapter builds on [Regression & Model Assessment](#) and a basic knowledge of probability theory – specifically random variables, normalization, and the metrics section below touches on calibration (empirical agreement of model distribution with true distribution). You can read [my notes](#) or any introductory probability text to learn these topics. After completing this chapter, you should be able to:

- Distinguish between types of classification
- Set-up and train a classifier with a cross-entropy loss function
- Characterize classifier performance
- Identify and address class imbalance

---

Note that in multi-class and binary classification  $\sum \hat{y} = 1$ , but in multi-label classification this is not the case. Multi-label classification is like doing  $K$  instances of binary classification.

---

The goal of classification is to find a function that describes the relationship between features and class,  $\hat{f}(\vec{x}) = \hat{y}$ . We'll see that this problem can be converted to regression by using probability or *distance from a decision boundary*. This means much of what we learned previously can be applied to this classification.

The classic application of classification in structure-activity relationship is in drug discovery, where we want to predict if a molecule will be active (positive class) as a function of structure. That dates back to the 1970s. Classification is widely used now in materials and chemistry. Many molecular design problems can be formulated as classification. For example, you can use it to design new organic photovoltaic materials [SZY+19] or antimicrobial peptides [BJW18].

## 4.1 Data

The dataset for this lecture was prepared by the MoleculeNet group [WRF+18]. It is a collection of molecules that succeeded or failed in clinical trials. The development of a new drug can cost well over a \$1 billion, so any way to predict if a molecule will fail during clinical trials is highly valuable. The reason molecules fail in clinical trials is often due to safety, so even though some of these drugs failed because they were not effective there may be something common to each of the failed ones that we can learn.

The labels will be the FDA\_Approved column which is a 1 or 0 indicating FDA approval status. This is an example of binary classification.

## 4.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

```
import pandas as pd
import matplotlib.pyplot as plt
import rdkit, rdkit.Chem, rdkit.Chem.Draw
import numpy as np
import jax.numpy as jnp
import mordred, mordred.descriptors
import jax
import dmol
```

Now we load the data. This is a little fancy because we're extracting the data file from a zip archive on a website.

```
# from zipfile import ZipFile
# from io import BytesIO
# from urllib.request import urlopen

# from web version
# url = 'https://deepchemdata.s3-us-west-1.amazonaws.com/datasets/clintox.csv.gz'
# file = urlopen(url).read()
# file = BytesIO(file)
# document = ZipFile(file)
```

(continues on next page)



(continued from previous page)

```
# toxdata = pd.read_csv(document.open('clintox.csv'))

# local version
toxdata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/clintox.csv.gz"
)
toxdata.head()
```

	smiles	FDA_APPROVED	CT_TOX
0	*C(=O) [C@H] (CCCCNC(=O) OCCOC) NC(=O) OCCOC	1	0
1	[C@@H]1 ([C@@H] ([C@@H] ([C@H] ([C@@H] ([C@@H]1Cl) C...	1	0
2	[C@H] ([C@@H] ([C@@H] (C(=O) [O-]) O) O) ([C@H] (C(=O) ...	1	0
3	[H] / [NH+] =C (/C1=CC(=O) /C(=C\C=c2ccc(=C ([NH3+]) ...	1	0
4	[H] / [NH+] =C (\N) /c1ccc(cc1) OCCCCOc2ccc(cc2) /C(...	1	0

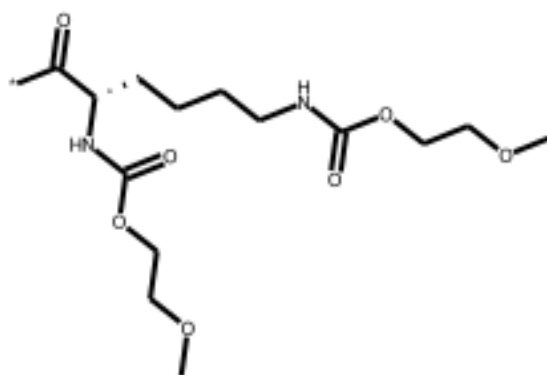
## 4.3 Molecular Descriptors

This time, our data does not come with pre-computed descriptors. We only have the SMILES string, which is a way of writing a molecule using letters and numbers (a string). We can use rdkit to convert the SMILES string into a molecule, and then we can use a package called Mordred [MTKT18] to compute a set of descriptors for each molecule. This package will compute around 1500 descriptors for each molecule.

We'll start by converting our molecules into rdkit objects and building a calculator to compute the descriptors.

```
# make object that can compute descriptors
calc = mordred.Calculator(mordred.descriptors, ignore_3D=True)
# make subsample from pandas df
molecules = [rdkit.Chem.MolFromSmiles(smi) for smi in toxdata.smiles]

# view one molecule to make sure things look good.
molecules[0]
```



Some of our molecules failed to be converted. We'll have to remove them. We need to remember which ones were deleted too, since we need to remove the failed molecules from the labels.

```
# the invalid molecules were None, so we'll just
# use the fact the None is False in Python
```

(continues on next page)

(continued from previous page)

```
valid_mol_idx = [bool(m) for m in molecules]
valid_mols = [m for m in molecules if m]
```

```
features = calc.pandas(valid_mols)
```

Now we just need to stitch everything back together so that our labels are consistent and standardize our features.

```
labels = toxdata[valid_mol_idx].FDA_APPROVED
features -= features.mean()
features /= features.std()

# we have some nans in features, likely because std was 0
features.dropna(inplace=True, axis=1)

print(f"We have {len(features.columns)} features per molecule")
```

```
We have 483 features per molecule
```

## 4.4 Classification Models

### 4.4.1 Linear Perceptron

We are able to predict single values from regression. How can we go from a predicted value to a class? The simplest answer is to use the same linear regression equation from chapter *Regression & Model Assessment*  $\hat{f}(\vec{x})$  and assign  $\hat{y} = 1$  when  $\hat{f}(\vec{x}) > 0$ ,  $\hat{y} = 0$  otherwise. Our model equation is then:

$$\hat{f}(\vec{x}) = \begin{cases} 1 & \vec{w} \cdot \vec{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

The term  $\vec{w} \cdot \vec{x} + b$  is called **distance from the decision boundary** where the decision boundary is at  $\vec{w} \cdot \vec{x} + b = 0$ . If it is large, we are far away from classifying it as 0. If it is small, we are close to classifying it as 0. It can be loosely thought of as “confidence”.

```
def perceptron(x, w, b):
    v = jnp.dot(x, w) + b
    y = jnp.where(v > 0, x=jnp.zeros_like(v), y=jnp.ones_like(v))
    return y
```

This particular model is called a **perceptron** and is the first neural network for classification. It was invented in 1958 by Frank Rosenblatt, a psychologist at Cornell University. It was not the first neural network, but is often the first one that students learn. The perceptron is an example of a **hard** classifier; it does not predict probability of the class and instead predicts exactly one class.

Now that we have a model, we must choose a loss function. We haven’t learned about many loss functions yet. We’ve only seen mean squared error. Let us begin with a related loss called mean absolute error (MAE). MAE measures disagreement between our class and the predicted class. This is like an accuracy – what percentage of the time we’re correct.

$$L = \frac{1}{N} \sum_i |y_i - \hat{y}_i| \quad (4.2)$$

(continues on next page)

71

(continued from previous page)

```
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 0.] , dtype=float32),  
DeviceArray(0., dtype=float32, weak_type=True))
```

It's all zeros! Why is that? It's because our `jnp.where` statement above is not differentiable, nor are any inequalities where the result is a constant (1 or 0 in our case). The perceptron actually has a special training procedure that is not related to its derivatives. One of the motivating reasons that deep learning is popular is that we do not need to construct a special training process for each model we construct – like the training procedure for the perceptron.

Rather than teach and discuss the special perceptron training procedure, we'll move to a more modern related classifier called a softmax binary classifier. This is a tiny change, the softmax binary classifier is:

$$\hat{f}(\vec{x}) = \sigma(\vec{w} \cdot \vec{x} + b) \quad (4.3)$$

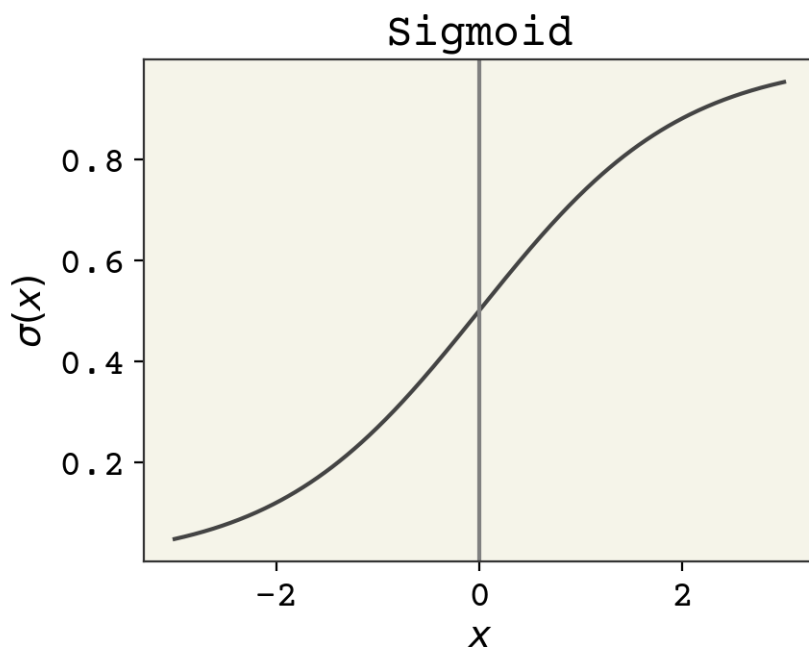


Fig. 4.1: The sigmoid function. Input is any real number and the output is a probability. Positive numbers map to probabilities greater than 0.5 and negative numbers to probabilities less than 0.5.

Softmax is the generalization of sigmoid to multiple classes. Although we call our binary classifier a softmax classifier, it doesn't use the softmax function.

where  $\sigma$  is the **sigmoid** function. The sigmoid has a domain of  $(-\infty, \infty)$  and outputs a probability  $(0, 1)$ . The input to the sigmoid can be viewed as log-odds, called **logits** for short. Odds are ratios of probability – odds of 1 means the probability of the class 1 is 0.5 and class 0 is 0.5. Odds of 2 means the probability of class 1 is 0.67 and class 0 is 0.33. Log-odds is the natural logarithm of that, so that log-odds of 0 means the odds are 1 and the output probability should be 0.5. One definition of the sigmoid is

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (4.4)$$

however in practice there are some complexities to implementing sigmoids to make sure they're numerically stable. This type of binary classifier is sometimes called **logistic regression** because we're regressing logits.

In essence, all we've done is replacing the inequality of the perceptron with a smooth differentiable version. Just like previously, a positive number indicated class 1 (FDA approved) but now it's a continuum of numbers from 0.5 to 1.0. This is **soft** classification – we give probabilities of class membership instead of hard assignment. However, our loss function now needs to be modified as well.

There is a different loss function that works better with classification called **cross-entropy**. You can experiment with mean absolute error or mean squared error with classification, but you'll find they are almost always worse than cross-entropy.

Cross-entropy is a loss function that describes distance between two probability distributions. When minimized, the two probability distributions are identical. Cross-entropy is a simplification of the [Kullback–Leibler divergence](#) which is a way to measure distance between two probability distributions. Technically it is not a distance since it's not symmetric with respect to its arguments. But in practice it is close enough to a distance that we treat it as one.

How is comparing predicted values  $\hat{y}$  and  $y$  like comparing two probability distributions? Even though these are both 1s and 0s in the case of hard classification, they do sum to 1, and so we consider them probability distributions. Cross-entropy is defined as:

$$L = - \sum_c^K y_c \log \hat{y}_c \quad (4.5)$$

where  $c$  indicates which class of the  $K$  we're considering, and it's assumed that  $\sum_c^K y_c = 1$  and  $\sum_c^K \hat{y}_c = 1$  like probabilities (and they are positive). In the case of binary classification (only two classes), this becomes:

$$L = - [y_0 \log \hat{y}_0 + y_1 \log \hat{y}_1] \quad (4.6)$$

where  $y_0$  is for the first class and  $y_1$  is for the second class. However, we also know that because these are probabilities that  $y_1 = 1 - y_0$ . We can rewrite to:

$$L = - [y_0 \log \hat{y}_0 + (1 - y_0) \log(1 - \hat{y}_0)] \quad (4.7)$$

Finally, we can drop the indication of the class:

$$L = - [y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad (4.8)$$

The correct way to avoid numerical instability in cross-entropy sigmoid classification is to have your model output the logits and you use a loss function that works on logits instead of probability. For example, `tf.nn.sigmoid_cross_entropy_with_logits`.

and this matches our data, where we have a single value for each label indicating if it is a class member. Now we have features, labels, loss, and a model. Let's create a batched gradient descent algorithm to train our classifier. Note, one change we need to do is use the built-in jax `jax.nn.sigmoid` function to avoid numerical instabilities and also add a small number to all logs to avoid numerical instabilities.

```

def bin_classifier(x, w, b):
    v = jnp.dot(x, w) + b
    y = jax.nn.sigmoid(v)
    return y

def cross_ent(y, yhat):
    return jnp.mean(-(y * jnp.log(yhat + 1e-10) + (1 - y) * jnp.log(1 - yhat + 1e-10)))

def loss_wrapper(w, b, x, y):
    yhat = bin_classifier(x, w, b)
    return cross_ent(y, yhat)

loss_grad = jax.grad(loss_wrapper, (0, 1))
w = np.random.normal(scale=0.01, size=len(features.columns))
b = 1.0

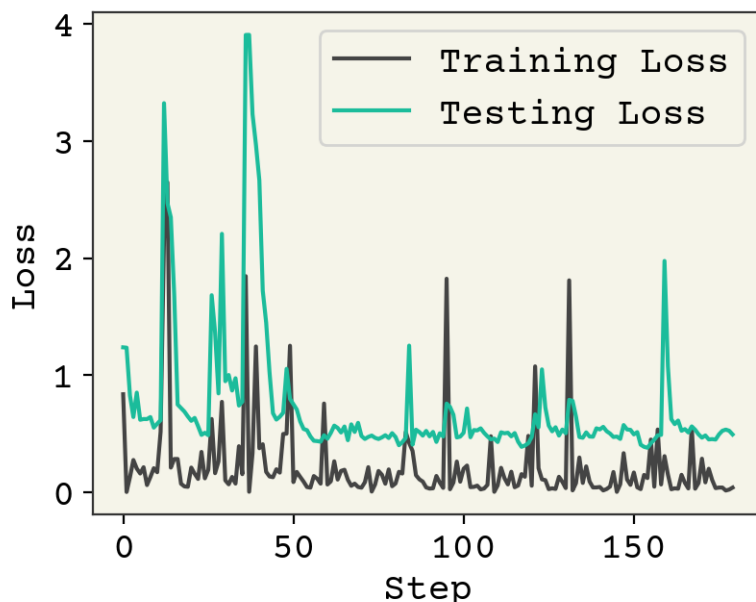
```

```

loss_progress = []
test_loss_progress = []
eta = 0.2
for epoch in range(5):
    for i in range(len(batch_idx) - 1):
        x = features[batch_idx[i] : batch_idx[i + 1]].values.astype(np.float32)
        y = labels[batch_idx[i] : batch_idx[i + 1]].values
        grad = loss_grad(w, b, x, y)
        w -= eta * grad[0]
        b -= eta * grad[1]
        loss_progress.append(loss_wrapper(w, b, x, y))
        test_loss_progress.append(loss_wrapper(w, b, test_x, test_y))
plt.plot(loss_progress, label="Training Loss")
plt.plot(test_loss_progress, label="Testing Loss")

plt.xlabel("Step")
plt.legend()
plt.ylabel("Loss")
plt.show()

```



We are making good progress with our classifier, as judged from testing loss. You can run the code longer, but I'll leave it at that. We have a reasonably well-trained model.

## 4.5 Classification Metrics

In regression, we assessed model performance with a parity plot, correlation coefficient, or mean squared error. In classification, we use slightly different metrics. The first metric is **accuracy**. Accuracy is the percentage of time that the predicted label matches the true label. We do not have a hard classifier, so we have to choose how to turn probability into a specific class. For now, we will choose the class with the highest probability. Let's see how this looks

```
def accuracy(y, yhat):
    # convert from prob to hard class
    hard_yhat = np.where(yhat > 0.5, np.ones_like(yhat), np.zeros_like(yhat))
    disagree = np.sum(np.abs(y - hard_yhat))
    return 1 - disagree / len(y)
```

```
accuracy(test_y, bin_classifier(test_x, w, b))
```

```
0.8269480756811194
```

An accuracy of 0.83 seems quite reasonable! However, consider this model:

```
def alt_classifier(x):
    return np.ones((x.shape[0]))
```

```
accuracy(test_y, alt_classifier(test_x))
```

```
0.9087837837837838
```

This model, which always returns 1, has better accuracy than our model. How is this possible?

**Answer**

If you examine the data, you'll see the majority of the molecules passed FDA clinical trials ( $y = 1$ ), so that just guessing 1 is a good strategy.

**4.5.1 Error Types**

Let's recall what we're trying to do. We're trying to predict if a molecule will make it through FDA clinical trials. Our model can be incorrect in two ways: it predicts a molecule will pass through clinical trials, but it actually fails. This is called a false positive. The other error is if we predict our drug will not make it through clinical trials, but it actually does. This is false negative.

False positive are sometimes known as Type I (pronounced type one) and false negatives as Type II false negatives

Our `alt_classifier` model, which simply reports everything as positive, has no false negative errors. It has many false positive errors. These two types of errors can be quantified. We're going to add one complexity – **threshold**. Our model provides probabilities which we're converting into hard class memberships – 1s and 0s. We have been choosing to just take the most probable class. However, we will now instead choose a threshold for when we report a positive (class 1). The rationale is that although we train our model to minimize cross-entropy, we may want to be more conservative or aggressive in our classification with the trained model. If we want to minimize false negatives, we can lower the threshold and report even predictions that have a probability of 30% as positive. Or, if we want to minimize false positives we may set our threshold so that our model must predict above 90% before we predict a positive.

```
def error_types(y, yhat, threshold):
    hard_yhat = np.where(yhat >= threshold, np.ones_like(yhat), np.zeros_like(yhat))
    # predicted 1, actually was 0 -> 1 (bool to remove predicted 0, actually was 1)
    fp = np.sum((hard_yhat - y) > 0)
    # predicted 0, actually was 1 -> 1 (bool to remove predicted 1, actually was 0)
    fn = np.sum((y - hard_yhat) > 0)
    return fp, fn
```

```
print("Alt Classifier", error_types(test_y, alt_classifier(test_x), 0.5))
print("Trained Classifier", error_types(test_y, bin_classifier(test_x, w, b), 0.5))
```

```
Alt Classifier (27, 0)
Trained Classifier (20, 20)
```

Now we have a better sense of how our model does in comparison. The number of errors is indeed larger for our trained model, but it has a bit of balance between the two errors. What is more important? In our case, I would argue doing clinical trials that fail is worse than mistakenly not starting them. That is, false positives are worse than false negatives. Let's see if we can tune our threshold value to minimize false positives.

```
print("Threshold 0.7", error_types(test_y, bin_classifier(test_x, w, b), 0.7))
print("Threshold 0.9", error_types(test_y, bin_classifier(test_x, w, b), 0.9))
print("Threshold 0.95", error_types(test_y, bin_classifier(test_x, w, b), 0.95))
print("Threshold 0.99", error_types(test_y, bin_classifier(test_x, w, b), 0.99))
```



```

Threshold 0.7 (18, 26)
Threshold 0.9 (9, 82)
Threshold 0.95 (6, 120)
Threshold 0.99 (1, 218)

```

By adjusting the threshold, we can achieve a balance of error more like what we desire for our model. We're able to have 1 false positives in fact, at the cost of missing 218 of the molecules. Now are we still predicting positives? Are we actually going to get some **true positives**? We can measure that as well

```

total_pos = np.sum(test_y)
print(
    "Total positives:",
    total_pos,
    "Predicted Positives:",
    np.sum(bin_classifier(test_x, w, b) > 0.99),
)

```

```
Total positives: 269 Predicted Positives: 52
```

Yes, our model is actually capable of predicting if molecules will pass FDA clinical trials with as few false positives as possible (1). A model that is capable of this tuning is an example of a good model. Our other model, that predicts 1s, has good accuracy but we cannot adjust it or try to better balance type I and type II errors.

## 4.5.2 Receiver-Operating Characteristic Curve

We can plot threshold, false positive rate, and true positive rate all together on one plot to capture model accuracy and balance between error type in a Receiver-Operating Characteristic Curve (ROC curve). The x-axis of ROC curve is false positive rate and the y-axis is true positive rate. Each point on the plot is our model with different thresholds. How do we choose which thresholds to use? It is the set of unique class probabilities we saw (namely, `np.unique`). We do need to add two extremes to this set though: all positive (threshold of 0.0) and all negative (1.0). Recall our alternate/baseline model of always predicting positive: it can only have a few points on the the ROC curve because it's unique set of probabilities is just 1.0. Let's make one and discuss what we're seeing.

```

unique_threshes = np.unique(bin_classifier(test_x, w, b))
fp = []
tp = []
total_pos = np.sum(test_y)
for ut in list(unique_threshes) + [-0.1, 1.01]:
    errors = error_types(test_y, bin_classifier(test_x, w, b), ut)
    fp.append(errors[0])
    tp.append(total_pos - errors[1])

# sort them so can plot as a line
idx = np.argsort(fp)
fpr = np.array(fp)[idx] / (len(test_y) - np.sum(test_y))
tpr = np.array(tp)[idx] / np.sum(test_y)

# now remove duplicate x-values
fpr_nd = []
tpr_nd = []
last = None
for f, t in zip(fpr, tpr):
    if last is None or f != last:

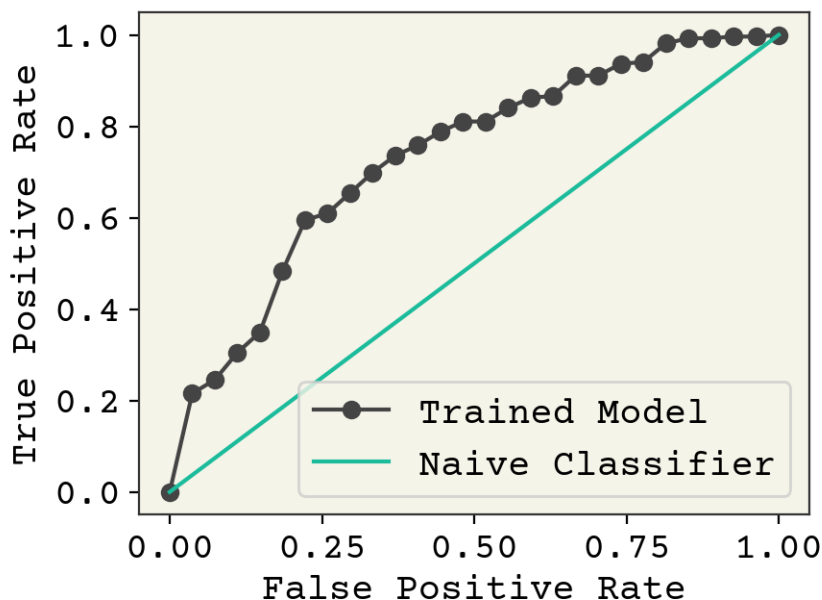
```

(continues on next page)

(continued from previous page)

```
last = f
fpr_nd.append(f)
tpr_nd.append(t)

plt.plot(fpr_nd, tpr_nd, "-o", label="Trained Model")
plt.plot([0, 1], [0, 1], label="Naive Classifier")
plt.ylabel("True Positive Rate")
plt.xlabel("False Positive Rate")
plt.legend()
plt.show()
```



This plot nicely shows how our trained model is actually sensitive to threshold, so that we could choose to more carefully screen for false negative or false positives. The best curves fall to the top-left of this plot. Our naive classifier is where we return a fixed percentage of examples randomly as positive or negative. You can plot the area under this curve with an integration and this is a good way to measure classifier performance and correctly capture the effect of both false negatives and false positives. The area under the ROC curve is known as the **ROC AUC score** and is preferred to accuracy because it captures the balance of Type I and II errors.

### 4.5.3 Other metrics

I will just mention that there are other ways to assess how your model balances the two error types. One major type is called **precision** and **recall**. Precision measures correctness of predicted positives and recall measures number of predicted positives. This can be a good viewpoint when doing molecular screening – you may want to be very precise in that your proposed molecules are accurate while sacrificing recall. Recall here meaning you do not return very many molecules. Models on the left of an ROC curve are precise. Models at the top have good recall. There are also F1 scores, likelihoods, Matthew's correlation coefficients, Jaccard index, Brier score, and balanced accuracy which all try to report one number which balances precision and recall. We will rarely explore these other measures but you should know they exist.

## Confusion Matrix

A confusion matrix is a table of counts indicating true and predicted classes. They are one of many methods for binary classification, but really stand-out as good visual assessment for multiclass classification. For example, consider we are categorizing molecules into three classes: insoluble, weakly soluble, and soluble. We can represent a classifier's performance in a table:

true\predicted	insoluble	weakly soluble	soluble
insoluble	121	8	1
weakly soluble	7	45	18
soluble	11	4	56

The diagonal elements show when the predicted and true labels agree. For example, 121 molecules were actually insoluble and predicted to be insoluble. We can also read how the classifier failed. One molecule was predicted to be soluble, but was actually insoluble. 4 molecules were predicted to be weakly soluble, but were actually soluble. This can help us understand *how* the classifier is failing.

## 4.6 Class Imbalance

The reason for this uneven amount of false positives and false negatives is that we have very few negative example – molecules which failed FDA clinical trials. This also explains why just predicting success has a high accuracy. How can we address this problem?

The first answer is do nothing. Is this imbalance a problem at all? Perhaps a drug in general will succeed at clinical trials and thus the imbalance in training data reflects what we expect to see in testing. This is clearly not the case, judging from the difficult and large expense of creating new drug molecules. However, this should be the first thing you ask yourself. If you're creating a classifier to detect lung cancer from X-ray images, probably you will have imbalanced training data and at test time, when evaluating patients, you'll also not have 50% of patients having lung cancer. This comes back to the discussion in the *Regression & Model Assessment* about training data distribution. If your testing data is within your training data distribution, then the class imbalance does not need to be explicitly addressed.

The second solution is to somehow weight your training data to appear more like your testing data when you think you do have **label shift**. There are two ways to accomplish this. You could “augment” your training data by repeating the minority class until the ratio of minority to majority examples matches the assumed testing data. There are research papers written on this topic, with intuitive results[CBHK02]. You can over-sample minority class but that can lead to a large dataset, so you can also under-sample the majority class. This is a robust approach that is independent to how you train. It also is typically as good as more sophisticated methods [YIAPLP21].

Another method of weighing data is to modify your loss function to increase the gradient updates applied to minority examples. This is equivalent to saying there is a difference in loss between a false positive vs a false negative. In our case, false positive are rarer and also more important in reality. We would rather skip a clinical trial (false negative) rather than start one and have it fail (false positive). We already tried minimizing false positives by changing the threshold on a trained model but let's see how this works during training. We'll create a weight vector that is high for negative labels so that if they are misclassified (false positive), there will be a bigger update.

```
def bin_classifier(x, w, b):
    v = jnp.dot(x, w) + b
    y = jax.nn.sigmoid(v)
    return y

def weighted_cross_ent(y, yhat, yw):
    # weights may not be normalized
```

(continues on next page)

(continued from previous page)

```

N = jnp.sum(yw)
# use weighted sum instead
return (
    jnp.sum(
        -(yw * y * jnp.log(yhat + 1e-10) + yw * (1 - y) * jnp.log(1 - yhat + 1e-
-10))
    )
    / N
)

def loss_wrapper(w, b, x, y, yw):
    yhat = bin_classifier(x, w, b)
    return weighted_cross_ent(y, yhat, yw)

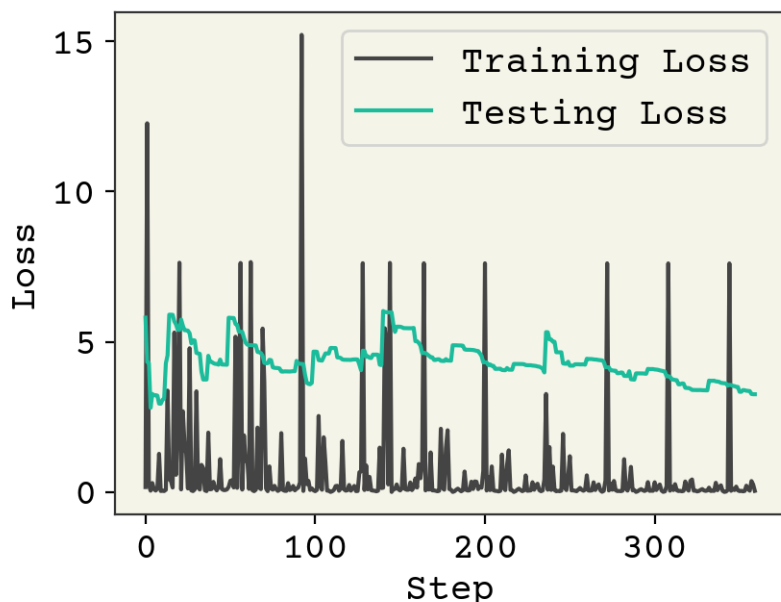
loss_grad = jax.grad(loss_wrapper, (0, 1))
w2 = np.random.normal(scale=0.01, size=len(features.columns))
b2 = 1.0
weights = np.ones_like(labels)
# make the labels = 0 values be much larger
weights[labels.values == 0] *= 1000
# now make weights be on average 1
# to keep our learning rate/avg update consistent
weights = weights * len(weights) / np.sum(weights)

loss_progress = []
test_loss_progress = []
eta = 0.2
# make epochs larger since this has
# very large steps that converge poorly
for epoch in range(10):
    for i in range(len(batch_idx) - 1):
        x = features[batch_idx[i] : batch_idx[i + 1]].values.astype(np.float32)
        y = labels[batch_idx[i] : batch_idx[i + 1]].values
        yw = weights[batch_idx[i] : batch_idx[i + 1]]
        grad = loss_grad(w2, b2, x, y, yw)
        w2 -= eta * grad[0]
        b2 -= eta * grad[1]
        loss_progress.append(loss_wrapper(w2, b2, x, y, yw))
        test_loss_progress.append(
            loss_wrapper(w2, b2, test_x, test_y, np.ones_like(test_y))
        )
plt.plot(loss_progress, label="Training Loss")
plt.plot(test_loss_progress, label="Testing Loss")

plt.xlabel("Step")
plt.legend()
plt.ylabel("Loss")
plt.show()

print("Normal Classifier", error_types(test_y, bin_classifier(test_x, w, b), 0.5))
print("Weighted Classifier", error_types(test_y, bin_classifier(test_x, w2, b2), 0.5))

```



Normal Classifier (20, 20)  
Weighted Classifier (6, 119)

The spikes in loss occur when we see a rare negative example, which are weighted heavily. Compared to the normal classifier trained above, we have fewer false positives at a threshold of 0.5. However, we also have more false negatives. We saw above that we could tweak this by changing our threshold. Let's see how our model looks on an ROC curve to compare our model trained with weighting with the previous model at all thresholds.

```
unique_threshes = np.unique(bin_classifier(test_x, w2, b2))
fp = []
tp = []
total_pos = np.sum(test_y)
for ut in list(unique_threshes) + [-0.1, 1.01]:
    errors = error_types(test_y, bin_classifier(test_x, w2, b2), ut)
    fp.append(errors[0])
    tp.append(total_pos - errors[1])

# sort them so can plot as a line
idx = np.argsort(fp)
fpr = np.array(fp)[idx] / (len(test_y) - np.sum(test_y))
tpr = np.array(tp)[idx] / np.sum(test_y)

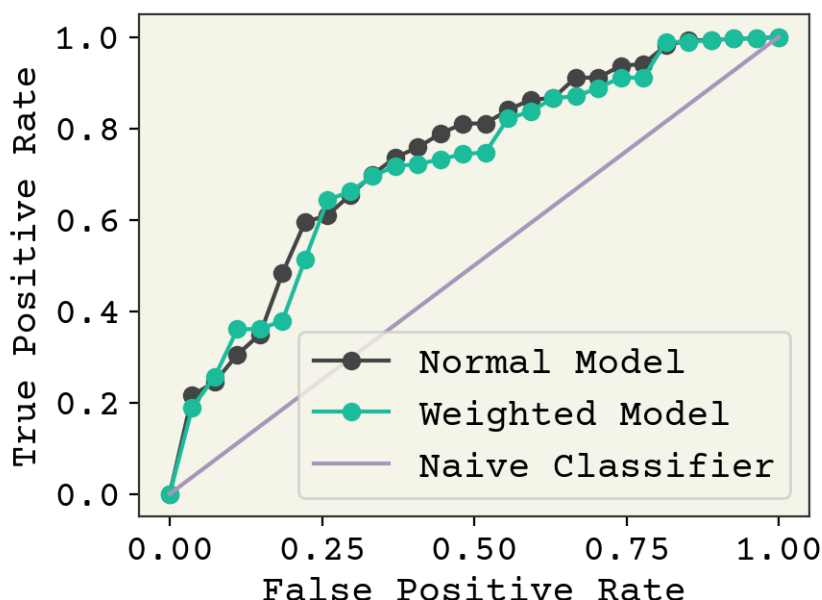
# now remove duplicate x-values
fpr_nd2 = []
tpr_nd2 = []
last = None
for f, t in zip(fpr, tpr):
    if last is None or f != last:
        last = f
        fpr_nd2.append(f)
        tpr_nd2.append(t)

plt.plot(fpr_nd, tpr_nd, "-o", label="Normal Model")
plt.plot(fpr_nd2, tpr_nd2, "-o", label="Weighted Model")
```

(continues on next page)

(continued from previous page)

```
plt.plot([0, 1], [0, 1], label="Naive Classifier")
plt.ylabel("True Positive Rate")
plt.xlabel("False Positive Rate")
plt.legend()
plt.show()
```



It appears our weighted training actually did not improve model performance, except in a small range between 0.25-0.4 false positive rate. It is even worse in the low false positive rate, which is where we would like to operate. In conclusion, we can modify the balance of false positive and false negative through modifications to training. However, we can also modify this after training by affecting the threshold for classification. This post-training procedure gives similar or even slightly better performance in our example.

This may not always be the case. An overview of methods are available in [HG09] and you can find a more recent discussion of the effects of reweighting, including when combined with regularization, in Bryrd and Lipton [BL19]. Bryrd and Lipton show that reweighting has little effect unless combined with L2 regularization and batch normalization, perhaps accounting for the small effect we observed.

#### 4.6.1 Screening: no negative examples

Class imbalance is common in peptide and drug discovery where screening is used to generate data. Screening typically only contains positive examples, meaning you have literally zero negative examples. This is an active topic of research in a field of **positive-unlabeled learning** [SBH+21]

## 4.7 Overfitting

The goal of this chapter is to introduce classification. For simplicity, we did not use any of the techniques from the last chapter, except training/testing splitting. You can and should use techniques like Jackknife+ and cross-validation to assess overfitting. Also, our descriptor number was very high, a few hundred, and regularization could be helpful for these models.

## 4.8 Chapter Summary

- We introduced classification, which is supervised learning with categorical labels. The labels can be single binary values - representing 2 classes which is binary classification.
- We can compute descriptors for molecules using Python packages and do not require them to be part of our dataset
- Cross-entropy loss should be used for classification tasks
- Classification models (called classifiers) can output distance from decision boundary or, more commonly, probability of class
- The Perceptron is an early example of a neural network classifier that has a special training procedure
- The sigmoid and soft-max functions convert real numbers into probabilities
- Binary classification error can be false positives or false negatives
- Accuracy does not distinguish these two errors, so receive-operator characteristic (ROC) curves can be used to assess model performance. Precision and recall are other commonly used measures.
- An imbalance of classes in training data is not necessarily a problem and can be addressed by weighting training examples

## 4.9 Exercises

### 4.9.1 Classification

1. Design your own examples of labels for binary, multi-class, and multi-label classification. For example, “A multi-class label is the country a person lives in. A label for this is a 225 element vector with one non-zero element indicating the country the person lives in.”
2. Write out the equations for cross-entropy in multi-class and multi-label settings.

### 4.9.2 Data

1. Use the dimensional reduction methods from our *first chapter* to plot the molecules here in 2D. Color the points based on their labels. Do you see any patterns?
2. Now, use clustering to color the molecules. Use an elbow plot to choose your cluster number.

### 4.9.3 Assessment

1. Repeat the model fitting with L1 and L2 regularization and plot them on a ROC curve. What effect does regularization have on these? Choose a strength of 0.1.
2. Could you use leave-one-class-out cross-validation in binary classification? Why or why not?
3. We said that class imbalance alone has little effect on model training, as long as the testing distribution matches the training distribution. However, can you make an argument using the bias-variance decomposition about why this may not be true with small dataset size?
4. Compute the area under the curve of an ROC curve using numerical trapezoidal integration.

### 4.9.4 Complete Model

Do your best to create a binary-classifier for this dataset with regularization and any other methods we learned from this chapter the previous ones. What is the best area under the curve you can achieve?

## 4.10 Cited References



## KERNEL LEARNING

Kernel learning is a way to transform features in either classification or regression problems. Recall in regression, we have the following model equation:

$$\hat{y} = \vec{w}\vec{x} + b \quad (5.1)$$

where  $\vec{x}$  is our feature vector of dimension  $D$ . In kernel learning, we transform our feature vector from dimension  $D$  features to *distances to training data points* of dimension  $N$ , where  $N$  is the number of training data points:

$$\hat{y} = \sum_i^N w_i \langle \vec{x}, \vec{x}_i \rangle + b \quad (5.2)$$

where  $\langle \cdot \rangle$  is distance between two feature vectors and  $\vec{x}_i$  is the  $i$ th training data point.  $\vec{x}$  is the function argument whereas  $\vec{x}_i$  are known values.

---

### Audience & Objectives

This chapter builds on *Regression & Model Assessment* and *Classification*. After completing this chapter, you should be able to

- Distinguish between primal and dual form
  - Choose when kernel learning could be beneficial
  - Understand the principles of training curves and the connection between training size and feature number
- 

Although we will use the word distance, the  $\langle \cdot \rangle$  function is actually an inner product which is more flexible than a distance.

One of the consequences of this transformation is that our training weight vector,  $\vec{w}$ , no longer depends on the number of features. **Instead the number of weights depends on the number of training data points.** This is the reason we use kernel learning. You might have few training data points but large feature vectors ( $N < D$ ). By using the kernel formulation, you'll reduce the number of weights. It might also be that your feature space is hard to model with a linear equation, but when you view it as distances from training data it becomes linear (often  $N > D$ ). Finally, it could be that your feature vector is infinite dimensional (i.e., it is a function not a vector) or for some reason you cannot compute it. In kernel learning, you only need to define your *kernel function*  $\langle \cdot \rangle$  and never explicitly work with the feature vector.

The distance function is called a *kernel function*  $\langle \cdot \rangle$ . A kernel function is a binary function (takes two arguments) that outputs a scalar and has the following properties:

1. Positive:  $\langle x, x' \rangle \geq 0$
2. Symmetric:  $\langle x, x' \rangle = \langle x', x \rangle$

3. Point-separating:  $\langle x, x' \rangle = 0$  if and only if  $x = x'$

The classic kernel function example is  $l_2$  norm (Euclidean distance):  $\langle \vec{x}, \vec{x}' \rangle = \sqrt{\sum_i^D (x_i - x'_i)^2}$ . Some of the most interesting applications of kernel learning though are when  $x$  is not a vector, but a function or some other structured object.

---

### Primal & Dual Form

**Dual Form** is what some call our model equation when it uses the kernels:  $\hat{y} = \sum_i^N w_i \langle \vec{x}, \vec{x}_i \rangle + b$ . To distinguish from the dual form, you can also refer to the usual model equation as the **Primal Form**  $\hat{y} = \vec{w}\vec{x} + b$ . It also sounds cool.

---

Kernel learning is a widely-used approach for learning potential energy functions and force fields in molecular modeling [RTMullerVL12, SSAB20].

## 5.1 Solubility Example

Let's revisit the solubility AqSolDB[SKE19] dataset from *Regression & Model Assessment*. Recall it has about 10,000 unique compounds with measured solubility in water (label) and 17 molecular descriptors (features).

## 5.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

As usual, the code below sets-up our imports.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import jax.numpy as jnp
import jax
import dmol
```

```
# soldata = pd.read_csv('https://dataverse.harvard.edu/api/access/datafile/3407241?
↳format=original&gbrecs=true')
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
↳csv"
)
features_start_at = list(soldata.columns).index("MolWt")
feature_names = soldata.columns[features_start_at:]
np.random.seed(0)

# standardize the features
```

(continues on next page)

(continued from previous page)

```
soldata[feature_names] -= soldata[feature_names].mean()
soldata[feature_names] /= soldata[feature_names].std()
```

## 5.3 Kernel Definition

We'll start by creating our kernel function. *Our kernel function does not need to be differentiable.* In contrast to the functions we see in deep learning, we can use sophisticated and non-differentiable functions in kernel learning. For example, you could use a two-component molecular dynamics simulation to compute the kernel between two molecules. We'll still implement our kernel functions in JAX for this example because it is efficient and consistent. Remember our kernel should take two feature vectors and return a scalar. In our example, we will simply use the  $l_2$  norm. I will add one small twist though: dividing by the dimension. This makes our kernel output magnitude independent of the number of dimensions of  $x$ .

```
def kernel(x1, x2):
    return jnp.sqrt(jnp.mean((x1 - x2) ** 2))
```

## 5.4 Model Definition

Since we're doing linear regression, you can compute the fit coefficients by just doing matrix algebra. We'll still approach this problem with gradient descent even though there are more efficient model-specific procedures.

Now we define our regression model equation in the *dual form*. Remember that our function must always take the training data in to compute the distance to a new given point. We will use the batch feature of JAX to compute all the kernels simultaneously for our new point.

```
def model(x, train_x, w, b):
    # make vectorized version of kernel
    vkernel = jax.vmap(kernel, in_axes=(None, 0), out_axes=0)
    # compute kernel with all training data
    s = vkernel(x, train_x)
    # dual form
    yhat = jnp.dot(s, w) + b
    return yhat

# make batched version that can handle multiple xs
batch_model = jax.vmap(model, in_axes=(0, None, None, None), out_axes=0)
```

## 5.5 Training

We now have trainable weights and a model equation. To begin training, we need to define a loss function and compute its gradient. We'll use mean squared error as usual for the loss function. We can use regularization, as we saw previously, but will skip it for now.

```
@jax.jit
def loss(w, b, train_x, x, y):
    return jnp.mean((batch_model(x, train_x, w, b) - y) ** 2)

loss_grad = jax.grad(loss, (0, 1))

# Get 80/20 split
N = len(soldata)
train = soldata[: int(N * 0.8)]
test = soldata[int(N * 0.8) :]

# convert from pandas dataframe to numpy arrays
train_x = train[feature_names].values
train_y = train["Solubility"].values
test_x = test[feature_names].values
test_y = test["Solubility"].values
```

We've defined our loss and split our data into training/testing. Now we will set-up the training parameters, including breaking up our training data into batches. An **epoch** is one iteration through the whole dataset.

```
eta = 1e-5
batch_size = 32
epochs = 10

# reshape into batches
batch_num = train_x.shape[0] // batch_size
# first truncate data so it's whole number of batches
trunc = batch_num * batch_size
train_x = train_x[:trunc]
train_y = train_y[:trunc]
# split into batches
x_batches = train_x.reshape(-1, batch_size, train_x.shape[-1])
y_batches = train_y.reshape(-1, batch_size)

# make trainable parameters
# w = np.random.normal(scale = 1e-30, size=train_x.shape[0])
w = np.zeros(train_x.shape[0])
b = np.mean(train_y) # just set to mean, since it's a good first guess
```

You may notice our learning rate,  $\eta$ , is unusually low at  $10^{-5}$ . It's because each training data point, for which we have about 8,000, contributes to the final  $\hat{y}$ . Thus if we take a large training step, it can create very big changes to  $\hat{y}$ .

```
loss_progress = []
test_loss_progress = []

for _ in range(epochs):
```

(continues on next page)

(continued from previous page)

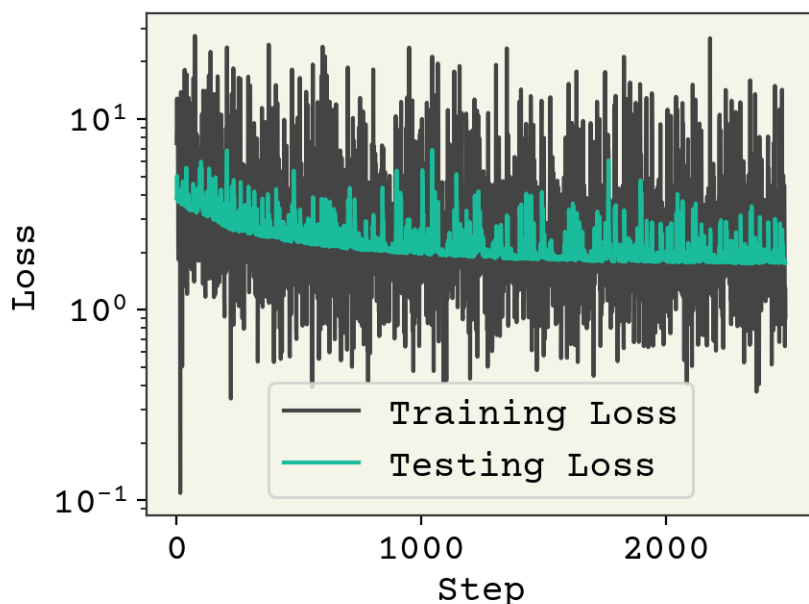
```

# go in random order
for i in np.random.randint(0, batch_num - 1, size=batch_num):
    # update step
    x = x_batches[i]
    y = y_batches[i]
    loss_progress.append(loss(w, b, train_x, x, y))
    test_loss_progress.append(loss(w, b, train_x, test_x, test_y))
    grad = loss_grad(w, b, train_x, x, y)
    w -= eta * grad[0]
    b -= eta * grad[1]
plt.plot(loss_progress, label="Training Loss")
plt.plot(test_loss_progress, label="Testing Loss")

plt.xlabel("Step")
plt.yscale("log")
plt.legend()
plt.ylabel("Loss")
plt.show()

```

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF\_CPP\_MIN\_LOG\_LEVEL=0 and rerun for more info.)

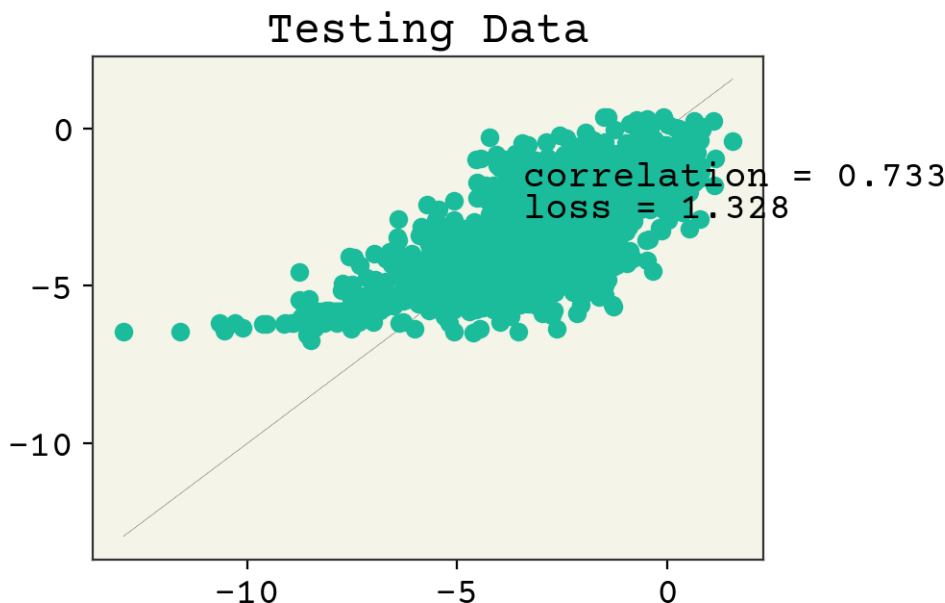


One small change from previous training loops is that we randomized our batches in the `for` loop.

```

yhat = batch_model(test_x, train_x, w, b)
plt.plot(test_y, test_y, ":", linewidth=0.2)
plt.plot(test_y, yhat, "o")
plt.text(min(y) + 1, max(y) - 2, f"correlation = {np.corrcoef(test_y, yhat)[0,1]:.3f}")
plt.text(min(y) + 1, max(y) - 3, f"loss = {np.sqrt(np.mean((test_y - yhat)**2)):.3f}")
plt.title("Testing Data")
plt.show()

```



We can see our results show underfitting. As usual, I want to make this code execute fast so I have not done many epochs. You can increase the epoch number and watch the loss and correlation improve over time.

## 5.6 Regularization

You'll notice that our trainable parameter number, by design, is equal to the number training data points. If we were to use a direct computation of the fit coefficients with a pseudo-inverse, we could run into problems because of this. Thus, most people add an additional regularization term to both make matrix algebra solutions tractable and because it seems wise with the large number of trainable parameters. Just like we saw in linear regression, L1 regression is known as **Lasso Ridge Regression** and L2 is known as **Kernel Ridge Regression**. Remember that L1 zeros-out specific parameters which was useful for interpreting the importance of features in linear regression. However, in the kernel setting this would only zero-out specific training data points and thus provides no real insight (usually, see [Explaining Predictions](#)). Kernel ridge regression is thus more popular in the kernel setting.

## 5.7 Training Curves

The bias-variance trade-off from [Regression & Model Assessment](#) showed how increasing model complexity could reduce model bias (more expressive and able to fit data better) at the cost of increased model variance (more sensitive to training data choice and amount). The model complexity was controlled by adjusting feature number. In kernel learning, we cannot control feature number because it is always equal to the number of training data points. Thus, we can only control hyperparameters like the choice of kernel, regularization, learning rate, etc. To assess these effects, we usually do not only compute test loss because that is highly-connected to the amount of training data you have. More training data means more sophisticated models and thus lower loss. So it is common in kernel learning especially to show how the test-loss changes a function of training data amount. These are presented as log-log plots due to the large magnitude changes in these. These are called **training curves** (or sometimes **learning curves**). Training curves can be applied broadly in ML and deep learning, but you'll most often see them in kernel learning.

Let's revisit our solubility model and compare L1 and L2 regularization with a training curve. Note that this code is very slow because we must compute  $M$  models, where  $M$  is the number of points we want on our training curve. To keep things efficient for this textbook, I'll use few points on the curve.

First, we'll turn our training procedure into a function.

```
def fit_model(loss, npoints, eta=1e-6, batch_size=16, epochs=25):

    sample_idx = np.random.choice(
        np.arange(train_x.shape[0]), replace=False, size=npoints
    )
    sample_x = train_x[sample_idx, :]
    sample_y = train_y[sample_idx]

    # reshape into batches
    batch_num = npoints // batch_size
    # first truncate data so it's whole nubmer of batches
    trunc = batch_num * batch_size
    sample_x = sample_x[:trunc]
    sample_y = sample_y[:trunc]
    # split into batches
    x_batches = sample_x.reshape(-1, batch_size, sample_x.shape[-1])
    y_batches = sample_y.reshape(-1, batch_size)

    # get loss grad
    loss_grad = jax.grad(loss, (0, 1))

    # make trainable parameters
    # w = np.random.normal(scale = 1e-30, size=train_x.shape[0])
    w = np.zeros(sample_x.shape[0])
    b = np.mean(sample_y) # just set to mean, since it's a good first guess
    for _ in range(epochs):
        # go in random order
        for i in np.random.randint(0, batch_num - 1, size=batch_num):
            # update step
            x = x_batches[i]
            y = y_batches[i]
            grad = loss_grad(w, b, sample_x, x, y)
            w -= eta * grad[0]
            b -= eta * grad[1]
    return loss(w, b, sample_x, test_x, test_y)

# test it out
fit_model(loss, 256)
```

```
DeviceArray(3.8454313, dtype=float32)
```

Now we'll create L1 and L2 version of our loss. We must choose the *strength* of the regularization. Since our weights are less than 1, I'll choose much stronger regularization for the L2. These are hyperparameters though and you can adjust them to improve your fit.

```
@jax.jit
def loss_l1(w, b, train_x, x, y):
    return jnp.mean((batch_model(x, train_x, w, b) - y) ** 2) + 1e-2 * jnp.sum(
        jnp.abs(w)
    )

@jax.jit
```

(continues on next page)

(continued from previous page)

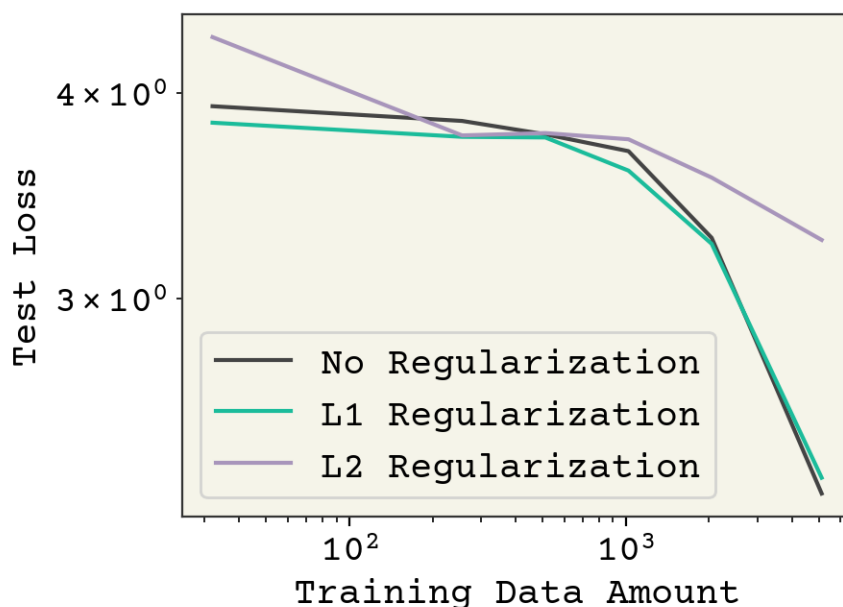
```
def loss_l2(w, b, train_x, x, y):
    return jnp.mean((batch_model(x, train_x, w, b) - y) ** 2) + 1e2 * jnp.sum(w**2)
```

And now we can generate the points necessary for our curves!

```
nvalues = [32, 256, 512, 1024, 2048, 1024 * 5]

nor_losses = [fit_model(loss, n) for n in nvalues]
l1_losses = [fit_model(loss_l1, n) for n in nvalues]
l2_losses = [fit_model(loss_l2, n) for n in nvalues]
```

```
plt.plot(nvalues, nor_losses, label="No Regularization")
plt.plot(nvalues, l1_losses, label="L1 Regularization")
plt.plot(nvalues, l2_losses, label="L2 Regularization")
plt.legend()
plt.xlabel("Training Data Amount")
plt.ylabel("Test Loss")
plt.gca().set_yscale("log")
plt.gca().set_xscale("log")
```



Finally, we see our training curves showing the different approaches. Regularization has some effect on final loss on the test data. It is hard to say if L1 and L2 are simply worse, or if I need to tune the regularization strength more. Nevertheless, this plot shows you how we typically evaluated kernel learning methods.



## 5.8 Exercises

1. Compute the analytical gradient for the dual form regression equation and use it to describe why the kernel function does not need to be differentiable.
2. Is it faster or slower to do training with kernel learning? Explain
3. Is it faster or slower to do inference with kernel learning? Explain
4. How can we modify Equation 4.2 to do classification?
5. Do the weight values give relative importance of training examples regardless of kernel?
6. Create a training curve from the above example showing 5 different L1 regularization strengths. Why might regularization not matter here?

## 5.9 Chapter Summary

- In this section we introduced kernel learning, which is a method to transform features into distance between samples.
- A kernel function takes two arguments and outputs a scalar. A kernel function must have three properties: positive, symmetric, and point-separating.
- The distance function (inner product) is a kernel function.
- Kernel functions do not need to be differentiable.
- Kernel learning is appropriate when you would like to use features that can only be specified as a binary kernel function.
- The number of trainable parameters in a kernel model is proportional to number of training points, not dimension of features.

## 5.10 Cited References



## **Part III**

### **C. Deep Learning**



## DEEP LEARNING OVERVIEW

**Deep learning** is a category of **machine learning**. Machine learning is a category of **artificial intelligence**. Deep learning is the use of neural networks to do machine learning, like classify and regress data. This chapter provides an overview and we will dive further into these topics in later chapters.

---

### Audience & Objectives

This chapter builds on *Regression & Model Assessment* and *Introduction to Machine Learning*. After completing this chapter, you should be able to

- Define deep learning
  - Define a neural network
  - Connect the previous regression principles to neural networks
- 

There are many good resources on deep learning to supplement these chapters. The goal of this book is to present a chemistry and materials-first introduction to deep learning. These other resources can help provide better depth in certain topics and cover topics we do not even cover, because I do not find them relevant to deep learning (e.g., image processing). I found the introduction the from [Ian Goodfellow's book](#) to be a good intro. If you're more visually oriented, Grant Sanderson has made a [short video series](#) specifically about neural networks that give an applied introduction to the topic. DeepMind has a high-level video showing what can be accomplished with [deep learning & AI](#). When people write "deep learning is a powerful tool" in their research papers, they typically cite [this Nature paper](#) by Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Zhang, Lipton, Li, and Smola have written a practical and example-driven [online book](#) that gives each example in Tensorflow, PyTorch, and MXNet. You can find many chemistry-specific examples and information about deep learning in chemistry via the excellent [DeepChem](#) project. Finally, some deep learning package provide a short introduction to deep learning via a tutorial of its API: [Keras](#), [PyTorch](#).

The main advice I would give to beginners in deep learning are to focus less on the neurological inspired language (i.e., connections between neurons), and instead view deep learning as a series of linear algebra operations where many of the matrices are filled with adjustable parameters. Of course nonlinear functions (activations) are used to join the linear algebra operations, but deep learning is essentially linear algebra operations specified via a "computation network" (aka computation graph) that vaguely looks like neurons connected in a brain.

---

### nonlinearity

A function  $f(\vec{x})$  is linear if two conditions hold:

$$f(\vec{x} + \vec{y}) = f(\vec{x}) + f(\vec{y}) \quad (6.1)$$

for all  $\vec{x}$  and  $\vec{y}$ . And

$$f(s\vec{x}) = sf(\vec{x}) \quad (6.2)$$

where  $s$  is a scalar. A function is **nonlinear** if these conditions do not hold for some  $\vec{x}$ .

---

## 6.1 What is a neural network?

The *deep* in deep learning means we have many layers in our neural networks. What is a neural network? Without loss of generality, we can view neural networks as 2 components: (1) a nonlinear function  $g(\cdot)$  which operates on our input features  $\mathbf{X}$  and outputs a new set of features  $\mathbf{H} = g(\mathbf{X})$  and (2) a linear model like we saw in our *Introduction to Machine Learning*. Our model equation for deep learning regression is:

$$\hat{y} = \vec{w}g(\vec{x}) + b \quad (6.3)$$

One of the main discussion points in our ML chapters was how arcane and difficult it is to choose features. Here, we have replaced our features with a set of trainable features  $g(\vec{x})$  and then use the same linear model as before. So how do we design  $g(\vec{x})$ ? That is the deep learning part.  $g(\vec{x})$  is a differentiable function composed of **layers**, which are themselves differentiable functions each with trainable weights (free variables). Deep learning is a mature field and there is a set of standard layers, each with a different purpose. For example, convolution layers look at a fixed neighborhood around each element of an input tensor. Dropout layers randomly inactivate inputs as a form of regularization. The most commonly used and basic layer is the **dense** or **fully-connected** layer.

Dense means each input element affects each output element. At one point, sparse layers were popular and had a nice analogy with how a brain is connected. However, dense layers do not require deciding which input/output connections to make and sparse layers are very rare now (except incidentally sparse layers, like convolutions).

A dense layer is defined by two things: the desired output feature shape and the **activation**. The equation is:

$$\vec{h} = \sigma(\mathbf{W}\vec{x} + \vec{b}) \quad (6.4)$$

where  $\mathbf{W}$  is a trainable  $D \times F$  matrix, where  $D$  is the input vector ( $\vec{x}$ ) dimension and  $F$  is the output vector ( $\vec{h}$ ) dimension,  $\vec{b}$  is a trainable  $F$  dimensional vector, and  $\sigma(\cdot)$  is the activation function.  $F$ , the number of output features, is an example of a **hyperparameter**: it is not trainable but is a problem dependent choice.  $\sigma(\cdot)$  is another hyperparameter. In principle, any differentiable function that has a domain of  $(-\infty, \infty)$  can be used for activation. However, the function should be nonlinear. If it were linear, then stacking multiple dense layers would be equivalent to one-big matrix multiplication and we'd be back at linear regression. So activations should be nonlinear. Beyond nonlinearity, we typically want activations that can “turn on” and “off”. That is, they have an output value of zero for some domain of input values. Typically, the activation is zero, or close to, for negative inputs.

The most simple activation function that has these two properties is the rectified linear unit (ReLU), which is

$$\sigma(x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

### 6.1.1 Universal Approximation Theorem

One of the reasons that neural networks are a good choice at approximating unknown functions ( $f(\vec{x})$ ) is that a neural network can approximate any function with a large enough network depth (number of layers) or width (size of hidden layers). There are many variations of this theorem – infinitely wide or infinitely deep neural networks. For example, any 1 dimensional function can be approximated by a depth 5 neural network with ReLU activation functions with infinitely wide layers (infinite hidden dimension) [LPW+17]. The universal approximation theorem shows that neural networks are, in the limit of large depth or width, expressive enough to fit any function.

## 6.1.2 Frameworks

Deep learning has lots of “gotchas” – easy to make mistakes that make it difficult to implement things yourself. This is especially true with numerical stability, which only reveals itself when your model fails to learn. We will move to a bit of a more abstract software framework than JAX for some examples. We'll use [Keras](#), which is one of many possible choices for deep learning frameworks.

## 6.1.3 Discussion

When it comes to introducing deep learning, I will be as terse as possible. There are good learning resources out there. You should use some of the reading above and tutorials put out by Keras (or PyTorch) to get familiar with the concepts of neural networks and learning.

## 6.2 Revisiting Solubility Model

We'll see our first example of deep learning by revisiting the solubility dataset with a two layer dense neural network.

## 6.3 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import dmol
```

### 6.3.1 Load Data

We download the data and load it into a [Pandas](#) data frame and then standardize our features as before.

```
# soldata = pd.read_csv('https://dataverse.harvard.edu/api/access/datafile/3407241?
↳format=original&gbrecs=true')
# had to rehost because dataverse isn't reliable
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
↳csv"
)
features_start_at = list(soldata.columns).index("MolWt")
feature_names = soldata.columns[features_start_at:]
# standardize the features
```

(continues on next page)

(continued from previous page)

```
soldata[feature_names] -= soldata[feature_names].mean()
soldata[feature_names] /= soldata[feature_names].std()
```

## 6.4 Prepare Data for Keras

The deep learning libraries simplify many common tasks, like splitting data and building layers. This code below builds our dataset from numpy arrays.

```
full_data = tf.data.Dataset.from_tensor_slices(
    (soldata[feature_names].values, soldata["Solubility"].values)
)
N = len(soldata)
test_N = int(0.1 * N)
test_data = full_data.take(test_N).batch(16)
train_data = full_data.skip(test_N).batch(16)
```

Notice that we used `skip` and `take` (See `tf.data.Dataset`) to split our dataset into two pieces and create batches of data.

## 6.5 Neural Network

Now we build our neural network model. In this case, our  $g(\vec{x}) = \sigma(\mathbf{W}^0 \vec{x} + \vec{b})$ . We will call the function  $g(\vec{x})$  a *hidden layer*. This is because we do not observe its output. Remember, the solubility will be  $y = \vec{w}g(\vec{x}) + b$ . We'll choose our activation,  $\sigma(\cdot)$ , to be `tanh` and the output dimension of the hidden-layer to be 32. The choice of `tanh` is empirical — there are many choices of nonlinearity and they are typically chosen based on efficiency and empirical accuracy. You can read more about this Keras [API here](#), however you should be able to understand the process from the function names and comments.

```
# our hidden layer
# We only need to define the output dimension - 32.
hidden_layer = tf.keras.layers.Dense(32, activation="tanh")
# Last layer - which we want to output one number
# the predicted solubility.
output_layer = tf.keras.layers.Dense(1)

# Now we put the layers into a sequential model
model = tf.keras.Sequential()
model.add(hidden_layer)
model.add(output_layer)

# our model is complete

# Try out our model on first few datapoints
model(soldata[feature_names].values[:3])
```

```
<tf.Tensor: shape=(3, 1), dtype=float32, numpy=
array([[ 1.0637524 ],
       [-0.02571592],
       [ 0.31304714]], dtype=float32)>
```



**Jax vs Keras**

We could have implemented this in Jax, but it would have been a few more lines of code. To keep the focus high level, I've used Keras for this chapter.

We can see our model predicting the solubility for 3 molecules above. There may be a warning about how our Pandas data is using float64 (double precision floating point numbers) but our model is using float32 (single precision), which doesn't matter that much. It warns us because we are technically throwing out a little bit of precision, but our solubility has much more variance than the difference between 32 and 64 bit precision floating point numbers. We can remove this warning by modifying the last line to be:

```
model(soldata[feature_names].values[:3].astype(float))
```

At this point, we've defined how our model structure should work and it can be called on data. Now we need to train it! We prepare the model for training by calling `model.compile`, which is where we define our optimization (typically a flavor of stochastic gradient descent) and loss

```
model.compile(optimizer="SGD", loss="mean_squared_error")
```

Look back to the amount of work it took to previously set-up loss and optimization process! Now we can train our model

```
model.fit(train_data, epochs=50)
```

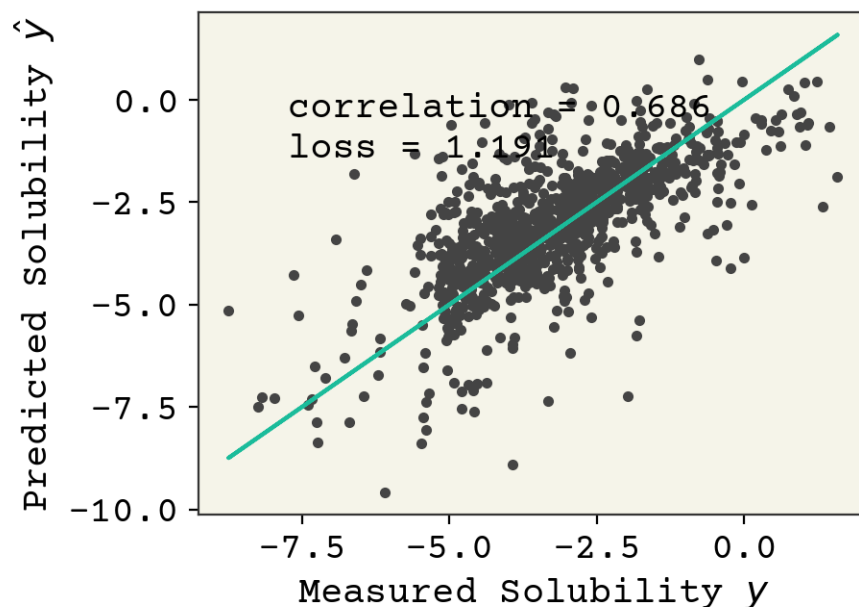
That was quite simple!

An epoch is one iteration over the whole dataset, regardless of batch size.

For reference, we got a loss about as low as 3 in our previous work. It was also much faster, thanks to the optimizations. Now let's see how our model did on the test data

```
# get model predictions on test data and get labels
# squeeze to remove extra dimensions
yhat = np.squeeze(model.predict(test_data))
test_y = soldata["Solubility"].values[:test_N]
```

```
plt.plot(test_y, yhat, ".")
plt.plot(test_y, test_y, "-")
plt.xlabel("Measured Solubility $y$")
plt.ylabel("Predicted Solubility $\hat{y}$")
plt.text(
    min(test_y) + 1,
    max(test_y) - 2,
    f"correlation = {np.corrcoef(test_y, yhat)[0,1]:.3f}",
)
plt.text(
    min(test_y) + 1,
    max(test_y) - 3,
    f"loss = {np.sqrt(np.mean((test_y - yhat)**2)):.3f}",
)
plt.show()
```



This performance is better than our simple linear model.

## 6.6 Exercises

1. Make a plot of the ReLU function. Prove it is nonlinear.
2. Try increasing the number of layers in the neural network. Discuss what you see in context of the bias-variance trade off
3. Show that a neural network would be equivalent to linear regression if  $\sigma(\cdot)$  was the identity function
4. What are the advantages and disadvantages of using deep learning instead of nonlinear regression for fitting data? When might you choose nonlinear regression over deep learning?

## 6.7 Chapter Summary

- Deep learning is a category of machine learning that utilizes neural networks for classification and regression of data.
- Neural networks are a series of operations with matrices of adjustable parameters.
- A neural network transforms input features into a new set of features that can be subsequently used for regression or classification.
- The most common layer is the dense layer. Each input element affects each output element. It is defined by the desired output feature shape and the activation function.
- With enough layers or wide enough hidden layers, neural networks can approximate unknown functions.
- Hidden layers are called such because we do not observe the output from one.
- Using libraries such as TensorFlow, it becomes easy to split data into training and testing, but also to build layers in the neural network.
- Building a neural network allows us to predict various properties of molecules, such as solubility.

## 6.8 Cited References



## STANDARD LAYERS

We will now see an overview of the enormous diversity in deep learning layers. This survey is necessarily limited to standard layers and we begin *without* considering the key layers that enable deep learning of molecules and materials. Almost all the layers listed below came out of a model for a specific task and were not thought-up independently. That means that some of the layers are suited to specific tasks and often the nomenclature around that layer is targeted towards a specific kinds of data.

---

### Audience & Objectives

This chapter builds on the overview from *Deep Learning Overview* and *Regression & Model Assessment*. After completing this chapter, you should be able to:

- Construct a neural network with various layers
- Understand how layers change shapes
- Recognize hyperparameters in a neural network
- Split data into train, test, and validation
- Regularize to prevent overfitting

---

The most common type is image data and we first begin with an overview of how image features are represented. Generally, an image is a rank 3 tensor with shape  $(H, W, C)$  where  $H$  is the height of the image,  $W$  is the width, and  $C$  is the number of channels (typically 3 – red, green, blue). Since all training is in batches, the input features shape will be  $(B, H, W, C)$ . Often layers will discuss input as having a batch axis, some number of shape axes, and then finally a channel axis. The layers will then operate on perhaps only the channels or only the shape dimensions. The layers are all quite flexible, so this is not a limitation in practice, but it's important to know when reading about layer types. Often the documentation or literature will mention *batch number* or *channels* and this is typically the first and last axes of a tensor, respectively.

---

**Note:** Everything and nothing is batched in deep learning. Practically, data is always batched. Even if your data is not batched, the first axis input to a neural network is of unspecified dimension and called the batch axis. Many frameworks make this implicit, meaning if you say the output from one layer is shape  $(4, 5)$ , it will be  $(B, 4, 5)$  when you actually inspect data. Or, for example in JAX, you can write your code without batching and make it batched through a function transform. So, all data is batched but often the math, frameworks, and documentation make it seem as if there is no batch axis.

---

An example of what a neural network looks like is shown in Fig. 7.1. In this case, its input is a 128x128 images with 3 channels (red, green, blue) and it outputs is a vector of probabilities of length 128 that indicate the class of the images. In other words, it takes in an image and gives it a probability of 128 possible labels like “cat” or “vase” or “crane”. The words annotating the figure indicate the different layer types we’ll learn about below.

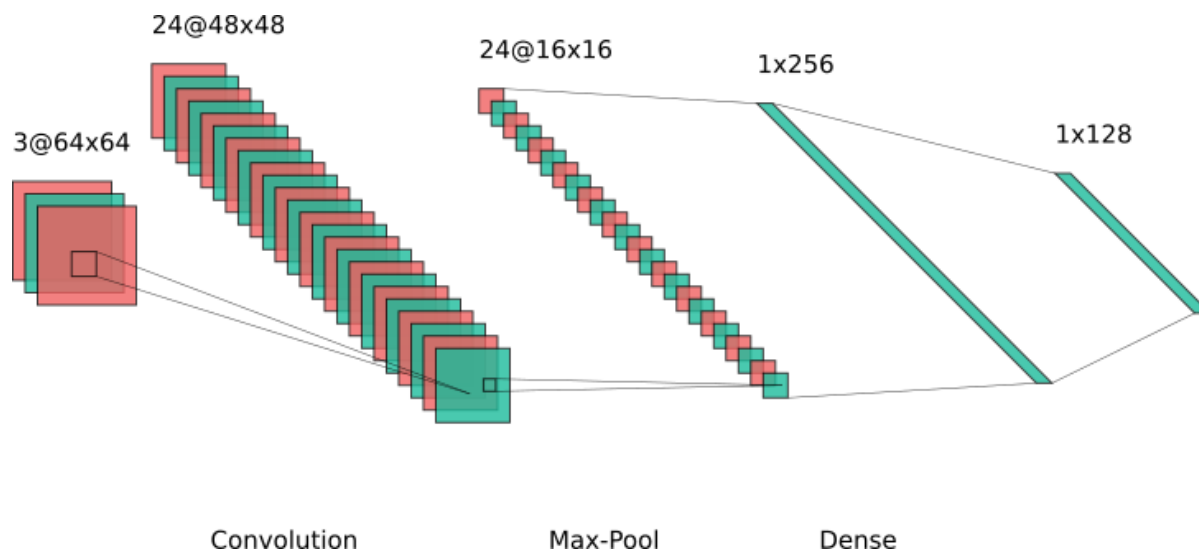


Fig. 7.1: A typical neural network architecture is composed of multiple layers. This network is used to classify images.

## 7.1 Hyperparameters

We saw from the Full connected (FC)/Dense layer that we have to choose if we use bias, the activation, and the output shape. As we learn about more complex layers, there will be more choices. These choices begin to accumulate and in a neural network you may have billions of possible combinations of them. These choices about shape, activation, initialization, and other layer arguments are called **hyperparameters**. They are parameters in the sense that they can be tuned, but they are not trained on our data so we call them hyperparameters to distinguish them from the “regular” parameters like value of weights and biases in the layers. The name is inherited from Bayesian statistics.

Choosing these hyperparameters is difficult and we typically rely on the body of existing literature to understand ranges of reasonable parameters. In deep learning, we usually are in a regime of hyperparameters which yield many trainable parameters (deep networks) and thus our models can represent any function. Our models are expressive. However, optimizing hyperparameters makes training faster and/or require less data. For example, papers have shown that carefully choosing the initial value of weights can be more effective than complex architecture [GB10]. Another example found that convolutions, which are thought to be the most important layer for image recognition, are not necessary if hyperparameters are chosen correctly for dense neural networks[CirecsanMGS10]. This is now changing, with options for tuning hyperparameters, but the current state-of-the art is to take hyperparameters from previous work as a starting guess and change a little if you believe it is needed.

### 7.1.1 Validation

The number of hyperparameters is high enough that overfitting can actually occur by choosing hyperparameters that minimize error on the test set. This is surprising because we don’t explicitly train hyperparameters. Nevertheless, you will find in your own work that if you use the test data extensively in hyperparameter tuning and for assessing overfitting of the regular training parameters, your performance will be overfit to the testing data. To combat this, we split our data three ways in deep learning:

1. Training data: used for trainable parameters.
2. Validation data: used to choose hyperparameters or measure overfitting of training data
3. Test data: data not used for anything except final reported error

To clean-up our nomenclature here, we use the word **generalization error** to refer to performance on a hypothetical

infinite stream of unseen data. So regardless of if you split three-ways or use other approaches, generalization error means error on unseen data.

You can replace this three-way split with cross-validation methods from previously, but remember that those require training  $k$ -times. Thus you rarely see  $k$ -fold cross-validation and even more rarely see leave-one-out or Jackknife because of how computationally expensive it is to train models.

### 7.1.2 Tuning

So how do you tune hyperparameters? The main answer is by hand, but this is an active area of research. Hyperparameters are continuous (e.g., regularization strength), categorical (e.g., which activation), and discrete variables (e.g., number of layers). One category of ways to tune hyperparameters is a topic called meta-learning[FAL17], which aims to learn hyperparameters by looking at multiple related datasets. Another area is auto-machine learning (auto-ML)[ZL17], where optimization strategies that do not require derivatives can tune hyperparameters. An important category of optimization related to hyperparameter tuning is **multi-armed bandit** optimization where we explicitly treat the fact that we have a finite amount of computational resources for tuning hyperparameters[LJD+18]. A comprehensive overview on hyperparameters and tuning techniques can be found in *Hyperparameter Tuning*.

## 7.2 Common Layers

Now that we have some understanding of hyperparameters and their role, let's now survey the common types of layers.

### 7.2.1 Convolutions

You can find a more thorough overview of [convolutions here](#) and [here with more visuals](#). Here is a [nice video on this](#). Convolutions are the most commonly used input layer when dealing with images or other data defined on a regular grid. In chemistry, you'll see convolutions on protein or DNA sequences, on 2D imaging data, and occasionally on 3D spatial data like average density from a molecular simulation. What makes a convolution different from a dense layer is that the number of trainable weights is more flexible than input grid shape  $\times$  output shape, which is what you would get with a dense layer. Since the trainable parameters don't depend on the input grid shape, you don't learn to depend on location in the image. This is important if you're hoping to learn something independent of location on the input grid – like if a specific object is present in the image independent of where it is located.

In a convolution, you specify a **kernel shape** that defines the size of trainable parameters. The kernel shape defines a window over your input data in which a dense neural network is applied. The rank of the kernel shape is the rank of your grid + 1, where the extra axis accounts for channels. For example, for images you might define a kernel shape of  $5 \times 5$ . The kernel shape will become  $5 \times 5 \times C$ , where  $C$  is the number of channels. When referring to a convolution as 1D, 2D, or 3D, we're referring to the grid of the input data and thus the kernel shape. A 2D convolution actually has an input of rank 4 tensors, the extra 2 axes accounting for batch and channels. The kernel shape of  $5 \times 5$  means that the output of a specific value in the grid will depend on its 24 nearest neighboring pixels (2 in each direction). Note that the kernel is used like a normal dense layer – it can have bias (dimension  $C$ ), output activation, and regularization.

Practically, convolutions are always grouped in parallel. You have a set of  $F$  kernels, where  $F$  is called the number of **filters**. Each of these filters is completely independent and if you examine what they learn, some filters will learn to identify squares and some might learn to identify color or others will learn textures. Filters is a term left-over from image processing, which is the field where convolutions were first explored. Combining all of these together, a 2D convolution will have an input shape of  $(B, H, W, C)$  and an output of  $(B, \approx H, \approx W, F)$ , where  $F$  is the number of filters chosen, and the  $\approx$  accounts for the fact that when you slide your kernel window over the input data, you'll lose some values on

the edge. This can either be treated by padding, so your input height and width match output height and width, or your dimensionality is reduced by a small amount (e.g., going from  $128 \times 128$  to  $125 \times 125$ ). A 1D convolution will have input shape  $(B, L, C)$  and output shape  $(B, \approx L, F)$ . As a practical example, consider a convolution on DNA.  $L$  is length of the sequence.  $C$ , your channels, will be **one-hot indicators** for the base (T, C, A, G).

### padding

Padding means insert some constants to make a tensor increase in shape. For example, if I want all my tensors to be of shape (32,32) and some are smaller, I could pad by adding 0s until the shape is (32,32).

One of the important properties we'll begin to discuss is **invariances** and **equivariances**. An invariance means the output from a neural network (or a general function) is insensitive to changes in input. For example, a translational invariance means that the output does not change if the input is translated. Convolutions and pooling should be chosen when you want to have **translation invariance**. For example, if you are identifying if a cat exists in an image, you want your network to give the same answer even if the cat is translated in the image to different regions. However, just because you use a convolution layer does not make a neural network automatically translationally invariant. You must include other layers to achieve this. Convolutions are actually translationally equivariant – if you translate all pixels in your input, the output will also be translated. People usually do not distinguish between equivariance and invariance. If you are trying to identify *where* a cat is located in an image you would still use convolutions but you want your neural network to be translationally equivariant, meaning your guess about where the cat is located is sensitive to where the cat is located in the input pixels. The reason convolutions have this property is that the trainable parameters, the kernel, are location independent. You use the same kernel on every region of the input.

### equivariance

It's a bit more complicated. Convolutions and pooling are *almost* translationally equivariant. There are edge effects because images are not infinitely wide so something special always must be done to deal with pixels near the edges of images, which prevents them from being fully equivariant.

## 7.2.2 Pooling

Convolutions are commonly paired with pooling layers because pooling also is translationally equivariant. If your goal is to produce a single number (regression) or class (classification) from an input image or sequence, you need to reduce the rank to 0, a scalar. After a convolution, you could use a reduction like average or maximum. It has been shown empirically that reducing the number of elements of your features more gradually is better. One way is through **pooling**. Pooling is similar to convolutions, in that you define a kernel shape (called window shape), but pooling has no trainable parameters. Instead, you run a window across your input grid and compute a reduction. Commonly an average or maximum is computed. If your pool window is a  $2 \times 2$  on an input of  $(B, H, W, F)$ , then your output will be  $(B, H/2, W/2, F)$ . In convolutional neural networks, often multiple **blocks** of convolutions and poolings are combined. For example, you might use three rounds of convolutions and pooling to take an image from  $32 \times 32$  down to a  $4 \times 4$ . Read more about [pooling here](#)



## 7.2.3 Embedding

Another important type of input layers are **embeddings**. Embeddings convert integers into vectors. They are typically used to convert characters or words into numerical vectors. The characters or words are first converted into **tokens** separately as a pre-processing step and then the input to the embedding layer is the indices of the token. The indices are integer values that index into a dictionary of all possible tokens. It sounds more complex than it is. For example, we might tokenize characters in the alphabet. There are 26 tokens (letters) in the alphabet (dictionary of tokens) and we could convert the word “hello” into the indices [7, 4, 11, 11, 14], where 7 means the 7th letter of the alphabet.

After converting into indices, an embedding layer converts these indices into dense vectors of a chosen dimension. The rationale behind embeddings is to go from a large discrete space (e.g., all words in the English language) into a much smaller space of real numbers (e.g., vectors of size 5). You might use embeddings for converting monomers in a polymer into dense vectors or atom identities in a molecule or DNA bases. We’ll see an embedding layer in the example below.

## 7.3 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

## 7.4 Example

At this point, we have enough common layers to try to build a neural network. We will combine these three layers to predict if a protein is soluble. Our dataset comes from [CSTR14] and consists of proteins known to be soluble or insoluble. As usual, the code below sets-up our imports.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import urllib
import dmol
```

Our task is binary classification. The data is split into two: positive and negative examples. We’ll need to rearrange a little into a normal dataset with labels and training/testing split. We also really really need to shuffle our data, so it doesn’t see all positives and then all negatives.

```
urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/master/data/solubility.npz",
    "solubility.npz",
)
with np.load("solubility.npz") as r:
    pos_data, neg_data = r["positives"], r["negatives"]

# create labels and stich it all into one
# tensor
```

(continues on next page)

(continued from previous page)

```

labels = np.concatenate(
    (
        np.ones((pos_data.shape[0], 1), dtype=pos_data.dtype),
        np.zeros((neg_data.shape[0], 1), dtype=pos_data.dtype),
    ),
    axis=0,
)
features = np.concatenate((pos_data, neg_data), axis=0)
# we now need to shuffle before creating TF dataset
# so that our train/test/val splits are random
i = np.arange(len(labels))
np.random.shuffle(i)
labels = labels[i]
features = features[i]
full_data = tf.data.Dataset.from_tensor_slices((features, labels))

# now split into val, test, train
N = pos_data.shape[0] + neg_data.shape[0]
print(N, "examples")
split = int(0.1 * N)
test_data = full_data.take(split).batch(16)
nontest = full_data.skip(split)
val_data, train_data = nontest.take(split).batch(16), nontest.skip(split).shuffle(
    1000
).batch(16)

```

18453 examples

Before getting to modeling, let's examine our data. The protein sequences have already been tokenized. There are 20 possible values at each position because there are 20 amino acids possible in proteins. Let's see a soluble protein

pos\_data[0]

```

array([[13, 17, 15, 16, 1, 1, 1, 17, 8, 9, 7, 1, 1, 4, 7, 6, 2,
        11, 2, 7, 11, 2, 8, 11, 17, 2, 6, 11, 15, 17, 8, 20, 1, 20,
        20, 17, 1, 6, 4, 8, 7, 20, 1, 9, 8, 1, 17, 20, 16, 17, 20,
        16, 20, 11, 16, 6, 6, 15, 11, 2, 10, 8, 20, 16, 11, 2, 2, 8,
        16, 19, 11, 17, 8, 11, 10, 2, 6, 2, 2, 20, 14, 1, 11, 3, 20,
        11, 16, 16, 2, 6, 16, 1, 20, 1, 4, 18, 14, 1, 3, 15, 7, 2,
        15, 2, 8, 18, 2, 6, 14, 4, 19, 20, 2, 18, 17, 1, 9, 15, 12,
        1, 8, 13, 15, 20, 11, 7, 4, 1, 11, 1, 6, 11, 9, 5, 2, 11,
        17, 4, 11, 10, 15, 11, 8, 1, 16, 4, 4, 11, 11, 20, 1, 7, 20,
        11, 4, 8, 2, 8, 2, 3, 8, 2, 15, 11, 20, 3, 14, 3, 8, 2,
        11, 9, 4, 20, 7, 14, 2, 8, 20, 20, 2, 20, 16, 2, 4, 6, 15,
        16, 1, 20, 17, 16, 11, 7, 0, 0, 0, 0, 0, 0])

```

Notice that integers/indices are used because our data is tokenized already. To make our data all be the same input shape, a special token (0) is inserted at the end indicating no amino acid is present. This needs to be treated carefully, because it should be zeroed throughout the network. Luckily this is built into Keras, so we do not need to worry about it.

This data is perfect for an embedding because we need to convert token indices to real vectors. Then we will use 1D convolutions to look for sequence patterns with pooling. We need to then make sure our final layer is a sigmoid, just like in *Classification*. This architecture is inspired by the original work on pooling with convolutions [LecunBottouBengio-Haffner98]. The number of layers and kernel sizes below are hyperparameters. You are encouraged to experiment with these or find improvements!

We begin with an embedding. We'll use a 2-dimensional embedding, which gives us two channels for our sequence. We'll just choose our kernel filter size for the 1D convolution to be 5 and we'll use 16 filters. Beyond that, the rest of the network is about distilling gradually into a final class.

```
model = tf.keras.Sequential()

# make embedding and indicate that 0 should be treated specially
model.add(
    tf.keras.layers.Embedding(
        input_dim=21, output_dim=16, mask_zero=True, input_length=pos_data.shape[-1]
    )
)

# now we move to convolutions and pooling
model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=5, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=4))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

# now we flatten to move to hidden dense layers.
# Flattening just removes all axes except 1 (and implicit batch is still in there as
# always!)

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

=====

embedding (Embedding)	(None, 200, 16)	336
-----------------------	-----------------	-----

conv1d (Conv1D)	(None, 196, 16)	1296
-----------------	-----------------	------

max_pooling1d (MaxPooling1D	(None, 49, 16)	0
)		
conv1d_1 (Conv1D)	(None, 47, 16)	784
max_pooling1d_1 (MaxPooling	(None, 23, 16)	0
1D)		
conv1d_2 (Conv1D)	(None, 21, 16)	784
max_pooling1d_2 (MaxPooling	(None, 10, 16)	0
1D)		
flatten (Flatten)	(None, 160)	0
dense (Dense)	(None, 256)	41216
dense_1 (Dense)	(None, 64)	16448
dense_2 (Dense)	(None, 1)	65

```
=====
```

```
Total params: 60,929
```

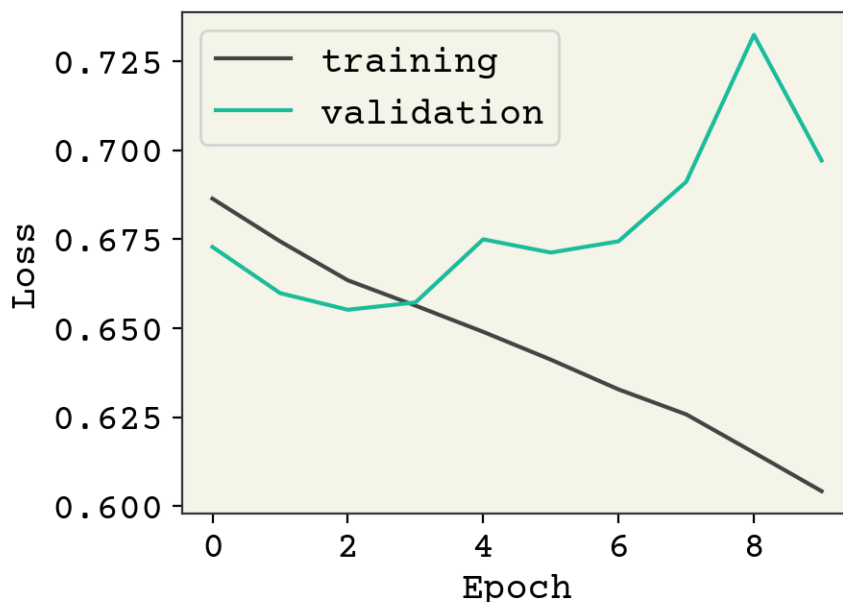
```
Trainable params: 60,929
```

```
Non-trainable params: 0
```

Take a moment to look at the model summary (shapes). This is a fairly complex neural network. If you can understand this, you'll have a grasp on most current networks used in deep learning. Now we'll begin training. Since we are doing classification, we'll also examine accuracy on validation data as we train.

```
model.compile("adam", loss="binary_crossentropy", metrics=["accuracy"])
result = model.fit(train_data, validation_data=val_data, epochs=10, verbose=0)
```

```
plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



You can see this is a classic case of overfitting, with the validation data rising quickly as we improve our loss on the training data. Indeed, our model is quite expressive in its capability to fit the training data but it is incidentally fitting the noise. We have 61,000 trainable parameters and about 15,000 training examples, so this is not a surprise. However, we still able to learn a little bit – our accuracy is above 50%. This is actually a challenging dataset and the state-of-the art result is 77% accuracy [KRK+18]. We need to expand our tools to include layers that can address overfitting.

## 7.5 Back propagation

At this stage, we should probably talk about back propagation and its connection to automatic gradient computation (autograds). This is how training “just works” when we take a gradient. This is actually a bit of a complicated topic, but it also nearly invisible to users of modern deep learning packages. Thus, I have chosen to not cover it in this book. You can find comprehensive discussions of modern autograd in [BPRS18] and in the [Jax manual](#).

## 7.6 Regularization

As we saw in the ML chapters, regularization is a strategy that changes your training procedure (often by adding loss terms) to prevent overfitting. There is a nice argument for it in the bias-variance trade-off regarding model complexity, however this doesn’t seem to hold in practice [NMB+18]. Thus, we view regularization as an empirical process. Regularization, like other hyperparameter tuning, is dependent on the layers, how complex your model is, your data, and especially if your model is underfit or overfit. Underfitting means you could train longer to improve validation loss. Adding regularization if your model is underfit will usually reduce performance. Consider training longer or adjusting learning rates if you observe this.

### 7.6.1 Early Stopping

The most commonly used and simplest form of regularization is **early stopping**. Early stopping means monitoring the loss on your validation data and stopping training once it begins to rise. Normally, training is done until converged – meaning the loss stops decreasing. Early stopping tries to prevent overfitting by looking at the loss on unseen data (validation data) and stopping once that begins to rise. This is an example of regularization because the weights are limited to move a fixed distance from their initial value. Just like in L2 regularization, we’re squeezing our trainable weights. Early stopping can be a bit more complicated to implement in practice than it sounds, so check out how frameworks do it before trying to implement yourself (e.g., `tf.keras.callbacks.EarlyStopping`).

### 7.6.2 Weight

**Weight regularization** is the addition of terms to the loss that depend on the trainable weights in the solubility model example. These can be L2 ( $\sqrt{\sum w_i^2}$ ) or L1 ( $\sum |w_i|$ ). You must choose the strength, which is expressed as a parameter (often denoted  $\lambda$ ) that should be much less than 1. Typically values of 0.1 to  $1 \times 10^{-4}$  are chosen. This may be broken into **kernel regularization**, which affects the multiplicative weights in a dense or convolution neural network, and **bias regularization**. Bias regularization is rarely seen in practice.

### 7.6.3 Activity

**Activity regularization** is the addition of terms to the loss that depend on the *output* from a layer. Activity regularization ultimately leads to minimizing weight magnitudes, but it makes the strength of that effect depend on the output from the layers. Weight regularization has the strongest effect on weights that have little effect on layer output, because they have no gradient if they have little effect on the output. In contrast, activity regularization has the strongest effect on weights that greatly affect layer output. Conceptually, weight regularization reduces weights that are unimportant but could harm generalization error if there is a shift in the type of features seen in testing. Activity regularization reduces weights that affect layer output and is more akin to early stopping by reducing how far those weights can move in training.

### 7.6.4 Batch Normalization

It is arguable if batch normalization is a regularization technique – there have been probably 10,000 papers on why it works. Batch normalization is a layer that is added to a neural network with trainable weights, but its trainable weights are not updated via gradient descent of the loss. Batch normalization has a layer equation of:

$$f(X) = \frac{X - \bar{X}}{S} \quad (7.1)$$

where  $\bar{X}$  and  $S$  are the sample mean and variance taken across the batch axis (zeroth axis of  $X$ ). This has the effect of “smoothing” out the magnitudes of values seen between batches. Remember that activations like ReLU depend on values being near 0 (since the nonlinear part is at  $x = 0$ ) and tanh has the most change in output around  $x = 0$ , so you typically want your intermediate layer outputs to be around 0. At inference time you may not have batches or your batches may be a different size, so  $\bar{X}$  and  $S$  are set to the average across all batches seen in training data. A common explanation of batch normalization is that it smooths out the optimization landscape by forcing layer outputs to be approximately normal[STIMkadyr18].

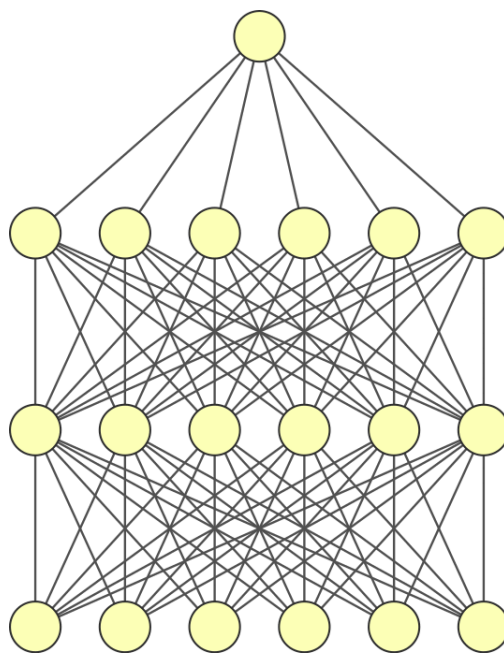
**Inference** is the word for when you use your model to make predictions. Training is when you train the model and inference is when you use the model.

### Layer Normalization

Batch normalization depends on there being a constant batch size. Some kinds of data, like text or a graphs, have different sizes and so the batch mean/variance can change significantly. **Layer normalization** avoids this problem by normalizing across the *features* (the non-zero axes) instead of the batch. This has a similar effect of making the layer output features behave well-centered at 0 but without having highly variable means/variances because of batch to batch variation. You’ll see these in graph neural networks and recurrent neural networks, with both take variable sized inputs.

### 7.6.5 Dropout

The last regularization type is **dropout**. Like batch normalization, dropout is typically viewed as a layer and has no trainable parameters. In dropout, we randomly zero-out specific elements of the input and then rescale the output so its average magnitude is unchanged. You can think of it like *masking*. There is a mask tensor  $M$  which contains 1s and 0s and is multiplied by the input. It is called masking because we mask whatever was in the elements that were multiplied by 0. Then the output is multiplied by  $|M| / \sum M$  where  $|M|$  is the number of elements in  $M$ . Dropout forces your neural network to learn to use different features or “pathways” by zeroing out elements. Weight regularization squeezes unused trainable weights through minimization. Dropout tries to force all trainable weights to be used by randomly negating weights. Dropout is more common than weight or activity regularization but has arguable theoretical merit. Some have proposed it is a kind of sampling mechanism for exploring model variations[GG16]. Despite it appearing ad-hoc, it is effective. Note that dropout is only used during training, not for inference. You need to choose the dropout rate when using it, another hyperparameter. Usually, you will want to choose a rate of 0.05–0.35. 0.2 is common. Too small of a value – meaning you rarely do dropout – makes the effect too small to matter. Too large of a value – meaning you often dropout values – can prevent you from actually learning. As fewer nodes get updated with dropout, larger learning rates with decay and a larger momentum can help with the model’s performance.



Standard Deep NN

Fig. 7.2: Dropout.

## 7.7 Residues

One last “layer” note to mention is residues. One of the classic problems in neural network training is **vanishing gradients**. If your neural network is deep and many features contribute to the label, you can have very small gradients during training that make it difficult to train. This is visible as underfitting. One way this can be addressed is through careful choice of optimization and learning rates. Another way is to add “residue” connections in the neural network. Residue connections are a fancy way of saying “adding” or “concatenating” later layers with early layers. The most common way to do this is:

$$X^{i+1} = \sigma(W^i X^i + b^i) + X^i \quad (7.2)$$

This is the usual equation for a dense neural network but we’ve added the previous layer output ( $X^i$ ) to our output. Now when you take a gradient of earlier weights from layer  $i - 1$ , they will appear through both the  $\sigma(W^i X^i + b^i)$  term via the chain rule and the  $X^i$  term. This goes around the activation  $\sigma$  and the effect of  $W^i$ . Note this continues at all layers and then a gradient can propagate back to earlier layers via either term. You can add the “residue” connection to the previous layer as shown here or go back even earlier. You can also be more complex and use a trainable function for how the residue term ( $X^i$ ) can be treated. For example:

$$X^{i+1} = \sigma(W^i X^i + b^i) + W'^i X^i \quad (7.3)$$

where  $W'^i$  is a set of new trainable parameters. We have seen that there are many hyperparameters for tuning and adjusting residue connections is one of the least effective things to adjust. So don’t expect much of an improvement. However, if you’re seeing underfitting and inefficient training, perhaps it’s worth investigating.



## 7.8 Blocks

You can imagine that we might join a dense layer with dropout, batch normalization, and maybe a residue. When you group multiple layers together, this can be called a **block** for simplicity. For example, you might use the word “convolution block” to describe a sequential layers of convolution, pooling, and dropout.

## 7.9 Dropout Regularization Example

Now let’s try to add a few dropout layers to see if we can do better on our example above.

```
model = tf.keras.Sequential()

# make embedding and indicate that 0 should be treated specially
model.add(
    tf.keras.layers.Embedding(
        input_dim=21, output_dim=16, mask_zero=True, input_length=pos_data.shape[-1]
    )
)

# now we move to convolutions and pooling
# NOTE: Keras doesn't respect masking here
# I should switch to PyTorch.
model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=5, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=4))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

# now we flatten to move to hidden dense layers.
# Flattening just removes all axes except 1 (and implicit batch is still in there as_
# always!)
model.add(tf.keras.layers.Flatten())

# Here is the dropout
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))

model.summary()
```

Model: "sequential\_1"

Layer (type)

Output Shape

Param #

=====

embedding_1 (Embedding)	(None, 200, 16)	336
-------------------------	-----------------	-----

conv1d_3 (Conv1D)	(None, 196, 16)	1296
-------------------	-----------------	------

max_pooling1d_3 (MaxPooling	(None, 49, 16)	0
-----------------------------	----------------	---

1D)

conv1d_4 (Conv1D)	(None, 47, 16)	784
-------------------	----------------	-----

max_pooling1d_4 (MaxPooling	(None, 23, 16)	0
-----------------------------	----------------	---

1D)

conv1d_5 (Conv1D)	(None, 21, 16)	784
-------------------	----------------	-----

max_pooling1d_5 (MaxPooling	(None, 10, 16)	0
-----------------------------	----------------	---

1D)

flatten_1 (Flatten)	(None, 160)	0
---------------------	-------------	---

dropout (Dropout)	(None, 160)	0
-------------------	-------------	---

dense_3 (Dense)	(None, 256)	41216
-----------------	-------------	-------

dropout_1 (Dropout)	(None, 256)	0
---------------------	-------------	---

dense_4 (Dense)	(None, 64)	16448
-----------------	------------	-------

dropout_2 (Dropout)	(None, 64)	0
---------------------	------------	---

dense_5 (Dense)	(None, 1)	65
-----------------	-----------	----

=====

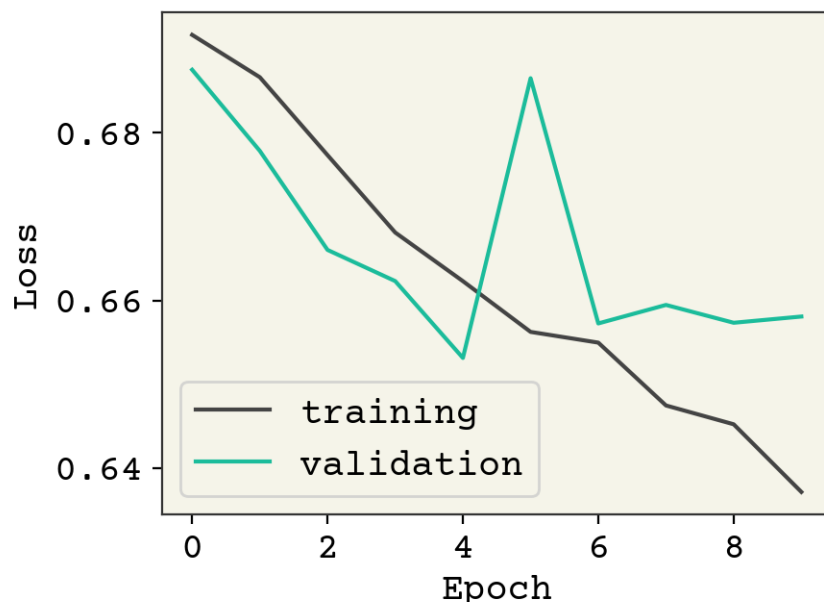
Total params: 60,929

Trainable params: 60,929

Non-trainable params: 0

```
model.compile("adam", loss="binary_crossentropy", metrics=["accuracy"])
result = model.fit(train_data, validation_data=val_data, epochs=10, verbose=0)
```

```
plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



We added a few dropout layers and now we can see the validation loss is a little better but additional training will indeed result in rising. Feel free to try the other ideas above to see if you can get the validation loss to decrease like the training loss.

## 7.10 L2 Weight Regularization Example

Now we'll demonstrate adding weight regularization.

```
model = tf.keras.Sequential()

# make embedding and indicate that 0 should be treated specially
model.add(
    tf.keras.layers.Embedding(
        input_dim=21, output_dim=16, mask_zero=True, input_length=pos_data.shape[-1]
    )
)

# now we move to convolutions and pooling
model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=5, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=4))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

# now we flatten to move to hidden dense layers.
# Flattening just removes all axes except 1 (and implicit batch is still in there as
# always!)
model.add(tf.keras.layers.Flatten())
```

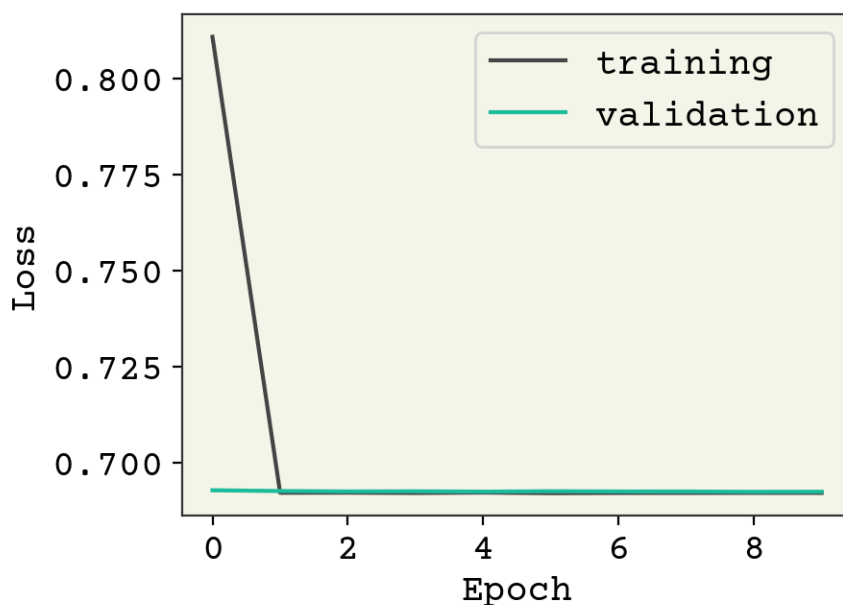
(continues on next page)

(continued from previous page)

```
# HERE IS THE REGULARIZATION:
model.add(tf.keras.layers.Dense(256, activation="relu", kernel_regularizer="l2"))
model.add(tf.keras.layers.Dense(64, activation="relu", kernel_regularizer="l2"))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))

model.compile("adam", loss="binary_crossentropy", metrics=["accuracy"])
result = model.fit(train_data, validation_data=val_data, epochs=10, verbose=0)

plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



L2 regularization is too strong it appears, preventing learning. You could go back and reduce the strength; here we're just using the default which doesn't look appropriate for our setting. Tuning hyperparameters like this is a favorite past time of neural network engineers and we could go on forever. We'll stop here and leave it as an exercise for the reader to continue exploring hyperparameters.

## 7.11 Activation Functions

Recall in *Deep Learning Overview* we mentioned that activation functions must be nonlinear and we often want them to have a region of input where the output value is zero. ReLU is the simplest example that satisfies these conditions - its output is zero for negative inputs and  $f(x) = x$  for positive values. Choosing activation is another hyperparameter and choice that we make. People used activations like tanh or sigmoids in early neural network research. ReLU began to dominate in modern deep learning because it's so efficient that models could be made larger for the same runtime speed.

Since 2019, this has been revisited because modern GPUs can run a variety of activation functions now quite quickly[EYG19]. Two commonly used modern activation functions are Gaussian Error Linear Units (GELU)[HG16] and Swish[EYG19]. They are shown in Fig. 7.3. They have these two properties of nonlinearity and an ability to turn-off

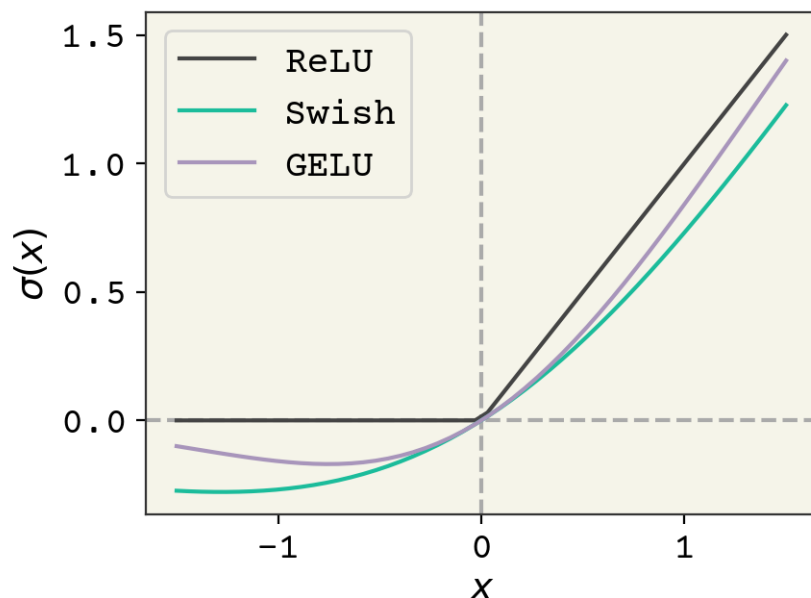


Fig. 7.3: Comparison of the usual ReLU activation function and GELU and Swish.

at negative values. They seem to give better results because of their non-zero gradient at negative values; they can continue to respond to gradients while they are turned off. It is more common now to see Swish than ReLU in most newer networks and GELU is specifically seen in transformers (discussed in [Deep Learning on Sequences](#)).

The equation for Swish is:

$$\sigma(x) = x \cdot \text{sigmoid}(x) = x \frac{1}{1 + e^{-x}}$$

and the equation for GELU is:

$$\sigma(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2} \left[ 1 + \text{erf} \left( \frac{x - \mu}{\sigma\sqrt{2}} \right) \right]$$

## 7.12 Discussion

Designing and training neural networks is a complex task. The best approach is to always start simple and work your way up in complexity. Remember, you have to write correct code, create a competent model, and have clean data. If you start with a complex model it can be hard to discern if learning problems are due to bugs, the model, or the data. My advice is to always start with a pre-trained or simple baseline network from a previous paper. If you find yourself designing and training your own neural network, read through Andrej Karpathy's [excellent guide](#) on how to approach this task.

## 7.13 Chapter Summary

- Layers are created for specific tasks, and given the variety of layers, there are a vast number of permutations of layers in a deep neural network.
- Convolution layers are used for data defined on a regular grid (such as images). In a convolution, one defines the size of the trainable parameters through the kernel shape.
- An invariance is when the output from a neural network is insensitive to spatial changes in the input (translation, rotation, rearranging order)
- An equivariance is when the output from a neural network changes the same way as the input. See *Input Data & Equivariances* and *Equivariant Neural Networks* for concrete definitions.
- Convolution layers are often paired with pooling layers. A pooling layer behaves similarly to a convolution layer, except a reduction is computed and the output is a smaller shape (same rank) than the input.
- Embedding layers convert indices into vectors, and are typically used as pre-processing steps.
- Hyperparameters are choices regarding the shape of the layers, the activation function, initialization parameters, and other layer arguments. They can be tuned but are not trained on the data.
- Hyperparameters must be tuned by hand, as they can be continuous, categorical, or discrete variables, but there are algorithms being researched that tune hyperparameters.
- Tuning the hyperparameters can make training faster or require less training data.
- Using a validation data set can measure the overfitting of training data, and is used to help choose the hyperparameters.
- Regularization is an empirical technique used to change training procedures to prevent overfitting. There are five common types of regularization: early stopping, weight regularization, activity regularization, batch normalization, and dropout.
- Vanishing gradient problems can be addressed by adding “residue” connections, essentially adding later layers with early layers in the neural network.

## 7.14 Cited References





## GRAPH NEURAL NETWORKS

Historically, the biggest difficulty for machine learning with molecules was the choice and computation of “descriptors”. Graph neural networks (GNNs) are a category of deep neural networks whose inputs are graphs and provide a way around the choice of descriptors. A GNN can take a molecule directly as input.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Regression & Model Assessment*. Although it is defined here, it would be good to be familiarize yourself with graphs/networks. After completing this chapter, you should be able to

- Represent a molecule in a graph
- Discuss and categorize common graph neural network architectures
- Build a GNN and choose a read-out function for the type of labels
- Distinguish between graph, edge, and node features
- Formulate a GNN into edge-updates, node-updates, and aggregation steps

---

GNNs are specific layers that input a graph and output a graph. You can find reviews of GNNs in Dwivedi *et al.* [DJL+20], Bronstein *et al.* [BBL+17], and Wu *et al.* [WPC+20]. GNNs can be used for everything from coarse-grained molecular dynamics [LWC+20] to predicting NMR chemical shifts [YCW20] to modeling dynamics of solids [XFLW+19]. Before we dive too deep into them, we must first understand how a graph is represented in a computer and how molecules are converted into graphs.

You can find an interactive introductory article on graphs and graph neural networks at [distill.pub](https://distill.pub/2021/slrpw21) [SLRPW21]. Most current research in GNNs is done with specialized deep learning libraries for graphs. As of 2022, the most common are PyTorch Geometric, Deep Graph library, Spektral, and TensorFlow GNNs.

### 8.1 Representing a Graph

A graph  $\mathbf{G}$  is a set of nodes  $\mathbf{V}$  and edges  $\mathbf{E}$ . In our setting, node  $i$  is defined by a vector  $\vec{v}_i$ , so that the set of nodes can be written as a rank 2 tensor. The edges can be represented as an adjacency matrix  $\mathbf{E}$ , where if  $e_{ij} = 1$  then nodes  $i$  and  $j$  are connected by an edge. In many fields, graphs are often immediately simplified to be directed and acyclic, which simplifies things. Molecules are instead undirected and have cycles (rings). Thus, our adjacency matrices are always symmetric  $e_{ij} = e_{ji}$  because there is no concept of direction in chemical bonds. Often our edges themselves have features, so that  $e_{ij}$  is itself a vector. Then the adjacency matrix becomes a rank 3 tensor. Examples of edge features might be covalent bond order or distance between two nodes.

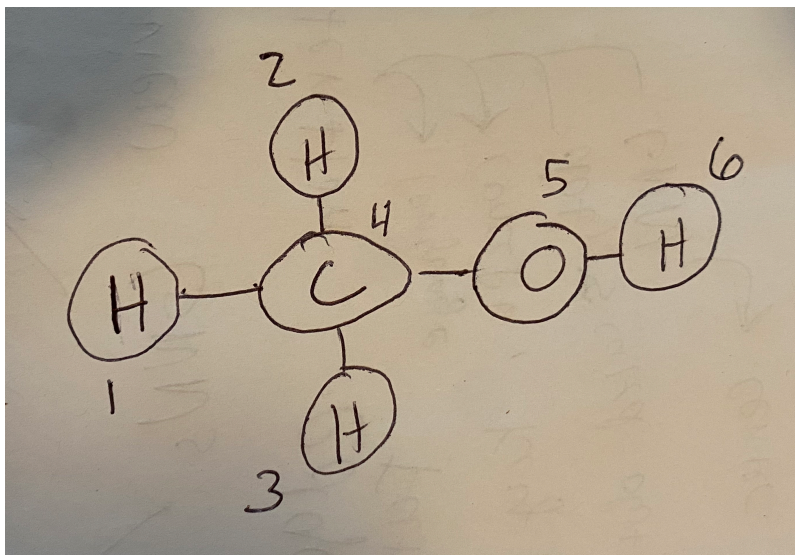


Fig. 8.1: Methanol with atoms numbered so that we can convert it to a graph.

### one-hot

Recall that a one-hot is a vector of all 0s and a single 1 -  $[0, 1, 0, 0]$ . The index of the non-zero element indicates the class. In this case, class is element.

Let's see how a graph can be constructed from a molecule. Consider methanol, shown in Fig. 8.1. I've numbered the atoms so that we have an order for defining the nodes/edges. First, the node features. You can use anything for node features, but often we'll begin with one-hot encoded feature vectors:

Node	C	H	O
1	0	1	0
2	0	1	0
3	0	1	0
4	1	0	0
5	0	0	1
6	0	1	0

$\mathbf{V}$  will be the combined feature vectors of these nodes. The adjacency matrix  $\mathbf{E}$  will look like:

	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	1	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Take a moment to understand these two. For example, notice that rows 1, 2, and 3 only have the 4th column as non-zero. That's because atoms 1-3 are bonded only to carbon (atom 4). Also, the diagonal is always 0 because atoms cannot be bonded with themselves.

You can find a similar process for converting crystals into graphs in Xie et al. [XG18]. We'll now begin with a function which can convert a smiles string into this representation.

## 8.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import tensorflow as tf
import pandas as pd
import rdkit, rdkit.Chem, rdkit.Chem.rdDepictor, rdkit.Chem.Draw
import networkx as nx
import dmol
```

```
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
    ↪CSV"
)
np.random.seed(0)
my_elements = {6: "C", 8: "O", 1: "H"}
```

The hidden cell below defines our function `smiles2graph`. This creates one-hot node feature vectors for the element C, H, and O. It also creates an adjacency tensor with one-hot bond order being the feature vector.

```
nodes, adj = smiles2graph("CO")
nodes
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

## 8.3 A Graph Neural Network

A graph neural network (GNN) is a neural network with two defining attributes:

1. Its input is a graph
2. Its output is permutation equivariant

We can understand clearly the first point. Here, a graph permutation means re-ordering our nodes. In our methanol example above, we could have easily made the carbon be atom 1 instead of atom 4. Our new adjacency matrix would then be:

	1	2	3	4	5	6
1	0	1	1	1	1	0
2	1	0	0	0	0	0
3	1	0	0	0	0	0
4	1	0	0	0	1	0
5	1	0	0	0	0	1
6	0	0	0	0	1	0

A GNN is permutation equivariant if the output change the same way as these exchanges. If you are trying to model a per-atom quantity like partial charge or chemical shift, this is obviously essential. If you change the order of atoms input, you would expect the order of their partial charges to similarly change.

Often we want to model a whole-molecule property, like solubility or energy. This should be **invariant** to changing the order of the atoms. To make an equivariant model invariant, we use read-outs (defined below). See [Input Data & Equivariances](#) for a more detailed discussion of equivariance.

### 8.3.1 A simple GNN

We will often mention a GNN when we really mean a layer from a GNN. Most GNNs implement a specific layer that can deal with graphs, and so usually we are only concerned with this layer. Let's see an example of a simple layer for a GNN:

$$f_k = \sigma \left( \sum_i \sum_j v_{ij} w_{jk} \right) \quad (8.1)$$

This equation shows that we first multiply every node ( $v_{ij}$ ) feature by trainable weights  $w_{jk}$ , sum over all node features, and then apply an activation. This will yield a single feature vector for the graph. Is this equation permutation equivariant? Yes, because the node index in our expression is index  $i$  which can be re-ordered without affecting the output.

Let's see an example that is similar, but not permutation equivariant:

$$f_k = \sigma \left( \sum_i v_{ij} w_{ik} \right) \quad (8.2)$$

This is a small change. We have one weight vector per node now. This makes the trainable weights depend on the ordering of the nodes. Then if we swap the node ordering, our weights will no longer align. So if we were to input two methanol molecules, which should have the same output, but we switched two atom numbers, we would get different answers. These simple examples differ from real GNNs in two important ways: (i) they give a single feature vector output, which throws away per-node information, and (ii) they do not use the adjacency matrix. Let's see a real GNN that has these properties while maintaining permutation equivariant.

## 8.4 Kipf & Welling GCN

One of the first popular GNNs was the Kipf & Welling graph convolutional network (GCN) [KW16]. Although some people consider GCNs to be a broad class of GNNs, we'll use GCNs to refer specifically to the Kipf & Welling GCN. Thomas Kipf has written an [excellent article introducing the GCN](#).

The input to a GCN layer is  $\mathbf{V}$ ,  $\mathbf{E}$  and it outputs an updated  $\mathbf{V}'$ . Each node feature vector is updated. The way it updates a node feature vector is by averaging the feature vectors of its neighbors, as determined by  $\mathbf{E}$ . The choice of averaging over neighbors is what makes a GCN layer permutation equivariant. Averaging over neighbors is not trainable, so we must add trainable parameters. We multiply the neighbor features by a trainable matrix before the averaging, which gives the GCN the ability to learn. In Einstein notation, this process is:

$$v_{il} = \sigma \left( \frac{1}{d_i} e_{ij} v_{jk} w_{lk} \right) \quad (8.3)$$

where  $i$  is the node we're considering,  $j$  is the neighbor index,  $k$  is the node input feature,  $l$  is the output node feature,  $d_i$  is the degree of node  $i$  (which makes it an average instead of sum),  $e_{ij}$  isolates neighbors so that all non-neighbor  $v_{jk}$ s are zero,  $\sigma$  is our activation, and  $w_{lk}$  is the trainable weights. This equation is a mouthful, but it truly just is the average over neighbors with a trainable matrix thrown in. One common modification is to make all nodes neighbors of themselves. This is so that the output node features  $v_{il}$  depends on the input features  $v_{ik}$ . We do not need to change our equation, just make the adjacency matrix have 1s on the diagonal instead of 0 by adding the identity matrix during pre-processing.

Building understanding about the GCN is important for understanding other GNNs. You can view the GCN layer as a way to “communicate” between a node and its neighbors. The output for node  $i$  will depend only on its immediate neighbors. For chemistry, this is not satisfactory. You can stack multiple layers though. If you have two layers, the output for node  $i$  will include information about node  $i$ 's neighbors' neighbors. Another important detail to understand in GCNs is that the averaging procedure accomplishes two goals: (i) it gives permutation equivariance by removing the effect of neighbor order and (ii) it prevents a change in magnitude in node features. A sum would accomplish (i) but would cause the magnitude of the node features to grow after each layer. Of course, you could ad-hoc put a batch normalization layer after each GCN layer to keep output magnitudes stable but averaging is easy.

To help understand the GCN layer, look at Fig. 8.2. It shows an intermediate step of the GCN layer. Each node feature is represented here as a one-hot encoded vector at input. The animation in Fig. 8.3 shows the averaging process over neighbor features. To make this animation easy to follow, the trainable weights and activation functions are not considered. Note that the animation repeats for a second layer. Watch how the “information” about there being an oxygen atom in the molecule is propagated only after two layers to each atom. All GNNs operate with similar approaches, so try to understand how this animation works.

### 8.4.1 GCN Implementation

Let's now create a tensor implementation of the GCN. We'll skip the activation and trainable weights for now. We must first compute our rank 2 adjacency matrix. The `smiles2graph` code above computes an adjacency tensor with feature vectors. We can fix that with a simple reduction and add the identity at the same time

```
nodes, adj = smiles2graph("CO")
adj_mat = np.sum(adj, axis=-1) + np.eye(adj.shape[0])
adj_mat
```

```
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]])
```

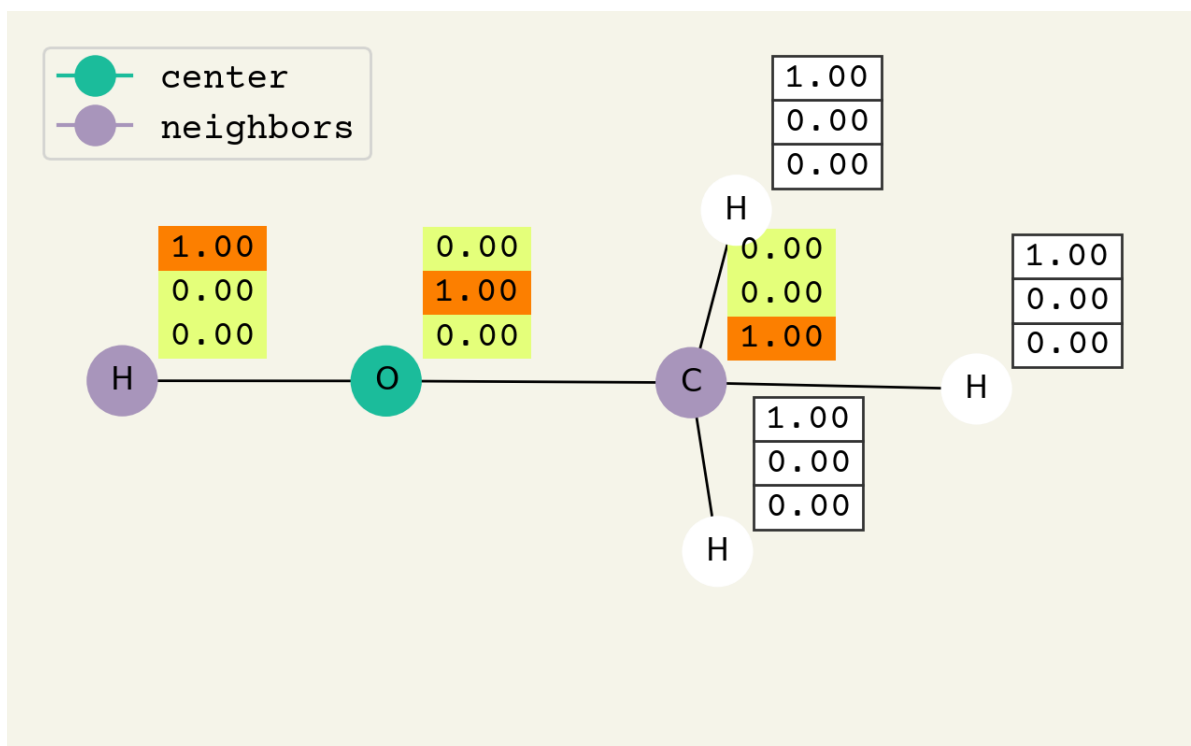


Fig. 8.2: Intermediate step of the graph convolution layer. The 3D vectors are the node features and start as one-hot, so a  $[1.00, 0.00, 0.00]$  means hydrogen. The center node will be updated by averaging its neighbors features.

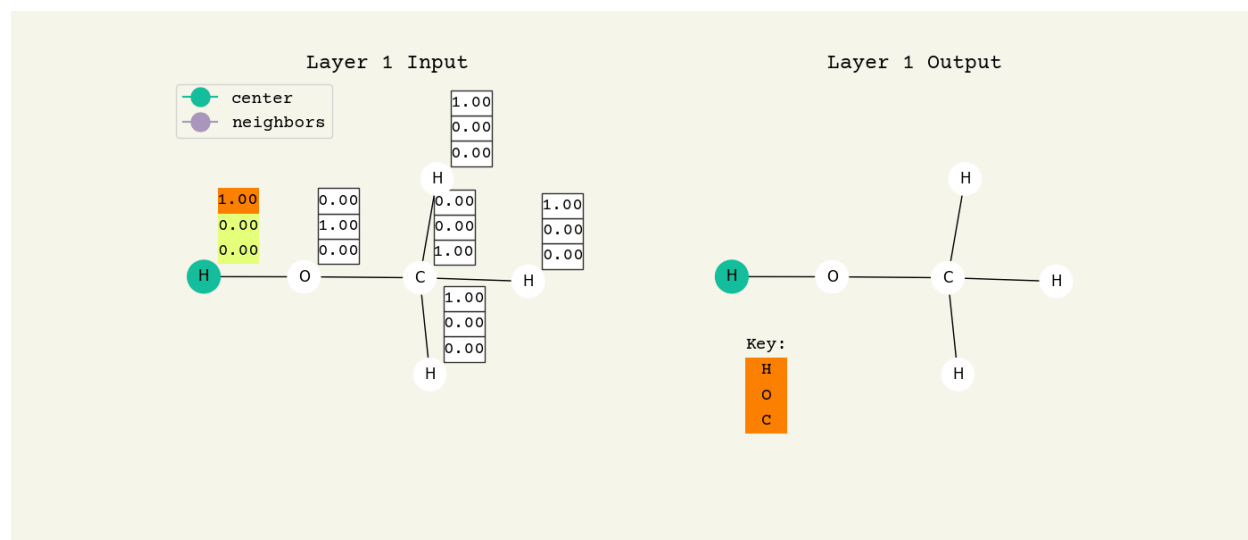


Fig. 8.3: Animation of the graph convolution layer operation. The left is input, right is output node features. Note that two layers are shown (see title change). As the animation plays out, you can see how the information about the atoms propagates through the molecule via the averaging over neighbors. So the oxygen goes from being just an oxygen, to an oxygen bonded to C and H, to an oxygen bonded to an H and CH<sub>3</sub>. The colors just reflect the same information in the numerical values.

To compute degree of each node, we can do another reduction:

```
degree = np.sum(adj_mat, axis=-1)
degree
```

```
array([5., 3., 2., 2., 2., 2.])
```

Now we can put all these pieces together into the Einstein equation

```
print(nodes[0])
# note to divide by degree, make the input 1 / degree
new_nodes = np.einsum("i,ij,jk->ik", 1 / degree, adj_mat, nodes)
print(new_nodes[0])
```

```
[1. 0. 0.]
[0.2 0.2 0.6]
```

To now implement this as a layer in Keras, we must put this code above into a new Layer subclass. The code is relatively straightforward, but you can read-up on the function names and Layer class in [this tutorial](#). The three main changes are that we create trainable parameters `self.w` and use them in the `tf.einsum`, we use an activation `self.activation`, and we output both our new node features and the adjacency matrix. The reason to output the adjacency matrix is so that we can stack multiple GCN layers without having to pass the adjacency matrix each time.

```
class GCNLayer(tf.keras.layers.Layer):
    """Implementation of GCN as layer"""

    def __init__(self, activation=None, **kwargs):
        # constructor, which just calls super constructor
        # and turns requested activation into a callable function
        super(GCNLayer, self).__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        # create trainable weights
        node_shape, adj_shape = input_shape
        self.w = self.add_weight(shape=(node_shape[2], node_shape[2]), name="w")

    def call(self, inputs):
        # split input into nodes, adj
        nodes, adj = inputs
        # compute degree
        degree = tf.reduce_sum(adj, axis=-1)
        # GCN equation
        new_nodes = tf.einsum("bi,bij,bjk,kl->bil", 1 / degree, adj, nodes, self.w)
        out = self.activation(new_nodes)
        return out, adj
```

A lot of the code above is Keras/TF specific and getting the variables to the right place. There are really only two key lines here. The first is to compute the degree by summing over the columns of the adjacency matrix:

```
degree = tf.reduce_sum(adj, axis=-1)
```

The second key line is to do the GCN equation (8.3) (without the activation)

```
new_nodes = tf.einsum("bi,bij,bjk,kl->bil", 1 / degree, adj, nodes, self.w)
```

We can now try our layer:

```
gcnlayer = GCNLayer("relu")
# we insert a batch axis here
gcnlayer((nodes[np.newaxis, ...], adj_mat[np.newaxis, ...]))
```

```
(<tf.Tensor: shape=(1, 6, 3), dtype=float32, numpy=
array([[0.          , 0.22815037, 0.11252856],
       [0.04706494, 0.4801577 , 0.41471443],
       [0.          , 0.22336864, 0.24838853],
       [0.          , 0.22336864, 0.24838853],
       [0.          , 0.22336864, 0.24838853],
       [0.07218027, 0.42193758, 0.20330799]]], dtype=float32)>,
<tf.Tensor: shape=(1, 6, 6), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]], dtype=float32)>)
```

It outputs (1) the new node features and (2) the adjacency matrix. Let's make sure we can stack these and apply the GCN multiple times

```
x = (nodes[np.newaxis, ...], adj_mat[np.newaxis, ...])
for i in range(2):
    x = gcnlayer(x)
print(x)
```

```
(<tf.Tensor: shape=(1, 6, 3), dtype=float32, numpy=
array([[0.05032339, 0.24143484, 0.12720497],
       [0.12059431, 0.36160874, 0.23188105],
       [0.05514492, 0.19730167, 0.10723374],
       [0.05514492, 0.19730167, 0.10723374],
       [0.05514492, 0.19730167, 0.10723374],
       [0.13564095, 0.4374843 , 0.2817377 ]]], dtype=float32)>, <tf.Tensor:
shape=(1, 6, 6), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]], dtype=float32)>)
```

It works! Why do we see zeros though? Probably because we had negative numbers that were removed by our ReLU activation. This will be solved by training and increasing our dimension number.



## 8.5 Solubility Example

We'll now revisit predicting solubility with GCNs. Remember before that we used the features included with the dataset. Now we can use the molecular structures directly. Our GCN layer outputs node-level features. To predict solubility, we need to get a graph-level feature. We'll see later how to be more sophisticated in this process, but for now let's just take the average over all node features after our GCN layers. This is simple, permutation invariant, and gets us from node-level to graph level. Here's an implementation of this

```
class GRLayer(tf.keras.layers.Layer):
    """A GNN layer that computes average over all node features"""

    def __init__(self, name="GRLayer", **kwargs):
        super(GRLayer, self).__init__(name=name, **kwargs)

    def call(self, inputs):
        nodes, adj = inputs
        reduction = tf.reduce_mean(nodes, axis=1)
        return reduction
```

The key line in that code is to just to compute the mean over the nodes (`axis=1`):

```
reduction = tf.reduce_mean(nodes, axis=1)
```

To complete our deep solubility predictor, we can add some dense layers and make sure we have a single-output without activation since we're doing regression. Note this model is defined using the [Keras functional API](#) which is necessary when you have multiple inputs.

```
ninput = tf.keras.Input(
    (
        None,
        100,
    )
)
ainput = tf.keras.Input(
    (
        None,
        None,
    )
)
# GCN block
x = GCNLayer("relu")([ninput, ainput])
x = GCNLayer("relu")(x)
x = GCNLayer("relu")(x)
x = GCNLayer("relu")(x)
# reduce to graph features
x = GRLayer()(x)
# standard layers (the readout)
x = tf.keras.layers.Dense(16, "tanh")(x)
x = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs=(ninput, ainput), outputs=x)
```

where does the 100 come from? Well, this dataset has lots of elements so we cannot use our size 3 one-hot encodings because we'll have more than 3 unique elements. We previously only had C, H and O. This is a good time to update our `smiles2graph` function to deal with this.

```
def gen_smiles2graph(sml):
    """Argument for the RD2NX function should be a valid SMILES sequence
    returns: the graph
    """
    m = rdkit.Chem.MolFromSmiles(sml)
    m = rdkit.Chem.AddHs(m)
    order_string = {
        rdkit.Chem.rdchem.BondType.SINGLE: 1,
        rdkit.Chem.rdchem.BondType.DOUBLE: 2,
        rdkit.Chem.rdchem.BondType.TRIPLE: 3,
        rdkit.Chem.rdchem.BondType.AROMATIC: 4,
    }
    N = len(list(m.GetAtoms()))
    nodes = np.zeros((N, 100))
    for i in m.GetAtoms():
        nodes[i.GetIdx(), i.GetAtomicNum()] = 1

    adj = np.zeros((N, N))
    for j in m.GetBonds():
        u = min(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        v = max(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        order = j.GetBondType()
        if order in order_string:
            order = order_string[order]
        else:
            raise Warning("Ignoring bond order" + order)
        adj[u, v] = 1
        adj[v, u] = 1
    adj += np.eye(N)
    return nodes, adj
```

```
nodes, adj = gen_smiles2graph("CO")
model((nodes[np.newaxis], adj_mat[np.newaxis]))
```

```
<tf.Tensor: shape=(1, 1), dtype=float32, numpy=array([[0.00021663]],
dtype=float32)>
```

We have switched from adjacency tensor to matrix only because a GCN cannot use edge features. Other architectures though can.

It outputs one number! That's always nice to have. Now we need to do some work to get a trainable dataset. Our dataset is a little bit complex because our features are tuples of tensors ( $\mathbf{V}$ ,  $\mathbf{E}$ ) so that our dataset is a tuple of tuples:  $((\mathbf{V}, \mathbf{E}), y)$ . We use a **generator**, which is just a python function that can return multiple times. Our function returns once for every training example. Then we have to pass it to the `from_generator` `tf.data.Dataset` constructor which requires explicit declaration of the shapes of these examples.

```
def example():
    for i in range(len(soldata)):
        graph = gen_smiles2graph(soldata.SMILES[i])
        sol = soldata.Solubility[i]
        yield graph, sol
```

(continues on next page)

(continued from previous page)

```
data = tf.data.Dataset.from_generator(
    example,
    output_types=(tf.float32, tf.float32), tf.float32),
    output_shapes=(
        (tf.TensorShape([None, 100]), tf.TensorShape([None, None])),
        tf.TensorShape([]),
    ),
)
```

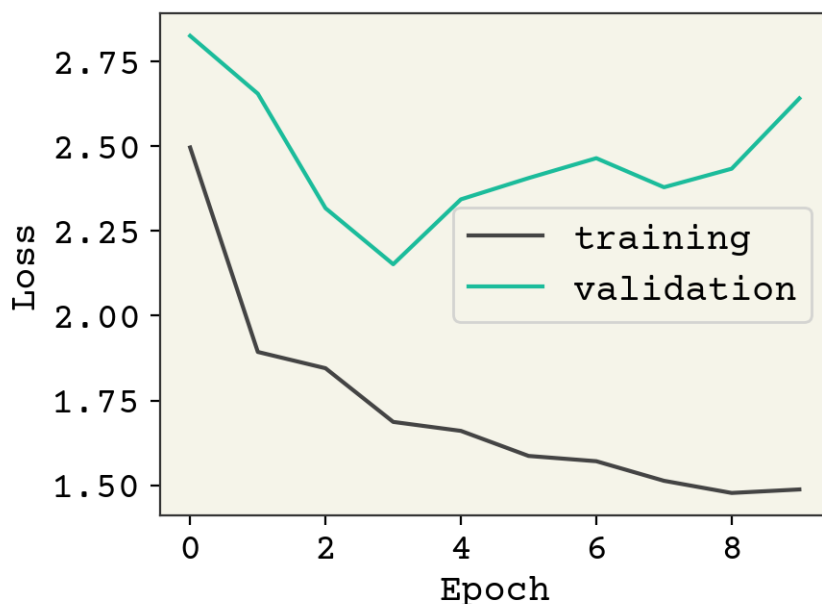
Whew, that's a lot. Now we can do our usual splitting of the dataset.

```
test_data = data.take(200)
val_data = data.skip(200).take(200)
train_data = data.skip(400)
```

And finally, time to train.

```
model.compile("adam", loss="mean_squared_error")
result = model.fit(train_data.batch(1), validation_data=val_data.batch(1), epochs=10)
```

```
plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



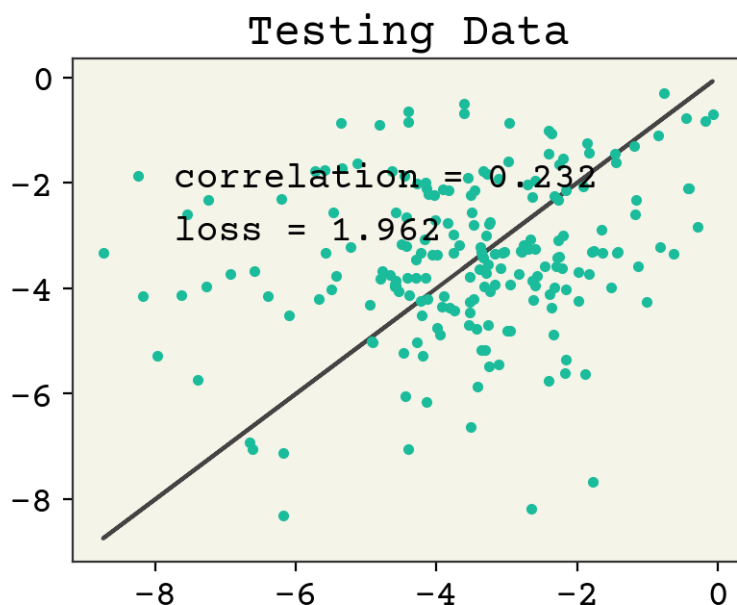
This model is definitely underfit. One reason is that our batch size is 1. This is a side-effect of making the number of atoms variable and then Keras/tensorflow has trouble batching together our data if there are two unknown dimensions. A standard trick is to group together multiple molecules into one graph, but making sure they are disconnected (no bonds between the molecules). That allows you to batch molecules without increasing the rank of your model/data.

Let's now check the parity plot.

```

yhat = model.predict(test_data.batch(1), verbose=0)[: , 0]
test_y = [y for x, y in test_data]
plt.figure()
plt.plot(test_y, test_y, "--")
plt.plot(test_y, yhat, ".")
plt.text(
    min(test_y) + 1,
    max(test_y) - 2,
    f"correlation = {np.corrcoef(test_y, yhat)[0,1]:.3f}",
)
plt.text(
    min(test_y) + 1,
    max(test_y) - 3,
    f"loss = {np.sqrt(np.mean((test_y - yhat)**2)):.3f}",
)
plt.title("Testing Data")
plt.show()

```



## 8.6 Message Passing Viewpoint

One way to more broadly view a GCN layer is that it is a kind of “message-passing” layer. You first compute a message coming from each neighboring node:

$$\vec{e}_{s_i j} = \vec{v}_{s_i j} \mathbf{W} \quad (8.4)$$

where  $v_{s_i j}$  means the  $j$ th neighbor of node  $i$ . The  $s_i$  means senders to  $i$ . This is how a GCN computes the messages, it's just a weight matrix times each neighbor node features. After getting the messages that will go to node  $i$ ,  $\vec{e}_{s_i j}$ , we aggregate them using a function which is permutation invariant to the order of neighbors:

$$\vec{e}_i = \frac{1}{|\vec{e}_{s_i j}|} \sum \vec{e}_{s_i j} \quad (8.5)$$

In the GCN this aggregation is just a mean, but it can be any permutation invariant (possibly trainable) function. Finally, we update our node using the aggregated message in the GCN:

$$\vec{v}'_i = \sigma(\vec{e}_i) \quad (8.6)$$

where  $v'$  indicates the new node features. This is simply the activated aggregated message. Writing it out this way, you can see how it is possible to make small changes. One important paper by Gilmer et al. explored some of these choices and described how this general idea of message passing layers does well in learning to predict molecular energies from quantum mechanics [GSR+17]. Examples of changes to the above GCN equations are to include edge information when computing the neighbor messages or use a dense neural network layer in place of  $\sigma$ . You can think of the GCN as one type of a broader class of message passing graph neural networks, sometimes abbreviated as MPNN.

## 8.7 Gated Graph Neural Network

One common variant of the message passing layer is the **gated graph neural network** (GGN) [LTBZ15]. It replaces the last equation, the node update, with

$$\vec{v}'_i = \text{GRU}(\vec{v}_i, \vec{e}_i) \quad (8.7)$$

where the  $\text{GRU}(\cdot, \cdot)$  is a gated recurrent unit [CGCB14]. A GRU is a binary (two input arguments) neural network that is typically used in sequence modeling. The interesting property of a GGN relative to a GCN is that it has trainable parameters in the node update (from the GRU), giving the model a bit more flexibility. In a GGN, the GRU parameters are kept the same at each layer, like how a GRU is used to model sequences. What's nice about this is that you can stack infinite GGN layers without increasing the number of trainable parameters (assuming you make  $\mathbf{W}$  the same at each layer). Thus GGNs are suited for large graphs, like a large protein or large unit cell.

You'll often see the prefix "gated" on GNNs and that means that the nodes are updated according to a GRU.

## 8.8 Pooling

Within the message passing viewpoint, and in general for GNNs, the way that messages from neighbors are combined is a key step. This is sometimes called **pooling**, since it's similar to the pooling layer used in convolutional neural networks. Just like in pooling for convolutional neural networks, there are multiple reduction operations you can use. Typically you see a sum or mean reduction in GNNs, but you can be quite sophisticated like in the Graph Isomorphism Networks [XHLJ18]. We'll see an example in our attention chapter of using self-attention, which can also be used for pooling. It can be tempting to focus on this step, but it's been empirically found that the choice of pooling is not so important [LDLio19, MSK20]. The key property of the pooling is permutation *invariance* - we want the aggregation operation to not depend on order of nodes (or edges if pooling over them). You can find a recent review of pooling methods in Grattarola et al. [GZBA21].

You can see a more visual comparison and overview of the various pooling strategies in this distill article by Daigavane et al. [DRA21].

## 8.9 Readout Function

GNNs output a graph by design. It is rare that our labels are graphs – typically we have node labels or a single graph label. An example of a node label is partial charge of atoms. An example of a graph label would be the energy of the molecule. The process of converting the graph output from the GNN into our predicted node labels or graph label is called the **readout**. If we have node labels, we can simply discard the edges and use our output node feature vectors from the GNN as the prediction, perhaps with a few dense layers before our predicted output label.

If we're trying to predict a graph-level label like energy of the molecule or net charge, we need to be careful when converting from node/edge features to a graph label. If we simply put the node features into a dense layer to get to the desired shape graph label, we will lose permutation equivariance (technically it's permutation invariance now since our output is graph label, not node labels). The readout we did above in the solubility example was a reduction over the node features to get a graph feature. Then we used this graph feature in dense layers. It turns out this is the only way [ZKR+17] to do a graph feature readout: a reduction over nodes to get graph feature and then dense layers to get predicted graph label from those graph features. You can also do some dense layers on the node features individually, but that already happens in GNN so I do not recommend it. This readout is sometimes called DeepSets because it is the same form as the DeepSets architecture, which is a permutation invariant architecture for features that are sets [ZKR+17].

You may notice that the pooling and readouts both use permutation invariant functions. Thus, DeepSets can be used for pooling and attention could be used for readouts.

### 8.9.1 Intensive vs Extensive

One important consideration of a readout in regression is if your labels are **intensive** or **extensive**. An intensive label is one whose value is independent of the number of nodes (or atoms). For example, the index of refraction or solubility are intensive. The readout for an intensive label should (generally) be independent of the number of nodes/atoms. So the reduction in the readout could be a mean or max, but not a sum. In contrast, an extensive label should (generally) use a sum for the reduction in the readout. An example of an extensive molecular property is enthalpy of formation.

## 8.10 Battaglia General Equations

As you can see, message passing layers is a general way to view GNN layers. Battaglia *et al.* [BHB+18] went further and created a general set of equations which captures nearly all GNNs. They broke the GNN layer equations down into 3 update equations, like the node update equation we saw in the message passing layer equations, and 3 aggregation equations (6 total equations). There is a new concept in these equations: graph feature vectors. Instead of having two parts to your network (GNN then readout), a graph level feature is updated at every GNN layer. The graph feature vector is a set of features which represent the whole graph or molecule. For example, when computing solubility it may have been useful to build up a per-molecule feature vector that is eventually used to compute solubility instead of having the readout. Any kind of per-molecule quantity, like energy, should be predicted with the graph-level feature vector.

The first step in these equations is updating the edge feature vectors, written as  $\vec{e}_k$ , which we haven't seen yet:

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) \quad (8.8)$$

where  $\vec{e}_k$  is the feature vector of edge  $k$ ,  $\vec{v}_{rk}$  is the receiving node feature vector for edge  $k$ ,  $\vec{v}_{sk}$  is the sending node feature vector for edge  $k$ ,  $\vec{u}$  is the graph feature vector, and  $\phi^e$  is one of the three update functions that define the GNN layer. Note that these are meant to be general expressions and you define  $\phi^e$  for your specific GNN layer.

Our molecular graphs are undirected, so how do we decide which node is receiving  $\vec{v}_{rk}$  and which node is sending  $\vec{v}_{sk}$ ? The individual  $\vec{e}'_k$  are aggregated in the next step as all the inputs into node  $v_{rk}$ . In our molecular graph, all bonds are both “inputs” and “outputs” from an atom (how else could it be?), so it makes sense to just view every bond as two directed edges: a C-H bond has an edge from C to H and an edge from H to C. In fact, our adjacency matrices already reflect that. There are two non-zero elements in them for each bond: one for C to H and one for H to C. Back to the original

question, what is  $\vec{v}_{rk}$  and  $\vec{v}_{sk}$ ? We consider every element in the adjacency matrix (every  $k$ ) and when we're on element  $k = \{ij\}$ , which is  $A_{ij}$ , then the receiving node is  $j$  and the sending node is  $i$ . When we consider the companion edge  $A_{ji}$ , the receiving node is  $i$  and the sending node is  $j$ .

$\vec{e}'_k$  is like the message from the GCN. Except it's more general: it can depend on the receiving node and the graph feature vector  $\vec{u}$ . The metaphor of a "message" doesn't quite apply, since a message cannot be affected by the receiver. Anyway, the new edge updates are then aggregated with the first aggregation function:

$$\vec{e}'_i = \rho^{e \rightarrow v} (E'_i) \quad (8.9)$$

where  $\rho^{e \rightarrow v}$  is our defined function and  $E'_i$  represents stacking all  $\vec{e}'_k$  from edges **into** node  $i$ . Having our aggregated edges, we can compute the node update:

$$\vec{v}'_i = \phi^v (\vec{e}'_i, \vec{v}_i, \vec{u}) \quad (8.10)$$

This concludes the usual steps of a GNN layer because we have new nodes and new edges. If you are updating the graph features ( $\vec{u}$ ), the following additional steps may be defined:

$$\vec{e}' = \rho^{e \rightarrow u} (E') \quad (8.11)$$

This equation aggregates all messages/aggregated edges across the whole graph. Then we can aggregate the new nodes across the whole graph:

$$\vec{v}' = \rho^{v \rightarrow u} (V') \quad (8.12)$$

Finally, we can compute the update to the graph feature vector as:

$$\vec{u}' = \phi^u (\vec{e}', \vec{v}', \vec{u}) \quad (8.13)$$

### 8.10.1 Reformulating GCN into Battaglia equations

Let's see how the GCN is presented in this form. We first compute our neighbor messages for all possible neighbors using (8.8). Remember in the GCN, messages only depend on the senders.

Even though we use the "edge update" function, remember in a GCN we ignore the edge features. We only care edges for defining the connectivity of the graph.

$$\vec{e}'_k = \phi^e (\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) = \vec{v}_{sk} \mathbf{W}$$

To aggregate our messages coming into node  $i$  in (8.9), we average them.

$$\vec{e}'_i = \rho^{e \rightarrow v} (E'_i) = \frac{1}{|E'_i|} \sum E'_i$$

Our node update is then the activation (8.10)

$$\vec{v}'_i = \phi^v (\vec{e}'_i, \vec{v}_i, \vec{u}) = \sigma(\vec{e}'_i)$$

we could include the self-loop above using  $\sigma(\vec{e}'_i + \vec{v}_i)$ . The other functions are not used in a GCN, so those three completely define the GCN.

## 8.11 The SchNet Architecture

One of the earliest and most popular GNNs is the SchNet network [SchuttSK+18]. It wasn't really recognized at publication time as a GNN, but it's now recognized as one and you'll see it often used as a baseline model. A **baseline** model is a well-accepted and accurate model that is compared with.

### Baseline Models

A common piece of wisdom is if you want to solve a real problem with deep learning, you should read the most recent popular paper in an area and use the baseline they compare against instead of their proposed model. The reason is that a baseline model usually must be easy, fast, and well-tested, which is generally more important than being the most accurate.

SchNet is for atoms represented as xyz coordinates (points) – not as a molecular graph. All our previous examples used the underlying molecular graph as the input. In SchNet we will convert our xyz coordinates into a graph, so that we can apply a GNN. SchNet was developed for predicting energies and forces from atom configurations without bond information. Thus, we need to first see how a set of atoms and their positions is converted into a graph. To get the nodes, we do a similar process as above and the atomic number is passed through an embedding layer, which is just means we assign a trainable vector to each atomic number (See *Standard Layers* for a review of embeddings).

Getting the adjacency matrix is simple too: we just make every atom be connected to every atom. It might seem confusing what the point of using a GNN is, if we're just connecting everything. *It is because GNNs are permutation equivariant.* If we tried to do learning on the atoms as xyz coordinates, we would have weights depending on the ordering of atoms and probably fail to handle different numbers of atoms.

There is one more missing detail: where do the xyz coordinates go? We make the model depend on xyz coordinates by constructing the edge features from the xyz coordinates. The edge  $\vec{e}$  between atoms  $i$  and  $j$  is computed purely from their distance  $r$ :

$$e_k = \exp\left(-\gamma(r - \mu_k)^2\right) \quad (8.14)$$

where  $\gamma$  is a hyperparameter (e.g.,  $10\text{\AA}$ ) and  $\mu_k$  is an equally-space grid of scalars - like  $[0, 5, 10, 15, 20]$ . The purpose of (8.14) is similar to turning a category feature like atomic number or covalent bond type into a one-hot vector. We cannot do a one-hot vector though, because there is an infinite number of possible distances. Thus, we have a kind of “smoothing” that gives us a pseudo one-hot for distance. Let's see an example to get a sense of it:

```
gamma = 1
mu = np.linspace(0, 10, 5)

def rbf(r):
    return np.exp(-gamma * (r - mu) ** 2)

print("input", 2)
print("output", np.round(rbf(2), 2))
```

```
input 2
output [0.02 0.78 0.    0.    0. ]
```

You can see that a distance of  $r = 2$  gives a vector with most of the activation for the  $k = 1$  position - which corresponds to  $\mu_1 = 2$ .

We have our nodes and edges and are close to defining the GNN update equations. We need a bit more notation. I'm going to use  $h(\vec{x})$  to indicate a multilayer perceptron (MLP) – basically a 1 to 2 dense layers neural network. The exact



number of dense layers and when/where activation is used in these MLPs will be defined in the implementation, because it is not so important for understanding. Recall, the definition of a dense layer is

$$h(\vec{x}) = \sigma(Wx + b)$$

We'll also use a different activation function  $\sigma$  called "shifted softplus" in SchNet:  $\ln(0.5e^x + 0.5)$ . You can see  $\sigma(x)$  compared with the usual ReLU activation in Fig. 8.4. The rationale for using shifted softplus is that it is smooth with respect to its input, so it could be used to compute forces in a molecular dynamics simulation which requires taking smooth derivatives with respect to pairwise distances.

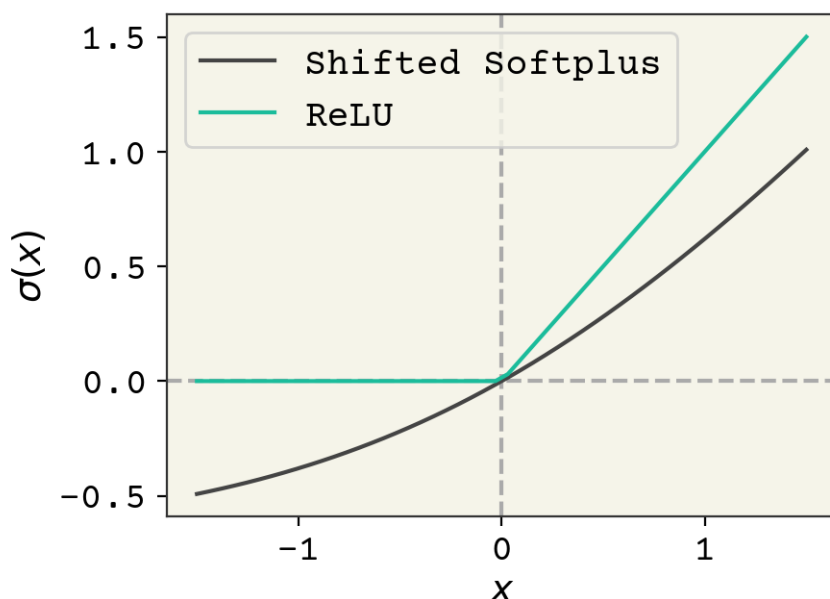


Fig. 8.4: Comparison of the usual ReLU activation function and the shifted softplus used in the SchNet model.

Now, the GNN equations! The edge update equation (8.8) is composed of two pieces. First, we run the incoming edge feature through an MLP and the atoms through an MLP. Then the result is run through an MLP:

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) = h_1(\vec{v}_{sk}) \cdot h_2(\vec{e}_k)$$

The next equation is the edge aggregation equation, (8.9). For SchNet, the edge aggregation is a sum over the neighbor atom features.

$$\vec{e}'_i = \sum E'_i$$

Finally, the node update equation for SchNet is:

$$\vec{v}'_i = \phi^v(\vec{e}'_i, \vec{v}_i, \vec{u}) = \vec{v}_i + h_3(\vec{e}'_i)$$

The GNN updates are applied typically 3-6 times. Although we have an edge update equation, like in GCN we do not actually override the edges and keep them the same at each layer. The original SchNet was for predicting energies and forces, so a readout can be done using sum-pooling or any other strategy described above.

These are sometimes changed, but in the original SchNet paper  $h_1$  is one dense layer without activation,  $h_2$  is two dense layers with activation, and  $h_3$  is 2 dense layers with activation on the first and not the second.

## What is SchNet?

The key GNN feature of a SchNet-like GNN are (1) use edge & node features in the edge update (message construction):

$$\vec{e}'_k = h_1(\vec{v}_{sk}) \cdot h_2(\vec{e}_k)$$

where  $h_i()$ s are some trainable functions and (2) use a residue in the node update:

$$\vec{v}'_i = \vec{v}_i + h_3(\vec{e}'_i)$$

---

All the other details about how to featurize the edges, how deep  $h_i$  is, what activation to choose, how to readout, and how to convert point clouds to graphs are about the specific SchNet model in [SchuttSK+18].

## 8.12 SchNet Example: Predicting Space Groups

Our next example will be a SchNet model that predict space groups of points. Identifying the space group of atoms is an important part of crystal structure identification, and when doing simulations of crystallization. Our SchNet model will take as input points and output the predicted space group. This is a classification problem; specifically it is multi-class because a set of points should only be in one space group. To simplify our plots and analysis, we will work in 2D where there are 17 possible space groups.

Our data for this is a set of points from various point groups. The features are xyz coordinates and the label is the space group. We will not have multiple atom types for this problem. The hidden cell below loads the data and reshapes it for the example.

Let's take a look at a few examples from the dataset

---

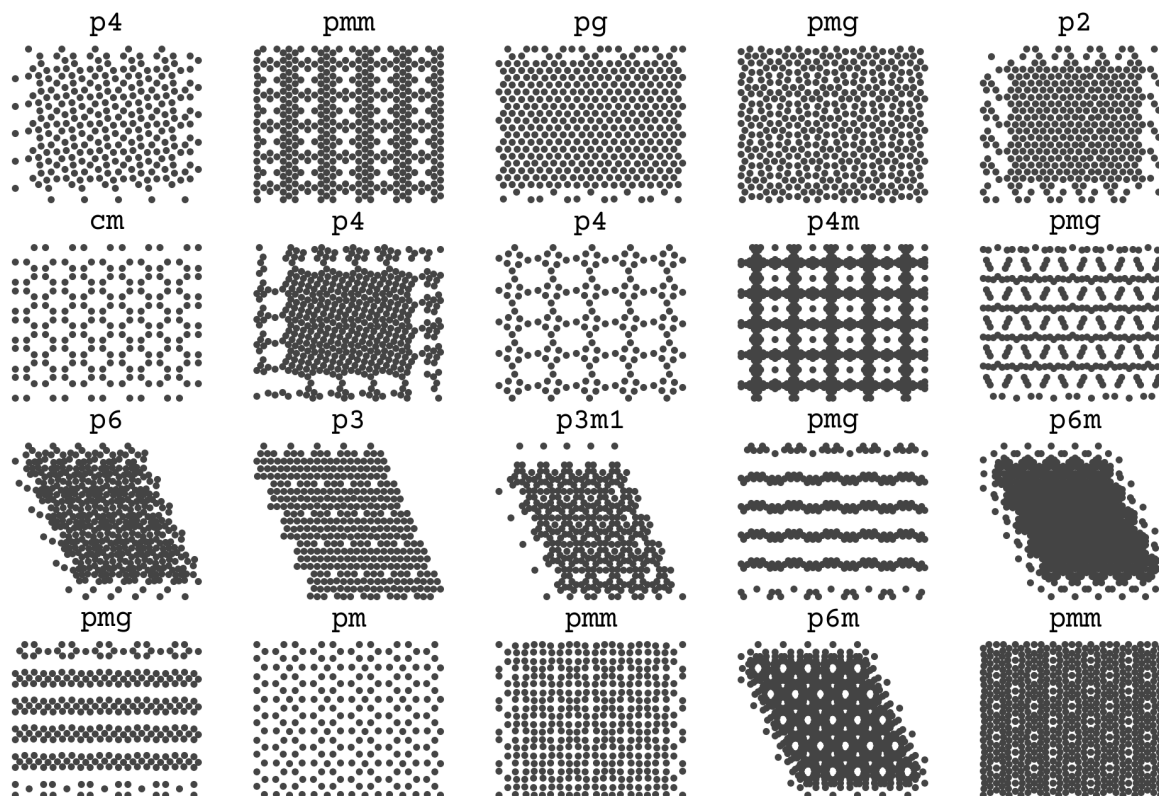
### The Data

This data was generated from [CW22] and all points are constrained to match the space group exactly during a molecular dynamics simulation. The trajectories were NPT with a positive pressure and followed the procedure in that paper for Figure 2. The force field is Lennard-Jones with  $\sigma = 1$  and  $\epsilon = 1$

---

```
fig, axs = plt.subplots(4, 5, figsize=(12, 8))
axs = axs.flatten()

# get a few example and plot them
for i, (x, y) in enumerate(data):
    if i == 20:
        break
    axs[i].plot(x[:, 0], x[:, 1], ".")
    axs[i].set_title(label_str[y.numpy()])
    axs[i].axis("off")
```



You can see that there is a variable number of points and a few examples for each space group. The goal is to infer those titles on the plot from the points alone.

### 8.12.1 Building the graphs

We now need to build the graphs for the points. The nodes are all identical - so they can just be 1s (we'll reserve 0 in case we want to mask or pad at some point in the future). As described in the SchNet section above, the edges should be distance to every other atom. In most implementations of SchNet, we practically add a cut-off on either distance or maximum degree (edges per node). We'll do maximum degree for this work of 16.

I have a function below that is a bit sophisticated. It takes a matrix of point positions in arbitrary dimension and returns the distances and indices to the nearest  $k$  neighbors - exactly what we need. It uses some tricks from *Tensors and Shapes*. However, it is not so important for you to understand this function. Just know it takes in points and gives us the edge features and edge nodes.

```
# this decorator speeds up the function by "compiling" it (tracing it)
# to run efficiently
@tf.function(
    reduce_retracing=True,
)
def get_edges(positions, NN, sorted=True):
    M = tf.shape(input=positions)[0]
    # adjust NN
    NN = tf.minimum(NN, M)
    qexpand = tf.expand_dims(positions, 1) # one column
    qTexpand = tf.expand_dims(positions, 0) # one row
    # repeat it to make matrix of all positions
```

(continues on next page)

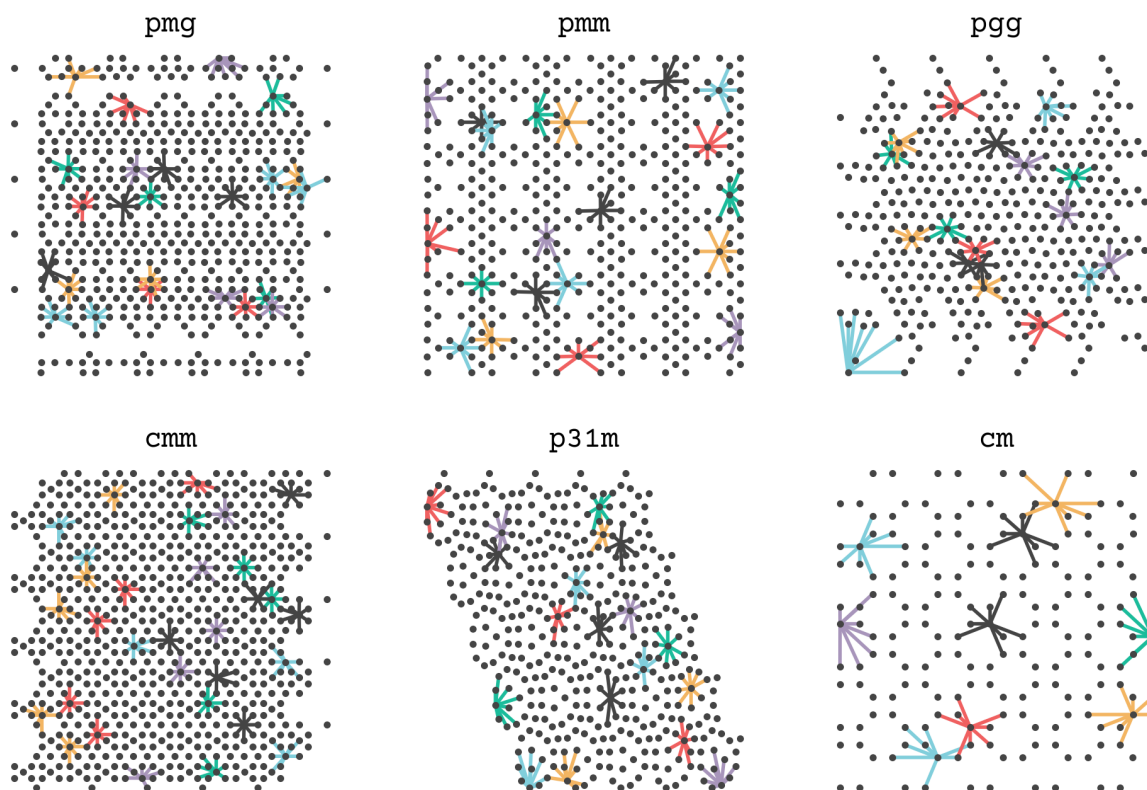
(continued from previous page)

```

qtile = tf.tile(qexpand, [1, M, 1])
qTtile = tf.tile(qTexpand, [M, 1, 1])
# subtract them to get distance matrix
dist_mat = qTtile - qtile
# mask distance matrix to remove zros (self-interactions)
dist = tf.norm(tensor=dist_mat, axis=2)
mask = dist >= 5e-4
mask_cast = tf.cast(mask, dtype=dist.dtype)
# make masked things be really far
dist_mat_r = dist * mask_cast + (1 - mask_cast) * 1000
topk = tf.math.top_k(-dist_mat_r, k=NN, sorted=sorted)
return -topk.values, topk.indices

```

Let's see how this function works by showing the connections between points in one of our examples. I've hidden the code below. It shows some point's neighbors and connects them so you can get a sense of how a set of points is converted into a graph. The complete graph will have all points' neighborhoods.



We will now add this function and the edge featurization of SchNet (8.14) to get the graphs for the GNN steps.

```

MAX_DEGREE = 16
EDGE_FEATURES = 8
MAX_R = 20

gamma = 1
mu = np.linspace(0, MAX_R, EDGE_FEATURES)

```

(continues on next page)

(continued from previous page)

```
def rbf(r):
    return tf.exp(-gamma * (r[... , tf.newaxis] - mu) ** 2)

def make_graph(x, y):
    edge_r, edge_i = get_edges(x, MAX_DEGREE)
    edge_features = rbf(edge_r)
    return (tf.ones(tf.shape(x)[0], dtype=tf.int32), edge_features, edge_i), y[None]

graph_train_data = train_data.map(make_graph)
graph_val_data = val_data.map(make_graph)
graph_test_data = test_data.map(make_graph)
```

Let's examine one graph to see what it looks like. We'll slice out only the first nodes.

```
for (n, e, nn), y in graph_train_data:
    print("first node:", n[1].numpy())
    print("first node, first edge features:", e[1, 1].numpy())
    print("first node, all neighbors", nn[1].numpy())
    print("label", y.numpy())
    break
```

```
first node: 1
first node, first edge features: [2.6852989e-01 5.3621467e-02 8.6928348e-10 1.
↵1441063e-24 0.0000000e+00
0.0000000e+00 0.0000000e+00 0.0000000e+00]
first node, all neighbors [143  8 145  3 10 106 207 202 150 196 204 191 214 149 ↵
↵159 203]
label [15]
```

### 8.12.2 Implementing the MLPs

Now we can implement the SchNet model! Let's start with the  $h_1, h_2, h_3$  MLPs that are used in the GNN update equations. In the SchNet paper these each had different numbers of layers and different decisions about which layers had activation. Let's create them now.

```
def ssp(x):
    # shifted softplus activation
    return tf.math.log(0.5 * tf.math.exp(x) + 0.5)

def make_h1(units):
    return tf.keras.Sequential([tf.keras.layers.Dense(units)])

def make_h2(units):
    return tf.keras.Sequential(
        [
            tf.keras.layers.Dense(units, activation=ssp),
            tf.keras.layers.Dense(units, activation=ssp),
        ]
    )
```

(continues on next page)

(continued from previous page)

```
def make_h3(units):
    return tf.keras.Sequential(
        [tf.keras.layers.Dense(units, activation=ssp), tf.keras.layers.Dense(units)]
    )
```

One detail that can be missed is that the weights in each MLP should change in each layer of SchNet. Thus, we've written the functions above to always return a new MLP. This means that a new set of trainable weights is generated on each call, meaning there is no way we could erroneously have the same weights in multiple layers.

### 8.12.3 Implementing the GNN

Now we have all the pieces to make the GNN. This code will be very similar to the GCN example above, except we now have edge features. One more detail is that our readout will be an MLP as well, following the SchNet paper. The only change we'll make is that we want our output property to be (1) multi-class classification and (2) intensive (independent of number of atoms). So we'll end with an average (intensive) and end with an output vector of logits the size of our labels.

```
class SchNetModel(tf.keras.Model):
    """Implementation of SchNet Model"""

    def __init__(self, gnn_blocks, channels, label_dim, **kwargs):
        super(SchNetModel, self).__init__(**kwargs)
        self.gnn_blocks = gnn_blocks

        # build our layers
        self.embedding = tf.keras.layers.Embedding(2, channels)
        self.h1s = [make_h1(channels) for _ in range(self.gnn_blocks)]
        self.h2s = [make_h2(channels) for _ in range(self.gnn_blocks)]
        self.h3s = [make_h3(channels) for _ in range(self.gnn_blocks)]
        self.readout_l1 = tf.keras.layers.Dense(channels // 2, activation=ssp)
        self.readout_l2 = tf.keras.layers.Dense(label_dim)

    def call(self, inputs):
        nodes, edge_features, edge_i = inputs
        # turn node types as index to features
        nodes = self.embedding(nodes)
        for i in range(self.gnn_blocks):
            # get the node features per edge
            v_sk = tf.gather(nodes, edge_i)
            e_k = self.h1s[i](v_sk) * self.h2s[i](edge_features)
            e_i = tf.reduce_sum(e_k, axis=1)
            nodes += self.h3s[i](e_i)
        # readout now
        nodes = self.readout_l1(nodes)
        nodes = self.readout_l2(nodes)
        return tf.reduce_mean(nodes, axis=0)
```

Remember that the key attributes of a SchNet GNN are the way that we use edge and node features. We can see the mixing of these two in the key line for computing the edge update (computing message values):

```
e_k = self.h1s[i](v_sk) * self.h2s[i](edge_features)
```

followed by aggregation of the edges updates (pooling messages):

```
e_i = tf.reduce_sum(e_k, axis=1)
```

and the node update

```
nodes += self.h3s[i](e_i)
```

Also of note is how we go from node features to multi-classes. We use dense layers that get the shape per-node into the number of classes

```
self.readout_l1 = tf.keras.layers.Dense(channels // 2, activation=ssp)
self.readout_l2 = tf.keras.layers.Dense(label_dim)
```

and then we take the average over all nodes

```
return tf.reduce_mean(nodes, axis=0)
```

Let's give now use the model on some data.

```
small_schnet = SchNetModel(3, 32, len(label_str))
```

```
for x, y in graph_train_data:
    yhat = small_schnet(x)
    break
print(yhat.numpy())
```

```
[ 0.00088388  0.00248148  0.01214652  0.0182933  -0.01964691  0.02274201
  0.02244076  0.01095641 -0.00631071 -0.02765592 -0.01235985  0.00388999
  0.01020311  0.00122206  0.00579039 -0.01482926 -0.00617345]
```

The output is the correct shape and remember it is logits. To get a class prediction that sums to probability 1, we need to use a softmax:

```
print("predicted class", tf.nn.softmax(yhat).numpy())
```

```
predicted class [0.05878644 0.05888043 0.05945227 0.05981884 0.05759181 0.06008555
 0.06006745 0.05938156 0.05836501 0.0571324  0.05801302 0.05896343
 0.05933684 0.05880633 0.05907559 0.05786994 0.05837303]
```

## 8.12.4 Training

Great! It is untrained though. Now we can set-up training. Our loss will be cross-entropy from logits, but we need to be careful on the form. Our labels are integers - which is called “sparse” labels because they are not full one-hot. Multi-class classification is also known as categorical classification. Thus, the loss we want is sparse categorical cross entropy from logits.

```
small_schnet.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
```

(continues on next page)

(continued from previous page)

```

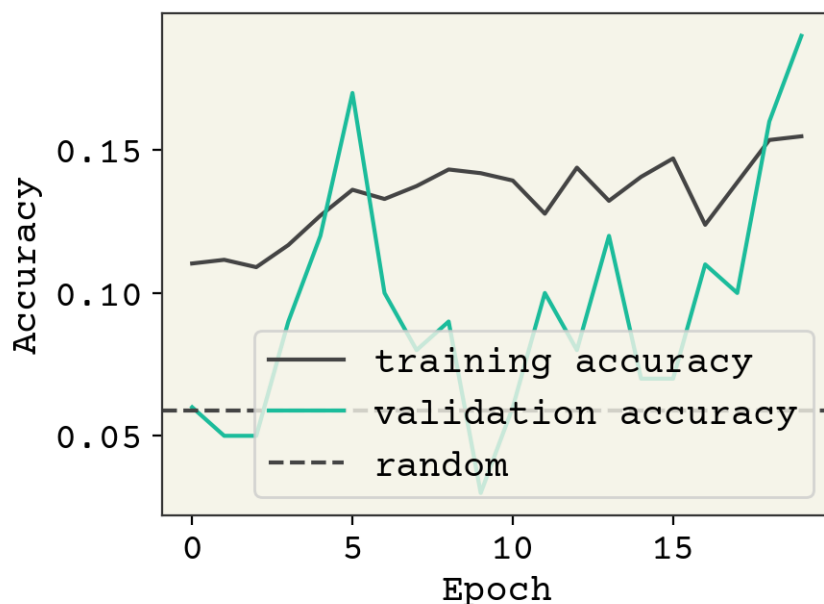
metrics="sparse_categorical_accuracy",
)
result = small_schnet.fit(graph_train_data, validation_data=graph_val_data, epochs=20)

```

```

plt.plot(result.history["sparse_categorical_accuracy"], label="training accuracy")
plt.plot(result.history["val_sparse_categorical_accuracy"], label="validation accuracy")
plt.axhline(y=1 / 17, linestyle="--", label="random")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.show()

```



The accuracy is not great, but it looks like we could keep training. We have a very small SchNet here. Standard SchNet described in [SchuttSK+18] uses 6 layers and 64 channels and 300 edge features. We have 3 layers and 32 channels. Nevertheless, we're able to get some learning. Let's visually see what's going on with the trained model on some test data

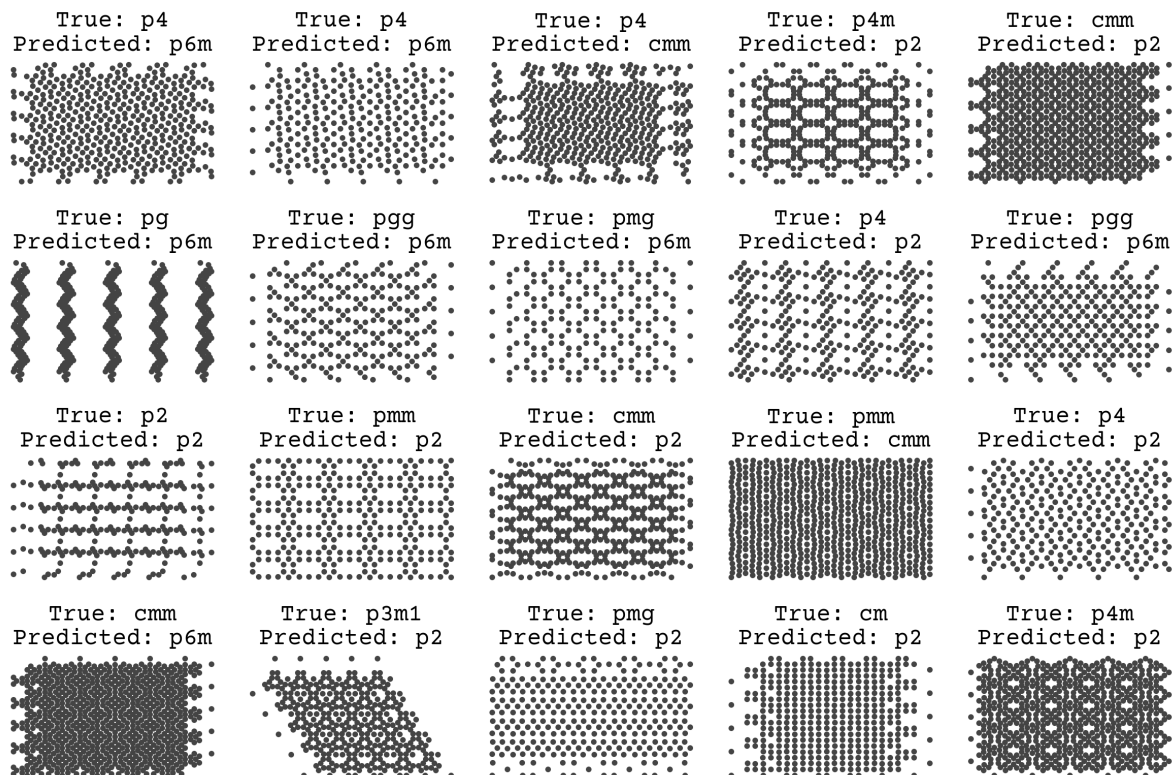
```

fig, axs = plt.subplots(4, 5, figsize=(12, 8))
axs = axs.flatten()

for i, ((x, y), (gx, _)) in enumerate(zip(test_data, graph_test_data)):
    if i == 20:
        break
    axs[i].plot(x[:, 0], x[:, 1], ".")
    yhat = small_schnet(gx)
    yhat_i = tf.math.argmax(tf.nn.softmax(yhat)).numpy()
    axs[i].set_title(f"True: {label_str[y.numpy()]} \n Predicted: {label_str[yhat_i]}")
    axs[i].axis("off")
plt.tight_layout()
plt.show()

```





We'll revisit this example later! One unique fact about this dataset is that it is *synthetic*, meaning there is no label noise. As discussed in [Regression & Model Assessment](#), that removes the possibility of overfitting and leads us to favor high variance models. The goal of teaching a model to predict space groups is to apply it on real simulations or microscopy data, which will certainly have noise. We could have mimicked this by adding noise to the labels in the data and/or by randomly removing atoms to simulate defects. This would better help our model work in a real setting.

## 8.13 Current Research Directions

### 8.13.1 Common Architecture Motifs and Comparisons

We've now seen message passing layer GNNs, GCNs, GGNs, and the generalized Battaglia equations. You'll find common motifs in the architectures, like gating, [Attention Layers](#), and pooling strategies. For example, Gated GNNs (GGNs) can be combined with attention pooling to create Gated Attention GNNs (GAANs)[[ZSX+18](#)]. GraphSAGE is similar to a GCN but it samples when pooling, making the neighbor-updates of fixed dimension[[HYL17](#)]. So you'll see the suffix "sage" when you sample over neighbors while pooling. These can all be represented in the Battaglia equations, but you should be aware of these names.

The enormous variety of architectures has led to work on identifying the "best" or most general GNN architecture [[DJL+20](#), [EPBM19](#), [SMBGunnemann18](#)]. Unfortunately, the question of which GNN architecture is best is as difficult as "what benchmark problems are best?" Thus there are no agreed-upon conclusions on the best architecture. However, those papers are great resources on training, hyperparameters, and reasonable starting guesses and I highly recommend reading them before designing your own GNN. There has been some theoretical work to show that simple architectures, like GCNs, cannot distinguish between certain simple graphs [[XHLJ18](#)]. How much this practically matters depends on your data. Ultimately, there is so much variety in hyperparameters, data equivariances, and training decisions that you should think carefully about how much the GNN architecture matters before exploring it with too much depth.

### 8.13.2 Nodes vs Edges

You'll find that most GNNs use the node-update equation in the Battaglia equations but do not update edges. For example, the GCN will update nodes at each layer but the edges are constant. Some recent work has shown that updating edges can be important for learning when the edges have geometric information, like if the input graph is a molecule and the edges are distance between the atoms [KGrossGunnemann20]. As we'll see in the chapter on equivariances (*Input Data & Equivariances*), one of the key properties of neural networks with point clouds (i.e., Cartesian xyz coordinates) is to have rotation equivariance. [KGrossGunnemann20] showed that you can achieve this if you do edge updates and encode the edge vectors using a rotation equivariant basis set with spherical harmonics and Bessel functions. These kind of edge updating GNNs can be used to predict protein structure [JES+20].

### 8.13.3 Beyond Message Passing

One of the common themes of GNN research is moving “beyond message passing,” where message passing is the message construction, aggregation, and node update with messages. Some view this as impossible – claiming that all GNNs can be recast as message passing [Velivckovic22]. Another direction is on disconnecting the underlying graph being input to the GNN and the graph used to compute updates. We sort of saw this above with SchNet, where we restricted the maximum degree for the message passing. More useful are ideas like “lifting” the graphs into more structured objects like simplicial complexes [BFO+21]. Finally, you can also choose where to send the messages beyond just neighbors [TZK21]. For example, all nodes on a path could communicate messages or all nodes in a clique.

### 8.13.4 Do we need graphs?

It is possible to convert a graph into a string if you're working with an adjacency matrix without continuous values. Molecules specifically can be converted into a string. This means you can use layers for sequences/strings (e.g., recurrent neural networks or 1D convolutions) and avoid the complexities of a graph neural network. SMILES is one way to convert molecular graphs into strings. With SMILES, you cannot predict a per-atom quantity and thus a graph neural network is required for atom/bond labels. However, the choice is less clear for per-molecule properties like toxicity or solubility. There is no consensus about if a graph or string/SMILES representation is better. SMILES can exceed certain graph neural networks in accuracy on some tasks. SMILES is typically better on generative tasks. Graphs obviously beat SMILES in label representations, because they have granularity of bonds/edges. We'll see how to model SMILES in *Deep Learning on Sequences*, but it is an open question of which is better.

## 8.14 Relevant Videos

### 8.14.1 Intro to GNNs

### 8.14.2 Overview of GNN with Molecule, Compiler Examples

## 8.15 Chapter Summary

- Molecules can be represented by graphs by using one-hot encoded feature vectors that show the elemental identity of each node (atom) and an adjacency matrix that show immediate neighbors (bonded atoms).
- Graph neural networks are a category of deep neural networks that have graphs as inputs.
- One of the early GNNs is the Kipf & Welling GCN. The input to the GCN is the node feature vector and the adjacency matrix, and returns the updated node feature vector. The GCN is permutation invariant because it averages over the neighbors.

- A GCN can be viewed as a message-passing layer, in which we have senders and receivers. Messages are computed from neighboring nodes, which when aggregated update that node.
- A gated graph neural network is a variant of the message passing layer, for which the nodes are updated according to a gated recurrent unit function.
- The aggregation of messages is sometimes called pooling, for which there are multiple reduction operations.
- GNNs output a graph. To get a per-atom or per-molecule property, use a readout function. The readout depends on if your property is intensive vs extensive
- The Battaglia equations encompasses almost all GNNs into a set of 6 update and aggregation equations.
- You can convert xyz coordinates into a graph and use a GNN like SchNet

## 8.16 Cited References



## INPUT DATA & EQUIVARIANCES

Molecular graphs and structures (xyz coordinates) are the fundamental features in molecules and materials. As discussed, often these are converted into *molecular descriptors* or some other representation. Why is that? Why can we not work with the data directly? For example, let's say we have a butane molecule and would like to predict its potential energy from its position. You could train a linear model  $\hat{E}$  that predicts energy

$$\hat{E} = \mathbf{X}\mathbf{W} + b \quad (9.1)$$

where  $\mathbf{X}$  is  $14 \text{ (atoms)} \times 3 \text{ (xyz coordinates)}$  matrix containing positions and  $\mathbf{W}$ ,  $b$  are trainable parameters. So far, this is all reasonable. Now what if I translate all the coordinates by -10:

$$(\mathbf{X} - 10) \mathbf{W} + b = \mathbf{X} \mathbf{W} + b - 10|\mathbf{W}| \quad (9.2)$$

We know the energy should not change if we translate all the coordinates equally – the molecule is not changing conformations. However, our linear regression will change by  $-10|\mathbf{W}|$ . We have accidentally made our model sensitive to the origin of our coordinate system, which is not physical. This is **translational variance** – our model changes when we translate the coordinates. I made this word up by the way; it's just easier than saying “non-translational invariant.” We want our models to be insensitive to this — **translationally invariant**.

---

### Audience & Objectives

This chapter builds on *Graph Neural Networks* and a basic knowledge of linear algebra. After completing this chapter, you should be able to

- Define invariance vs equivariance
  - Define translation, rotation, and permutation equivariance
  - Choose features that have desired invariances
- 

Consider another example from our butane molecule. What if we swapped the order of the atoms in our  $\mathbf{X}$  matrix. There is no such thing as a “left” or “right” side of our molecule, so it should not matter. However, you'll see that changing the order of  $\mathbf{X}$  changes which weights are multiplied and thus the predicted energy will change. This is called a **permutation variance**. Our model changes if we re-order our inputs, even though from our knowledge of chemistry this should not matter. Similarly, our output energy should not be sensitive to a rotation of the molecular coordinates. Namely they should be permutation invariant and rotationally invariant.

You could teach your model to learn permutation variance of left/right in this example, either by making your training data contain multiple orderings of  $\mathbf{X}$  or somehow making your  $\mathbf{W}$  be symmetric. You can accomplish this via data augmentation. However, this is inefficient because the number of permutations you want to train the model to ignore is combinatorially large.

## 9.1 Equivariances

Models that work with molecules should be permutation equivariant if they are outputting a per-atom or per-bond quantity. Permutation equivariant means that if you rearrange the order of atoms, the output changes in the same way. For example, if you're predicting partial charge per atom  $\vec{y} = f(\mathbf{X})$  where  $f(\mathbf{X})$  is our model function. If you rearrange  $\mathbf{X}$ , you expect the  $\vec{y}$  to rearrange to match that. Let's try to state this with an equation. Consider  $\mathcal{P}_{34}$  to be the *permutation operator*. It swaps indices 3 and 4 in axis 0 of a tensor. Then a permutation equivariant model equation should have:

$$f(\mathcal{P}_{ij}[\mathbf{X}]) = \mathcal{P}_{ij}[\vec{y}], \quad \forall i, j \quad (9.3)$$

where  $i, j$  are indices of  $\mathbf{X}$  (atom indices). For example, consider  $f(\mathbf{X})$  to predict partial charge given a molecule. If the input is water, our input atoms could be arranged HOH and  $f(\mathbf{X}) = (0.3, -0.6, 0.3)$ . Now if we swap atoms 0 and 1, our input is arranged OHH. If our function is permutation equivariant, then it should output  $f(\mathcal{P}_{01}[\mathbf{X}]) = \mathcal{P}_{01}[\vec{y}] = (-0.6, 0.3, 0.3)$ .

You can find a more general form of permutation equivariance in [TSK+18]. Now what happens if we output a scalar like energy? Then our permutation operator does nothing:

$$f(\mathcal{P}_{ij}[\mathbf{X}]) = \mathcal{P}_{ij}[\hat{E}] = \hat{E} \quad (9.4)$$

we call this case a **permutation invariance**.

The classic way to introduce equivariance is through group theory. Rather than teach group theory here, I'll use simpler equations that do not quite fully capture the ideas and power of equivariances but get the point across quickly.

**Note:** An invariance is special type of equivariance. If something is equivariant, you can easily make it invariant (e.g., averaging over your equivariant axes).

## 9.2 Equivariances of Coordinates

When we work with molecular coordinates as features we need to be a bit more careful in distinguishing between the “features” that might be element identity and those which specify the location in space. This kind of data is referred to as **point clouds** in computer science. Let's break our features into  $(\vec{r}, \vec{x})$  where  $\vec{r}$  is the location of the atom/point and  $\vec{x}$  are its features (e.g., element, charge, spin, etc.). Similarly, we may have labels at each point so that we write our labels as  $(\vec{r}', \vec{y})$ . The label might be direction  $\vec{r}'$  and force magnitude  $y$  or perhaps a field at  $\vec{r}'$  with vectors  $\vec{y}$ . You may not have  $\vec{r}'$  like if you're predicting energy. It may be that  $\vec{r} = \vec{r}'$ . With this notation, we can write out **translation equivariance** as:

$$f(\vec{r} + \vec{t}, \vec{x}) = (\vec{r}' + \vec{t}, \vec{y}), \quad \forall \vec{t} \quad (9.5)$$

You can also write this out if you have a matrix of atom positions (like a molecule):

$$f(\mathbf{R} + \vec{t}, \mathbf{X}) = (\mathbf{R}' + \vec{t}, \vec{y}_i), \quad \forall \vec{t} \quad (9.6)$$

where  $\mathbf{R}$  is the matrix of all position vectors  $\text{vec}\{\mathbf{r}\}_i$ . In the case that you do not have output coordinates  $\vec{r}'$ , then it becomes:

$$f(\mathbf{R} + \vec{t}, \mathbf{X}) = \vec{y}_i, \quad \forall \vec{t} \quad (9.7)$$

which we call **translation invariance**. It's important to note that these equations do not apply to some specific  $\vec{t}$ , but any  $\vec{t}$ .

**Rotational equivariance** can be similarly defined. Consider  $\mathcal{R}$  to be a rotation operator (e.g., a quaternion). Then our rotation equivariance equation is

$$f(\mathcal{R}[\mathbf{R}], \mathbf{X}) = (\mathcal{R}[\mathbf{R}'], \vec{y}_i), \quad \forall \mathcal{R} \quad (9.8)$$

an example might be again that  $(\mathbf{R}', \vec{y}_i)$  defines some field and our equivariance says that if we rotate our input points, our output points will obey the same rotation. Again, we can also have **rotation invariance** if our model does not output  $\mathbf{R}'$ .

## 9.3 Constructing Equivariant Models

There has been some work to unify equivariances into a single “layer” type so that you can just pick what equivariances you want like you would a hyperparameter [TSK+18, WGW+18]. The chapter *Equivariant Neural Networks* covers these and a recent review may be found in [Est20]. A popular implementation is [available here](#). A recent application in molecules was in predicting dipole moments, a great example of where rotation invariance would fail because dipole moments should rotate the same way when the molecule rotates [MGSNoe20].

If you do not need full equivariances, you can often use a data transformation to make the model invariant regardless of its architecture. Some of the common transforms are summarized in the table below and you can find a much more detailed discussion of data transformations in Musil et al. [MGB+21]. The list below omits [data augmentation](#) where you try to teach your model these invariances through training — which is a common and effective strategy in image deep learning. Alternatively, if your invariants are finite (e.g., only 90 degree rotations or you have small permutation invariant sets) you can just apply each possible transformation (rotation/permutation) and average the results to make a quick and easy invariant network [ZKR+17]. That is sometimes called **test augmentation**.

Data Transformation	Equivariance
Matrix Determinant	Permutation Invariance
Eigendecomposition	Permutation Invariance
Reduction (sum, mean)	Permutation Invariance
Pairwise Vector/Distance	Translation/Rotation Invariance
Angles	Translation/Rotation Invariance
Atom-centered Symmetry Functions	Rotation/Translation Invariance
Trajectory Alignment	Rotation/Translation Invariance
Molecular Descriptors	All invariant

### 9.3.1 Matrix Determinant

A matrix determinant is not quite permutation invariant. If you swap two rows in a matrix, it makes the determinant change signs. However, you can easily just square the determinant to remove the sign change. How would you use a determinant in a model? It’s just a building block. You could build a matrix of all neighbor features in a molecular graph and then do a determinant to arrive at a permutation invariant output. The determinant has two major disadvantages: (i) it outputs a single number (not that expressive) and (ii) it’s really expensive  $O(n^3)$ . Thus you won’t see a determinant too frequently in deep learning.

### 9.3.2 Eigendecomposition

The eigendecomposition is when you factorize a matrix into its eigenvalues and eigenvectors. The (sorted) eigenvalues of a matrix are permutation invariant. Compared with the determinant, you get more eigenvalues from a matrix than determinants so there is less loss of information. Computing eigenvalues is still expensive at  $O(n^3)$  and they are not differentiable. Nevertheless, you will see this strategy in kernel learning where you do not need to propagate derivatives through the kernel. One important application of this is in some of the early work on quantum machine learning [RTMullerVL12].

### 9.3.3 Reductions

An obvious way to remove permutation variance is to just sum over the points or atoms. More generally, you can use any kind of reduction like a mean or product. Like a determinant, this results in a single number or at least removal of an axis. Reductions are fast and differentiable.

### 9.3.4 Pairwise Distance

Using pairwise distances or vectors is the standard solution to translation invariance. Rarely are we actually that concerned with translational equivariance. If using pairwise vectors between atoms instead of the xyz coordinates, this naturally removes the choice of origin and thus makes the model translation invariant. If we go further and use pairwise distance instead of vectors, this also removes the effect of rotations giving a rotation invariance. This is fast, differentiable, and the usual approach to add translation and rotation invariance.

### 9.3.5 Angles

Angles are rotation, translation, and scale invariant. You can define angles any way, but often they are done between consecutive bonds (3 atoms total). Combining angles and pairwise distances (along bonds only) are called **internal coordinates**.

### 9.3.6 Convolutional Layers

Convolutional layers are well-known to be translationally equivariant. Remember that they work in 3D as well, so they can be an input layer if translational equivariance is desired. However, convolutions work on pixels or voxels in 3D, so you must first bin your coordinates into a voxel grid. See Chew et al. for an example [CJZ+20].

### 9.3.7 Atom-centered Symmetry Functions

These are a large class of functions described by Behler[Beh11] that transform from the input coordinate/features  $(\mathbf{R}, \mathbf{X})$  to a new set of features  $\mathbf{X}'$  that obey rotational and translational symmetry. This makes them translational and rotationally invariant. Behler didn't propose a single function to get these features, but instead explored the choices and theory. Bartók et al. [BartokKCsanyi13] provided a specific recipe which is called a SOAP descriptor. These are drop-in replacements for  $(\mathbf{R}, \mathbf{X})$  that are translation and rotation invariant but do not lose much information. They are differentiable, although complex to implement.



### 9.3.8 Trajectory Alignment

One special case is when your points are part of a time-dependent trajectory. This usually implies that the order of the points does not change. Thus, we do not need to worry about permutation invariance. This is the case when analyzing results from a molecular dynamics trajectory, like a dynamic simulation of a protein. One way to make our data translation and rotation invariant in this case is to align it to some reference coordinates. For example, you could always make the center of mass of the trajectory be at the origin. This will make the points translation invariant, because you always center a new set of points. A natural question is if it must be the center of mass? No, because your points are permutation invariant, you could just pick point 0 as the origin. Then if the points are translated, this will be undone when you move the points such that point 0 is the origin. To be more precise, we always apply a centering function  $c(\mathbf{R})$  that redefines the origin before processing a set of points. We can also define a rotation function  $r(\mathbf{R})$  that will be applied where we align to some definite set of three Euler angles (two vectors). See the example below for this.

### 9.3.9 Molecular Descriptors

Molecular descriptors are the classic way to convert molecules into translation/rotation/permutation invariant features. There exists 3D descriptors as well that can treat structure. They are also called **fingerprints**. Fingerprints is a broad term for converting a molecular structure to a binary sequence. Commonly, each bit indicates the presence or absence of a specific substructure. We won't focus on these in this course because they are untrainable and choosing the correct combination of descriptors is an unsolved problem which has no clear process.

## 9.4 Examples

Let's demonstrate with some code how to go about creating functions that obey these equivariances. We won't be training these models because training has no effect on equivariances, but you should train your models if you're doing learning [?]

There are ways to make training *enforce* equivariances[RSPoczros17], but it's a somewhat complex and rarely used strategy. This is different than data augmentation, where we hope it learns these.

I'll define my butane molecule as a set of coordinates  $\mathbf{R}_i$  and features  $\mathbf{X}_i$ . My features are just one-hot vectors indicating if a particular point is a carbon atom  $[1, 0]$  or a hydrogen atom  $[0, 1]$ . In our example, we will just be interested in predicting energy. We will not train our models, so the energy will not be accurate.

## 9.5 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

### 9.5.1 No equivariances

A one-hidden layer dense neural network is an example of a model with no equivariances. To fit our data into this dense layer, we'll just stack the positions and features into a large input tensor and output energy. We'll use a tanh as activation, 16 hidden layer dimension, and our output layer has no activation because we're doing regression to energy. Our weights will always be randomly initialized.

```
# our 1-hidden layer model
def hidden_model(r, x, w1, w2, b1, b2):
    # stack into one large input
    i = np.concatenate((r, x), axis=1).flatten()
    v = np.tanh(i @ w1 + b1)
    v = v @ w2 + b2
    return v

# initialize our weights
w1 = np.random.normal(size=(N * 5, 16)) # 5 -> 3 xyz + 2 features
b1 = np.random.normal(size=(16,))
w2 = np.random.normal(size=(16,))
b2 = np.random.normal()
```

Let's see what the predicted energy is with our coordinates.

```
hidden_model(R_i, X_i, w1, w2, b1, b2)
```

```
2.2814890918459536
```

This is not trained, so we aren't that concerned about the value. Now let's see if our model is sensitive to translation.

```
hidden_model(R_i + np.array([1, 2, 3]), X_i, w1, w2, b1, b2)
```

```
-2.79306064480369
```

As expected, it is sensitive to translation. I added the vector (1, 2, 3) to all input points and the energy changed. This model is not translation invariant. The choice of (1, 2, 3) is arbitrary, the model should not change output regardless of the choice of the translation vector if the model is translation invariant. Rotations can be done using the `scipy` transformation library which takes care of some of the messiness of working with quaternions, which are the operators that perform rotations.

```
import scipy.spatial.transform as trans

# rotate around x coord by 45 degrees
rot = trans.Rotation.from_euler("x", 45, degrees=True)

print("No rotation", hidden_model(R_i, X_i, w1, w2, b1, b2))
print("Rotated", hidden_model(rot.apply(R_i), X_i, w1, w2, b1, b2))
```

```
No rotation 2.2814890918459536
Rotated -0.7302631811530622
```

Our model is affected by the rotation, meaning it is not rotation invariant. Permutation invariance comes from swapping indices.

```
# swap 0, 1 rows
perm_R_i = np.copy(R_i)
perm_R_i[0], perm_R_i[1] = R_i[1], R_i[0]
# we do not need to swap X_i 0,1 because they are identical

print("original", hidden_model(R_i, X_i, w1, w2, b1, b2))
print("permuted", hidden_model(perm_R_i, X_i, w1, w2, b1, b2))
```

```
original 2.2814890918459536
permuted 0.0609968150217437
```

Our model is not permutation invariant!

## 9.5.2 Permutation Invariant

We will use a reduction to achieve permutation invariance. All that is needed is to ensure that weights are not a function of our atom number axis and then do a reduction (sum) prior to the output layer. Here is the implementation.

```
# our 1-hidden layer model with perm inv
def hidden_model_pi(r, x, w1, w2, b1, b2):
    # stack into one large input
    i = np.concatenate((r, x), axis=1)
    v = np.tanh(i @ w1 + b1)
    # reduction
    v = np.sum(v, axis=0)
    v = v @ w2 + b2
    return v

# initialize our weights
w1 = np.random.normal(size=(5, 16)) # note it no longer has N!
b1 = np.random.normal(size=(16,))
w2 = np.random.normal(size=(16,))
b2 = np.random.normal()
```

We made three changes: we kept the atom axis (no more flatten), we removed the atom axis after the hidden layer (sum), and we made our weights not depend on the atom axis. Now let's observe if this is indeed permutation invariant.

```
print("original", hidden_model_pi(R_i, X_i, w1, w2, b1, b2))
print("permuted", hidden_model_pi(perm_R_i, X_i, w1, w2, b1, b2))
```

```
original -3.61539748295704
permuted -3.61539748295704
```

It is!

### 9.5.3 Translation Invariant

The next change we will make is to convert our  $N \times 3$  shaped coordinates into  $N \times N \times 3$  pairwise vectors. This gives us translation invariance. This causes an issue because our distance features went from being 3 per atom to  $N \times 3$  per atom. Thus we've introduced a dependence on atom number in our distance features and that means it's easy to accidentally break our permutation invariance. We can just sum over this new axis though. Let's see an implementation:

```
# our 1-hidden layer model with perm inv, trans inv
def hidden_model_pti(r, x, w1, w2, b1, b2):
    # compute pairwise distances using broadcasting
    d = r - r[:, np.newaxis]
    # stack into one large input of N x N x 5
    # concatenate doesn't broadcast, so I manually broadcast the Nx2 x matrix
    # into N x N x 2
    i = np.concatenate((d, np.broadcast_to(x, (d.shape[:-1] + x.shape[-1:])), axis=-
-1)
    v = np.tanh(i @ w1 + b1)
    # reduction over both axes
    v = np.sum(v, axis=(0, 1))
    v = v @ w2 + b2
    return v
```

```
print("original", hidden_model_pti(R_i, X_i, w1, w2, b1, b2))
print("permuted", hidden_model_pti(perm_R_i, X_i, w1, w2, b1, b2))
print("translated", hidden_model_pti(R_i + np.array([-2, 3, 4]), X_i, w1, w2, b1, b2))
print("rotated", hidden_model_pti(rot.apply(R_i), X_i, w1, w2, b1, b2))
```

```
original -30.794047959277147
permuted -30.794047959277126
translated -30.79404795927714
rotated -28.389440679751605
```

It is now translation and permutation invariant. But not yet rotation invariant.

### 9.5.4 Rotation Invariant

It is a simple change to make it rotationally invariant. We just convert the pairwise vectors into pairwise distances. We'll use squared distances for simplicity.

```
# our 1-hidden layer model with perm, trans, rot inv.
def hidden_model_ptri(r, x, w1, w2, b1, b2):
    # compute pairwise distances using broadcasting
    d = r - r[:, np.newaxis]
    # x^2 + y^2 + z^2 of pairwise vectors
    # keepdims so we get an N x N x 1 output
    d2 = np.sum(d**2, axis=-1, keepdims=True)
    # stack into one large input of N x N x 3
    # concatenate doesn't broadcast, so I manually broadcast the Nx2 x matrix
    # into N x N x 2
    i = np.concatenate(
        (d2, np.broadcast_to(x, (d2.shape[:-1] + x.shape[-1:])), axis=-1
    )
    v = np.tanh(i @ w1 + b1)
    # reduction over both axes
```

(continues on next page)

(continued from previous page)

```

v = np.sum(v, axis=(0, 1))
v = v @ w2 + b2
return v

# initialize our weights
w1 = np.random.normal(size=(3, 16)) # now just 1 dist feature
b1 = np.random.normal(size=(16,))
w2 = np.random.normal(size=(16,))
b2 = np.random.normal()

# test it

print("original", hidden_model_ptr_i(R_i, X_i, w1, w2, b1, b2))
print("permuted", hidden_model_ptr_i(perm_R_i, X_i, w1, w2, b1, b2))
print("translated", hidden_model_ptr_i(R_i + np.array([-2, 3, 4]), X_i, w1, w2, b1,
-b2))
print("rotated", hidden_model_ptr_i(rot.apply(R_i), X_i, w1, w2, b1, b2))

```

```

original 358.2899463342588
permuted 358.28994633425873
translated 358.2899463342588
rotated 358.2899463342588

```

We have achieved our invariances! Remember that you could use different choices to achieve these invariances. Also, you may not want an invariance sometimes. You may want equivariances or are not concerned at all. For example, if you're always working with one molecule you may never need to switch around atom orders.

Finally as a sanity check, let's make sure that if we change the coordinates our predicted energy changes.

```

R_i[0] = 2.0
print("changed", hidden_model_ptr_i(R_i, X_i, w1, w2, b1, b2))

```

```

changed 380.9895875398029

```

Our model is still sensitive to the input features

## 9.6 Trajectory Alignment Example

As described above, if you're working with a trajectory there is no requirement to have permutation equivariance. To achieve translation and rotation invariance, we can align to some fixed points that will be present in all structures (like center of mass). As we'll see below, trajectory alignment is fraught with issues like rotation ambiguities, unphysical rotations, and creating fictitious covariances between far away points due to alignment. Internal coordinates (pairwise dist, angles) are almost always better. However, trajectory alignment has good scaling properties.

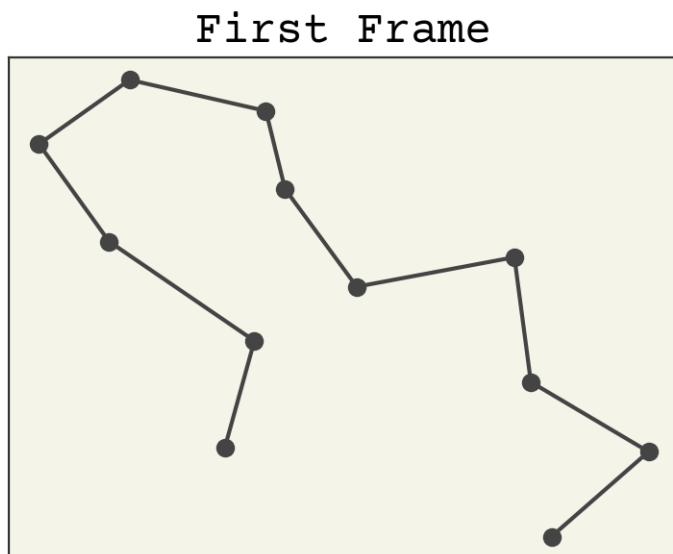
The movie below shows an example trajectory that we'll examine. I've made it 2D to make it simple to visualize, but the same principles apply to 3D.

Let's start by loading the trajectory and defining/testing our centering function. The trajectory is a tensor that is shape time, point number, xy -> (2048, 12, 2). I'll examine two centering functions: align to center of mass and align to point 0 to see what the two look like.

```

url = urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/master/data/paths.npz", "paths.npz"
)
paths = np.load("paths.npz")["arr"]
# plot the first point
plt.title("First Frame")
plt.plot(paths[0, :, 0], paths[0, :, 1], "o-")
plt.xticks([])
plt.yticks([])
plt.show()

```



```

def center_com(paths):
    """Align paths to COM at each frame"""
    coms = np.mean(paths, axis=-2, keepdims=True)
    return paths - coms

def center_point(paths):
    """Align paths to particle 0"""
    return paths - paths[:, :1]

ccpaths = center_com(paths)
cppaths = center_point(paths)

```

To compare, we'll draw a sample of frames on top of one another to see the now translationally invariant coordinates.

```

fig, axs = plt.subplots(ncols=3, squeeze=True, figsize=(16, 4))

axs[0].set_title("No Center")
axs[1].set_title("COM Center")
axs[2].set_title("Point 0 Center")
cmap = plt.get_cmap("cool")
for i in range(0, 2048, 16):
    axs[0].plot(paths[i, :, 0], paths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048))

```

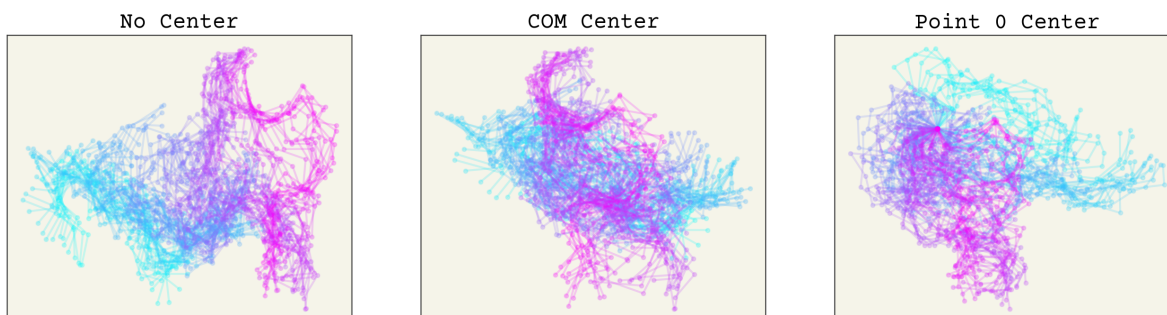
(continues on next page)

(continued from previous page)

```

    axs[1].plot(
        ccpaths[i, :, 0], ccpaths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048)
    )
    axs[2].plot(
        cppaths[i, :, 0], cppaths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048)
    )
for i in range(3):
    axs[i].set_xticks([])
    axs[i].set_yticks([])

```



The color indicates time. You can see that having no alignment makes the spatial coordinates depend on time implicitly because the points drift over time. Both aligning to COM or point 0 removes this effect. The COM implicitly removes 2 degrees of freedom. Point 0 alignment makes point 0 have no variance (also remove degrees of freedom), which could affect how you design or interpret your model.

Now we will align by rotation. We need to define a unique rotation. A simple way is to choose 1 (or 2 in 3D) vectors that define our coordinate system directions. For example, we could choose that the vector from point 0 to point 1 defines the positive direction of the x-axis. A more sophisticated way is to find the principal axes of our points and align along these. For 2D, we only need to align to one of them. Again, this implicitly removes a degree of freedom. We will examine both. Computing principle axes requires an eigenvalue decomposition, so it's a bit more numerically intense [FR00].

**Warning:** For all rotation alignment methods, you must have already centered the points.

```

def make_2drot(angle):
    mats = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(angle)]])
    # swap so batch axis is first
    return np.swapaxes(mats, 0, -1)

def align_point(paths):
    """Align to 0-1 vector assuming 2D data"""
    vecs = paths[:, 0, :] - paths[:, 1, :]
    # find angle to rotate so these are pointed towards pos x
    cur_angle = np.arctan2(vecs[:, 1], vecs[:, 0])
    rot_angle = -cur_angle
    rot_mat = make_2drot(rot_angle)
    # to mat mult at each frame
    return paths @ rot_mat

def find_principle_axis(points, naxis=2):
    """Compute single principle axis for points"""

```

(continues on next page)

(continued from previous page)

```

inertia = points.T @ points
evals, evecs = np.linalg.eig(inertia)
order = np.argsort(evals)[::-1]
# return largest vectors
return evecs[:, order[:naxis]].T

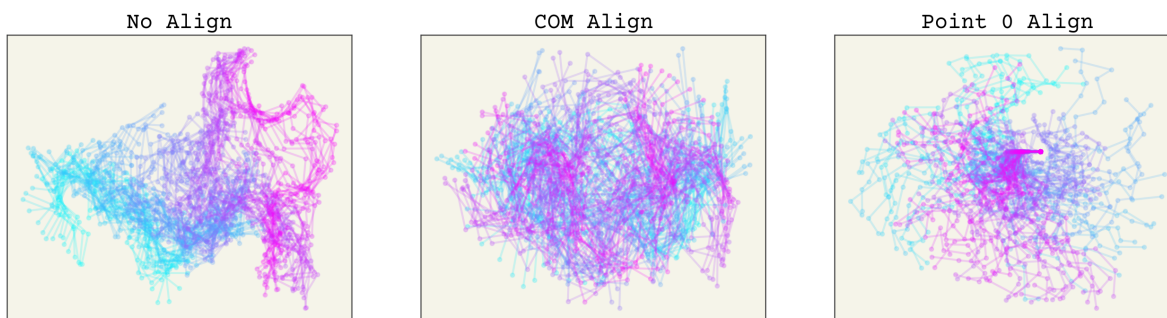
def align_principle(paths, axis_finder=find_principle_axis):
    # someone should tell me how to vectorize this in numpy
    vecs = [axis_finder(p) for p in paths]
    vecs = np.array(vecs)
    # find angle to rotate so these are pointed towards pos x
    cur_angle = np.arctan2(vecs[:, 0, 1], vecs[:, 0, 0])
    cross = np.cross(vecs[:, 0], vecs[:, 1])
    rot_angle = -cur_angle - (cross < 0) * np.pi
    rot_mat = make_2drot(rot_angle)
    # rotate at each frame
    rpaths = paths @ rot_mat
    return rpaths

appaths = align_point(cppaths)
apapaths = align_principle(ccpaths)

fig, axs = plt.subplots(ncols=3, squeeze=True, figsize=(16, 4))

axs[0].set_title("No Align")
axs[1].set_title("COM Align")
axs[2].set_title("Point 0 Align")
for i in range(0, 2048, 16):
    axs[0].plot(paths[i, :, 0], paths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048))
    axs[1].plot(
        apapaths[i, :, 0], apapaths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048)
    )
    axs[2].plot(
        appaths[i, :, 0], appaths[i, :, 1], "-.", alpha=0.2, color=cmap(i / 2048)
    )
for i in range(3):
    axs[i].set_xticks([])
    axs[i].set_yticks([])

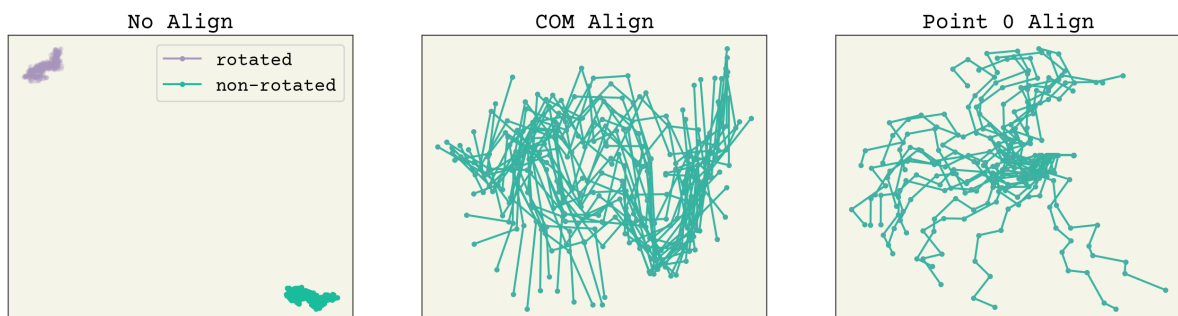
```



You can see how points far away on the chain from 0 have much more variance in the point 0 align, whereas the COM alignment looks better spread. Remember, to apply these methods you must do them to your both your training data and any prediction points. Thus, they should be viewed as part of your neural network. We can now check that rotating has



no effect on these. The plots below have the trajectory rotated by 1 radian and you can see that both alignment methods have no change (the lines are overlapping).



Now which method is better? Aligning based on arbitrary points is indeed easier, but it creates an unusual new variance in your features. For example, let's see what happens if we make a small perturbation to one conformation. The code is hidden for simplicity. We try changing point 1, then point 0, then point 11 to see the effects of perturbations along the chain.



As you can see, perturbing one point alters all others after alignment. This makes these transformed features sensitive to noise, especially aligning to point 0 or 1. More importantly, this effect is uneven in the alignment to point 0. This can in-turn make training quite difficult. Of course, neural networks are universal approximators so in theory this should not

matter. However, I expect that using the COM alignment approach will give better training because the network will not need to account for this unusual variance structure.

The alignment changes due to small changes in input points is described by the Jacobian of our transform which measures how changes to one input dimension affects all output dimension.

The final analysis shows a video of the two frames. One thing you'll note are the jumps, when the principle axes swap direction. You can see that the ambiguity caused by these can create artifacts.

Some people refer to principle axes finding as a kind of PCA. It is. But when we discuss PCA, we mean identifying the sources of variances across the trajectory, not across points in a single frame. You could do PCA in a single frame and use that to define your trans/rot invariant frame. It's mathematically equivalent to principle axes finding.

### 9.6.1 Using Unsupervised Methods for Alignment

There are additional methods for aligning trajectories. You could define one frame as the “reference” and find the translation and rotations that best align with that reference. This could give some interpretability to your rotation and translation alignment. A tempting option is to use dimensionality reduction (like PCA), but these are not rotation invariant. This is confusing at first, because remember PCA should remove translation and rotation. It removes it though from the training data and not an arbitrary frame because it examines motion along a trajectory. You can easily see this getting principle components and then trying to align a new frame to them and a rotated version of the new frame. You'll get different results. Another important consideration is if the unsupervised method can handle new data. Manifold embeddings do not provide a linear transform that can handle new data for inference. Manifold embeddings are also not necessarily rotation invariant or equivariant.

## 9.7 Distance Features

One more topic on parsing data is treating distances. As we saw above, pairwise distance is a wise transformation because it is translation and rotation invariant. However, we may want to sometimes transform this further. One obvious choice is to use  $1/r$  as the input to our neural network. This is because most properties of atoms (and thus molecules) are affected by their closest nearby atoms. An oxygen nearby a carbon is much more important than an oxygen that is 100 nanometers away. Choosing  $1/r$  as an input makes it easier for a neural network to train because it encodes this physical insight about local interactions being the most important. Of course, a neural network could learn to turn  $r$  into  $1/r$  because they are universal approximators. Yet this approach means we do not need to waste training data and weights on learning to change  $r$  into  $1/r$ . *This approach will not work on non-local or mildly-non-local effects like electrostatics or multi-scale phenomena.*

Another detail on distances is that we often want to “featurize” them; we'd like to go from one a single number like  $r = 3.2$  to a vector of reals. Why? Well that's just how neural networks learn. Hidden-layers need to have more than 1 dimension to be expressive enough to model any function. This seems like an obvious point though: if you used  $r$  in a neural network it would obviously get larger as it goes through hidden layers. However, there are a few “standard” ways that people like to do this process. There are valid reasons, like making it smoothly differentiable, that you might choose one of these special “featurizing” functions.

### 9.7.1 Repeating

The first approach is to just repeat  $r$  up to the desired hidden dimension. This is representable as a dense neural network with no activation.

### 9.7.2 Binning

As explored in Gilmer et al. [GSR+17] and others, you can bin your distances to be a one-hot vector. Essentially, you histogram  $r$  into fixed bins so that you only have one bin being “hot”. Each bin will represent a segment of positions (e.g., 4.5-5.0). This has discontinuous derivatives with respect to distance and is rarely used.

### 9.7.3 Radial Basis Functions

Radial basis functions are a commonly used procedure for converting a scalar into a fixed number of features and were first used in interpolation[Pow77]. Radial basis functions use the following equation:

$$e_i = \exp \left[ -(r - d_i)^2 / w \right] \quad (9.9)$$

where  $d_i$  is an equally spaced vector of distances (e.g., [1.5, 3.0, 4.5, 6.0]) and  $w$  is a trainable (or hyper) parameter. This computes a Gaussian kernel between  $r$  and all distances  $d_i$ . What is nice about this expression is the smooth well-behaved derivatives with respect to  $r$  and lack of trainable parameters. You can (almost) represent this with a dense layer and a softmax activation.

### 9.7.4 Sub NN

Another strategy used in Gilmer et al. [GSR+17] is to just put your distances through a series of dense layers to get features. For example, if you’re going to use the distance in a graph neural network you could run it through three dense layers first to get a larger feature dimension. Remember that repeating and radial basis functions are equivalent to dense layers (assuming correct activation choice), so this strategy can be a simple solution to the above choice.

## 9.8 Chapter Summary

- Machine learning models that work with molecules must be permutation invariant, such that if the atoms are rearranged, the output will not change.
- Translational invariance of molecular coordinates is when the coordinates are shifted and the resulting output does not change.
- Rotational invariance is similar, except the molecular coordinates are rotated.
- Data augmentation is when you try to teach your model the various types of equivariances by rotating and translating your training data to create additional examples.
- There are various techniques, such as eigendecomposition or pairwise distance to make molecular coordinates invariant.
- A one-hidden layer dense neural network is an example of a model with no equivariances.
- You can try alignment for trajectories, where each training example has the same ordering and number of atoms.

## 9.9 Cited References

## EQUIVARIANT NEURAL NETWORKS

The previous chapter *Input Data & Equivariances* discussed data transformation and network architecture decisions that can be made to make a neural network equivariant with respect to translation, rotation, and permutations. However, those ideas limit the expressibility of our networks and are constructed ad-hoc. Now we will take a more systematic approach to defining equivariances and prove that there is only one layer type that can preserve a given equivariance. The result of this section will be layers that can be equivariant with respect to any transform, even for more esoteric cases like points on a sphere or mirror operations. To achieve this, we will need tools from group theory, representation theory, harmonic analysis, and deep learning. Equivariant neural networks are part of a broader topic of **geometric deep learning**, which is learning with data that has some underlying geometric relationships. Geometric deep learning is thus a broad-topic and includes the “5Gs”: grids, groups, graphs, geodesics, and gauges. However, you’ll see papers with that nomenclature concentrated on point clouds (gauges), whereas graph learning and grids are usually called graph neural networks and convolutions neural networks respectively.

---

### Audience & Objectives

This chapter builds on *Input Data & Equivariances* and a strong background in math. Although not required, a background on Hilbert spaces, group theory, representation theory, Fourier series, and Lie algebra will help. After completing this chapter, you should be able to

- Derive and understand the mathematical foundations of equivariant neural networks
- Reason about equivariances of neural networks
- Know common symmetry groups
- Implement G-equivariant neural network layers
- Understand the shape, purpose, and derivation of irreducible function representations
- Know how weight-constraints can be used as an alternative

**Danger:** This chapter teaches how to add equivariance for point clouds, but not permutations. To work with real molecules we need to combine ideas from this chapter with permutation equivariance from the *Graph Neural Networks* chapter. That combination is explored in *Modern Molecular NNs*.

## 10.1 Do you need equivariance?

I'm being a bit unfair, these papers have some slightly different application areas (lie vs compact vs finite groups) and differ mostly in their nonlinearity.

Before we get too far, let me first try to talk you out of equivariant networks. The math required is advanced, especially because the theory of these is still in flux. There are five papers in the last few years that propose a general theory for equivariant networks and they each take a slightly different approach [FSIW20, CGW19, KT18, LW20, FWW21]. It is also easy to make mistakes in implementations due to the complexity of the methods. You must also do some of the implementation details yourself, because general efficient implementations of groups is still not solved (although we are [getting close now for specifically SE\(3\)](#)). You will also find that equivariant networks are not in general state of the art on point clouds – although that is starting to change with recent benchmarks set in point cloud segmentation [WAR20], molecular force field prediction [BSS+21], molecular energy predictions [KGrossGunnemann20], and 3D molecular structure generation [SHF+21].

Alternatives to equivariant networks are training and testing augmentation. Both are powerful methods for many domains and are easy to implement [SK19]. You can find details in the [Input Data & Equivariances](#) chapter. However, augmentation does not work for locally compact symmetry groups (e.g.,  $SO(3)$ ) — so you cannot use them for rotationally equivariant data. You can do data transformations like discussed in [Input Data & Equivariances](#) to avoid equivariance and only work with invariance.

So why would you study this chapter? I think these ideas are important and incorporating the equivariant layers into other network architectures can dramatically reduce parameter numbers and increase training efficiency.

## 10.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

### 10.2.1 Outline

We have to lay some mathematical foundations before we can grasp the equations and details of equivariant networks. We'll start with a brief overview of group theory so we can define the principle of equivariance generally. Then we'll show how any equivariance can be enforced in a neural network via a generalization of convolutions. Then we'll visit representation theory to see how to encode groups into matrices. Then we'll see how these convolutions can be more easily represented using the generalization of Fourier transforms. Finally, we'll examine some implementations. Throughout this chapter we'll see three examples that capture some of the different settings.

## 10.3 Group Theory

A modern treatment of group theory can be found in [Zee16]. You can watch a short fun primer video on group theory from 3Blue1Brown [here](#).

A group is a general object in mathematics. A group is a set of elements that can be combined in a binary operation whose output is another member of the group. The most common example are the integers. If you combine two integers in a binary operation, the output is another integer. Of course, it depends on the operation ( $1 \div 2$  does not give an integer), so specifically consider addition. Integers are not the example we care about though. We're interested in groups of **transformations** that move points in a space. Operations like rotation, scaling, mirroring, or translating of single points. As you read about groups here, remember that the elements of the groups are *not* numbers or points. The group elements are transformations that act on points in the space. Notice I'm being a bit nebulous on what the space is for now. Let's first define a group:

### Group Definition

A group  $G$  is a set of elements (e.g.,  $\{a, b, c, i, e\}$ ) equipped with a binary operation ( $a \cdot b = c$ ) whose output is another group element and the following conditions are satisfied:

1. **Closure** The output of the binary operation is always a member of the group
2. **Associativity**  $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
3. **Identity** There is a single identity element  $e$  such that  $ex = x \forall x \in G$
4. **Inverse** There exists exactly one inverse element  $i$  for each  $x$  such that  $xi = e$

This is quite a bit of nice structure. We always have an inverse available. Applying the binary operations never accidentally leaves our group. One important property that is missing from this list is **commutativity**. In general, a group is not commutative so that  $a \cdot b \neq b \cdot a$ . If the group does have this extra property, we call the group **abelian**. Another detail is how big the set is. It can indeed be infinite, which is why the integers or all possible rotations of rotations of points on a sphere can be represented as a group. One notational convenience we'll make is that the binary operation “.” will just be referred to as “dot” or sometimes multiplication if I get sloppy. The number of elements in a group  $|G|$  is known as the **order**.

If you multiply two transforms  $a \cdot b$ , we always apply  $b$  first and then  $a$ . This is important to remember for non-commutative groups (non-abelian).

The point of introducing the groups is so that they can transform elements of our space. This is done through a **group action**

### Group Action

A group action  $\pi(g, v)$  is a mapping from a group  $G$  and a space  $\mathcal{X}$  to the space  $\mathcal{X}$ :

$$\pi : G \times \mathcal{X} \rightarrow \mathcal{X} \quad (10.1)$$

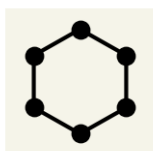
function arrow

$G \times \mathcal{X}$  means there are two input arguments, one from group  $G$  and one from space  $\mathcal{X}$ .  $\rightarrow \mathcal{X}$  shows our function outputs a value in the space  $\mathcal{X}$ .

So a group action takes in two arguments (binary): a group element and a point in a space  $\mathcal{X}$  and transforms the point to a new one:  $\pi(g, x_0) = x_1$ . This is just a more systematic way of saying it transforms a point. The group action is neither unique to the space nor group. Often we'll omit the function notation for the group action and just write  $gx = x'$ .

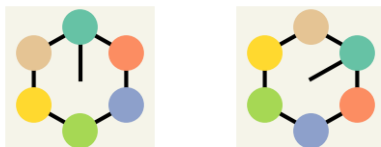
Let's introduce our three example groups that we'll refer to throughout this chapter.

### 10.3.1 Finite Group $Z_6$



The first group is about rotations of a hexagon

. Our basic group member will be rotating the hexagon



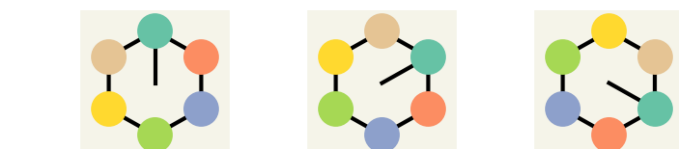
enough to shift all the vertices:

$\rightarrow$

. Notice I've colored the vertices and added a line so we can easily distinguish the orientation of the hexagon. Remember the hexagon, its colors, and if it is actually symmetric have nothing to do with the group. *The group elements are transformations we apply to the hexagon.*

One group action for this example can use modular arithmetic. If we represent a point in our space as  $\{0, \dots, 5\}$  then the rotation transformation is  $x' = x + 1 \pmod{6}$ . For example, if we start at 5 and rotate, we get back to 0.

Our group must contain our rotation transformation  $r$  and the identity:  $\{e, r\}$ . This set is not closed though: rotating twice



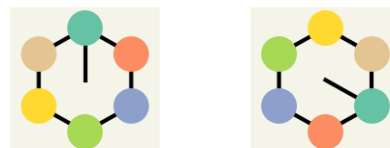
$r \cdot r$

$\rightarrow$

$\rightarrow$

need to have  $\{e, r, r^2, r^3, r^4, r^5\}$ .

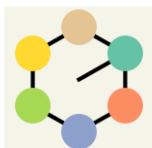
gives a new group element  $r^2$ . To close the group we



Is this closed? Consider rotating twice and then five times  $r^5 \cdot r^2$

$\rightarrow$

$\rightarrow$



You can see that this is the same as  $r$ , so  $r^5 \cdot r^2 = r$ . What about the inverses element? The inverse of  $r$  is  $r^5$ .  $r \cdot r^5 = e$ . You can indeed see that each element has an inverse ( $e$  is its own inverse).

In general, we can write out the group as a multiplication table that conveys all group elements and defines the output of



all binary outputs:

	$e$	$r$	$r^2$	$r^3$	$r^4$	$r^5$
$e$	$e$	$r$	$r^2$	$r^3$	$r^4$	$r^5$
$r$	$r$	$r^2$	$r^3$	$r^4$	$r^5$	$e$
$r^2$	$r^2$	$r^3$	$r^4$	$r^5$	$e$	$r$
$r^3$	$r^3$	$r^4$	$r^5$	$e$	$r$	$r^2$
$r^4$	$r^4$	$r^5$	$e$	$r$	$r^2$	$r^3$
$r^5$	$r^5$	$e$	$r$	$r^2$	$r^3$	$r^4$

You can also see that the group is abelian (commutative). For example,  $r \cdot r^3 = r^3 \cdot r$ .

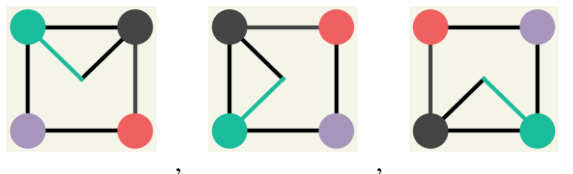
This kind of table is called a **Cayley table**. Although it doesn't matter for this example, we'll see later that the order of look-up matters. Specifically if our group is non-abelian. *The row factor comes first and the column factor second.* So  $r \cdot r^5$  means we look at row  $r$  and column  $r^5$  to get the group element, which in this case is  $e$ .

This group of rotations is an example of a **cyclic group** and is isomorphic (same transformations, but operates on different objects) to integers modulo 6. Meaning, you could view rotation  $r^n$  as operating on integers  $(x + n) \bmod 6$ . Cyclic groups are written as  $Z_n$ , so this group is  $Z_6$ .

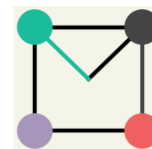
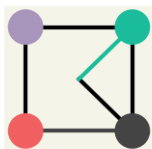
### 10.3.2 p4m

p4m strictly speaking only includes integer translations but many of the principles apply for continuous infinite groups (locally compact) and integer (countably) infinite groups

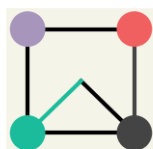
The second group contains translation,  $90^\circ$  rotations, and horizontal/vertical mirroring. We're now operating on real numbers  $x, y$ , so we're in  $\mathbb{R}^2$ . Let's ignore the translation for now and just consider mirroring ( $s$ ) and rotation by  $90^\circ$  ( $r$ ) about the origin. What powers of  $r$  and  $s$  do we need to have a closed group? Considering rotations alone first, like last time



we should only need up to  $r^3$ . Here are the rotations visually:



What about mirroring on horizontal/vertical? Mirroring along the horizontal axis:



→ is actually the same as rotating twice and then mirroring along the vertical. In fact, you only need to have mirroring along one axis. We'll choose the vertical axis by convention and denote that as  $s$ .

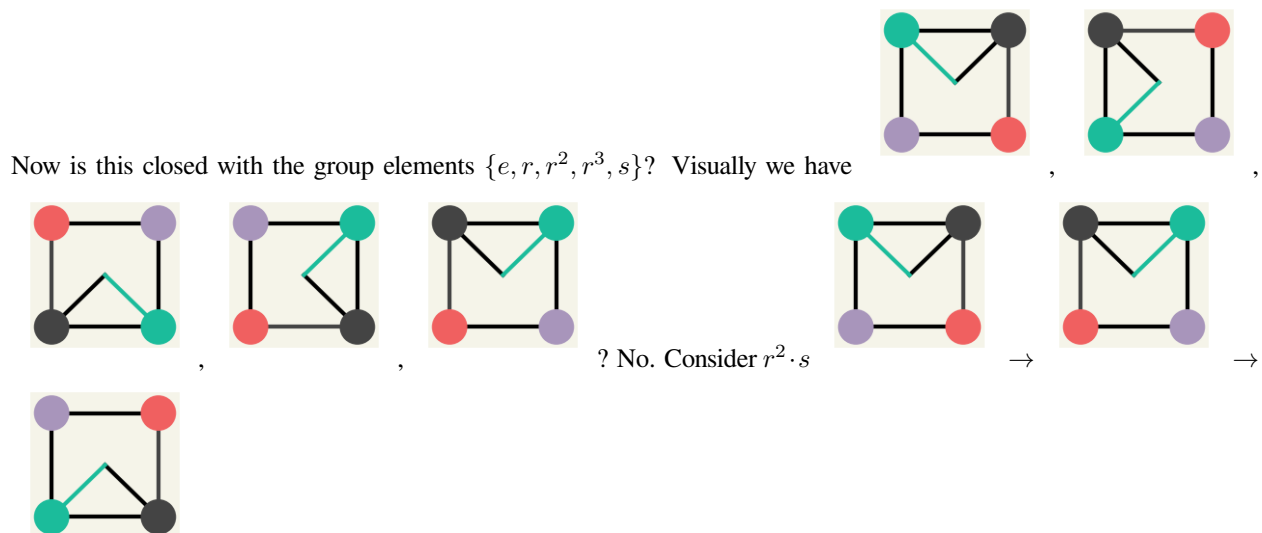
We can build the group action piece by piece. The group action for rotation can be represented as a 2D rotation matrix acting a point  $(x, y)$ :

$$\begin{bmatrix} \cos \frac{k2\pi}{4} & -\sin \frac{k2\pi}{4} \\ \sin \frac{k2\pi}{4} & \cos \frac{k2\pi}{4} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, k \in \{0, 1, 2, 3\}$$

where  $k$  can allow us to do two rotations at once ( $k = 2$ ) or the identity ( $k = 0$ ). The vertical axis mirror action can be represented by

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

These two group actions can be ordered to correctly represent rotation then mirroring or vice-versa.

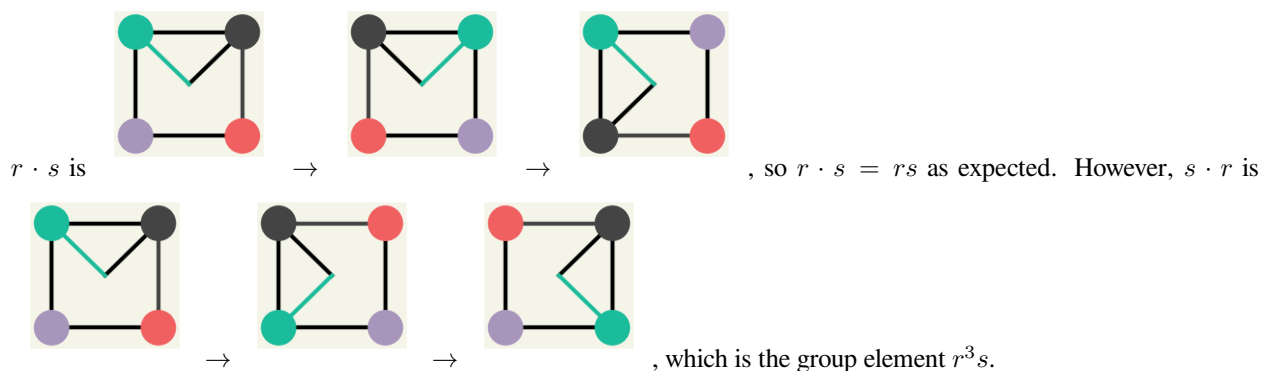


which is not an element. To close the group, we need  $\{e, r, r^2, r^3, s, rs, r^2s, r^3s\}$ . The multiplication table (which defines the elements too) is:

	$e$	$r$	$r^2$	$r^3$	$s$	$rs$	$r^2s$	$r^3s$
$e$	$e$	$r$	$r^2$	$r^3$	$s$	$rs$	$r^2s$	$r^3s$
$r$	$r$	$r^2$	$r^3$	$e$	$rs$	$r^2s$	$r^3s$	$s$
$r^2$	$r^2$	$r^3$	$e$	$r$	$r^2s$	$r^3s$	$s$	$rs$
$r^3$	$r^3$	$e$	$r$	$r^2$	$r^3s$	$s$	$rs$	$r^2s$
$s$	$s$	$r^3s$	$r^2s$	$rs$	$e$	$r^3$	$r^2$	$r$
$rs$	$rs$	$s$	$r^3s$	$r^2s$	$r$	$e$	$r^3$	$r^2$
$r^2s$	$r^2s$	$rs$	$s$	$r^3s$	$r^2$	$r$	$e$	$r^3$
$r^3s$	$r^3s$	$r^2s$	$rs$	$s$	$r^3$	$r^2$	$r$	$e$

This is a **Cayley table**. Remember *The row factor comes first and the column factor second*. So  $rs \cdot r^3$  means we look at row  $rs$  and column  $r^3$  to get the group element, which in this case is  $r^2s$ .

As you can see from the Cayley table, the group is closed. Remember, elements like  $rs$  are not a binary operation. They are group elements, hence the missing binary operation symbol. We also see that the group is not commutative.



We can also read the inverses off the table. For example, the inverse of  $r$  is the column which gives the identity element:  $r^3$ . This group is known as the dihedral group 4  $D_4$ . It has order 8.

Now consider the translation group elements. For simplicity, let's only consider integer translations. We can label them as  $t_{w,h}$ . So  $t_{3,4}$  means translate by  $x + 3$  and  $y + 4$ . Is this a proper group? Certainly it is associative, there is an identity  $t_{0,0}$  and an inverse for each element  $t_{-x,-y}$ . What about closure? Yes, since translating twice is equivalent to one larger translation:  $t_{w,h} \cdot t_{w',h'} = t_{w+w',h+h'}$ . This expression also shows group action for translation.

What about when we combine with our other elements from the  $D_4$  group? Consider the product  $r \cdot t_{3,4}$ . This means translating by  $(3, 4)$  and then rotating by  $90^\circ$  about the origin. If you consider this acting on a single point  $(0, 0)$ , you could get  $(0, 0) \rightarrow (3, 4) \rightarrow (-4, 3)$ . What element of our group would this represent? At first it seems like it could be  $t_{-3,4}$ . However,  $t_{-3,4}$  would only work specifically for starting at  $(0, 0)$ . If you started at  $(1, 1)$ , you would get to  $(-4, 5)$  with  $r \cdot t_{3,4}$  and  $(-2, 5)$  with  $t_{-3,4}$ . To be correct for *any point*, we need a different group element. So the product  $r \cdot t_{3,4}$  actually cannot be a product but instead must be a group element. In fact, our new combined group is just going to be  $ab$  where  $a$  is an element from  $D_4$  and  $b$  is a translation. Thus  $r \cdot t_{3,4} = rt_{3,4}$ .

Combining these two groups, the translation and  $D_4$ , is an example of a **semidirect product**. A semidirect product just means that we create a new group by combining all possible group elements. There is some machinery for this, like the identity element in our new group is something like  $et_{0,0}$ , and it has some other structure. It is called semidirect, instead of direct, because we can actually mix our group elements. The elements both act on points in the same space  $(x, y)$  plane, so this makes sense. Another condition is that we can only have a semidirect product when one subgroup is normal and the translation subgroup is the normal subgroup. It is coincidentally abelian, but these two properties are not always identical. This semidirect product group is called p4m.

Below, is an optional section that formalizes the idea of combining these two groups into one larger group.

### Normal Subgroup

A normal subgroup is a group of elements  $n$  from the group  $G$  called  $N$ . Each  $n \in N$  should have the property that  $g \cdot n \cdot g^{-1}$  gives an element in  $N$  for any  $g$ .

This does not mean  $g \cdot n \cdot g^{-1} = n$ , but instead that  $g \cdot n \cdot g^{-1} = n'$  where  $n'$  is some other element in  $N$ . For example, in p4m the translations form a normal subgroup. Rotating, translating, then doing the inverse of the rotation is equivalent to some translation. Notice that  $D_4$  is not a normal subgroup of p4m. If you do an inverse translation, rotate, then do a translation you may not have something equivalent to a rotation. It may be strange that we're talking about the group p4m when we haven't yet described how it's defined (identity, inverse, binary op). We'll do that with the semidirect product and then we could go back and verify that the translations are a normal subgroup more rigorously. I do not know the exact connection, but it seems that normal subgroups are typically abelian.

### Semidirect Product

Given a normal subgroup of  $G$  called  $N$  and a subgroup  $H$ , we can define  $G$  using the semidirect product. Each element in  $G$  is a tuple of two elements in  $N, H$  written as  $(n, h)$ . The identity is  $(e_n, e_h)$  and the binary operation is:

$$(n_1, h_1) \cdot (n_2, h_2) = (n_1 \cdot \phi(h_1)(n_2), h_1 \cdot h_2) \quad (10.2)$$

where  $\phi(h)(n)$  is the conjugation of  $n$   $\phi(h)(n) = h \cdot n \cdot h^{-1}$ . When a transform  $(n, h)$  is applied, we follow the normal convention that  $h$  is applied first followed by  $n$ .

We are technically doing an outer semidirect product: combining them under the assumption that both  $D_4$  and  $T$  are part of a larger group which contains both. This is a bit of a semantic detail, but they are actually both part of  $p4m$  and a larger group called the affine group which includes, rotation, shear, translation, mirror, and scale operations on points. You could also argue they are part of groups which can be represented by 3x3 invertible matrices. Thus, you can combine these and get something that is still smaller than their larger containing group ( $p4m$  is smaller than all affine transformations).

One consequence of the semidirect product is that if you have a group element  $(n, h)$  but want to instead apply  $n$  first (instead of  $h$ ), you can use the binary operation:

$$(e_n, h) \cdot (n, e_h) = (e_n \cdot \phi(h)(n), h \cdot e_h) = (\phi(h)(n), h) \quad (10.3)$$

so  $\phi(h)(n)$  somehow captures the effect of switching the order applying elements from  $H$  and  $N$ . In our case, this means swapping the order of rotation/mirroring and translation.

To show what effect the semidirect product has in p4m, we can clean-up our example above about  $r \cdot t_{3,4}$ . We should write the first element of this binary product  $r$  as a tuple of group elements: one from the  $D_4$  and one from the translations. Since there is no translation for  $r$ , we use the identity. Thus we write  $r$  as  $(t_{0,0}, r)$  in our semidirect product group p4m. Note that the normal subgroup comes first (applied last) by convention. Similarly,  $t_{3,4}$  is written as  $(t_{3,4}, e)$ . Our equation becomes:

$$(t_{0,0}, r) \cdot (t_{3,4}, e) = (t_{0,0} \cdot \phi(r)(t_{3,4}), r \cdot e) = (t_{0,0} \cdot \phi(r)(t_{3,4}), r)$$

where  $\phi$  is the automorphism that distinguishes a semidirect product from a direct product. The direct product has  $\phi(h)(n) = n$  so that the binary operation for the direct product group is just the element-wise binary products.  $\phi(h)(n) = hnh^{-1}$  for semidirect products. In our equation, this means  $\phi(r)(t_{3,4}) = r \cdot t_{3,4} \cdot r^3$ . Substituting this and using the fact that both groups have the same binary operation (matrix multiplication, as we'll see shortly):

$$(t_{0,0} \phi(r)(t_{3,4}), r) = (r \cdot t_{3,4} \cdot r^3, r) = r \cdot t_{3,4} \cdot r^3 \cdot r = r \cdot t_{3,4}$$

Thus we've proved that translating by 3, 4 followed by rotating can be expressed as  $r \cdot t_{3,4}$ , which seems like a lot of work for an obvious result. I won't cover the semidirect product of the group action, but we'll see that we do not necessarily need to build a group action encapsulating both translation and rotation/mirroring.

### 10.3.3 SO(3) Group

SO(3) is the group for analyzing 3D point clouds like trajectories or crystal structures (with no other symmetries). SO(3) is the group of all rotations about the origin in 3D. The group is non-abelian because rotations in 3D are not commutative. The group order is infinite, because you can rotate in this group by any angle (or sets of angles). If you are interested in allowing translations, you can use SE(3) which is the semidirect product of SO(3) and the translation group (like p4m), which is a normal subgroup.

The SO(3) name is a bit strange. SO stands for "special orthogonal" which are two properties of square matrices. In this case, the matrices are  $3 \times 3$ . Orthogonal means the columns sum to one and special means the determinant is 1. Interestingly, all rotations in 3D around the origin are also the SO(3) matrices.

One detail is that since we're rotating (no scale or translation) the distance to origin will not change. We cannot move the radius. The group action is the product of 3 3D rotation matrices (using [Euler angles](#))  $R_z(\alpha)R_y(\beta)R_z(\gamma)$  where  $\alpha, \gamma \in [0, 2\pi]$ ,  $\beta \in [0, \pi]$  and

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

### 10.3.4 Groups on Spaces

We've defined transforms and their relationships to one another via group theory. Now we need to actually connect the transforms to a space. It is helpful to think about the space as Euclidean with a concept of distance and coordinates, but we'll see that this is not required. Our space could be vertices on a graph or integers or classes. There are *some* requirements though. The first is that our space must be **homogeneous**. Homogeneous means that from any point in our space  $x$  we can reach any other point with a transform  $g$  from our group  $G$ . The second requirement is that if our group is infinite, the space must **locally compact**. This is a concept from topology and we won't really ever be troubled by it. Most spaces we'll see in chemistry or materials science (Euclidean spaces) are locally compact. This doesn't matter for finite groups, so we can use non-compact spaces.

**Lie group**

If the group transforms are further smooth and have smooth inverses, the group (and associated space) are called a **lie group**.

**Finite Group  $Z_6$** 

The space is homogeneous because our group includes “compound” rotations like  $r^4$ . This is a finite group, so we do not require the space to be compact.

**Locally Compact p4m**

The space is homogeneous since we can use a translation to get to any other point. The space is locally compact because we are in 2D Euclidean geometry.

**SO(3) Group**

The space is homogeneous because we restrict ourselves to being on the sphere. The space is locally compact because we are in 3D Euclidean geometry.

The requirement of space being homogeneous is fairly strict. It means we cannot work with  $\mathbb{R}^2$  with a finite group like mirror and fixed rotations (i.e., p4m without translations). For example, going from  $x = (0, 0)$  to  $x = (1, 1)$  cannot be done with rotations/mirror group elements alone. As you can see, working in a Euclidean space thus requires a locally compact group. Similarly, a finite group implies a finite space because of the homogeneous requirement.

This may seem like a ton of work. We could have just started with  $xyz$  coordinates and rotation matrices. Please continue to wait though, we’re about to see something incredible.

## 10.4 Equivariance Definition

You should be thinking now about how we can define equivariance using our new groups. That’s where we’re headed. We need to do a bit of work now to “lift” neural networks and our features into the framework we’re building. First, in *Input Data & Equivariances* we defined our features as being composed of tuples  $(\vec{r}_i, \vec{x}_i)$  where  $\vec{r}_i$  is a spatial point and  $\vec{x}_i$  are the features at that point. Let’s now view these input data as functions, defined as  $f(\vec{r}) = \vec{x}$  and assume if a point  $\vec{r}'$  isn’t in our training data then  $f(\vec{r}') = \vec{0}$ . More formally, our training data is a function  $f : \mathcal{X} \rightarrow \mathbb{R}^n$  that maps from our homogeneous space  $x$  to real vector (or complex vectors) of dimension  $n$ .

We have promoted our data into a function and now a neural network can no longer be just function since its input is a function. Our neural network will be also promoted to an **linear map**, which has an input of a function and an output of a function. Formally, our network  $\psi : f(\mathcal{X}) \rightarrow f'(\mathcal{X})$ . Notice the input and output spaces of the functions should be the same (we cannot switch from 2D to 3D). Linear maps are also called **operators**, depending on which branch of mathematics you’re in.

The last piece of equivariance is to promote our group elements, which transform points, to work on functions.

**G-Function Transform Definition**

An element  $g$  of group  $G$  on the homogeneous space  $\mathcal{X}$  can act on a function  $f : \mathcal{X} \rightarrow \mathbb{R}^n$  via the group transform linear map  $\mathbb{T}_g : f(\mathcal{X}) \rightarrow f'(\mathcal{X})$  defined as

$$f'(gx) = f(x) \Rightarrow f'(x) = f(g^{-1}x) \quad (10.4)$$

This definition takes a moment to think about. Consider a translation of an image. You want to move an image to the left by 10 pixels, so  $g = t_{10,0}$ . The image is defined by the function  $f(x, y) = (r, g, b)$ , where  $r, g, b$  is the color. We want  $T_g f(x, y)$ . Without knowing about groups, you can intuit that translating can be done by creating a new function  $f'(x', y') = f(x - 10, y)$ . Notice that the inverse of  $g^{-1} = t_{-10,0}$  acts on the points, not  $g$ . Recall that a group requires there to be an inverse for any group element.

Now we have all the pieces to define an equivariant neural network:

### Equivariant Neural Network Definition

Given a group  $G$  that has actions on two homogeneous space  $\mathcal{X}_1$  and  $\mathcal{X}_2$ , a  $G$ -equivariant neural network is a linear map  $\psi : f(\mathcal{X}_1) \rightarrow f'(\mathcal{X}_2)$  that has the property[KT18]:

$$\psi [\mathbb{T}_g f(x)] = \mathbb{T}'_g \psi [f(x)] \quad \forall f(x) \quad (10.5)$$

where  $\mathbb{T}_g$  and  $\mathbb{T}'_g$  are  $G$ -function transforms on the two spaces. If  $\mathbb{T}'_g = \text{id}$ , meaning the transform is the identity in the output space regardless of  $g$ , then  $\psi$  is a  $G$ -invariant neural network.

The definition means that we get the same output if we transform the input function to the neural network or transform the output (in the equivariant case). In a specific example, if we rotate the input by 90 degrees, that's the same result as rotating the output by 90 degrees. Take a moment to ensure that matches your idea of what equivariance means. After all this math, we've generalized equivariance to arbitrary spaces and groups.

What the two input and output spaces? It's easiest to think about them as the same space for equivariant neural networks. For an invariant, the output space is typically a scalar. Another example for an invariant one could be aligning a molecular structure to a reference. The neural network should align to the same reference regardless of how the input is transformed.

## 10.5 G-Equivariant Convolution Layers

Kondor and Trivedi showed that there is *only one* way to make a  $G$ -equivariant neural network:

### G-Equivariant Convolution Theorem

A neural network layer (linear map)  $\psi$  is  $G$ -equivariant if and only if its form is a convolution operator \*

$$\psi(f) = (f * \omega)(u) = \sum_{g \in G} f \uparrow^G (ug^{-1}) \omega \uparrow^G (g) \quad (10.6)$$

where  $f : H \rightarrow \mathbb{R}^n$  and  $\omega : H' \rightarrow \mathbb{R}^n$  are functions of quotient spaces  $H$  and  $H'$ . If the group  $G$  is locally compact (infinite elements), then the convolution operator is

$$\psi(f) = (f * \omega)(u) = \int_G f \uparrow^G (ug^{-1}) \omega \uparrow^G (g) d\mu(g) \quad (10.7)$$

where  $\mu$  is the group Haar measure. A [Haar measure](#) is a generalization of the familiar integrand factor you see when doing integrals in polar coordinates or spherical coordinates.

This is one of the strongest theorems in deep learning. It says there is only one way to achieve equivariance in a neural network. This may seem counter-intuitive since there are many competing approaches to convolutions. These other approaches are actually equivalent to a convolution, just it can be hard to notice.

As you can see from the theorem, we must introduce more new concepts. The first important detail is that all our functions are over our group elements (technically the quotient space  $G/H_0$ ), not our space. This should seem strange. We will easily fix this because there is a (bijective) way to assign one group element to each point in the space. The second detail is the  $f \uparrow^G$ . The order of the group  $G$  is greater than or equal to the number of points in our space, so if the function is defined on our space, we must “lift” it up to the group  $G$  which has more elements. The last detail is the point about **quotient spaces**. Quotient spaces are how we cut-up our group  $G$  into subgroups so that one has the same order as the number of points in our space. Below I detail these new concepts just enough so that we can implement and understand these convolutions.

**Warning:** To actually learn, you need to put in a nonlinearity in after the convolution. A simple (and often used) case is to just use a standard activation function like ReLU. We’ll look at more complex examples below.

## 10.6 Converting between Space and Group

Let’s see how we can convert between functions on the space  $\mathcal{X}$  and functions on the group  $G$ .  $|G| \geq |\mathcal{X}|$  (because the space is homogeneous) so it is rare that we can uniquely replace each point in space with a group in  $G$ . Instead, we will construct a partitioning of  $G$  into  $|\mathcal{X}|$  sets called a quotient space  $G/H$  such that  $|G/H| = |\mathcal{X}|$ . It turns out, there is a well-studied approach to arranging elements in a group called **cosets**. Constructing cosets is a two-step process. First we define a subgroup  $H$ . A **subgroup** means it is itself a group; it has identities and inverses. We cannot accidentally leave  $H$ ,  $h_1 \cdot h_2 \in H$ . For example, translation transformations are a subgroup because you cannot accidentally create a rotation when combining two translations.

This process of constructing cosets and then using that to lift our function is closely related to the process of finding an induced representation on  $G$  via a representation on  $H$ .

After constructing a subgroup  $H$ , we can apply an element  $g$  to every element in  $H$ , written as

$$gH = \{g \cdot h \mid h \in H\} \quad (10.8)$$

If this sounds strange, wait for an example.  $gH$  is called a **left coset**. We mention the direction because  $G$ ’s binary operation may not be commutative (non-abelian). What happens if  $g$  is in  $H$ ? No problem;  $H$  is a group so applying an element to every element in  $H$  just gives back  $H$  (i.e.  $hH = H$ ). Cosets are not groups, they are definitely not closed or have inverses. What’s the point of making all these cosets? Remember our goal is to partition  $G$  into a bunch of smaller sets so that we have one for each point in  $\mathcal{X}$ . Constructing cosets partitions  $G$  for sure, but do we get enough? Could we accidentally have overlaps between cosets, where  $g_1H$  and  $g_2H$  contain the same elements?

If your group involves rotations, make life easy on yourself and always choose  $x_0$  as the origin (or center of the rotations).

It turns out if our space is homogeneous we can construct our cosets in a special way so that we have exactly one coset for each point in the space  $\mathcal{X}$ . To get our group, we pick an arbitrary point in the space  $x_0$ . Often this will be the origin. Then we choose our subgroup  $H$  to be all group elements that leave  $x_0$  unchanged. This is called a stabilizer subgroup  $H_0$  and is defined as

$$H_0 = \{g \in G \text{ such that } gx_0 = x_0\} \quad (10.9)$$

We will not prove that this is a group itself. This defines our subgroup. Here’s the remarkable thing: we will have exactly enough cosets with this stabilizer as there are points in  $\mathcal{X}$ . However, multiple  $g$ ’s will give the same coset (as expected, since  $|G| > |\mathcal{X}|$ ).

This set of all cosets is itself a group and it is written as  $G/H_0$ . The fact that the cosets is a group is just weird. What is the identity coset? How do you define binary operations on cosets? It turns out we do not need these items but it is fascinating.

Now comes the details, how do we match-up points in  $\mathcal{X}$  to the cosets? We know that the space is homogeneous so each point in  $x$  can be reached from our arbitrary origin by a group element  $gx_0 = x$ . That's one way to connect points to group elements, but which coset will  $g$  be in? There also may be multiple  $g$ s that satisfy the equation. It turns out that all the group elements that satisfy the equation will be in the same coset. The reason why is that  $g \cdot hx_0 = gx_0$  because all elements  $h$  of the stabilizer group do not move  $x_0$ . Quite elegant.

How do we find which coset we need? Since the identity  $e$  is in  $H_0$  (by definition), the coset  $gH_0$  will contain  $g$  itself. Thus, we can convert a function  $f(x)$  from the space to be a function on the quotient space  $f(g)$  via what we call **lifting**:

$$f \uparrow^G (g) = f(gx_0) \quad (10.10)$$

One point to note is that you can plug any element  $g$  from the group into  $f \uparrow^G (g)$  but it is bijective only over  $G/H$  (the cosets). Your null space will be the whole subgroup  $H_0$ .

A coset can have multiple labels in this system.  $g_1H_0$  and  $g_2H_0$  could be the same coset. There are no consequences of this, but just be aware.

Going the opposite, from a function on the group to the space, is called **projecting** because it will have a smaller domain. We can use the same process as above. We create the quotient space and then just take the average over a single coset to get a single value for the point  $x$ :

$$f \downarrow_{\mathcal{X}(x)=\frac{1}{|H_0|} \sum_{u \in gH_0} f(u), gx_0=x} (10.11)$$

where we've used the fact that  $|gH_0| = |H_0|$ . Note that the coset generating element  $g$  is found by solving  $gx_0 = x$ , where of course  $g$  is not a stabilizing element (otherwise  $gx_0 = x_0$  by definition). Let's see some examples now to make all of these easier to understand.

## Finite Group $Z_6$



Our function is the color of the vertices in our picture  $f(x) = (r, g, b)$  where  $r, g, b$  are fractions of the color red, blue green. If we define the vertices to start at the line pointing up, we can label them  $0, \dots, 5$ . So for example  $f(0) = (0.11, 0.74, 0.61)$ , which is the color of the top vertex.

We can define the origin as  $x_0 = 0$ .  $|G| = |\mathcal{X}|$  for this finite group and thus our stabilizer subgroup only contains the identity  $H_0 = \{e\}$ . Our cosets and their associated points will be  $(eH_0, x = 0), (rH_0, x = 1), (r^2H_0, x = 2), (r^3H_0, x = 3), (r^4H_0, x = 4), (r^5H_0, x = 5)$ . The lifted  $f \uparrow^G (g)$  can be easily defined using these cosets.



## Locally Compact p4m

p4m is intended for images, so our example will be a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$  that represents a color image. This group contains rotations about the origin, so if we choose the origin as our stabilizer it will cleanly separate our group. Namely:

$$H_0 = \{e_n e_r, e_n r, e_n r^2, e_n r^3, e_n s, e_n r s, e_n r^2 s, e_n r^3 s\}$$

where our elements have been written out as the semidirect product of translations and  $D_4$  as discussed previously. Let's compute a coset to get a sense of this process. Consider the group element  $t_{1,0} e_r$  creating the coset  $t_{1,0} e_r H_0$ . The first element of the coset is  $t_{1,0} e_r \cdot e_n e_r = t_{1,0} e_r$ . The second element is  $t_{1,0} e_r \cdot t_{0,0} r = t_{1,0} r$ . The rest of the elements of this coset are:

$$t_{1,0} e_r H_0 = \{t_{1,0} e_r, t_{1,0} r, t_{1,0} r^2, t_{1,0} r^3, t_{1,0} s, t_{1,0} r s, t_{1,0} r^2 s, t_{1,0} r^3 s\}$$

Note these were simple to compute because  $\phi(g)(e_n) = g e_n g^{-1} = e_n$ . Now what point is this associated with? Consider the first non-identity coset element  $t_{1,0} r$  acting on the origin:  $(0, 0) \rightarrow (0, 0) \rightarrow (1, 0)$ . You'll see similarly that all elements in the coset will follow the same pattern: the first element from  $H_0$  doesn't move the origin (by definition) and the second element is the same in the coset (translation by  $x + 1$ ). Thus, the first coset  $t_{1,0} e_r H_0$  is associated with the point  $(1, 0)$ .

Now consider a coset that involves a  $D_4$  element:  $t_{1,0} r s H_0$ . You can compute its elements as:

$$t_{1,0} r s H_0 = \{t_{1,0} r s, t_{1,0} s, t_{1,0} r^3 s, t_{1,0} r^2 s, t_{1,0} r, t_{1,0} e_r, t_{1,0} r^3, t_{1,0} r^2\}$$

This contains all the same elements as the coset  $t_{1,0} e_r H_0$ ! This is because we have more group elements than space in  $\mathcal{X}$ ; multiple  $g$ 's result in the same coset. This doesn't change our intuition though: the translation transform still defines the connection between our coset and the space. Our lifting function will be

$$f \uparrow^G (g) = f \uparrow^G ((t_{x,y}, h)) = f(x, y)$$

## SO(3) Lie Group

For this example, our function will be points on the sphere  $f(x) = \sum_i \delta(x - x_i)$ . We can represent the group element rotations (among other choices) as being the product of three rotations about the  $y$  and  $z$  axes:  $R_z(\alpha) R_y(\beta) R_z(\gamma)$ . If that seems surprising, remember that rotations are not commutative. Santa lives in the north pole, so let's choose the north pole  $(0, 0, 1)$  as our stabilizer. You cannot choose  $(0, 0, 0)$  remember because it is not in the space. Our subgroup is rotations that only involve  $\gamma$ , for example  $R_z(0) R_y(0) R_z(90)$  is in our subgroup  $H_0$ . Let's generate a coset, say for the group element  $g = R_z(120) R_y(0) R_z(60)$ . The coset  $g H_0$  will be rotations of  $R_z(120) R_y(0) R_z(60) R_z(0) R_y(0) R_z(\gamma)$ , which can be simplified to  $R_z(120) R_y(0) R_z(60 + \gamma)$ . Thus the coset is  $g H_0 = \{R_z(120) R_y(0) R_z(60 + \gamma) \forall \gamma \in [0, 2\pi]\}$

Now what point is associated with this coset? It will be this rotation applied to the origin:  $R_z(120) R_y(0) R_z(60 + \gamma) x_0$ . The first rotation has no effect, by definition, so it becomes  $R_z(120) R_y(0) x_0$ . The general form is that the coset for a point  $x$  is the rotation such that  $R_z(\alpha) R_y(\beta) x_0 = x$ . This quotient space happens to be identical to  $SO(2)$ , rotations in 2D, because it's defined by two angles. The lifting functions is defined as:

$$f \uparrow^G (g) = f \uparrow^G (R_z(\alpha) R_y(\beta) R_z(\gamma)) = f(R_z(\alpha) R_y(\beta) x_0)$$

## 10.7 G-Equivariant Convolutions on Finite Groups

We now have all the tools to build an equivariant network for a finite group. We'll continue with our example group  $Z_6$  on vertices of a hexagon. The cells below does our imports.

```
import numpy as np
import matplotlib.pyplot as plt
import dmol
from dmol import color_cycle
```

Let's start by defining our input function:

```
# make our colors (nothing to do with the model)

vertex_colors = []
for c in color_cycle:
    hex_color = int(c[1:], 16)
    r = hex_color // 256**2
    hex_color = hex_color - r * 256**2
    g = hex_color // 256
    hex_color = hex_color - g * 256
    b = hex_color
    vertex_colors.append((r / 256, g / 256, b / 256))
vertex_colors = np.array(vertex_colors)

def z6_fxn(x):
    return vertex_colors[x]

z6_fxn(0)
```

```
array([0.265625, 0.265625, 0.265625])
```

If we assume our group is indexed already by our vertex coordinates  $\{0, \dots, 5\}$  then our function is already defined on the group. Now we need our trainable kernel function. It will be defined like our other function.

```
# make weights be 3x3 matrices at each group element
# 3x3 so that we have 3 color channels in and 3 out
weights = np.random.normal(size=(6, 3, 3))

def z6_omega(x):
    return weights[x]

z6_omega(3)
```

```
array([[ -0.18718385,  1.53277921,  1.46935877],
       [ 0.15494743,  0.37816252, -0.88778575],
       [-1.98079647, -0.34791215,  0.15634897]])
```

Now we can define our group convolution operator from Equation 8.6. We do need one helper function to get an inverse group element. Remember too that this returns a *function*

```
def z6_inv(g):
    return (6 - g) % 6

def z6_prod(g1, g2):
    return (g1 + g2) % 6

def conv(f, p):
    def out(u):
        g = np.arange(6)
        # einsum is so we can do matrix product for elements of f and g,
        # since we have one matrix per color
        c = np.sum(np.einsum("ij,ijk->ik", f(z6_prod(u, z6_inv(g))), p(g)), axis=0)
        return c

    return out

conv(z6_fxn, z6_omega)(0)
```

```
array([ 1.5752359 ,  3.70837565, -3.68896212])
```

At this point, we can now verify that the CNN is equivariant by comparing transforming the input function and the output function. We'll need to define our function transforms as well.

```
def z6_fxn_trans(g, f):
    return lambda h: f(z6_prod(z6_inv(g), h))

z6_fxn(0), z6_fxn_trans(2, z6_fxn)(0)
```

```
(array([0.265625, 0.265625, 0.265625]),
 array([0.94921875, 0.70703125, 0.3828125 ]))
```

First we'll compute  $\psi[\mathbb{T}_2 f(x)]$  – the network acting on the transformed input function

```
trans_element = 2
trans_input_fxn = z6_fxn_trans(trans_element, z6_fxn)
trans_input_out = conv(trans_input_fxn, z6_omega)
```

Now we compute  $\mathbb{T}_2 \psi[f(x)]$  – the transform acting on the network output

```
output_fxn = conv(z6_fxn, z6_omega)
trans_output_out = z6_fxn_trans(trans_element, output_fxn)

print("g -> psi[f(g)], g -> psi[Tg f(g)], g-> Tg psi[f(g)]")
for i in range(6):
    print(
        i,
        np.round(conv(z6_fxn, z6_omega)(i), 2),
        np.round(trans_input_out(i), 2),
        np.round(trans_output_out(i), 2),
    )
```

```

g -> psi[f(g)], g -> psi[Tgf(g)], g-> Tg psi[f(g)]
0 [ 1.58  3.71 -3.69] [ 4.16  0.82 -2.78] [ 4.16  0.82 -2.78]
1 [ 4.06  2.55 -3.2 ] [ 2.9   2.33 -2.6 ] [ 2.9   2.33 -2.6 ]
2 [ 2.59  2.34 -1.97] [ 1.58  3.71 -3.69] [ 1.58  3.71 -3.69]
3 [ 2.66  2.25 -1.13] [ 4.06  2.55 -3.2 ] [ 4.06  2.55 -3.2 ]
4 [ 4.16  0.82 -2.78] [ 2.59  2.34 -1.97] [ 2.59  2.34 -1.97]
5 [ 2.9   2.33 -2.6 ] [ 2.66  2.25 -1.13] [ 2.66  2.25 -1.13]

```

We can see that the outputs indeed match and therefore our network is G-equivariant. One last detail is that it would be nice to visualize this, so we can add a nonlinearity to remap our output back to color space. Our colors should be between 0 and 1, so we can use a sigmoid to put the activations back to valid colors. I'll hide the input since it contains irrelevant code, but here is the visualization of the previous numbers showing the equivariance.

```

c1 = conv(z6_fxn, z6_omega)
c2 = trans_input_out
c3 = trans_output_out
titles = [
    r"$\psi\left[f(g)\right]$",
    r"$\psi\left[\mathbb{T}_2f(g)\right]$",
    r"$\mathbb{T}_2\psi\left[f(g)\right]$",
]

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

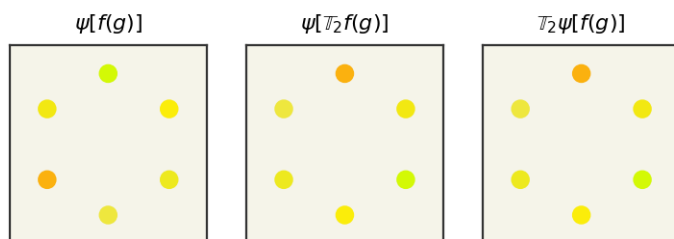
def convert_color(r, g, b):
    h = int(sigmoid(r) * 256**3 + sigmoid(g) * 256**2 + sigmoid(b) * 256)
    return "#{:6X}".format(h)

c1 = [sigmoid(c1(i)) for i in range(6)]
c2 = [sigmoid(c2(i)) for i in range(6)]
c3 = [sigmoid(c3(i)) for i in range(6)]

fig, axs = plt.subplots(1, 3, squeeze=True)
points = np.array(
    [
        (0, 1),
        (0.5 * np.sqrt(3), 0.5),
        (0.5 * np.sqrt(3), -0.5),
        (0, -1),
        (-0.5 * np.sqrt(3), -0.5),
        (-0.5 * np.sqrt(3), 0.5),
    ]
)

for i in range(3):
    axs[i].scatter(points[:, 0], points[:, 1], color=[c1, c2, c3][i])
    # plt.plot([0, points[0,0]], [0, points[0, 1]], color='black', zorder=0)
    axs[i].set_xticks([])
    axs[i].set_yticks([])
    axs[i].set_xlim(-1.4, 1.4)
    axs[i].set_ylim(-1.4, 1.4)
    axs[i].set_aspect("equal")
    axs[i].set_title(titles[i], fontsize=8)
plt.show()

```



As you can see, our output looks the same if we apply the rotation either before or after, so our network is G-equivariant.

## 10.8 G-Equivariant Convolutions with Translation

How can we treat the p4m group? We cannot directly use the continuous convolution definition because the rotations/mirror subgroup is finite and we cannot use the finite convolution because the translation subgroup is locally compact (infinitely many elements). Instead, we will exploit the structure of the group: it is constructed via a semidirect product so each group element is a pair of elements. Namely we can rewrite Equation 8.6 using the constituent subgroups  $N \rtimes H$  and writing elements  $g = hn, g^{-1} = n^{-1}h^{-1}$ .

Remember that  $g = nh$  is fine to use because  $(n, e_r) \cdot (e_n, h) = (n, h)$ , whereas the reverse requires using the conjugation  $\phi(h)(n)$ .

$$(f * \omega)(u) = \sum_{n \in N} \sum_{h \in H} f \uparrow^G (un^{-1}h^{-1}) \omega(hn) \quad (10.12)$$

Now we must treat the fact that there are an infinite number of elements in  $N$  (the translations). We can simply choose the kernel function ( $\omega$ ) to only have support ( $\omega(g) > 0$ ) at locations we want and that will simplify the integration. This may seem ad-hoc – but remember we already made choices like not including 45° rotations. There do exist ways to systematically treat how to narrow the kernels into “neighborhoods” of groups in [FSIW20] or you can find a rigorous derivation specifically for p4 in [RBTH20] or [CW16].

I have a hidden cell below which does a bit of magic. It makes the group elements be hashable. That in turn allows me to cache functions, enabling much faster speeds. This code would be unusable otherwise due to all the nested loops.

Our goal for the p4m group is image data, so we'll limit the support of the kernel to only integer translations (like pixels) and limit the distance to 5 units. This simply reduces our sum over the normal subgroup ( $N$ ). We can now begin our implementation. We'll start by loading an image which will serve as our function. It is a  $32 \times 32$  RGB image. Remember that we need to allow points to have 3 dimensions, where the third dimension is always 1 to accommodate our augmented space.

```
# load image and drop alpha channel
W = 32
try:
    func_vals = plt.imread("quadimg.png")[..., :3]
except FileNotFoundError as e:
    # maybe on google colab
    import urllib.request
```

(continues on next page)

(continued from previous page)

```

urllib.request.urlretrieve(
    "https://raw.githubusercontent.com/whitead/dmol-book/master/dl/quadimg.png",
    "quadimg.png",
)
func_vals = plt.imread("quadimg.png")[:, :, :3]
# we pad it with zeros to show boundary
func_vals = np.pad(
    func_vals, ((1, 1), (1, 1), (0, 0)), mode="constant", constant_values=0.2
)

def pix_func(x):
    # clip & squeeze & round to account for transformed values
    xclip = np.squeeze(np.clip(np.round(x), -W // 2 - 1, W // 2)).astype(int)
    # points are centered, fix that
    xclip += [W // 2, W // 2, 0]
    # add 1 to account for padding
    return func_vals[xclip[:, 0] + 1, xclip[:, 1] + 1]

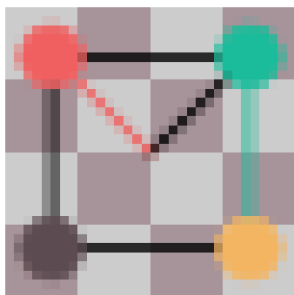
```

```

def plot_func(f, ax=None):
    if ax is None:
        plt.figure(figsize=(2, 2))
        ax = plt.gca()
    gridx, gridy = np.meshgrid(
        np.arange(-W // 2, W // 2), np.arange(-W // 2, W // 2), indexing="ij"
    )
    # make it into batched x,y indices and add dummy 1 indices for augmented space
    batched_idx = np.vstack(
        (gridx.flatten(), gridy.flatten(), np.ones_like(gridx.flatten()))
    ).T
    ax.imshow(f(batched_idx).reshape(W, W, 3), origin="upper")
    ax.axis("off")

plot_func(pix_func)

```



Now let's define our G-function transform so that we can transform our function with group elements. We'll apply a  $rst_{12,-8}$  element to our function.

```

def make_h(rot, mirror):
    """Make h subgroup element"""
    m = np.eye(3)

```

(continues on next page)

(continued from previous page)

```

if mirror:
    m = np.array([[ -1, 0, 0], [0, 1, 0], [0, 0, 1]])
    r = np.array(
        [[np.cos(rot), -np.sin(rot), 0], [np.sin(rot), np.cos(rot), 0], [0, 0, 1]]
    )
    return r @ m

def make_n(dx, dy):
    """Make normal subgroup element"""
    return np.array([[1, 0, dx], [0, 1, dy], [0, 0, 1]])

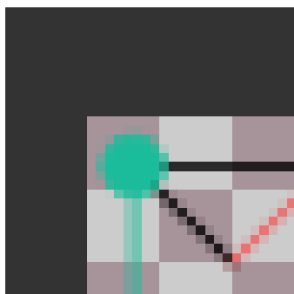
def g_func_trans(g, f):
    """compute g-function transform"""

    @np_cache(maxsize=W*3)
    def fxn(x, g=g, f=f):
        ginv = np.linalg.inv(g)
        return f(ginv.reshape(1, 3, 3) @ x.reshape(-1, 3, 1))

    return fxn

g = make_h(np.pi, 1) @ make_n(12, -8)
tfunc = g_func_trans(g, pix_func)
plot_func(tfunc)

```



Now we need to create our lifting and projecting maps to go from functions over points to functions over group elements. Remember, our lifting function just takes the translation element and makes that point.

```

# enumeraet stabilizer subgroup (rotation/mirrors)
stabilizer = []
for i in range(4):
    for j in range(2):
        stabilizer.append(make_h(i * np.pi / 2, j))

def lift(f):
    """lift f into group"""
    # create new function from original
    # that is f(gx_0)
    @np_cache(maxsize=W*3)
    def fxn(g, f=f):

```

(continues on next page)

(continued from previous page)

```

        return f(g @ np.array([0, 0, 1]))

    return fxn

def project(f):
    """create projected function over space"""

    @np_cache(maxsize=W**3)
    def fxn(x, f=f):
        # x may be batched so we have to allow it to be N x 3
        x = np.array(x).reshape((-1, 3))
        out = np.zeros((x.shape[0], 3))
        for i, xi in enumerate(x):
            # find coset gH
            g = make_n(xi[0], xi[1])
            # loop over coset
            for h in stabilizer:
                ghi = g @ h
                out[i] += f(ghi)
            out[i] /= len(stabilizer)
        return out

    return fxn

# try them out
print("lifted", lift(pix_func)(g))
print("projected", project(lift(pix_func))([12, -8, 0]))

```

```

lifted [0.93333334 0.7176471 0.43137255]
projected [[0.72941178 0.71764708 0.72156864]]

```

We now need to create our kernel functions  $\phi$ . Rather than make a function of the group elements, we'll use indices to represent the different group elements. Remember we need to apply a sigmoid at the end so that we stay in color space.

```

kernel_width = 5 # must be odd
# make some random values for kernel (untrained)
# kernel is group elements x 3 x 3. The group elements are structured (for
  ↪simplicity) as a N x 5 x 5
# the 3 x 3 part is because we have 3 color channels coming in and 3 going out.
kernel = np.random.uniform(
    -0.5, 0.5, size=(len(stabilizer), kernel_width, kernel_width, 3, 3)
)

def conv(f, p=kernel):
    @np_cache(maxsize=W**4)
    def fxn(u):
        # It is possible to do this without inner for
        # loops over convolution (use a standard conv),
        # but we do this for simplicity.
        result = 0
        for hi, h in enumerate(stabilizer):
            for nix in range(-kernel_width // 2, kernel_width // 2 + 1):

```

(continues on next page)



(continued from previous page)

```

        for niy in range(-kernel_width // 2, kernel_width // 2 + 1):
            result += (
                f(u @ make_n(-nix, -niy) @ np.linalg.inv(h))
                @ kernel[hi, nix + kernel_width // 2, niy + kernel_width // 2]
            )
        return sigmoid(result)

    return fxn

# compute convolution
cout = conv(lift(pix_func))
# try it out on a group element
cout(g)

```

```
array([0.08769476, 0.82036708, 0.99128031])
```

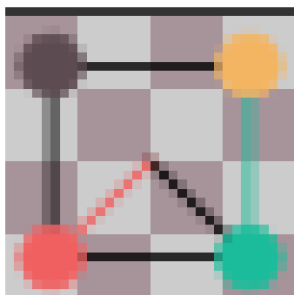
At this point our convolution layer has returned a function over all group elements. We can visualize this by viewing each stabilizer element individually across the normal subgroup. This is like plotting each coset with a choice of representative element.

```

def plot_coset(h, f, ax):
    """plot a function over group elements on cosets given representative g"""
    gridx, gridy = np.meshgrid(
        np.arange(-W // 2, W // 2), np.arange(-W // 2, W // 2), indexing="ij"
    )
    # make it into batched x,y indices and add dummy 1 indices for augmented space
    batched_idx = np.vstack(
        (gridx.flatten(), gridy.flatten(), np.ones_like(gridx.flatten()))
    ).T
    values = np.zeros((W**2, 3))
    for i, bi in enumerate(batched_idx):
        values[i] = f(h @ make_n(bi[0], bi[1]))
    ax.imshow(values.reshape(W, W, 3), origin="upper")
    ax.axis("off")

# try it with mirror
plt.figure(figsize=(2, 2))
plot_coset(make_h(0, 1), lift(pix_func), ax=plt.gca())

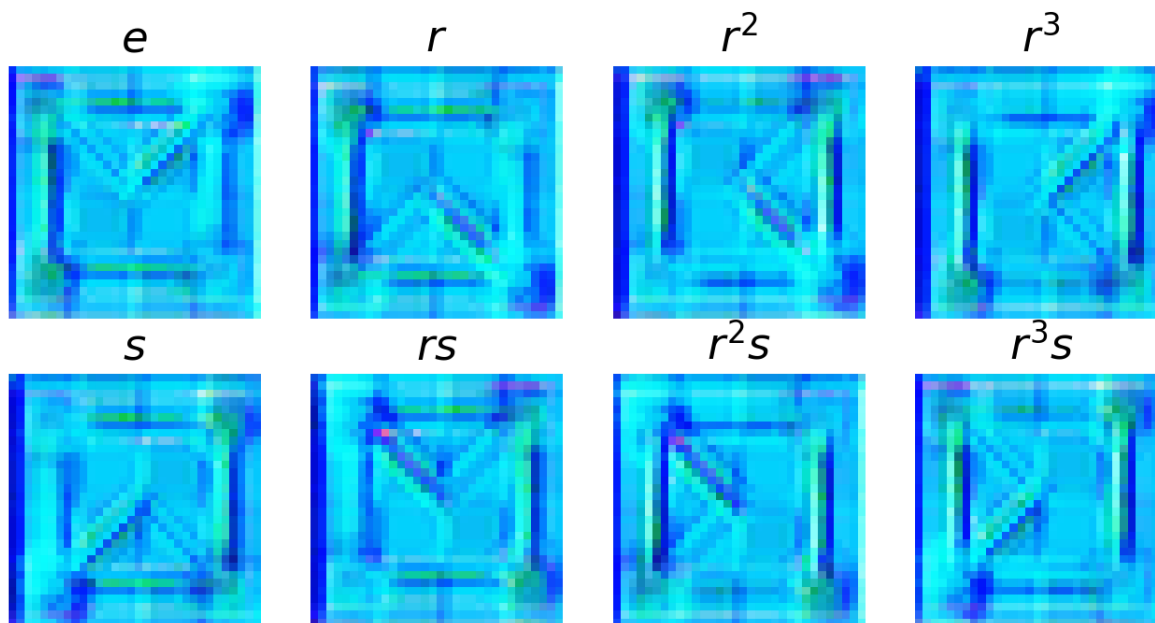
```



Now we will plot our convolution for each possible coset representative. This code is *incredibly* inefficient because we have so many loops in plotting and the convolution. This is where the `np_cache` from above helps.

```

stabilizer_names = ["$e$", "$r$", "$r^2$", "$r^3$", "$s$", "$rs$", "$r^2s$", "$r^3s$"]
fig, axs = plt.subplots(2, 4, figsize=(8, 4))
axs = axs.flatten()
for i, (n, h) in enumerate(zip(stabilizer_names, stabilizer)):
    ax = axs[i]
    plot_coset(h, cout, ax)
    ax.set_title(n)
    
```



These convolutions are untrained, so it's sort of a diffuse random combination of pixels. You can see each piece of the function broken out by stabilizer group element (the rotation/mirroring). We can stack multiple layers of these convolution if we wanted. At the end, we want to get back to our space with the projection. Let us now show our layers are equivariant by applying a G-function transform to input and output.

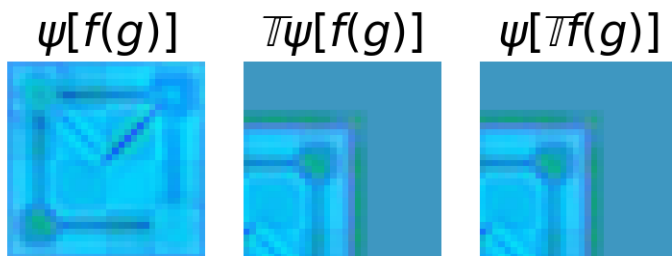
```

fig, axs = plt.subplots(1, 3, squeeze=True)
plot_func(project(cout), ax=axs[0])
axs[0].set_title(r"$\psi\left[f(g)\right]$")

# make a transformation for visualization purposes
g = make_h(np.pi, 0) @ make_n(-10, 16)
tfunc = g_func_trans(g, project(cout))
plot_func(tfunc, ax=axs[1])
axs[1].set_title(r"$\mathbb{T}\psi\left[f(g)\right]$")

tcout = project(conv(lift(g_func_trans(g, pix_func))))

plot_func(tcout, ax=axs[2])
axs[2].set_title(r"$\psi\left[\mathbb{T}f(g)\right]$")
plt.show()
    
```



This shows that the convolution layer is indeed equivariant. Details not covered here are how to do pooling (if desired) and the choice of nonlinearity. You can find more details on this for the p4m group in Cohen et al. [CW16]. This implementation is also quite slow! Kondor et al. [KT18] show how you can reduce the number of operations by identifying sparsity in the convolutions.

## 10.9 Group Representation

p4m was an infinite group but we restricted ourselves to a finite subset. Before we can progress to move to truly infinite locally compact groups, like  $SO(3)$ , we need to learn how to systematically represent the group element binary operation. You can find a detailed description of representation theory in Serre [Ser77] and it is covered in Zee [Zee16]. Thus far, we've discussed the group actions – how they affect a point. Now we need to describe how to represent them with matrices. This will be a very quick overview of this topic, but representation of groups is a large area with well-established references. There is specifically a great amount of literature about building up these representations, but we'll try to focus on using them since you generally can look-up the representations for most groups we'll operate in.

Let us first define a representation on a group:

### Linear Representation of a Group

Let  $G$  be a group on an  $n$ -dimensional vector space  $\mathcal{X}$ . A linear representation of  $G$  is a group homomorphism:  $\rho : G \rightarrow GL(m, \mathbb{C})$  where  $GL(m, \mathbb{C})$  is the space of  $m \times m$  square invertible matrices with complex numbers. The representation  $\rho$  should satisfy the following equation

$$\rho(g_1 \cdot g_2) = \rho(g_1) \rho(g_2) \quad \forall g_1, g_2 \in G \quad (10.13)$$

where the term  $\rho(g_1) \rho(g_2)$  is a matrix product.

There are a few things to note about this definition. First, the representation assigns matrices to group elements in such a way that multiplying matrices gives the same representation as getting the representation of the binary operation ( $\rho(g_1 \cdot g_2)$ ). Second, the matrices have to be square and invertible. This follows from the requirement that group elements must have an inverse, so naturally we need invertible matrices. The invertible requirement also means often need to allow complex numbers. Third, the **degree** of the representation ( $m$ ) need not be the same size as the vector space.

There is a big detail missing from this definition. Does this have anything to do with how the group element affect a point? No. Consider that  $\rho(g_i) = 1$  is a valid representation, as in it satisfies the definition. Yet 1 is not the correct way to transform points with group elements. If we go further and say that the representation is *injective* (one to one), then we must have a unique representation for every group element. That is called a **faithful representation**. This is better, but it turns out there are still multiple faithful representations for a group.

Remember the way a group affects a point is a **group action**, which maps from the direct product of  $G, \mathcal{X}$  (i.e., a tuple like  $(g_2, x)$  to  $\mathcal{X}$ ). A group action, if it is linear, can also be a representation. Consider that we write the group action  $\pi$  (how a group element affects a point) as  $\pi(g)(x) = x'$ . You can convert this into a square matrix in  $\mathcal{X} \times \mathcal{X}$  by considering how each element of  $x'$  is affected the element in  $x$ . This matrix can be further shown to be in  $GL(m, \mathcal{X})$  and a representation by relying on its linearity. There isn't a special word for this, but often groups are defined in terms

of these special matrices that both transforms points and are valid representations (e.g.,  $SO(3)$ ). They are then called the **defining representation** or **fundamental representation**.

Let's now see group representations on the examples above that are both group actions and representations.

## Finite Group $Z_6$

Our group action defined above was modular arithmetic, which is not linear and so we cannot use it to construct representation. There are multiple representation for cyclic groups like  $Z_6$ . If you're comfortable with complex numbers, you can build circulant matrices of 6th roots of unity. If that confuses you, like it does me, then you can also just view this group like a rotation group. Just like how if you rotate enough times you get back to the beginning, you can also use rotation matrices of  $360/6 = 60^\circ$ . This requires a 2D vector representation though for the space. With this choice, a representation is:

$$\begin{bmatrix} \cos \frac{k2\pi}{6} & -\sin \frac{k2\pi}{6} \\ \sin \frac{k2\pi}{6} & \cos \frac{k2\pi}{6} \end{bmatrix}, k \in \{0, 1, 2, 3, 4, 5\}$$

Let's verify that this is a representation by checking that  $r^2 \cdot r^4 = e$

$$\begin{bmatrix} \cos \frac{4\pi}{6} & -\sin \frac{4\pi}{6} \\ \sin \frac{4\pi}{6} & \cos \frac{4\pi}{6} \end{bmatrix} \begin{bmatrix} \cos \frac{8\pi}{6} & -\sin \frac{8\pi}{6} \\ \sin \frac{8\pi}{6} & \cos \frac{8\pi}{6} \end{bmatrix} = \begin{bmatrix} \cos \frac{12\pi}{6} & -\sin \frac{12\pi}{6} \\ \sin \frac{12\pi}{6} & \cos \frac{12\pi}{6} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

You can also verify that this is a group action by repeated application to the point  $(1, 0)$ , which will rotate around the unit circle.

## Locally Compact p4m

Our group action defined above for the translation elements is not linear. To define a representation, we can use **Affine Matrices** which are  $3 \times 3$  invertible square matrices. That means even though our goal is 2D data, we need to introduce a 3rd dimension:  $(x, y, 1)$ . The 3rd dimension is always 1 and is called the augmented dimension. To specify a group representation we simply need to multiply an affine matrix for rotation, reflection, and translation (*in that order!*). These are:

Rotation:

$$\begin{bmatrix} \cos \frac{k\pi}{4} & -\sin \frac{k\pi}{4} & 0 \\ \sin \frac{k\pi}{4} & \cos \frac{k\pi}{4} & 0 \\ 0 & 0 & 1 \end{bmatrix}, k \in \{0, 1, 2, 3\}$$

Reflection:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation:

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

It is a bit more involved to verify this is a group representation, but you can try a few group element products to convince yourself. Do not forget the special homomorphism (conjugate  $\phi(h)(n)$ ) for semidirect products when multiplying group element, which ensures the correct behavior if rearrange the order of the matrices.

## SO(3) Group

A representation of the  $SO(3)$  group is just its usual group action: the product of 3 3D rotation matrices  $R_z(\alpha)R_y(\beta)R_z(\gamma)$  where  $\alpha, \gamma \in [0, 2\pi]$ ,  $\beta \in [0, \pi]$  and the matrices are defined above.

### 10.9.1 Unitary Representations

One minor detail is that if we have some representation  $\rho(g_1)\rho(g_2) = \rho(g_1 \cdot g_2)$ , then we could make a “new” representation  $\rho'$  by inserting some invertible matrix  $\mathbf{S}$ :

$$\rho'(g) = \mathbf{S}^{-1}\rho(g)\mathbf{S}$$

because

$$\begin{aligned}\rho'(g_1)\rho'(g_2) &= \mathbf{S}^{-1}\rho(g_1)\mathbf{S}\mathbf{S}^{-1}\rho(g_2)\mathbf{S} \\ &= \mathbf{S}^{-1}\rho(g_1)\rho(g_2)\mathbf{S} = \rho'(g_1 \cdot g_2)\end{aligned}$$

There is a theorem, the Unitarity Theorem, that says we can always choose an  $\mathbf{S}$  (for finite groups) such that we make our representation **unitary**. Unitary means that  $\rho(g)^{-1} = \rho^\dagger(g)$  for any  $g$ . Remember that  $\rho(g)$  is a matrix, so  $\rho^\dagger(g)$  is the adjoint (transpose and complex conjugate) of the matrix. Thus, without any loss of generality we can assume all representations we use are unitary or can be trivially converted to unitary.

### 10.9.2 Irreducible representations

These representations that both describe the group action and how group elements affect on another are typically **reducible**, meaning if you drop the requirement that they also describe group action they can be simplified. The process of reducing representations is again a topic better explored in other references [Ser77], but here I will sketch out the important ideas. The main idea is that we can form decomposable unitary representation matrices that are composed of smaller block matrices and zero blocks. These smaller blocks,  $\rho_i(g)$ , are *irreducible* — they cannot be broken into smaller blocks and zeros

$$\rho(g) = \mathbf{S}^{-1} \begin{pmatrix} \rho_0(g) & 0 & \dots & 0 \\ 0 & \rho_1(g) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \rho_k(g) \end{pmatrix} \mathbf{S} \quad (10.14)$$

This block notation is consistent, regardless of  $g$ . That is a strong statement because  $\rho(g_1)\rho(g_2)$  should give back an element in  $\mathcal{G}$  —  $\rho(g')$  — with the same block structure. What is interesting about this notation is that each block is then *itself* a representation. We could just pick  $\rho_0(g)$  as a representation, and if this block structure is true for all  $g$ , then  $\rho_0(g_1)\rho_0(g_2)$  should give back something with non-zero elements only in the rows/columns of the  $\rho_0(g)$  block. We could also combine  $\rho_0(g)$  and  $\rho_1(g)$  or even  $\rho_0(g)$  and  $\rho_2(g)$ . Thus, these irreducible representations (**irreps**) are the pieces that we use to build any other representation. The irreducible representations are all dimension 1 if  $\mathcal{G}$  is abelian, but otherwise irreducible representations are square matrices.

To add some notation, we use **direct sums** to write the bigger unitary representation:

$$\rho(g) = \mathbf{S}^{-1} \begin{pmatrix} \rho_0(g) & 0 & \dots & 0 \\ 0 & \rho_1(g) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \rho_k(g) \end{pmatrix} \mathbf{S} = \rho_0(g) \oplus \rho_1(g) \oplus \dots \oplus \rho_k(g) \quad (10.15)$$

and we could just stop the direct sum wherever we would like. The number of irreducible representations is finite for finite groups and infinite for locally compact groups. These irreducible representations are like orthonormal basis-functions or basis-vectors from Hilbert spaces. From the Peter-Weyl theorem, they specifically can be transformed to create a complete

basis-set for integrable ( $L^2$ ) functions of the group. “Transformed” because irreducible representations are representations of  $g$  (output a matrix), but we need them to output a scalar to be a basis-set for an integrable function.

Where do we get these integrable functions? Recall we can use lifting to move functions of our space to our group and then these irreducible representations enable us to represent the functions as a (direct) sum of coefficients of the irreducible representations. Remember, each individual irreducible representation is itself a valid representation, but they are not all faithful and so you need some of them to uniquely represent all group elements and all of them to represent arbitrary functions over the group. One final note, these irreducible representations have been essentially mapped out for all groups and thus we look them up in table rather than try to construct them.

## 10.10 G-Equivariant Convolutions on Compact Groups

Now we can represent functions on groups as a direct sum (list of increasing length vectors) of coefficients on the irreps. Namely  $f(g) = f_0 \cdot \rho_0(g) \oplus \vec{f}_1 \cdot \rho_1(g) \oplus \dots \oplus \vec{f}_k \cdot \rho_k(g)$ . Remember the direct sum notation  $\oplus$  is just a shorthand that means put all this stuff into a big matrix of increasingly large blocks. The individual  $\vec{f}_i$ s are called **fragments** to distinguish them from the actual irreps (which are functions).

Since the group is not changing, we could even more compactly write this as  $f(g) = f_0 \oplus \vec{f}_1 \oplus \dots \oplus \vec{f}_k$ . We’d like to revise the G-Equivariant convolution layer equation:

$$\psi(f) = (f * \omega)(u) = \int_G f \uparrow^G (ug^{-1}) \omega \uparrow^G (g) d\mu(g) \quad (10.16)$$

to use irreps now. *It turns out that the convolutional integral becomes a product of irreps.* [KT18]. Our expression simplifies to:

$$\psi(f) = f_0 w_0 \oplus \vec{f}_1 w_1 \oplus \dots \oplus \vec{f}_k w_k \quad (10.17)$$

This result says we just multiply the irreducible representations by weights, but do not mix across irreps. The weights become matrices if we start to allow multiple channels (multiple fragments). An important point then is how we actually can learn if there is no communication between irreps. That’s where the nonlinearity comes in. It is discussed in more depth below, but the most common nonlinearity is to take a tensor product (all irreps times all irreps) and then reduce that by multiplying the larger rank tensor by a special tensor for the group called Clebsch-Gordan coefficients that reduces it equivariantly back down to the direct sum of irreps. This enables mixing between the irreps and is nonlinear.

### 10.10.1 Irreducible representations on SO(3)

There is an infinite sequence of possible irreducible representations for the SO(3) group known as the Wigner D-matrices. They must be of odd dimension, and so are traditionally written as the sequence  $2l + 1$  where  $l$  is an integer that serves as the irreducible representation index. The Wigner D-matrices are square with dimension  $2l + 1$  and are a function of the group element (e.g., the angles of the rotation). This may be surprising, that the irreducible representations can be of greater dimension than our reducible representation  $R_z(\alpha)R_y(\beta)R_z(\gamma)$ . The easiest way to think about this is to consider SO(3) acting on 3 dimensional  $n$ th degree monomials rather than points:  $x^i y^j z^k$  where  $l = i + j + k$ . The trivial representation works on the 0th degree monomial ( $l = 0$ ), the  $l = 1$  irrep has three possible monomials ( $x, y, z$ ), the  $l = 2$  irrep has 5 possible monomials (excluding a redundant term). You can find a nice description of [this here](#).

What  $l$  should you choose? It depends on your input and output. If you choose  $l = 0$ , you can only represent scalars (not points/vectors). If you choose  $l = 1$ , you can represent vectors. You can always pick a larger  $l$ , but if you pick a lower  $l$  you will become invariant to the higher-order geometric structure.

The reason you just plug in the values into the spherical harmonics is you’re (1) turning your data into a dirac-delta function and (2) taking its integral over all the irreps, which turns out to be the same as plugging in the value.

Our choice of Euler Angles (zyz rotation) means that the Wigner D-matrices turn into spherical harmonics. Now how do we get our input data, into the irrep for the group SO(3)? You just plug the input coordinates/features into the spherical harmonics. Have multiple scalar features (e.g., charge, atomic number) – you simply add another axis to the irreps and create multiple “channels”. Another detail is that our weight size seems to be set by the irrep size. How do we get wider layers (more weights)? The same way: by adding more channels to each irrep.

### 10.10.2 SO(3) Nonlinearity & Mixing

There are two equations for equivariant nonlinearity in SO(3), and they are sometimes combined. The first nonlinearity is a **Clebsch-Gordan tensor product** and enables mixing between irreps. The equation is

$$\vec{f}'_i = \sum_j \sum_k \text{CG}_{j,k,i} \cdot \vec{f}_j \vec{f}_k \quad (10.18)$$

where  $\text{CG}_{j,k,i}$  are the Clebsch-Gordan coefficients that ensure we maintain equivariance after multiplying all irreps with all irreps (and do a change of basis). This expression is sometimes written as  $\text{CG}_{j,k,i} \vec{f} \otimes \vec{f}$

As before,  $\vec{f}_i$  are the fragments (coefficients) on the irreps that represent our function  $f(g)$  of the group. The fragments are usually computed directly by plugging in the coordinates into spherical harmonic equations. [KLT18] showed that this is itself nonlinear, and thus a complete layer with nonlinearity would be that equation combined with Equation (10.17). We may also choose to skip some of the terms, since this is an expensive equation.

There is another kind nonlinearity that is equivariant called gated nonlinearities [WGW+18]. The equation is simple; just compute the magnitude of each of the irrep fragments  $\vec{f}_i$  and put it through a traditional neural network nonlinearity (e.g., ReLU):

$$\sigma_{\text{gated}}(\vec{f}_i) = \sigma(|\vec{f}_i|) \vec{f}_i$$

The gated nonlinearity is sometimes used instead of Equation (10.18) or as an extra step after it [TSK+18].

At the end of the network, most of the time we simply take the  $f_0$  scalar (the  $l = 0$  irrep fragment). We may have multiple channels, so we don't have one  $f_0$ . But, often we are doing classification or predicting energy (and its derivative, forces) and thus want a  $l = 0$  feature. The Clebsch-Gordan tensor products are essential to ensure mixing between the spatial information at the higher dimensional irrep fragments and scalar input features like atomic number.

## 10.11 SO(3) Equivariant Example

Let's implement a non-differentiable version of the equations above for the SO(3) group. To begin, let's write the code to convert our points into their irreps. Our code is not differentiable, so we won't be able to train.

```
from scipy.special import sph_harm

def cart2irreps(x, l):
    # convert to spherical coords to eval
    N = x.shape[0]
    r = np.linalg.norm(x, axis=-1)
    azimuth = np.arctan2(x[:, 1], x[:, 0])
    polar = np.arccos(x[:, 2] / r)
    f = []
    for li in range(1):
        fi = []
        for m in range(-li, li + 1):
```

(continues on next page)

(continued from previous page)

```

        y = sph_harm(m, li, azimuth, polar)
        # convert to real
        if m < 0:
            y = np.sqrt(2) * (-1) ** m * y.imag
        elif m > 0:
            y = np.sqrt(2) * (-1) ** m * y.real
        fi.append(y.real)
    fi = np.array(fi)
    f.append(fi.reshape(N, 2 * li + 1))
return f

def print_irreps(f):
    for i in range(len(f)):
        print("irrep", i)
        print(f[i])

points = np.random.rand(2, 3)
# make them be on unit sphere
points /= np.linalg.norm(points, axis=-1)[:, np.newaxis]
M = 3 # number of irreps
f = cart2irreps(points, M)
print_irreps(f)

```

```

irrep 0
[[0.28209479]
 [0.28209479]]
irrep 1
[[0.30938929 0.35978765 0.34151954]
 [0.25147992 0.16240408 0.21457659]]
irrep 2
[[-0.229949   -0.3533116   0.48355975  0.41407486  0.14687347]
 [-0.06474223  0.25382933  0.24695337 -0.15868095 -0.19084746]]

```

We chose to use 3 irreps here and get a fragment vector for each particle at each irrep. This gives a scalar for  $l = 0$  irrep, a 3 dimensional vector for  $l = 1$ , and a 5 dimensional vector for  $l = 2$  irrep. It's a choice, but usually you'll see networks encode a 3D point into the  $l = 1$  irrep because it's the smallest irrep that won't become invariant. This also allows a sort of separation of input features, where we can put scalar properties like mass or element into the  $l = 0$  irrep and the point position into the  $l = 1$  irrep. Often, you'll also see multiple channels (multiple sets of fragments at a given irrep) to add expressiveness.

Notice that  $l = 0$  is the same for both points - that's because that spherical harmonic is constant. Another reason why we put scalar quantities there.

Let's now implement the linear part - Equation (10.17). We'll have channels now, since otherwise we get a single weight.

```

def linear(f, W):
    for l in range(len(f)):
        # promote to have channels, if not yet
        if len(f[l].shape) == 2:
            f[l] = f[l][..., None]
        f[l] = np.einsum("ijk,kl->ijl", f[l], W[l])
    return f

```

(continues on next page)



(continued from previous page)

```
def init_weights(cin, cout):
    return np.random.randn(M, cin, cout)
```

```
weights = init_weights(1, 4)
h = linear(f, weights)
print(h)
```

```
[array([[[-0.31297784, -0.154452, 0.18786586, -0.71498461]]],
       [[[-0.31297784, -0.154452, 0.18786586, -0.71498461]]], array([[[-0.
↪42546735, 0.15500163, -0.14858391, 0.28962165],
       [-0.49477439, 0.18025082, -0.17278767, 0.33679993],
       [-0.46965237, 0.17109864, -0.16401443, 0.31969902]]],

       [[[-0.34583128, 0.12598949, -0.12077299, 0.23541225],
       [-0.22333557, 0.08136318, -0.0779944, 0.15202768],
       [-0.2950824, 0.10750121, -0.1030502, 0.20086677]]], array([[[-0.
↪1860702, 0.27550026, -0.09351039, -0.27632922],
       [-0.28589279, 0.42330012, -0.14367667, -0.4245738 ],
       [ 0.39128702, -0.57934949, 0.196643, 0.58109271],
       [ 0.33506122, -0.49610014, 0.16838648, 0.49759287],
       [ 0.11884712, -0.17596806, 0.05972714, 0.17649754]]],

       [[[-0.05238814, 0.0775672, -0.02632789, -0.0778006 ],
       [ 0.20539369, -0.30411111, 0.10322149, 0.30502616],
       [ 0.1998298, -0.29587308, 0.10042534, 0.29676334],
       [-0.1284015, 0.19011452, -0.06452873, -0.19068656],
       [-0.15443, 0.22865298, -0.07760947, -0.22934098]]]])]
```

You can see that we now have multiple channels at each irrep.

Now we'll implement the Clebsch-Gordan nonlinearity, Equation (10.18). We'll use the coefficients in sympy.

```
from sympy.physics.quantum.cg import CG
from sympy import S
from functools import lru_cache

# to speed-up repeated calls, put a cache around it
@lru_cache
def cg(i, j, k, l, m, n):
    # to get a float, we wrap input in symbol (S), call
    # doit, and evalf.
    r = CG(S(i), S(j), S(k), S(l), S(m), S(n)).doit().evalf()
    return float(r)

# As you can see, the Clebsch-Gordan nonlinearity is a lot!!
def cgnl(f):
    output = [np.zeros_like(fi) for fi in f]
    # m,n -> outputs
    for i in range(len(f)):
        for j in range(-i, i + 1):
```

(continues on next page)

(continued from previous page)

```

    for k in range(len(f)):
        for l in range(-k, k + 1):
            for m in range(len(f)):
                for n in range(-m, m + 1):
                    output[m][:, n] += (
                        f[l][:, j] * f[k][:, l] * cg(i, j, k, l, m, n)
                    )
    return output

```

cgnl(h)

```

[array([[0.42489967, 0.4032353 , 0.0964734 , 0.94690111]],
       [[0.11165005, 0.03936491, 0.03783205, 0.52913038]]]),
array([[ 0.24885549, -0.05730307, -0.05889344, -0.39649116],
       [-0.05000888, -0.24971277, -0.12805358, -0.11797142],
       [ 0.14781777, -0.13169458, -0.08727785, -0.30940054]]),

array([[ 0.16494686, -0.06671324, -0.05442168, -0.28454164],
       [ 0.13365903, -0.0284449 , -0.03038242, -0.21118876],
       [ 0.18040701, -0.0355278 , -0.03947416, -0.28288486]]]),
array([[ 0.55749603,  0.18656926,  0.03216769,  0.77978823],
       [ 0.50911544, -0.02010282, -0.00947985,  0.81664941],
       [-0.13146835, -0.07648033,  0.07056881, -1.00717992],
       [-0.06505638,  0.01613839,  0.07100045, -0.77672149],
       [ 0.09459735, -0.15717732,  0.02821465, -0.37198278]]),

array([[ 0.1384026 , -0.10437677, -0.00300055,  0.08031154],
       [-0.05399183,  0.03246938,  0.04335287, -0.46199106],
       [-0.11401494,  0.01293701,  0.03401445, -0.48684213],
       [ 0.15902584, -0.06563252, -0.01575307,  0.29445021],
       [ 0.2021304 , -0.13665679, -0.02137265,  0.30913128]]])

```

Now we can make our complete layer! We won't use a gated nonlinearity here, just the Clebsch-Gordan nonlinearity.

```

def cg_net(x, W, l, num_layers):
    f = cart2irreps(x, l)
    for i in range(num_layers):
        f = linear(f, W[i])
        f = cgnl(f)
    return np.squeeze(f[0])

num_layers = 3
L = 3
channels = 4
weights = (
    [init_weights(1, channels)]
    + [init_weights(channels, channels) for _ in range(num_layers - 2)]
    + [init_weights(channels, 1)]
)

cg_net(points, weights, L, num_layers)

```

```
array([5.71056675e+01, 2.06055946e-02])
```

Now we have our irrep features. How do we get an output? If we're trying to output a scalar (regression/classification), we would just take the  $l = 0$  irrep fragment. If we instead would like something more geometric, we can integrate over the spherical harmonics (analogous to inverse Fourier Transform). Or, you can read out the point at which the spherical harmonics are maximized.

Let us now check that our network is indeed invariant (we're outputting a single value per point, so invariant). We'll make a rotation and check if our output changes.

```
# random 3x3 matrix
R = np.random.rand(3, 3)
# make it a member of SO(3)
U, _, V = np.linalg.svd(R)
R = np.dot(U, V)
```

```
print(cg_net(points, weights, L, num_layers))
print(cg_net(points @ R, weights, L, num_layers))
```

```
[5.71056675e+01 2.06055946e-02]
[4.52543928 0.30593967]
```

*As you can see, something is broken and I need to fix it when I have time.*

## 10.12 Equivariant Neural Networks with Constraints

You do not need to use irreducible representations. It is currently in 2022 the dominant paradigm due to its good accuracy. One alternative is to work in the defining/faithful representation and put equivariant constraints on your network weights. This approach is quite nice because the implementation is independent of the group. It also works for finite groups. Let's see an example of this approach via the library released by the authors called Equivariant MLP (emlp)[FWW21]

We'll create an  $SO(3)$  equivariant neural network and check that it is equivariant to rotations. We begin by defining our group and its representation. I'll show a few elements too, to demonstrate that this is the faithful representation and not the irreducible.

```
from emlp.groups import SO, S
import emlp.reps as reps
import emlp
import haiku as hk
import emlp.nn.haiku as ehk
import jax.numpy as jnp

so3_rep = reps.V(SO(3))
# grab a random group element
sampled_g = SO(3).sample()
dense_rep = so3_rep.rho(sampled_g)
# check its a member of SO(3)
# g @ g.T = I
print(dense_rep @ dense_rep.T)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)
```

```
[[ 1.0000001e+00 -9.0145235e-08 -3.6942627e-08]
 [-9.0145235e-08  1.0000000e+00  3.4498189e-08]
 [-3.6942627e-08  3.4498189e-08  9.9999994e-01]]
```

Now we'll apply our group element to a point to see it rotate the point. The norm should be unchanged, because it's a rotation.

```
point = np.array([0, 0, 1])
print("new point", dense_rep @ point.T)
print("norm", np.sqrt(np.sum((dense_rep @ point) ** 2)))
```

```
new point [ 0.2110719 -0.73052925 -0.6494426 ]
norm 1.0
```

Now let's assume our input function consists of 5 points (e.g., methanol molecule) defined by features (e.g., 1D element embedding) and positions. We'll create that as a direct sum of 5 scalars and 5 vectors. Our output will be a vector (e.g., dipole). Equivariance here will then mean that if rotate the input points, our output vector should undergo the same rotation.

```
input_rep = 5 * so3_rep**0 + 5 * so3_rep**1
print("input rep", input_rep)
print("output rep", so3_rep)

input_point = np.random.randn(5 + 5 * 3)
print("input features", input_point[:5])
print("input positions\n", input_point[5:].reshape(5, 3))
```

```
input rep 5V0+5V
output rep V
input features [-0.30458234 -1.89067604  0.97765169 -0.18302214  0.59400493]
input positions
[[ 0.54096229 -0.44211156 -0.5074754 ]
 [-0.71190593  1.1881366  -1.81726978]
 [-1.48572616 -0.69015372  3.02716867]
 [-1.24053084  0.07303629  0.19194001]
 [ 0.59835643 -0.70481736  1.01966388]]
```

```
model = emlp.nn.EMLP(input_rep, so3_rep, group=SO(3), num_layers=1)
output_point = model(input_point)
print("output", output_point)
```

```
output [ 0.0028723  0.00271391 -0.02050202]
```

Now we'll transform the input points according to a random element in the group. We could convert the input into the five spatial vectors and apply the group element to them individually and put them back together. However, `emlp` has a convenience function for exactly that. We can change our group element to the input representation.

```
trans_input_point = input_rep.rho_dense(sampled_g) @ input_point
print("transformed input features", trans_input_point[:5])
print("transformed input positions\n", trans_input_point[5:].reshape(5, 3))
```

```
transformed input features [-0.30458233 -1.890676    0.9776517 -0.18302214  0.
↪5940049 ]
transformed input positions
[[ 0.5648738  0.3965097  0.5189717 ]
 [-1.7186793  0.9593498  1.1604906 ]
 [ 0.20919645 -1.2322842 -3.207048  ]
 [-0.8010346  0.31936038 -0.91512   ]
 [ 1.1189423 -0.5973959 -0.5344146  ]]
```

Now we compare running the transformed input through the model against applying the group element to the output from the untransformed input.

```
model(trans_input_point), sampled_g @ output_point
```

```
(DeviceArray([-0.00452228,  0.01232371,  0.01623649], dtype=float32),
 DeviceArray([-0.00452229,  0.01232371,  0.01623649], dtype=float32))
```

Indeed they are equivalent – meaning this model is equivariant. The constraint approach is quite simple to use and can handle arbitrary groups. However, it may not be efficient when working with many input points (like a protein) and it may make sense to use an implementation specific to  $E(3)$  or  $SO(3)$ .

### 10.12.1 How the constraints work

How does this magic happen? Rather than explicitly setting constraints on the dense layer weights, `emlp` always first projects the network weights into an **equivariant subspace**. This means that the cost of equivariance is paid when constructing the model when this projection matrix is found but not later during training and inference. The equivariant subspace is the space of allowed weights that respect the equivariance. Let's see what this looks like.

Recall that a dense layer has the equation:

$$y = \sigma(Wx + b) \quad (10.19)$$

where  $\sigma$  is a special nonlinearity for equivariant neural networks we won't discuss here (see [WGW+18]). To respect the equivariance,  $W, b$  will need to be projected into an equivariant subspace that depends on our group and input/output representations. So our modified equation would look like:

$$y = \sigma(P_w Wx + P_b b) \quad (10.20)$$

Let's start by making these projectors.  $P_b$  only will need to consider the output rep, since  $b$  is the bias (same representation as output).

```
Pw = (input_rep >> so3_rep).equivariant_projector()
Pb = (so3_rep).equivariant_projector()

print("Pw shape is", Pw.shape, "Pb shape is", Pb.shape)
```

```
Pw shape is (60, 60) Pb shape is (3, 3)
```

Note that they are square because they should leave the underlying dimension of  $W$  unchanged – we are not projecting to a reduce dimension, but a subspace within the space of possible values of the weights. Remember too our representations are flattened - that 60 comes from the fact that our weight matrix is  $3 \times (5 + 15)$ .

Now let's show how these projectors can convert an arbitrary weight matrix into one that is equivariant.

```
W = np.random.randn(3, 5 + 5 * 3)
b = np.random.randn(3)

print(
    "W is not alone equivariant",
    W @ trans_input_point.flatten(),
    "!=" ,
    sampled_g @ W @ input_point,
)
```

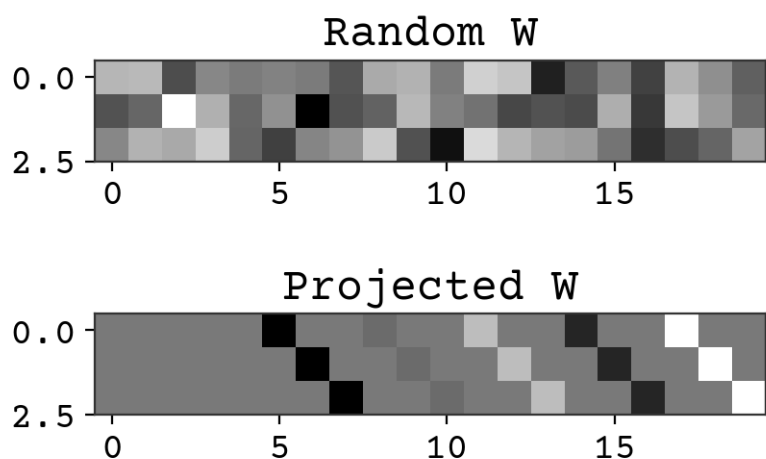
```
W is not alone equivariant [ -3.1314962 -12.084191  11.542714 ] != [12.0744295 -2.
↪836445   3.0263994]
```

```
Proj_W = (Pw @ W.flatten()).reshape(W.shape)
print(
    "Projected W is equivariant",
    Proj_W @ trans_input_point.flatten(),
    "==",
    sampled_g @ Proj_W @ input_point,
)
```

```
Projected W is equivariant [-1.0771619  1.4464636  1.667451 ] == [-1.077162   1.
↪4464636  1.6674511]
```

You may be wondering how much the projection affects  $W$ . Is there enough flexibility that you can learn? We can compare the full *random* matrix  $W$  vs it's projection.

```
plt.title("Random W")
plt.imshow(W)
plt.show()
plt.title("Projected W")
plt.imshow(Proj_W)
plt.show()
```



It appears that there are only a few unique values in  $W$  after projection, so that our weight space is effectively much lower dimensional. This is why it's important to have multiple channels! This also demonstrates why `emlp` can be more expensive. We're training 180 values but we could have just used a few. Similarly, the projected bias is zero for our system.

```
Pb @ b
```

```
DeviceArray([0., 0., 0.], dtype=float32)
```

## 10.12.2 Including Permutation Groups

In real molecules, we also need to have permutation equivariance with respect to the atom ordering and bond ordering – which is not true of our above example about computing dipole moment. `emlp` also supports permutation groups, which are usually written as  $S_n$ , where  $n$  is the number of permutable elements in the group. We'll work on that in the next chapter.

## 10.13 Chapter Summary

- Equivariant neural networks guarantee equivariance by construction for arbitrary groups, which removes the need to align trajectories, work in special coordinate systems, or use pairwise distances.
- Equivariance can be achieved by parameter sharing or testing/training data augmentation, but here we focused on equivariant layers that can be composed into a neural network.
- Equivariance requires definition of a group and homogeneous space. We must view our input data as functions and our models as operators that transform functions.
- Finite groups can be treated with G-equivariant layers that have an additional sum across the number of group elements.
- Infinite groups like  $SO(3)$  can be made finite by working with a direct sum (list of vectors) of the irreducible representations. This requires converting the input data though to the irreducible representation and there are complexities in nonlinearities and implementations typically must be written per-group.
- Constraint-based equivariant layers are flexible, general, and quick to implement but may not scale well with respect to size of input group or number of points.
- Recent work has also shown you can put irreducible representation direct sums into the edges of graph neural networks, gaining input size independence, permutation invariance, and spatial equivariance in one model.

## 10.14 Relevant Videos

### 10.14.1 Intro to Geometric Deep Learning

### 10.14.2 Equivariant Networks

### 10.14.3 Equivariant Network Tutorial

[watch here](#)

## 10.15 Cited References



## MODERN MOLECULAR NNS

We have seen two chapters about equivariances in *Input Data & Equivariances* and *Equivariant Neural Networks*. We have seen one chapter on dealing with molecules as objects with permutation equivariance *Graph Neural Networks*. We will combine these ideas and create neural networks that can treat arbitrary molecules with point clouds and permutation equivariance. We already saw SchNet is able to do this by working with an invariant point cloud representation (distance to atoms), but modern networks mix in ideas from *Equivariant Neural Networks*. This is a highly-active research area, especially for predicting energies, forces, and relaxed structures of molecules.

---

### Audience & Objectives

This chapter assumes you have read *Input Data & Equivariances*, *Equivariant Neural Networks*, and *Graph Neural Networks*. You should be able to

- Categorize a task (features/labels) by equivariance
- Understand body-ordered expansions
- Differentiate models based on their message passing, message type, and body-ordering

---

<b>Warning:</b> This chapter is in progress
---



## EXPLAINING PREDICTIONS

Neural network predictions are not interpretable in general. In this chapter, we explore how to explain predictions. This is part of the broader topic of explainable AI (XAI). These explanations should help us understand why particular predictions are made. This is a critical topic because being able to understand model predictions is justified from a practical, theoretical, and increasingly a regulatory stand-point. It is practical because it has been shown that people are more likely to use predictions of a model if they can understand the rationale [LS04]. Another practical concern is that correctly implementing methods is much easier when one can understand how a model arrived at a prediction. A theoretical justification for transparency is that it can help identify incompleteness in model domains (i.e., covariate shift)[DVK17]. It is now becoming a compliance problem because both the European Union [GF17] and the G20 [Dev19] have recently adopted guidelines that recommend or require explanations for machine predictions. The European Union is considering going further with more [strict draft legislation](#) being considered.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Deep Learning on Sequences*. It also assumes a good knowledge of probability theory, including conditional probabilities. You can read [my notes](#) or any introductory probability text to get an overview. After completing this chapter, you should be able to

- Justify why explanations are important
- Distinguish between justification, interpretation, and explanation
- Compute feature importance and Shapley values
- Define a counterfactual and compute them
- Know which models are interpretable and how to fit interpretable surrogate models

---

A famous example on the need for explainable AI is found in Caruana et al.[CLG+15] who built an ML predictor to assess mortality risk of patients in the ER with pneumonia. The idea is that patients with pneumonia are screened with this tool and it helps doctors know which patients are more at risk of dying. It was found to be quite accurate. When the interpretation of its predictions were examined though, the reasoning was medically insane. The model surprisingly suggested patients with asthma (called asthmatics) have a reduced mortality risk when coming to the ER with pneumonia. Asthma, a condition which makes it difficult to breathe, was found to *make pneumonia patients less likely to die*. This was incidental; asthmatics are actually more at risk of dying from pneumonia but doctors are acutely aware of this and are thus more aggressive and attentive with them. Thanks to the increase care and attention from doctors, there are fewer mortalities. From an empirical standpoint, the model predictions are correct. However if the model were put into practice, it could have cost lives by incorrectly characterizing asthmatics as low mortality risk. Luckily the interpretability of their model helped researchers identify this problem. Thus, we can see that interpretation should always be a step in the construction of predictive models.

## 12.1 What is an explanation?

We'll use the definition of explanation from Miller [Mil19]. Miller distinguishes between interpretability, justification, and explanation with the following definitions:

- **interpretability** “the degree to which an observer can understand the cause of a decision”. Miller considers this synonymous with explainability. *This is a generally a property of a model.*
- **justification** evidence or explanation of why a decision is good, like testing error or accuracy of a model. *This is a property of a model.*
- **explanation** explanations are a presentation of information intended for humans that give the context and cause for an outcome. These are the major focus of this chapter. *This is generally something extra we generate and not a property of a model.*

We will dig deeper into what constitutes an *explanation*, but note an explanation is different than justifying a prediction. Justification is what we've focused on previously: empirical evidence for why we should believe model predictions are accurate. An explanation provides a *cause* for the prediction. Ultimately, explanations are intended to be understood by humans.

Deep learning alone is a black box modeling technique. It not interpretable or explainable. Examining the weights or model equation provides little insight into why predictions are made. Thus, interpretability is an extra task and means adding an explanation to predictions from the model. This is a challenge because of both the black box nature of deep learning and because there is no consensus on what exactly constitutes an “explanation” for model predictions [DVK17]. For some, interpretability means having a natural language explanation justifying each prediction. For others, it can be simply showing which features contributed most to the prediction.

There are two broad approaches to interpretation of ML models: post hoc interpretation via explanations and self-explaining models [MSK+19]. Self-explaining models are constructed so that an expert can view output of the model and connect it with the features through reasoning. They inherently interpretable. Self-explaining models are highly dependent on the task model [MSMuller18]. A familiar example would be a physics based simulation like molecular dynamics or a single-point quantum energy calculation. You can examine the molecular dynamics trajectory, look at output numbers, and an expert can explain why, for example, the simulation predicts a drug molecule will bind to a protein.

It may seem like self-explaining models would be useless for deep learning interpretation. However, we will see later that we can create a **surrogate model** (sometimes **proxy model**) that is self-explaining and train it to agree with the deep learning model. Why will this training burden be any less than just using the surrogate model from the beginning? We can generate an infinite amount of training data because our trained neural network can label arbitrary points. You can also construct deep learning models which have self-explaining features in them, like attention [BCB14]. This allows you to connect the input features to the prediction based on attention. There is also work within machine learning called **symbolic regression**, which tries to construct self-explaining models by working with mathematical equations that can be directly interpreted [AGFW21, BD00, UT20]. Symbolic regression is then used to generate the surrogate model [CSGB+20].

Post hoc interpretation by creating explanations can be approached in a number of ways, but the most common are training data importance, feature importance, and counterfactual explanations [WSW22, RSG16a, RSG16b, WMR17]. An example of a post hoc interpretation based on data importance is identifying the most influential training data to explain a prediction [KL17]. It is perhaps arguable if this gives an *explanation*, but it certainly helps understand which data is relevant for a prediction. Feature importance is probably the most common XAI approach and frequently appears in computer vision research where the pixels most important for the class of an image are highlighted.

Counterfactual explanations are an emerging post hoc interpretation method. Counterfactuals are new data point that serve as an explanation. A counterfactual gives insight into how important and sensitive the features are. An example might be in a model that recommends giving a loan. A model could produce the following counterfactual explanation (from [WMR17]):

You were denied a loan based on your annual income, zip code, and assets. If your annual income had been \$45,000, you would have been offered a loan.

The second sentence is the counterfactual and shows how the features could be changed to affect the model outcome. Counterfactuals provide a nice balance of complexity and explanatory power.

This was a brief overview of large field of XAI. You can find a recent review of interpretable deep learning in Samek et al. [SML+21] and Christopher Molnar has a [broad online book](#) about interpretable machine learning, including deep learning [Mol19]. Prediction error and confidence in predictions is not covered here, since they are more about justification, but see the methods from *Regression & Model Assessment* which apply.

## 12.2 Feature Importance

Feature importance is the most straightforward and common method of interpreting a machine learning model. The output of feature importance is a ranking or numerical values for each feature, typically for a single prediction. If you are trying to understand the feature importance across the whole model, this is called **global** feature importance and **local** for a single prediction. Global feature importance and global interpretability is relatively rare because accurate deep learning models change which features are important in different regions of feature space.

Let's start with a linear model to see feature importance:

$$\hat{y} = \vec{w}\vec{x} + b \quad (12.1)$$

where  $\vec{x}$  is our feature vector. A simple way to assess feature importance is to simply look at the weight value  $w_i$  for a particular feature  $x_i$ . The weight  $w_i$  shows how much  $\hat{y}$  would change if  $x_i$  is increased by 1, while all other features are constant. If the magnitude of our features are comparable, then this would be a reasonable way to rank features. However, if our features have units, this approach is sensitive to unit choices and relative magnitude of features. For example if our temperature was changed from Celsius to Fahrenheit, a 1 degree increase will have a smaller effect.

To remove the effect of feature magnitude and units, a slightly better way to assess feature importance is to divide  $w_i$  by the **standard error** in the feature values. Recall that standard error is just the ratio of sum of squared error in predicted value divided by the total deviation in the feature. Standard error is a ratio of prediction accuracy to feature variance.  $w_i$  divided by standard error is called the  $t$ -statistic because it can be compared with the  $t$ -distribution for assessing feature importance.

$$t_i = \frac{w_i}{S_{w_i}}, S_{w_i}^2 = \frac{1}{N-D} \sum_j \frac{(\hat{y}_j - y_j)^2}{(x_{ij} - \bar{x}_i)^2} \quad (12.2)$$

where  $N$  is the number of examples,  $D$  is the number of features, and  $\bar{x}_i$  is the average value of the  $i$ th feature. The  $t_i$  value can be used to rank features and it can be used for a hypothesis test: if  $P(t > t_i) < 0.05$  then that feature is significant, where  $P(t)$  is Student's  $t$ -distribution. Note that a feature's significance is sensitive to which features are present in a model; if you add new features some may become redundant.

If we move to a nonlinear learned function  $\hat{f}(\vec{x})$ , we must compute how the prediction changes if a feature value increases by 1 via the derivative approximation:

$$\frac{\Delta \hat{f}(\vec{x})}{\Delta x_i} \approx \frac{\partial \hat{f}(\vec{x})}{\partial x_i}$$

so a change by 1 is

$$\Delta \hat{f}(\vec{x}) \approx \frac{\partial \hat{f}(\vec{x})}{\partial x_i}. \quad (12.3)$$

In practice, we make a slight variation on this equation – instead of a Taylor series centered at 0 approximating this change, we center at some other root (point where the function is 0). This “grounds” the series at the decision boundary (a root) and then you can view the partials as “pushing” the predicted class away or towards the decision boundary. Another way to think about this is that we use the first-order terms of the Taylor series to build a linear model. Then we just apply what we did above to that linear model and use the coefficients as the “importance” of features. Specifically, we use this surrogate function for  $\hat{f}(\vec{x})$ :

$$\text{cancel} \hat{f}(\vec{x}) \approx 0f(\vec{x}') + \nabla \hat{f}(\vec{x}') \cdot (\vec{x} - \vec{x}') \quad (12.4)$$

where  $\vec{x}'$  is the root of  $\hat{f}(\vec{x})$ . In practice people may choose the trivial root  $\vec{x}' = \vec{0}$ , however a nearby root is ideal. This root is often called the **baseline** input. Note that as opposed to the linear example above, we consider the product of the partial  $\frac{\partial \hat{f}(\vec{x})}{\partial x_i}$  and the increase above baseline  $(x_i - x'_i)$ .

### 12.2.1 Neural Network Feature Importance

In neural networks, the partial derivatives are a poor approximation of the real changes to the output. Small changes to the input can have discontinuous changes (because of nonlinearities like ReLU), making the terms above have little explanatory power. This is called the **shattered gradients** problem [BFL+17]. Breaking down each feature separately also misses correlations between features – which don't exist in a linear model. Thus the derivative approximation works satisfactorily in locally linear models, but not deep neural networks.

There are a variety of techniques that get around the issue of shattered gradients in neural networks. Two popular methods are integrated gradients [STY17] and SmoothGrad [STK+17]. Integrated gradients creates a path from  $\vec{x}'$  to  $\vec{x}$  and integrates Equation 4 along that path:

$$\text{IG}_i = (\vec{x} - \vec{x}') \int_0^1 \left[ \nabla \hat{f}(\vec{x}' + t(\vec{x} - \vec{x}')) \right]_i dt \quad (12.5)$$

where  $t$  is some increment along the path such that  $\vec{x}' + t(\vec{x} - \vec{x}') = \vec{x}'$  when  $t = 0$  and  $\vec{x}' + t(\vec{x} - \vec{x}') = \vec{x}$  when  $t = 1$ . This gives us the integrated gradient for each feature  $i$ . The integrated gradients are the importance of each feature, but without the complexity of shattered gradients. There are some nice properties too, like  $\sum_i \text{IG}_i = f(\vec{x}) - f(\vec{x}')$  so that the integrated gradients provide a complete partition of the change from the baseline to the prediction [STY17].

Implementing integrated gradients is actually relatively simple. You approximate the path integral with a Riemann sum by breaking the path into a set of discrete inputs between the input features  $\vec{x}$  and the baseline  $\vec{x}'$ . You compute the gradient of these inputs with the neural network. Then you multiply that by the change in features above baseline:  $(\vec{x} - \vec{x}')$ .

SmoothGrad is a similar idea to the integrated gradients. Rather than summing up the gradients along a path though, we sum gradients from random points nearby our prediction. The equation is:

$$\text{SG}_i = \sum_j^M \left[ \nabla \hat{f}(\vec{x}' + \vec{\epsilon}) \right]_i \quad (12.6)$$

where  $M$  is a choice of sample number and  $\vec{\epsilon}$  is sampled from  $D$  zero-mean Gaussians [STK+17]. The only change in implementation here is to replace the path with a series of random perturbations.

Beyond these gradient based approaches, Layer-wise Relevance Propagation (LRP) is another popular approach for feature importance analysis in neural networks. LRP works by doing a backwards propagation through the neural network that partitions the output value of one layer to the input features. It “distributes relevance.” What is unusual about LRP is that each layer type needs its own implementation. It doesn't rely on the analytic derivative, but instead a Taylor series expansion of the layer equation. There are variants for GNNs and sequence models, so that LRP can be used in most settings in materials and chemistry [MBL+19].

### 12.2.2 Shapley Values

A model agnostic way to treat feature importance is with **Shapley values**. Shapley values come from game theory and are a solution to how to pay a coalition of cooperating players according to their contributions. Imagine each feature is a player and we would like to “pay” them according to their contribution to the predicted value. A Shapley value  $\phi_i(x)$  is the pay to feature  $i$  at instance  $x$ . We break-up the predicted function value  $\hat{f}(x)$  into the Shapley values so that the sum of the pay is the function value:  $\sum_i \phi_i(x) = \hat{f}(x)$ . This means you can interpret the Shapley value of a feature as its numerical contribution to the prediction. Shapley values are powerful because their calculation is agnostic to the model, they partition the predicted value among each feature, and they have other attributes that we would desire in an explanation of a prediction (symmetry, linearity, permutation invariant, etc.). Their disadvantage are that exact

computation is combinatorial with respect to feature number and they have no sparsity, making them less helpful as feature number grows. Most methods we discuss here also have no sparsity. You can always force your model to be sparse to achieve sparse explanations, like with L1 regularization (see [Standard Layers](#)).

Shapley values are computed as

$$\phi_i(x) = \frac{1}{Z} \sum_{S \in N \setminus x_i} v(S \cup x_i) - v(S) \quad (12.7)$$

$$Z = \frac{|S|! (N - |S| - 1)!}{N!}$$

where  $S \in N \setminus x_i$  means all sets of features that exclude feature  $x_i$ ,  $S \cup x_i$  means putting back feature  $x_i$  into the set, and  $v(S)$  is the value of  $\hat{f}(x)$  using only the features included in  $S$ , and  $Z$  is a normalization value. The formula can be interpreted as the mean of all possible differences in  $\hat{f}$  formed by adding/removing feature  $i$ .

One immediate concern though is how can we “remove” feature  $i$  from a model equation? We marginalize out feature  $i$ . Recall a marginal is a way to integrate out a random variable  $P(x) = \int P(x, y) dy$ . That integrates over all possible  $x$  values. Marginalization can be used on functions of random variables, which obviously are also random variables, by taking an expectation:  $E_y[f|X=x] = \int f(X=x, y)P(X=x, y) dy$ . I’ve emphasized that the random variable  $X$  is fixed in the integral and thus  $E_y[f]$  is a function of  $x$ .  $y$  is removed by computing the expected value of  $f(x, y)$  where  $x$  is fixed (the function argument). We’re essentially replacing  $f(x, y)$  with a new function  $E_y[f]$  that is the average of all possible  $y$  values. I’m over-explaining this though, it’s quite intuitive once you see the code below. The other detail is that *value* is the change relative to the average of  $\hat{f}$ . You can typically ignore this extra term - it cancels, but I include it for completeness. Thus the value equation becomes [vStrumbeljK14]:

$$v(x_i) = \int f(x_0, x_1, \dots, x_i, \dots, x_N) P(x_0, x_1, \dots, x_i, \dots, x_N) dx_i - E[\hat{f}(\vec{x})] \quad (12.8)$$

How do we compute the marginal  $\int f(x_0, x_1, \dots, x_i, \dots, x_N) P(x_0, x_1, \dots, x_i, \dots, x_N) dx_i$ ? We do not have a known probability distribution  $P(\vec{x})$ . We can sample from  $P(\vec{x})$  by considering our data as an **empirical distribution**. That is, we can sample from  $P(\vec{x})$  by sampling data points. There is a little bit of complexity here because we need to sample the  $\vec{x}$ ’s jointly, we cannot just mix together individual features randomly because there are correlations between features that will be removed.

Strumbelj et al. [vStrumbeljK14] showed that we can directly estimate the  $i$ th Shapley value with:

$$\phi_i(\vec{x}) = \frac{1}{M} \sum \hat{f}(\vec{z}_{+i}) - \hat{f}(\vec{z}_{-i}) \quad (12.9)$$

where  $\vec{z}$  is a “chimera” example constructed from the real example  $\vec{x}$  and a randomly drawn example  $\vec{x}'$ . We randomly select from  $\vec{x}$  and  $\vec{x}'$  to construct  $\vec{z}$ , except  $\vec{z}_{+i}$  specifically has the  $i$ th feature from the example  $\vec{x}$  and  $\vec{z}_{-i}$  has the  $i$ th feature from the random example  $\vec{x}'$ .  $M$  is chosen large enough to get a good sample for this value. [vStrumbeljK14] gives guidance on choosing  $M$ , but basically as large  $M$  as computationally feasible reasonable. One change in this approximation though is that we end-up with an explicit term for the expectation (sometimes denoted  $\phi_0$ ) so that our “completeness” equation is:

$$\sum_i \phi_i(\vec{x}) = \hat{f}(\vec{x}) - E[\hat{f}(\vec{x})] \quad (12.10)$$

Or if you explicitly include expectation as  $\phi_0$ , which is independent of  $\vec{x}$

$$\phi_0 + \sum_{i=1} \phi_i(\vec{x}) = \hat{f}(\vec{x}) \quad (12.11)$$

Marginalizing features is *not* the same as replacing features with their average.

With this efficient approximation method, the strong theory, and independence of model choice, Shapley values are an excellent choice for describing feature importance for predictions.

## 12.3 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

---

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

```
import haiku as hk
import jax
import tensorflow as tf
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
import urllib
from functools import partial
from jax.example_libraries import optimizers as opt
import dmol
```

```
np.random.seed(0)
tf.random.set_seed(0)
```

```
ALPHABET = [
    "-",
    "A",
    "R",
    "N",
    "D",
    "C",
    "Q",
    "E",
    "G",
    "H",
    "I",
    "L",
    "K",
    "M",
    "F",
    "P",
    "S",
    "T",
    "W",
    "Y",
    "V",
]
```

We now define a few functions we'll need to convert between amino acid sequence and one-hot vectors.

```
def seq2array(seq, L=200):
    return np.pad(list(map(ALPHABET.index, seq)), (0, L - len(seq))).reshape(1, -1)
```

(continues on next page)



(continued from previous page)

```
def array2oh(a):
    a = np.squeeze(a)
    o = np.zeros((len(a), 21))
    o[np.arange(len(a)), a] = 1
    return o.astype(np.float32).reshape(1, -1, 21)

url = urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/master/data/hemolytic.npz",
    "hemolytic.npz",
)
with np.load("hemolytic.npz", "rb") as r:
    pos_data, neg_data = r["positives"], r["negatives"]
```

## 12.4 Feature Importance Example

Let's see an example of these feature importance methods on a peptide prediction task to predict if a peptide will kill red blood cells (hemolytic). This is similar to the solubility prediction example from *Standard Layers*. The data is from [BW21]. The model takes in peptide sequences (e.g., DDFRD) and predicts the probability that the peptide is hemolytic. The goal of the feature importance method here will be to identify which amino acids matter most for the hemolytic activity. The hidden-cell below loads and processes the data into a dataset.

We rebuild the convolution model in Jax (using *Haiku*) to make working with gradients a bit easier. We also make a few changes to the model – we pass in the sequence length and amino acid fractions as extra information in addition to the convolutions.

```
def binary_cross_entropy(logits, y):
    """Binary cross entropy without sigmoid. Works with logits directly"""
    return (
        jnp.clip(logits, 0, None) - logits * y + jnp.log(1 + jnp.exp(-jnp.
        ↪abs(logits)))
    )

def model_fn(x):
    # get fractions, excluding skip character
    aa_fracs = jnp.mean(x, axis=1)[:, 1:]
    # compute convolutions/poolings
    mask = jnp.sum(x[... , 1:], axis=-1, keepdims=True)
    for kernel, pool in zip([5, 3, 3], [4, 2, 2]):
        x = hk.Conv1D(16, kernel)(x) * mask
        x = jax.nn.tanh(x)
        x = hk.MaxPool(pool, pool, "VALID")(x)
        mask = hk.MaxPool(pool, pool, "VALID")(mask)
    # combine fractions, length, and convolution outputs
    x = jnp.concatenate((hk.Flatten()(x), aa_fracs, jnp.sum(mask, axis=1)), axis=1)
    # dense layers. no bias, so zeros give P=0.5
    logits = hk.Sequential(
        [
            hk.Linear(256, with_bias=False),
            jax.nn.tanh,
            hk.Linear(64, with_bias=False),
```

(continues on next page)

(continued from previous page)

```

        jax.nn.tanh,
        hk.Linear(1, with_bias=False),
    ]
)(x)
return logits

model = hk.without_apply_rng(hk.transform(model_fn))

def loss_fn(params, x, y):
    logits = model.apply(params, x)
    return jnp.mean(binary_cross_entropy(logits, y))

@jax.jit
def hemolytic_prob(params, x):
    logits = model.apply(params, x)
    return jax.nn.sigmoid(jnp.squeeze(logits))

@jax.jit
def accuracy_fn(params, x, y):
    logits = model.apply(params, x)
    return jnp.mean((logits >= 0) * y + (logits < 0) * (1 - y))

```

```

rng = jax.random.PRNGKey(0)
xi, yi = features[:batch_size], labels[:batch_size]
params = model.init(rng, xi)

opt_init, opt_update, get_params = opt.adam(1e-2)
opt_state = opt_init(params)

@jax.jit
def update(step, opt_state, x, y):
    value, grads = jax.value_and_grad(loss_fn)(get_params(opt_state), x, y)
    opt_state = opt_update(step, grads, opt_state)
    return value, opt_state

```

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF\_CPP\_MIN\_LOG\_LEVEL=0 and rerun for more info.)

```

epochs = 32
for e in range(epochs):
    avg_v = 0
    for i, (xi, yi) in enumerate(train_data):
        v, opt_state = update(i, opt_state, xi.numpy(), yi.numpy())
        avg_v += v
    opt_params = get_params(opt_state)

def predict(x):
    return jnp.squeeze(model.apply(opt_params, x))

```

(continues on next page)

(continued from previous page)

```
def predict_prob(x):
    return hemolytic_prob(opt_params, x)
```

If you're having trouble following the code, that's OK! The goal of this chapter is to show how to get explanations of a model, not necessarily how to build the model. So focus on the next few lines where I show how to use the model to get predictions and explain them. The model is called via `predict(x)` for logits or `predict_prob` for probability.

## Sequence Models

Review *Deep Learning on Sequences* to refresh ideas about one-hots and sequence models.

Let's try an amino acid sequence, a peptide, to get a feel for the model. The model outputs logits (logarithm of odds), which we put through a sigmoid to get probabilities. The peptides must be converted from a sequence to a matrix of one-hot column vectors. We'll try two known sequences: Q is known to be common in hemolytic residues and the second sequence is poly-G, which is the simplest amino acid.

```
s = "QQQQQ"
sm = array2oh(seq2array(s))
p = predict_prob(sm)
print(f"Probability {s} of being hemolytic {p:.2f}")

s = "GGGGG"
sm = array2oh(seq2array(s))
p = predict_prob(sm)
print(f"Probability {s} of being hemolytic {p:.2f}")
```

```
Probability QQQQQ of being hemolytic 1.00
Probability GGGGG of being hemolytic 0.00
```

It looks reasonable – the model matches our intuition about these two sequences

Now we compute the accuracy of our model, which is quite good.

```
acc = []
for xi, yi in test_data:
    acc.append(accuracy_fn(opt_params, xi.numpy(), yi.numpy()))
print(jnp.mean(np.array(acc)))
```

```
0.95208335
```

### 12.4.1 Gradients

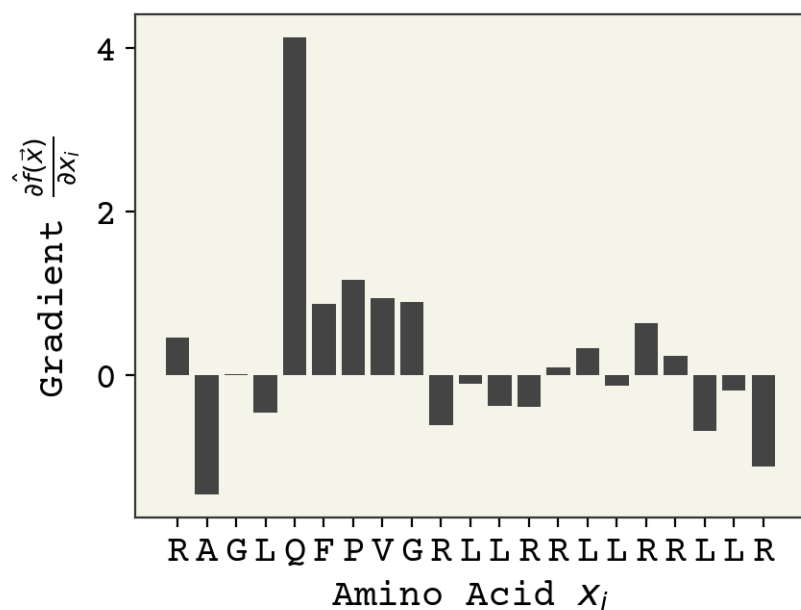
Now to start examining *why* a particular sequence is hemolytic! We'll begin by computing the gradients with respect to input – the naive approach that is susceptible to shattered gradients. Computing this is a component in the process for integrated and smooth gradients, so not wasted effort. We will use a more complex peptide sequence that is known to be hemolytic to get more interesting analysis.

```
s = "RAGLQFPVGRLLRRLRLRLR"
sm = array2oh(seq2array(s))
p = predict_prob(sm)
print(f"Probability {s} of being hemolytic {p:.2f}")
```

```
Probability RAGLQFPVGRLLRRLRLRLR of being hemolytic 1.00
```

The code is quite simple, just a gradient computation.

```
gradient = jax.grad(predict, 0)
g = gradient(sm)
plot_grad(g, s)
```



Remember that the model outputs logits. Positive value of the gradient mean this amino acid is responsible for pushing hemolytic probability higher and negative values mean the amino acid is pushing towards non-hemolytic. Interestingly, you can see a strong position dependence on the leucine (L) and arginine (R).

## 12.4.2 Integrated Gradients

We'll now implement the integrated gradients method. We go through three basic steps:

1. Create an array of inputs going from baseline to input peptide
2. Evaluate gradient on each input
3. Compute the sum of the gradients and multiply it by difference between baseline and peptide

The baseline for us is all zeros – which gives a probability of 0.5 (logits = 0, a model root). This baseline is exactly on the decision boundary. You could use other baselines like all glycines or all alanines, just they should be at or near probability of 0.5. You can find a detailed and interactive exploration of the baseline choice in [SLL20].

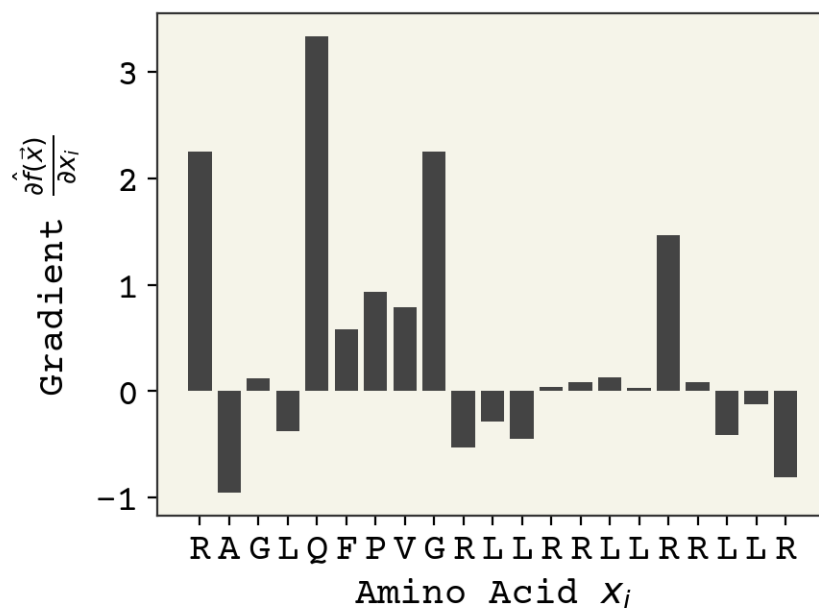
```
def integrated_gradients(sm, N):
    baseline = jnp.zeros((1, L, 21))
    t = jnp.linspace(0, 1, N).reshape(-1, 1, 1)
    path = baseline * (1 - t) + sm * t

    def get_grad(pi):
        # compute gradient
        # add/remove batch axes
        return gradient(pi[jnp.newaxis, ...])[0]

    gs = jax.vmap(get_grad)(path)
    # sum pieces (Riemann sum), multiply by (x - x')
    ig = jnp.mean(gs, axis=0, keepdims=True) * (sm - baseline)
    return ig
```

```
ig = integrated_gradients(sm, 1024)
```

```
plot_grad(ig, s)
```



We see that the position dependence has become more pronounced, with arginine being very sensitive to position. Relatively little has qualitatively changed between this and the vanilla gradients.

### 12.4.3 SmoothGrad

To do SmoothGrad, our steps are almost identical:

1. Create an array of inputs that are random perturbations of the input peptide
2. Evaluate gradient on each input
3. Compute the mean of the gradients

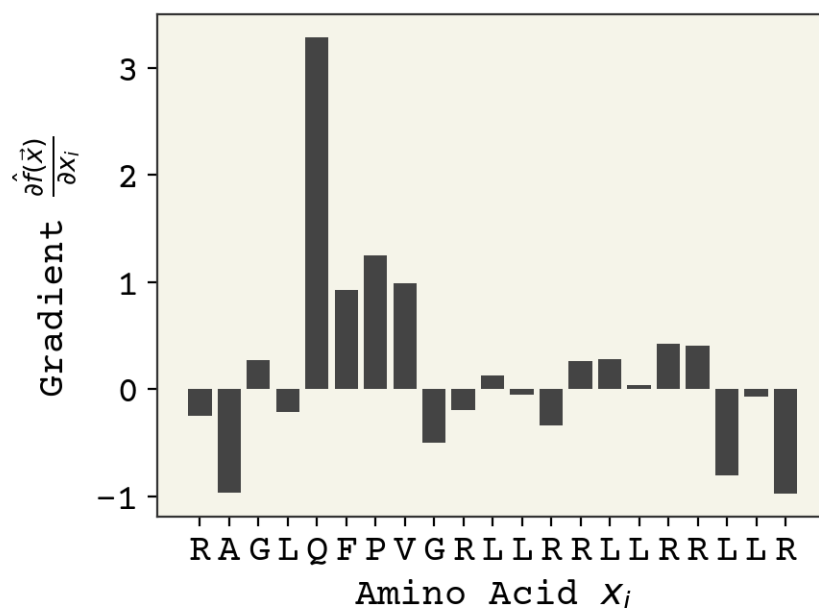
There is one additional hyperparameter,  $\sigma$ , which in principle should be as small as possible while still causing the model output to change.

```
def smooth_gradients(sm, N, rng, sigma=1e-3):
    baseline = jnp.zeros((1, L, 21))
    t = jax.random.normal(rng, shape=(N, sm.shape[1], sm.shape[2])) * sigma
    path = sm + t
    # remove examples that are negative and force summing to 1
    path = jnp.clip(path, 0, 1)
    path /= jnp.sum(path, axis=2, keepdims=True)

    def get_grad(pi):
        # compute gradient
        # add/remove batch axes
        return gradient(pi[jnp.newaxis, ...])[0]

    gs = jax.vmap(get_grad)(path)
    # mean
    ig = jnp.mean(gs, axis=0, keepdims=True)
    return ig

sg = smooth_gradients(sm, 1024, jax.random.PRNGKey(0))
plot_grad(sg, s)
```



It looks remarkably similar to the vanilla gradient setting – probably because our 1D input/shallow network is not as sensitive to shattered gradients.

### 12.4.4 Shapley Value

Now we will approximate the Shapley values for each feature using Equation 10.9. The Shapley value computation is different than previous approaches because it does not require gradients. The basic algorithm is:

1. select random point  $x'$
2. create point  $z$  by combining  $x$  and  $x'$
3. compute change in predicted function

One efficiency change we make is to prevent modifying the sequence in its padding – basically prevent exploring making the sequence longer.

```
def shapley(i, sm, sampled_x, rng, model):
    M, F, *_ = sampled_x.shape
    z_choice = jax.random.bernoulli(rng, shape=(M, F))
    # only swap out features within length of sm
    mask = jnp.sum(sm[..., 1:], -1)
    z_choice *= mask
    z_choice = 1 - z_choice
    # construct with and w/o ith feature
    z_choice = z_choice.at[:, i].set(0.0)
    z_choice_i = z_choice.at[:, i].set(1.0)
    # select them via multiplication
    z = sm * z_choice[..., jnp.newaxis] + sampled_x * (1 - z_choice[..., jnp.newaxis])
    z_i = sm * z_choice_i[..., jnp.newaxis] + sampled_x * (
        1 - z_choice_i[..., jnp.newaxis]
    )
    v = model(z_i) - model(z)
    return jnp.squeeze(jnp.mean(v, axis=0))

# assume data is already shuffled, so just take M
M = 4096
sl = len(s)
sampled_x = train_data.unbatch().batch(M).as_numpy_iterator().next()[0]
# make batched shapley so we can compute for all features
bshapley = jax.vmap(shapley, in_axes=(0, None, None, 0, None))
sv = bshapley(
    jnp.arange(sl),
    sm,
    sampled_x,
    jax.random.split(jax.random.PRNGKey(0), sl),
    predict,
)

# compute global expectation
eyhat = 0
for xi, yi in full_data.batch(M).as_numpy_iterator():
    eyhat += jnp.mean(predict(xi))
eyhat /= len(full_data)
```

One nice check on Shapley values is that we can check that their sum is equal to the value of model function minus the expect value across all instances. Note we made approximations to use the Equation from [vStrumbeljK14] so that we cannot expect perfect agreement. That value is computed as:

```
print(np.sum(sv), predict(sm))
```

```
6.7905803 8.086109
```

which is *some* disagreement. This is an effect of the approximation method we're using. We can check that by examining how sample number effects the sum of Shapley values.

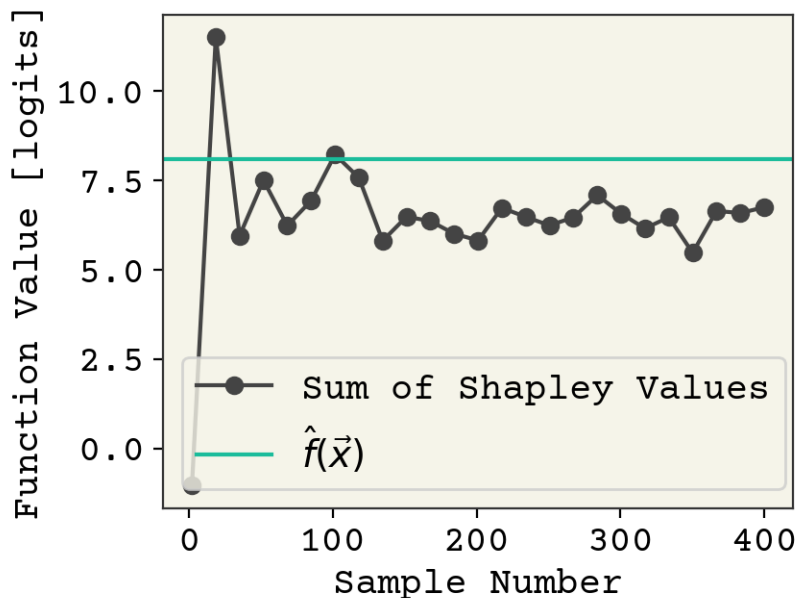
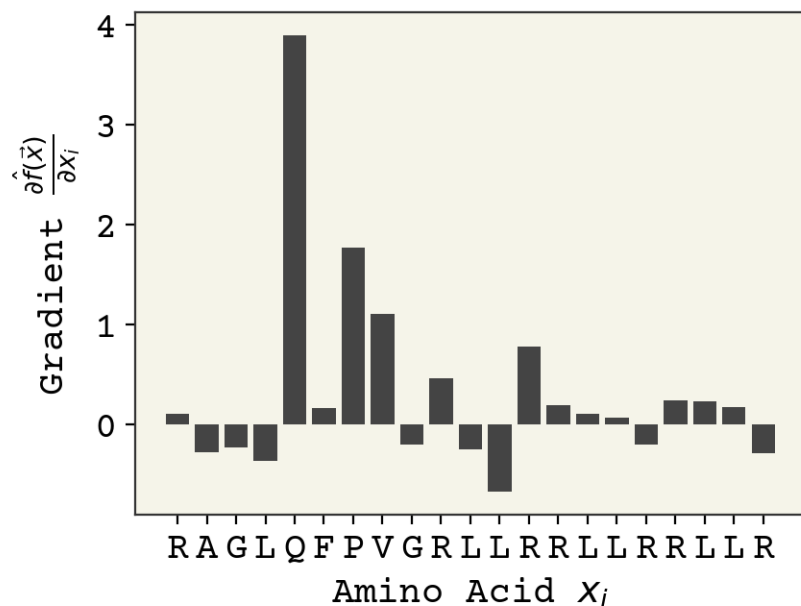


Fig. 12.1: A comparison of sum of Shapley values and function value as a function of samples number in the Shapley value approximation.

It is slowly converging. Finally we can view the individual Shapley values, which is our explanation.

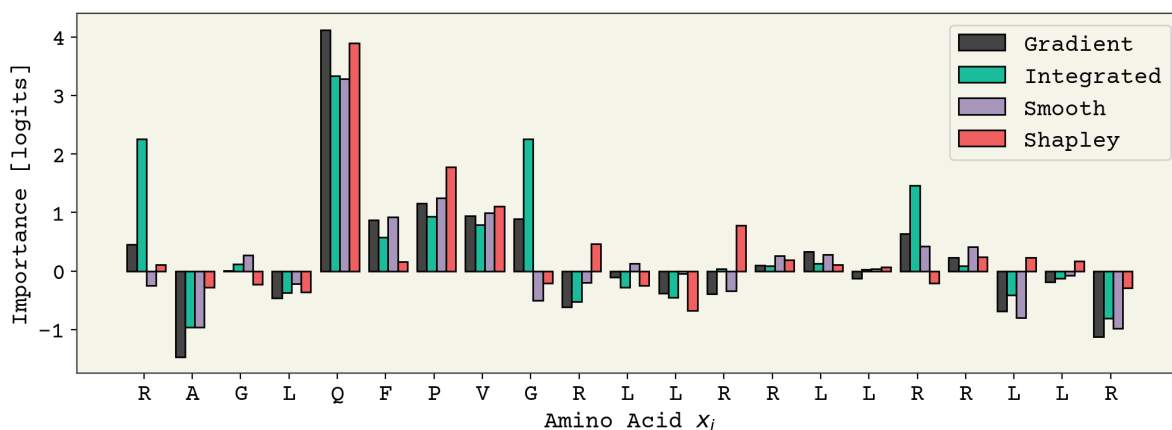
```
plot_grad(sv, s)
```





The four methods are shown side-by-side below.

```
heights = []
plt.figure(figsize=(12, 4))
x = np.arange(len(s))
for i, (gi, l) in enumerate(zip([g, ig, sg], ["Gradient", "Integrated", "Smooth"])):
    h = gi[0, np.arange(len(s)), list(map(ALPHABET.index, s))]
    plt.bar(x + i / 5 - 1 / 4, h, width=1 / 5, edgecolor="black", label=l)
plt.bar(x + 3 / 5 - 1 / 4, sv, width=1 / 5, edgecolor="black", label="Shapley")
ax = plt.gca()
ax.set_xticks(range(len(s)))
ax.set_xticklabels(s)
ax.set_xlabel("Amino Acid  $x_i$ ")
ax.set_ylabel("Importance [logits]")
plt.legend()
plt.show()
```



As someone who works with peptides, I believe the Shapley is the most accurate here. I wouldn't expect the pattern of L and R to be that significant, which is what the Shapley values show. Another difference is that the Shapley values do not show the phenylalanine (F) as have a significant effect.

What can we conclude from this information? We could perhaps add an explanation like this: "The sequence is predicted to be hemolytic primarily because of the glutamine, proline, and arrangement of leucine and arginine."

## 12.5 What is feature importance for?

Feature importance rarely leads to a clear explanation that gives the cause for a prediction or insight that is actionable. The lack of causality can lead us to find meaning in feature explanations that do not exist[CK18]. Another caveat is remember that we are explaining *model*, not the actual chemical systems. For example, avoid saying "Hemolytic activity was caused by the glutamine in position 5." Instead: "Our model predicted hemolytic activity because of glutamine in position 5."

An **actionable** explanation is one that shows how to modify the features to affect the outcome — similar to saying we know the cause for an outcome. Thus, there is ongoing debate about if feature importance is an explanation [Lip18]. A popular line of work that tries to connect feature importance to human *concepts* is called Quantitative testing with concept activation vectors (TCAV) [KWG+18]. I personally have moved away from feature importance for XAI because the explanations are not actionable or causal and often can add additional confusion.

## 12.6 Training Data Importance

Another kind of explanation or interpretation we might desire is *which* training data points contribute most to a prediction. This is a more literal answer to the question: “Why did my model predict this?” – neural networks are a result of training data and thus the answer to why a prediction is made can be traced to training data. Ranking training data for a given prediction helps us understand which training examples are causing the neural network to predict a value. This is like an influence function,  $\mathcal{I}(x_i, x)$ , which gives a score of influence for training point  $i$  and input  $x$ . The most straightforward way to compute the influence would be to train the neural network with (i.e.,  $\hat{f}(x)$ ) and without  $x_i$  (i.e.,  $\hat{f}_{-x_i}(x)$ ) and define the influence as

$$\mathcal{I}(x_i, x) = \hat{f}_{-x_i}(x) - \hat{f}(x) \quad (12.12)$$

For example, if a prediction is higher after removing the training point  $x_i$  from training, we would say that point has a positive influence. Computing this influence function requires training the model as many times as you have points – typically computationally infeasible. [KL17] show a way to approximate this by looking at infinitesimal changes to the *weights* of each training point. Computing these influence functions do require computing a Hessian with respect to the loss function and thus are not commonly used. If you’re using JAX though, this is simple to do.

If using a kernel model, the features are the training data. The above methods like integrated gradients, give training data importance.

Training data importance provides an interpretation that is useful for deep learning experts. It tells you which training examples are most influential for a given predictions. This can help troubleshoot issues with data or tracing explanations for spurious predictions. However, a typical user of predictions from a deep learning model will probably be unsatisfied with a ranking of training data as an explanation.

## 12.7 Surrogate Models

One of the more general ideas in interpretability is to fit an interpretable model to a black box model *in the neighborhood of a specific example*. We assume that an interpretable model cannot be fit globally to a black box model – otherwise we could just use the interpretable model and throw away the black box model. However, if we fit the interpretable model to just a small region around an example of interest, we can provide an explanation through the locally correct interpretable model. We call the interpretable model a **local surrogate model**. Examples of local surrogate models that are inherently interpretable include decision trees, linear models, sparse linear models (for succinct explanations), a Naive Bayes Classifier, etc.

A popular algorithm for this process of fitting a local surrogate model is called Local Interpretable Model-Agnostic Explanations (LIME) [RSG16a]. LIME fits the local surrogate model in the neighborhood of the example of interest utilizing the loss function that trained the original black box model. The loss function for the local surrogate model is weighted so that we value points closer to the example of interest as we regress the surrogate model. The LIME paper includes sparsifying the surrogate model in its notation, but we’ll omit that from the loss equation since that is more of an attribute of the local surrogate model. Thus, our definition for the local surrogate model loss is

$$l^s(x') = w(x', x) l(\hat{f}_s(x'), \hat{f}(x')) \quad (12.13)$$

where  $w(x', x)$  is a weight kernel function that weights points near example of interest  $x$ ,  $l(\cdot, \cdot)$  is the original black box model loss,  $\hat{f}(\cdot)$  is the black box model, and  $\hat{f}_s(\cdot)$  is the local surrogate model.

LIME as formulated in [RSG16a] gives feature importance descriptions, but some surrogate models may be interpretable already. Like a decision tree.

The weight function is a bit ad hoc – it depends on the data type. For regression tasks with scalar labels, we use a kernel function and you have a variety of choices: Gaussian, cosine, Epanechnikov. For text, the LIME implementations use a **Hamming distance** which just counts number of text tokens which do not match between two strings. Images use the same distance but with superpixels being the same as the example or blank.

How are the points  $x'$  generated? In the continuous case  $x'$  is sampled *uniformly*, which is quite a feat since feature spaces are often unbounded. You could sample  $x'$  according to your weight function and then omit the weighting (since it was sampled according to that) to avoid issues like unbounded feature spaces. In general, LIME is a bit subjective in continuous vector feature spaces. For images and text,  $x'$  is formed by masking tokens (words) and zeroing (making black) superpixels. This leads to explanations that should feel quite similar to Shapley values – and indeed you can show LIME is equivalent to Shapley values with some small notation changes.

## 12.8 Counterfactuals

### Counterfactuals

Our optimization formulation is what's used in XAI, but counterfactuals in other contexts do not have the “nearness” criterion

A counterfactual is a solution to an optimization problem: find an example  $x'$  that has a different label than  $x$  and as close as possible to  $x$  [WMR17]. You can formulate this like:

$$\begin{aligned} &\text{minimize} && d(x, x') \\ &\text{such that} && \hat{f}(x) \neq \hat{f}(x') \end{aligned} \tag{12.14}$$

In regression settings where  $\hat{f}(x)$  outputs a scalar, you need to modify your constraint to be some  $\Delta$  away from  $\hat{f}(x)$ .  $x'$  that satisfies this optimization problem is the counterfactual: a condition that did not occur and would have led to a different outcome. Typically finding  $x'$  is treated as a derivative-free optimization. You can calculate  $\frac{\partial \hat{f}}{\partial x'}$  and do constrained optimization, but in practice it can be faster to just randomly perturb  $x$  until  $\hat{f}(x) \neq \hat{f}(x')$  like a Monte Carlo optimization. You can also use a generative model that can propose new  $x'$  via unsupervised training. See [WSW22] for a universal counterfactual generator for molecules. See [NB20] for a method specifically for graph neural networks of molecules.

Defining distance is an important subjective concern, that we saw above for LIME. A common example for molecular structures is Tanimoto similarity (also known as Jaccard index) of molecular fingerprints/descriptors like Morgan fingerprints [RH10].

Counterfactuals have one disadvantage compared to Shapley values: they do not provide a *complete* explanation. Shapley values sum to the prediction, meaning we are not missing any part of the explanation. Counterfactuals modify as few features as possible (minimizing distance) and so may omit information about features that still contribute to a prediction. Of course, one advantage of Shapley values is that they are actionable. You can use the counterfactual directly.

### 12.8.1 Example

We can quickly implement this idea for our peptide example above. We can define our distance as the Hamming distance. Then the closes  $x'$  would be a single amino acid substitution. Let's just try enumerating those and see if we can achieve a label swap. We'll define a function that does a single substitution:

```
def check_cf(x, i, j):
    # copy
    x = jnp.array(x)
    # substitute
    x = x.at[:, i].set(0)
    x = x.at[:, i, j].set(1)
    return predict(x)
```

```
check_cf(sm, 0, 0)
```

```
DeviceArray(8.560182, dtype=float32)
```

Then build all possible substitutions with `jnp.meshgrid` and apply our function over that with `vmap`. `.ravel()` makes our array of indices be a single dimensions, so we do not need to worry about doing a complex `vmap`.

```
ii, jj = jnp.meshgrid(jnp.arange(sl), jnp.arange(21))
ii, jj = ii.ravel(), jj.ravel()
x = jax.vmap(check_cf, in_axes=(None, 0, 0))(sm, ii, jj)
```

Now we'll display all the single amino acid substitutions which resulted in a negative prediction - the logits are less than zero.

```
from IPython.core.display import display, HTML

out = ["<tt>"]
for i, j in zip(ii[jnp.squeeze(x) < 0], jj[jnp.squeeze(x) < 0]):
    out.append(f'{s[:i]}<span style="color:red;">{ALPHABET[j]}</span>{s[i+1:]}<br/>')
out.append("</tt>")
display(HTML("".join(out)))
```

```
<IPython.core.display.HTML object>
```

We have a few to choose from, but the interpretation is essentially exchange the glutamine with a hydrophobic group or replace the proline with V, F, A, or C to make the peptide non-hemolytic. Stated as a counterfactual: "If the glutamine were exchanged with a hydrophobic amino acid, the peptide would not be hemolytic".

## 12.9 Specific Architectures Explanations

The same principles above apply to GNNs, but there are competing ideas about how best to translate these ideas to work on graphs. See [AZL21] for a discussion of theory of interpretability specifically for GNNs and [YYGJ20] for a survey of the methods available for constructing explanations in GNNs.

NLP is another area where there are specific approaches to constructing explanations and interpretation. See [MRC21] for a recent survey.

## 12.10 Model Agnostic Molecular Counterfactual Explanations

The main challenge associated with counterfactuals in chemistry is the difficulty in computing the derivative in (12.14). Therefore, most methods which focus on this task are specific to model architectures as we saw previously. Wellawatte et. al [WSW22] have introduced a method named Molecular Model Agnostic Counterfactual Explanations (MMACE) to do this for molecules regardless of model architecture.

The MMACE method is implemented in the `exmol` package. Given a molecule and a model, `exmol` is able to generate local counterfactual explanations. There are two main steps involved in the MMACE method. First, a local chemical space is expanded around the given base molecule. Next, each sample point is labeled with the user given model architecture. These labels are then used to identify the counterfactuals in the local chemical space. As the MMACE method is model agnostic, `exmol` package is able to generate counterfactuals for both classification and regression tasks.

Now let's see how to generate molecular counterfactuals using `exmol`. In this example, we will train a random forest model which predicts clinical toxicology of molecules. For this binary classification task, we'll be using the same dataset we used in the *Classification* chapter presented by the MoleculeNet group [WRF+18].

## 12.11 Running This Notebook

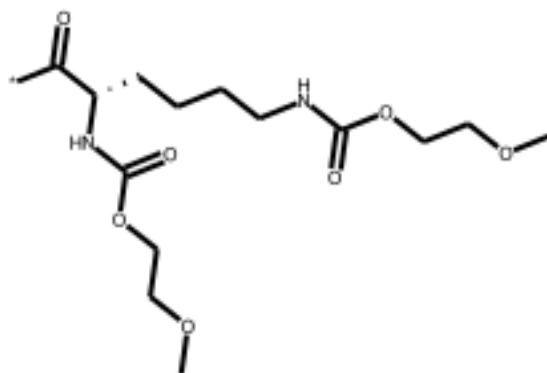
Click the [link](#) above to launch this page as an interactive Google Colab. See details below on installing packages, either on your own environment or on Google Colab

**Tip:** To install packages, execute this code in a new cell

```
!pip install exmol jupyter-book matplotlib numpy pandas seaborn sklearn mordred[full]
↳rdkit-pypi
```

```
# make object that can compute descriptors
calc = mordred.Calculator(mordred.descriptors, ignore_3D=True)
# make subsample from pandas df
molecules = [rdkit.Chem.MolFromSmiles(smi) for smi in toxdata.smiles]

# view one molecule to make sure things look good.
molecules[0]
```



After importing the data we generate input descriptors with `Mordred` package.

```
# Get valid molecules from the sample
valid_mol_idx = [bool(m) for m in molecules]
valid_mols = [m for m in molecules if m]
# Compute molecular descriptors using Mordred
features = calc.pandas(valid_mols, quiet=True)
labels = toxdata[valid_mol_idx].FDA_APPROVED
# Standardize the features
features -= features.mean()
features /= features.std()

# we have some nans in features, likely because std was 0
features = features.values.astype(float)
features_select = np.all(np.isfinite(features), axis=0)
features = features[:, features_select]
print(f"We have {len(features)} features per molecule")
```

```
We have 1478 features per molecule
```

In this example, we are using a simple dense neural network classifier implemented with Keras. First, let's train this simple classifier and use it to generate labels for the counterfactuals in `exmol`. By improving the performance of the trained model, you can expect more accurate results. But the following example is sufficient to understand the workings of `exmol` for now.

```
# Train and test split
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.2, shuffle=True
)
ft_shape = X_train.shape[-1]

# reshape data
X_train = X_train.reshape(-1, ft_shape)
X_test = X_test.reshape(-1, ft_shape)
```

Now let's build our model and compile! You can find an in depth introduction to dense models in the [Deep Learning Overview](#) chapter.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(ft_shape,)))
model.add(tf.keras.layers.Dense(32, activation="relu"))
model.add(tf.keras.layers.Dense(32))
model.add(Dense(1, activation="sigmoid"))
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

```
# Model training
model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0)
_, accuracy = model.evaluate(X_test, y_test)
print(f"Model accuracy: {accuracy*100:.2f}%")
```

```
1/10 [==>.....] - ETA: 0s - loss: 0.1121 - accuracy: 0.9688
```

```
10/10 [=====] - 0s 740us/step - loss: 0.2182 - accuracy: 0.9324
```

```
Model accuracy: 93.24%
```

Seems like our model has a good accuracy!

Now we'll write a wrapper function that takes in SMILES and/or SELFIES molecule representations and output label predictions from the trained classifier. A detailed description on SELFIES can be found in *Deep Learning on Sequences* chapter. This wrapper function is given as an input to `exmol.sample_space` function in `exmol` to create a local chemical space around a given base molecule. `exmol` uses Superfast Traversal, Optimization, Novelty, Exploration and Discovery (STONED) algorithm [NPK+21] as a generative algorithm to expand the local space. Given a base molecule, the STONED algorithm randomly mutate SELFIES representations of the molecules. These mutations can be string substitutions, additions or deletions.

```
def model_eval(smiles, selfies):
    molecules = [rdkit.Chem.MolFromSmiles(smi) for smi in smiles]
    features = calc.pandas(molecules)
    features = features.values.astype(float)
    features = features[:, features_select]
    labels = np.round(model.predict(np.nan_to_num(features).reshape(-1, ft_shape)))
    return labels
```

Now we use STONED to sample local chemical space with `exmol.sample_space`. In this example, we will modify the size of the sample space with `num_samples` argument. The base molecule selected here is a non-FDA approved molecule.

```
space = exmol.sample_space("C1CC(=O)NC(=O)C1N2CC3=C(C2=O)C=CC=C3N", model_eval);
```

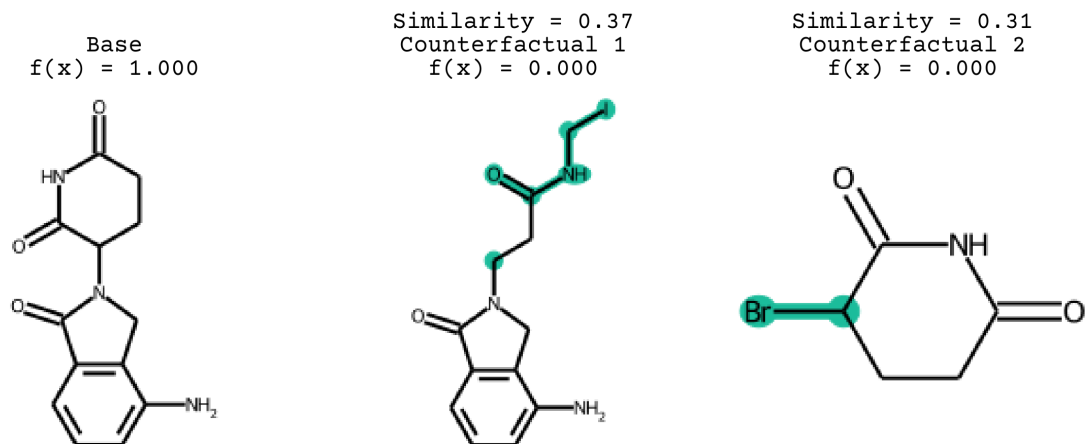
Once the sample space is created, we can identify counterfactuals in the local chemical space using `exmol.sample_space` function. Each counterfactual is a python dataclass that contain additional information.

```
exps = exmol.cf_explain(space, 2)
exps[1]
```

```
Example(smiles='Nc1cccc2c1CN(CCC(=O)NCI)C2=O', selfies=
↳ '[N][C][=C][C][=C][C][=C][Ring1][Branch1_2][C][N][Branch1_1][Branch2_
↳ 3][C][C][C][Branch1_2][C][=O][N][C][I][C][Ring1][N][=O]', similarity=0.
↳ 37333333333333335, yhat=array(0., dtype=float32), index=771, position=array([1.
↳ 83485072, 5.25947839]), is_origin=False, cluster=8, label='Counterfactual 1')
```

You can easily visualize the generated counterfactuals using the plotting codes in `exmol`: `exmol.sample_space` and `exmol.sample_space`. Similarity between the base and counterfactuals is the Tanimoto similarity of ECFP4 fingerprints. Top 3 counterfactuals are the shown here.

```
exmol.plot_cf(exps, n_rows=1)
```

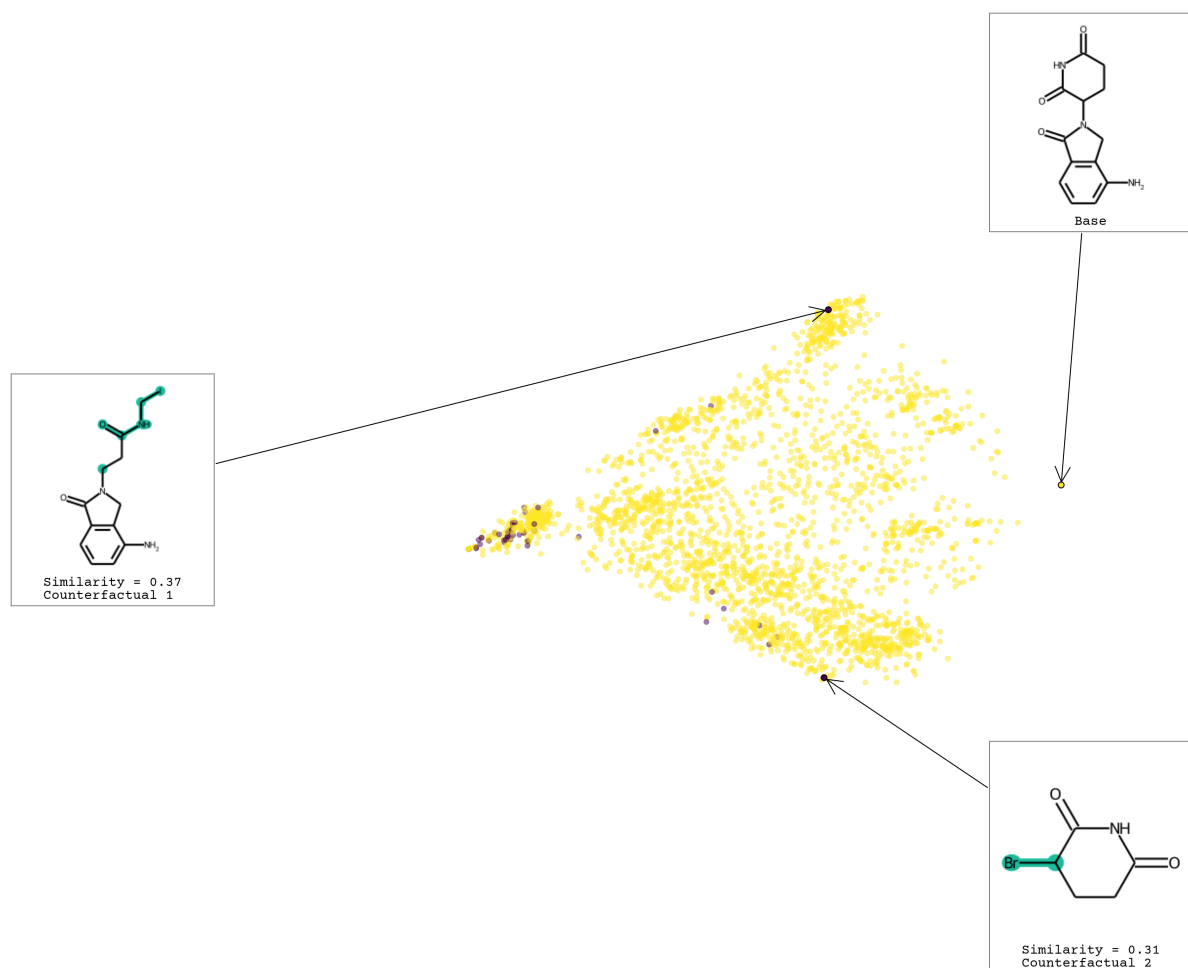


The base molecule which we selected here is NOT FDA approved. By looking at the generated counterfactuals we can conclude that, the heterocyclic group has an impact on the toxicity of the base. Therefore, by altering the heterocyclic group, the base molecule might be made non-toxic according to our model. This also shows why counterfactual explanations give actionable insight into how modifications can be made.

We can also visualize the generated chemical space!

```
exmol.plot_space(space, exps)
```





## 12.12 Summary

- Interpretation of deep learning models is imperative for ensuring model correctness, making predictions useful to humans, and can be required for legal compliance.
- Interpretability of neural networks is part of a broader topic of explainability in AI (XAI), a topic that is in its infancy
- An *explanation* is still ill-defined, but most often is expressed in terms of model features.
- Strategies for explanations include feature importance, training data importance, counterfactuals, and surrogate models that are locally accurate,
- Most explanations are generated per-example (at inference).
- The most systematic but expensive to compute explanations are Shapley values.
- Some argue that counterfactuals provide the most intuitive and satisfying explanations, but they may not be complete explanations.
- `exmol` is a software that generate model agnostic molecular counterfactual explanations.

## 12.13 Cited References

## ATTENTION LAYERS

Attention is a concept in machine learning and AI that goes back many years, especially in computer vision[BP97]. Like the word “neural network”, attention was inspired by the idea of attention in how human brains deal with the massive amount of visual and audio input[TG80]. **Attention layers** are deep learning layers that evoke the idea of attention. You can read more about attention in deep learning in Luong et al. [LPM15] and get a practical [overview here](#). Attention layers have been empirically shown to be so effective in modeling sequences, like language, that they have become indispensable[VSP+17]. The most common place you’ll see attention layers is in **transformer** neural networks that model sequences. We’ll also sometimes see attention in graph neural networks.

Attention can be confusing because of the three inputs. We’ll see later though that these inputs are actually often identical. The query is one key and the keys and values are equal. Then if you batch the queries together (one for each key), then you’ll see the query, keys, and values are equal. This is self-attention.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Tensors and Shapes*. You should be comfortable with broadcasting, matrices, and tensor shapes. After completing this chapter, you should be able to

- Correctly specify shapes and input/output of attention layers
- Implement attention layers
- Understand how attention can be put into other layer types

---

Attention layers are fundamentally a weighted mean reduction. It is just computing a mean, where you somehow weight each element contributing to the mean. Since it is a mean, attention decreases the rank of an input tensor. Attention is unusual among layers because it takes three inputs, whereas most layers in deep learning take just one or perhaps two. These inputs are called the **query**, the **values**, and the **keys**. The reduction occurs over the values; so if the values are rank 3, the output will be rank 2. The query should be one less rank than the keys. The keys should be the same rank as the values. The keys and query determine how to weight the values according to the **attention mechanism** – a fancy word for equation.

The table below summarizes these three input arguments. Note that often the query is batched, so that its rank will be 2 if batched. If the input query is batched, then the output’s rank will be batched as well and be 2 instead of 1.

	Rank	Axes	Purpose	Example
Query	1	(# of attn features)	input for checking against keys	One word represented as feature vector
Keys	2	(sequence length, # of attn features)	used to compute attention against query	All words in a sentence represented as matrix of feature vectors
Values	2	(sequence length, # of value features)	used to compute value of output	A vector of numbers for each word in a sentence
Output	1	(# of value features)	attention-weighted mean over values	single vector

## 13.1 Example

Attention is best conceptualized as operating on a sequence. Let's use a sentence like "The sleepy child reads a book". The words in the sentence correspond to the keys. If we represent our words as embeddings, our keys will be rank 2. For example, the word "sleepy" might be represented by an embedding vector of length 2:  $[2, 0, 1]$ , where these embedding values are trained or taken from a standard language embedding. By convention, the zeroth axis of keys will be the position in the sequence and the first axis contains these vectors. The query is often an element from the keys, like the word "book." The point of attention is to see what parts of the sentence the query should be influenced by. "Book" should have strong attention on "child" and "reads," but probably not to "sleepy." You'll see soon that we will actually compute this as a vector, called the attention vector  $\vec{b}$ . The output from the attention layer will be a reduction over the values where each element of values is weighted by the attention between the query and the key. Thus there should be one key and one value for each element in our sentence. The values could be identical to the keys, which is common.

Let's see how this looks mathematically. The attention layer consists of two steps: (1) computing the attention vector  $\vec{b}$  using the **attention mechanism** and (2) the reduction over the values using the attention vector  $\vec{b}$ . Attention mechanism is a fancy word for the attention equation. Consider our example above. We'll use a 3-dimensional embedding for our words

Keys and queries as one-hot encodings will not work as inputs to attention layers because the dot product will give zeros, unless the key and query are equal.

Index	Embedding	Word
0	0,0,0	The
1	2,0,1	Sleepy
2	1,-1,-2	Child
3	2,3,1	Reads
4	-2,0,0	A
5	0,2,1	Book

The keys will be a rank 2 tensor (matrix) putting all these together. Note that these are only integers to make this example clearer, typically words are represented with floating point numbers when embedded.

$$\mathbf{K} = \begin{bmatrix} 0 & 2 & 1 & 2 & -2 & 0 \\ 0 & 0 & -1 & 3 & 0 & 2 \\ 0 & 1 & -2 & 1 & 0 & 1 \end{bmatrix} \quad (13.1)$$

They keys are shape (6, 3) because our sentence has 6 words and each word is represented with a 3 dimensional embedding vector. Let's make our values simple, we'll have one for each word. These values are what determine our output. Perhaps

they could be the sentiment of the word: is it a positive word (“happy”) or a negative word (“angry”).

$$\mathbf{V} = [0, -0.2, 0.3, 0.4, 0, 0.1] \quad (13.2)$$

Note that the values  $\mathbf{V}$  should be the same rank as the keys, so its shape is interpreted as  $(6, 1)$ . Finally, the query which should be one rank less than the keys. Our query is the word “book:”

$$\vec{q} = [0, 2, 1] \quad (13.3)$$

## 13.2 Attention Mechanism Equation

The attention mechanism equation uses query and keys arguments only. It outputs a tensor one rank less than the keys, giving a scalar for each key corresponding to the attention the query should have for the key. This attention vector should be normalized. The most common attention mechanism a dot product and softmax:

$$\vec{b} = \text{softmax}(\vec{q} \cdot \mathbf{K}) = \text{softmax}\left(\sum_j q_j k_{ij}\right) \quad (13.4)$$

where index  $i$  is the position in the sequence and  $j$  is the index of the feature. Softmax is defined as

$$\text{softmax}(\vec{x}) = \frac{e^{\vec{x}}}{\sum_i e^{\vec{x}_i}} \quad (13.5)$$

and ensures that  $\vec{b}$  is normalized. Substituting our values from the example above:

$$\vec{b} = \text{softmax}\left([0, 2, 1] \times \begin{bmatrix} 0 & 2 & 1 & 2 & -2 & 0 \\ 0 & 0 & -1 & 3 & 0 & 2 \\ 0 & 1 & -2 & 1 & 0 & 1 \end{bmatrix}\right) = \text{softmax}([0, 1, -4, 7, 0, 5]) \quad (13.6)$$

$$\vec{b} = [0, 0, 0, 0.88, 0, 0.12] \quad (13.7)$$

I’ve rounded the numbers here, but essentially the attention vector only gives weight to the word itself (book) and the verb “read”. I made this up, remember, but it gives you an idea of how attention gives you a way to connect words. It may even remind you of our graph neural network’s idea of neighbors.

## 13.3 Attention Reduction

After computing the attention vector  $\vec{b}$ , this is used to compute a weighted mean over the values:

$$\mathbf{V}\vec{b} = [0, 0, 0, 0.88, 0, 0.12]^T [0, -0.2, 0.3, 0.4, 0, 0.1] = 0.36 \quad (13.8)$$

Conceptually, our example computed the attention-weighted sentiment of the query word “book” in our sentence. You can see that attention layers do two things: compute an attention vector with the attention mechanism and then use it to take the attention-weighted average over the values.

## 13.4 Tensor-Dot

This dot product, softmax, and reduction is called a tensor-dot and is the most common attention layer[LPM15]. One common modification is to divide by the dimension of the keys (last axis dimension). Remember the keys are not normalized. If they are random numbers, the magnitude of the output from the dot product scales with the square root of the

dimension of the keys due to the central limit theorem. This can make the soft-max behave poorly, since you're taking  $e^{\vec{q} \cdot \mathbf{K}}$ . Putting this all together, the equation is:

$$\vec{b} = \text{softmax} \left( \frac{1}{\sqrt{d}} \vec{q} \cdot \mathbf{K} \right) \quad (13.9)$$

where  $d$  is the dimension of the query vector.

## 13.5 Soft, Hard, and Temperature Attention

One possible change to attention is to replace the softmax with a one at the position of highest attention and zero at all others. This is called **hard attention**. The equation for hard attention is to replace softmax with a “hardmax”, defined as

$$\text{hardmax}(\vec{x}) = \lim_{T \rightarrow 0} \frac{e^{\vec{x}/T}}{\sum_i e^{x_i/T}} \quad (13.10)$$

which is a mathematical way to formulate putting 1 in the position of the largest element of  $\vec{x}$  and a 0 at all others. The choice of  $T$  is for the word temperature, because this equation is similar to Boltzmann's distribution from statistical mechanics. You can see that limit  $T = 0$  is the hard attention,  $T = 1$  is the soft attention, and  $T = \infty$  means uniform attention. you could tune  $T$  to some intermediate values as well.

## 13.6 Self-Attention

Remember how everything is batched in deep learning? The batched input to an attention layer is usually the query. So although in the above discussion it was a tensor of one rank less than the keys (typically a query *vector*), once it has been batched it will be the same rank as the keys. Almost always, the query is in fact equal to the keys. Like in our example, our query was the embedding vector for the word “book”, which is one of the keys. If you consider the query to be batched so that you consider every word in the sentence, the query becomes equal to the keys. A further special case is when the query, values and keys are equal. This is called **self-attention**. This just means our attention mechanism uses the values directly and there is no extra set of “keys” input to the layer.

## 13.7 Trainable Attention

There are no trainable parameters in our definitions above. How can you do learning with attention? Typically, you don't have trainable parameters in equations directly. Instead, you put the keys, values, and query through a dense layer (see *Standard Layers*) before the attention. So when viewed as a layer, attention has no trainable parameters. When viewed as a block with a dense layer and attention layer, it is trainable. We'll see this now explicitly below.

## 13.8 Multi-head Attention Block

Inspired by the idea of convolutions with multiple filters, there is a block (group of layers) that splits to multiple parallel attentions. These are called “multi-head attention”. If your values are shape  $(L, V)$ , you will get back a  $(H, V)$  tensor, where  $H$  is the number of parallel attention layers (heads). If there are no trainable parameters in attention layers, what's the point of this though? Well, you must introduce weights. These are *square* weight matrices because we need all shapes to remain constant among all the attention heads.


Consider an attention layer to be defined by  $A(\vec{q}, \mathbf{K}, \mathbf{V})$ . The multi-head attention is

$$[A(\mathbf{W}_q^0 \vec{q}, \mathbf{W}_k^0 \mathbf{K}, \mathbf{W}_v^0 \mathbf{V}), A(\mathbf{W}_q^1 \vec{q}, \mathbf{W}_k^1 \mathbf{K}, \mathbf{W}_v^1 \mathbf{V}), \dots, A(\mathbf{W}_q^H \vec{q}, \mathbf{W}_k^H \mathbf{K}, \mathbf{W}_v^H \mathbf{V})] \quad (13.11)$$

where each element of the output vector [...] is itself an output vector from an attention layer, making  $H (L, V)$  shaped tensors. So the whole output is an  $(H, L, V)$  tensor. The most famous example of the multi-head attention block is in transformers[VSP+17] where they use self-attention multi-head attention blocks.

Typically we apply multiple sequential blocks of attention, so need the values input to the next block to be of rank 2 again (instead of the rank 3  $(H, L, V)$  tensor). Thus the output from the multi-head attention is often reduced by matrix multiplication with an  $(H, V, V)$  weight tensor or a  $(H)$  tensor of weights so that you get back to rank 2. If this seems confusing, see the example below.

## 13.9 Running This Notebook

Click the  above to launch this page as an interactive Google Colab.

### 13.10 Code Examples

Let's see how attention can be implemented in code. I will use random variables here for the different quantities but I will indicate which variables should be trained with  $w_{\cdot}$  and which should be inputs with  $i_{\cdot}$ .

#### 13.10.1 Tensor-Dot Mechanism

We'll begin with implementing the tensor-dot attention mechanism first. As an example, we'll use a sequence length of 11 and a keys feature length of 4 and a values feature dimension of 2. Remember the keys and query must share feature dimension size.

```
import numpy as np

def softmax(x, axis=None):
    return np.exp(x) / np.sum(np.exp(x), axis=axis)

def tensor_dot(q, k):
    b = softmax((k @ q) / np.sqrt(q.shape[0]))
    return b

i_query = np.random.normal(size=(4,))
i_keys = np.random.normal(size=(11, 4))

b = tensor_dot(i_query, i_keys)
print("b = ", b)
```

```
b = [0.07087918 0.07219164 0.06015269 0.10149468 0.10727669 0.09912065
      0.05998734 0.11274782 0.11425083 0.0857242  0.1161743 ]
```

As expected, we get out a vector  $\vec{b}$  whose sum is 1.

### 13.10.2 General Attention

Now let's put this attention mechanism into an attention layer.

```
def attention_layer(q, k, v):
    b = tensor_dot(q, k)
    return b @ v

i_values = np.random.normal(size=(11, 2))
attention_layer(i_query, i_keys, i_values)
```

```
array([-0.05282571, -0.31739908])
```

We get two values, one for each feature dimension.

### 13.10.3 Self-attention

The change in self-attention is that we make queries, keys, and values equal. We need to make a small change in that the queries are batched in this setting, so we should get a rank 2 output.

```
def batched_tensor_dot(q, k):
    # a will be batch x seq x feature dim
    # which is N x N x 4
    # batched dot product in einstein notation
    a = np.einsum("ij,kj->ik", q, k) / np.sqrt(q.shape[0])
    # now we softmax over sequence
    b = softmax(a, axis=1)
    return b

def self_attention(x):
    b = batched_tensor_dot(x, x)
    return b @ x

i_batched_query = np.random.normal(size=(11, 4))
self_attention(i_batched_query)
```

```
array([[ 1.97386118e-01,  1.08634210e+00,  4.33386250e-01,
        -3.35260258e-01],
       [-1.37192214e-01, -1.78211543e-01, -2.69559605e-01,
         8.99183346e-01],
       [-2.51258035e-01,  2.87696723e-01, -8.26667694e-01,
        -8.04712276e-01],
       [ 2.72029726e-01,  1.45580621e+00,  7.64246022e-02,
        -7.18977958e-01],
       [ 1.45461898e+00, -2.11470358e-01, -1.10760529e-01,
        -1.89908712e-01],
       [-4.65778961e-01, -1.89541037e-01, -8.71743678e-02,
        -6.86076184e-02],
       [-2.97814193e-01,  4.06323035e-02, -1.01299609e-01,
        -1.19522594e-01],
       [-5.13777234e-01,  4.67803325e-01, -1.14513341e+00,
```

(continues on next page)



(continued from previous page)

```

3.04912259e+00],
[ 8.75646635e-01, -8.42179012e-01,  2.06737738e-01,
 2.95851809e-03],
[-1.62593449e-01,  1.47639900e-01,  2.22949149e-01,
 2.42255767e-01],
[-1.83462799e-01, -4.84462962e-01,  4.53552354e-02,
-1.29851417e-01]])

```

We are given as output an  $11 \times 4$  matrix, which is correct.

### 13.10.4 Adding Trainable Parameters

You can add trainable parameters to these steps by adding a weight matrix. Let's do this for the self-attention. Although keys, values, and query are equal in self-attention, I can multiply them by different weights. Just to demonstrate, I'll have the values change to feature dimension 2.

```

# weights should be input feature_dim -> desired output feature_dim
w_q = np.random.normal(size=(4, 4))
w_k = np.random.normal(size=(4, 4))
w_v = np.random.normal(size=(4, 2))

def trainable_self_attention(x, w_q, w_k, w_v):
    q = x @ w_q
    k = x @ w_k
    v = x @ w_v
    b = batched_tensor_dot(q, k)
    return b @ v

trainable_self_attention(i_batched_query, w_q, w_k, w_v)

```

```

array([[ -8.58958244e-01, -5.13178623e-01],
 [ 1.10934020e+00, -9.60125897e-01],
 [-1.03148431e+02, -5.87849015e+01],
 [-7.45899826e+02, -4.25956711e+02],
 [-8.41749719e-01,  7.38908114e+00],
 [ 1.00737653e-01, -4.30026225e-02],
 [ 2.55700480e-02, -5.37187276e-02],
 [ 2.20041720e+00, -1.83378265e+00],
 [-3.58311728e-01,  2.57271825e+00],
 [ 3.43232307e-01, -1.98047339e-01],
 [ 3.80090445e-02,  1.48158328e-02]])

```

Since we had our values change to feature dimension 2 with the weights, we get out an  $11 \times 2$  output.

### 13.10.5 Multi-head

The only change for multi-head attention is that we have one set of weights for each head and we agree on how to combine after running through the heads. I'll just use a length  $H$  vector of trainable weights. Other strategies are to concatenate them or use a reduction (e.g., mean, max).

```
w_q_h1 = np.random.normal(size=(4, 4))
w_k_h1 = np.random.normal(size=(4, 4))
w_v_h1 = np.random.normal(size=(4, 2))
w_q_h2 = np.random.normal(size=(4, 4))
w_k_h2 = np.random.normal(size=(4, 4))
w_v_h2 = np.random.normal(size=(4, 2))
w_h = np.random.normal(size=2)

def multihead_attention(x, w_q_h1, w_k_h1, w_v_h1, w_q_h2, w_k_h2, w_v_h2):
    h1_out = trainable_self_attention(x, w_q_h1, w_k_h1, w_v_h1)
    h2_out = trainable_self_attention(x, w_q_h2, w_k_h2, w_v_h2)
    # join along last axis so we can use dot.
    all_h = np.stack((h1_out, h2_out), -1)
    return all_h @ w_h

multihead_attention(i_batched_query, w_q_h1, w_k_h1, w_v_h1, w_q_h2, w_k_h2, w_v_h2)
```

```
array([[ -6.63016684e-03,  1.27043347e+01],
       [ 2.95100409e+00, -2.37259549e+01],
       [ 1.43063566e-01, -1.54043340e+00],
       [ 9.04642783e-01,  4.07777159e+00],
       [ 8.16130174e+02,  1.30157842e+03],
       [-2.11198155e+00,  6.61297066e-02],
       [-5.95874193e-01, -5.99715526e-01],
       [ 3.30108456e+01, -3.06059541e+02],
       [ 2.12945954e+02,  3.73960944e+02],
       [-1.86914546e-01, -1.51180116e+00],
       [ 7.55107371e-02,  1.13459883e+00]])
```

As expected, we do get an  $11 \times 2$  rank 2 output.

## 13.11 Attention in Graph Neural Networks

Recall that the key attribute of a graph neural network is permutation equivariance. We used reductions like sum or mean over neighbors as the way to make the graph neural network layers be permutation equivariant. Attention layers are also permutation invariant (when not batched) and equivariant (when batched). This has made attention a popular choice for how to aggregate neighbor information. Attention layers are good at finding important neighbors and so are important with high-degree graphs (lots of neighbors). This is rare in molecules, but you can just define all atoms to be connected and then put distances as the edge attributes. Recall that graph convolution layers (GCN layer), and most GNN layers, only allow information to propagate one-bond per layer. Thus joining all atoms and using attention can give you long-range communication without so many layers. The disadvantage is that your network must now learn how to give attention to the correct bonds/atoms.

Let's see how attention fits into the Battaglia equations[BHB+18]. Recall that the Battaglia equations are general standard equations for defining a GNN. Attention can appear in multiple places, but as discussed above it appears when considering neighbors. Specifically, the query will be the  $i$ th node, and the keys/values will be some combination of neighboring node

and edge features. There is no step in the Battaglia equations where this fits neatly, but we can split up the attention layer as follows. Most of the attention layer will fit into the edge update equation:

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) \quad (13.12)$$

Recall that this is a general equation and our choice of  $\phi^e()$  defines the GNN.  $\vec{e}_k$  is the feature vector of edge  $k$ ,  $\vec{v}_{rk}$  is the receiving node feature vector for edge  $k$ ,  $\vec{v}_{sk}$  is the sending node feature vector for edge  $k$ ,  $\vec{u}$  is the global graph feature vector. We will use this step for attention mechanism where the query is the receiving node  $\vec{v}_{rk}$  and the keys/values are composed of the senders and edges vectors. To be specific, we'll use the approach from Zhang et al. [ZSX+18] with a tensor-dot mechanism. They only considered node features and set the keys and values to be identical as the node features. However, they put trainable parameters at each layer that translated the node features in to the keys/query.

$$\vec{q} = \mathbf{W}_q \vec{v}_{rk} \quad (13.13)$$

$$\mathbf{K} = \mathbf{W}_k \vec{v}_{sk} \quad (13.14)$$

$$\mathbf{V} = \mathbf{W}_v \vec{v}_{sk} \quad (13.15)$$

$$\vec{b}_k = \text{softmax}\left(\frac{1}{\sqrt{d}} \vec{q} \cdot \mathbf{K}\right) \quad (13.16)$$

$$\vec{e}'_k = \vec{b}_k \mathbf{V} \quad (13.17)$$

Putting it compactly into one equation:

$$\vec{e}'_k = \text{softmax}\left(\frac{1}{\sqrt{d}} \mathbf{W}_q \vec{v}_{rk} \cdot \mathbf{W}_k \vec{v}_{sk}\right) \mathbf{W}_v \vec{v}_{sk} \quad (13.18)$$

Now we have weighted edge feature vectors from the attention. Finally, we sum over these edge features in the edge aggregation step.

$$\vec{e}'_i = \rho^{e \rightarrow v}(E'_i) = \sum E'_i \quad (13.19)$$

In Zhang et al. [ZSX+18], they used multi-headed attention as well. How would multi-headed attention work? Your edge feature matrix  $E'_i$  now becomes an edge feature tensor, where axis 0 is edge ( $k$ ), axis 1 is feature, and axis 2 is the head. Recall that the “head” just means which set of  $\mathbf{W}_q^h, \mathbf{W}_k^h, \mathbf{W}_v^h$  we used. To reduce the tensor back to the expected matrix, we simply use another weight matrix that maps from the last two axes (feature, head) down to features only. I will write-out the indices explicitly to be more clear:

$$\vec{e}'_{il} = \rho^{e \rightarrow v}(E'_i) = \sum_k e'_{ikjh} w_{jhl} \quad (13.20)$$

where  $j$  is edge feature input index,  $l$  is our output edge feature matrix, and  $k, h, i$  are defined as before. **Transformer** is another name for a network built on multi-headed attention, so you'll also see transformer graph neural networks [MDM+20] building.

## 13.12 Chapter Summary

- Attention layers are inspired by human ideas of attention, but is fundamentally a weighted mean reduction.
- The attention layer takes in three inputs: the query, the values, and the keys. These inputs are often identical, where the query is one key and the keys and the values are equal.
- They are good at modeling sequences, such as language.
- The attention vector should be normalized, which can be achieved using a softmax activation function, but the attention mechanism equation is a hyperparameter.

- Attention layers compute an attention vector with the attention mechanism, and then reduce it by computing the attention-weighted average.
- Using hard attention (hardmax function) returns the maximum output from the attention mechanism.
- The tensor-dot followed by a softmax is the most common attention mechanism.
- Self-attention is achieved when the query, values, and the keys are equal.
- Attention layers by themselves are not trainable.
- Multi-head attention block is a group of layers that splits to multiple parallel attentions.

### 13.13 Cited References

## DEEP LEARNING ON SEQUENCES

Deep learning on sequences is part of a broader long-term effort in machine learning on sequences. *Sequences* is a broad term that includes text, integer sequences, DNA, and other ordered data. Deep learning on sequences often intersects with another field called natural language processing (NLP). NLP is a much broader field than deep learning, but there is quite a bit of overlap with sequence modeling.

We'll focus on the application of deep learning on sequences and NLP to molecules and materials. NLP in chemistry would at first appear to be a rich area, especially with the large amount of historic chemistry data existing only in plain text. However, the most work in this area has been on representations of molecules *as text* via the SMILES[Wei88] (and recently SELFIES [KHN+20]) encoding. There is nothing *natural* about SMILES and SELFIES though, so we should be careful to discriminate between work on natural language (like identifying names of compounds in a research article) from predicting the solubility of a compound from its SMILES. I hope there will be more NLP in the area, but publishers prevent bulk access/ML on publications. Few corpuses (collections of natural language documents) exist for NLP on chemistry articles.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Attention Layers*. After completing this chapter, you should be able to

- Define natural language processing and sequence modeling
- Recognize and be able to encode molecules into SMILES or other string encodings
- Understand RNNs and know some layer types
- Construct RNNs in seq2seq or seq2vec configurations
- Know the transformer architecture
- Understand how design can be made easier in a latent space

---

One advantage of working with molecules as text relative to graph neural networks (GNNs) is that existing ML frameworks have many more features for working with text, due to the strong connection between NLP and sequence modeling. Another reason is that it is easier to train generative models, because generating valid text is easier than generating valid graphs. You'll thus see generative/unsupervised learning of chemical space more often done with sequence models, whereas GNNs are typically better for supervised learning tasks and can incorporate spatial features (e.g., [YCW20, KGrossGunnemann20]). Outside of deep learning, graphical representations are in viewed as more robust than text encodings when used in methods like genetic algorithms and chemical space exploration [BFSV19]. NLP with sequence models can also be used to understand natural language descriptions of materials and molecules, which is essential for *materials* that are defined with more than just the molecular structure.

In sequence modeling, unsupervised learning is very common. We can predict the probability of the next token (word or character) in a sequence, like guessing the next word in a sentence. This does not require labels, because we only need examples of the sequences. It is also called pre-training, because it precedes training on sequences with labels. For chemistry this could be predicting the next atom in a SMILES string. For materials, this might be predicting the next

word in a synthesis procedure. These pre-trained have statistical model of a language and can be fine-tuned (trained a second time) for a more specific task, like predicting if a molecule will bind to a protein.

Another common task in sequence modeling (including NLP) is to convert sequences into continuous vectors. This doesn't always involve deep learning. Models that can embed sequences into a vector space are often called seq2vec or x2vec, where x might be molecule or synthesis procedure.

Finally, we often see *translation* tasks where we go from one sequence language to another. A sequence to sequence model (seq2seq) is similar to pre-training because it actually predicts probabilities for the output sequence.

## 14.1 Converting Molecules into Text

Before we can begin to use neural networks, we need to convert molecules into text. Simplified molecular-input line-entry system (SMILES) is a de facto standard for converting molecules into a string. SMILES enables molecular structures to be correctly saved in spreadsheets, databases, and input to models that work on sequences like text. Here's an example SMILES string: CC(NC)CC1=CC=C(OCO2)C2=C1. SMILES was crucial to the field of cheminformatics and is widely used today beyond deep learning. Some of the first deep learning work was with SMILES strings because of the ability to apply NLP models to SMILES strings.

Let us imagine SMILES as a function whose domain is molecular graphs (or some equivalent complete description of a molecule) and the image is a string. This can be thought of as an **encoder** that converts a molecular graph into a string. The SMILES encoder function is not surjective – there are many strings that cannot be reached from decoding graphs. The SMILES encoder function is injective – each graph has a different SMILES string. The inverse of this function, the **SMILES decoder**, cannot have the domain of all strings because some strings do not decode to valid molecular graphs. This is because of the syntax rules of SMILES. Thus, we can regard the domain to be restricted to *valid* SMILES string. In that case, the decoder is surjective – all graphs are reachable via a SMILES string. The decoder is not injective – multiple graphs can be reached by SMILES string.

This last point, the non-injectivity of a SMILES decoder, is a problem identified in database storage and retrieval of compounds. Since multiple SMILES strings map to the same molecular graph, it can happen that multiple entries in a database are actually the same molecule. One way around this is **canonicalization** which is a modification to the encoder to make a unique SMILES string. It can fail though [OBoyle12]. If we restrict ourselves to valid, canonical SMILES, then the SMILES decoder function is injective and surjective – bijective.

### idempotent

One way to assess correctness of the canonicalization process is by testing the idempotent property of the SMILES encoder/decoder pair. That is, if we decode/encode a canonical SMILES string we should get back the same string.

The difficulty of canonicalization and thus perceived weakness of SMILES in creating unique strings led (in part) to the creation of InChi strings. InChI is an alternative that is inherently canonical. InChI strings are typically longer and involve more tokens, which seems to affect their use in deep learning. InChI as a representation is often worse with the same amount of data vs SMILES.

If you've read the previous chapters on equivariances (*Input Data & Equivariances* and *Equivariant Neural Networks*), a natural question is if SMILES is permutation invariant. That is, if you change the order of atoms in the molecular graph that has no effect on chemistry, is the SMILES string identical? Yes, if you use the canonical SMILES. So in a supervised setting, using canonical SMILES gives an atom ordering permutation invariant neural network because the representation *will not* be permuted after canonicalization. Be careful; you should not trust that SMILES you find in a dataset are canonical.

### 14.1.1 SELFIES

Recent work from Krenn et al. developed an alternative approach to SMILES called SELF-referencIng Embedded Strings (SELFIES)[[KHN+20](#)]. Every string is a valid molecule. Note that the characters in SELFIES are not all ASCII characters, so it's not like every sentence encodes a molecule (would be cool though). SELFIES is an excellent choice for generative models because any SELFIES string automatically decodes to a valid molecule. SELFIES, as of 2021, is not directly canonicalized though and thus is not permutation invariant by itself. However, if you add canonical SMILES as an intermediate step, then SELFIES are canonical. It seems that models which output a molecule (generative or supervised) benefit from using SELFIES instead of SMILES because the model does not need to learn how to make valid strings – all strings are already valid SELFIES [[RZS20](#)]. This benefit is less clear in supervised learning and no difference has been observed empirically[[CGR20](#)]. Here's a blog post giving an [overview of SELFIES and its applications](#).

### 14.1.2 Demo

You can get a sense for SMILES and SELFIES in this [demo page](#) that uses a RNN (discussed below) to generate SMILES and SELFIES strings.

### 14.1.3 Stereochemistry

SMILES and SELFIES can treat stereoisomers, but there are a few complications. `rdkit`, the dominant Python package, [cannot treat non-tetrahedral chiral centers with SMILES](#) as of 2021. For example, even though SMILES according to its specification can correctly distinguish cisplatin and transplatin, the implementation of SMILES in `rdkit` cannot. Other examples of chirality that are present in the SMILES specification but not implementations are planar and axial chirality. SELFIES relies on SMILES (specifically the `rdkit` implementation) and thus is also susceptible to this problem. This is an issue for any organometallic compounds. In organic chemistry though, most chirality is tetrahedral and correctly treated by `rdkit`.

### 14.1.4 What is a chemical bond?

More broadly, the idea of a chemical bond is a concept created by chemists [[Bal11](#)]. You cannot measure the existence of a chemical bond in the lab and it is not some quantum mechanical operator with an observable. There are certain molecules which cannot be represented by classic single,double,triple,aromatic bonded representations, like ferrocene or diborane. This bleeds over to text encoding of a molecule where the bonding topology doesn't map neatly to bond order. The specific issue this can cause is that multiple unique molecules may appear to have the same encoding (non-injective). In situations like this, it is probably better to just work with the exact 3D coordinates and then bond order or type is less important than distance between atoms.

## 14.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

## 14.3 Recurrent Neural Networks

String is a synonym for sequence here. Character and symbol are synonyms for token (single element of the string).

Recurrent neural networks (RNN) have been by far the most popular approach to working with molecular strings. RNNs have a critical property that they can have different length input sequences, making it appropriate for SMILES or SELFIES which both have variable length. RNNs have recurrent layers that consume an input sequence element-by-element. Consider an input sequence  $\mathbf{X}$  which is composed of a series of vectors (recall that characters or words can be represented with one-hot or embedding vectors)  $\mathbf{X} = [\vec{x}_0, \vec{x}_1, \dots, \vec{x}_L]$ . The RNN layer function is binary and takes as input the  $i$ th element of the input sequence and the output from the  $i - 1$  layer function. You can write it as:

$$f(f \dots f(\vec{x}_0, \vec{0}), \vec{x}_1, \vec{x}_2) \dots \vec{x}_L) \quad (14.1)$$

Commonly we would like to actually see and look at these intermediate outputs from the layer function  $f_4(\vec{x}_4, f_3(\dots)) = \vec{h}_4$ . These  $\vec{h}$ s are called the hidden state because of the connection between RNNs and Markov State Models. We can **unroll** our picture of an RNN to be:

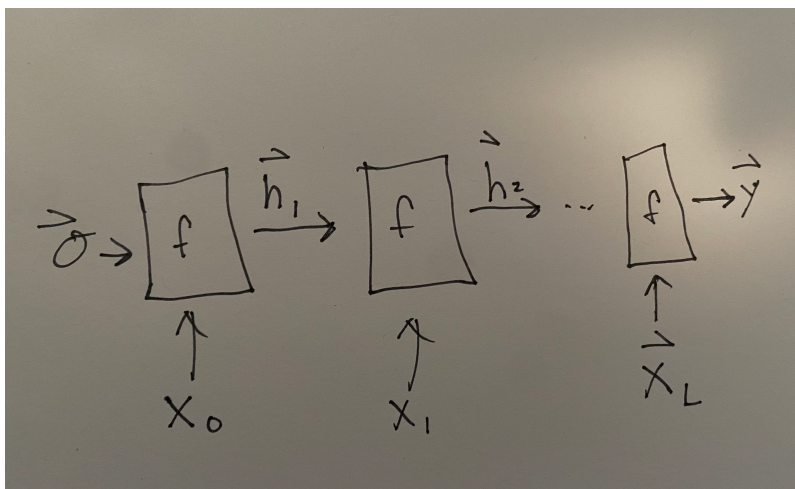


Fig. 14.1: Unrolled picture of RNN.

where the initial hidden state is assumed to be  $\vec{0}$ , but could be trained. The output at the end is shown as  $\vec{y}$ . Notice there are no subscripts on  $f$  because we use the same function and weights at each step. This re-use of weights makes the choice of parameter number independent of input lengths, which is also necessary to make the RNN accommodate arbitrary length input sequences. It should be noted that the length of  $\vec{y}$  may be a function of the input length, so that the  $\vec{h}_i$  may be increasing in length at each step to enable an output  $\vec{y}$ . Some diagrams of RNNs will show that by indicating a growing output sequence as an additional output from  $f(\vec{x}_i, \vec{h}_{i-1})$ .

Interestingly, the form of  $f(\vec{x}, \vec{h})$  is quite flexible based on the discussion above. There have been hundreds of ideas for the function  $f$  and it is problem dependent. The two most common are long short-term memory (LSTM) units and gated recurrent unit (GRU). You can spend quite a bit of time trying to reason about [these functions](#), understanding how [gradients propagate nicely through them](#), and there is an analogy about how they are inspired by human memory. Ultimately, they are used because they perform well and are widely-implemented so we do not need to spend much time on these details. The main thing to know is that GRUs are simpler and faster, but LSTMs seem to be better at more difficult sequences. Note that  $\vec{h}$  is typically 1-3 different quantities in modern implementations. Another details is the word **units**. Units are like the hidden state dimension, but because the hidden state could be multiple quantities (e.g., LSTM) we do not call it dimension.



Actually, they are not used so much anymore because transformers seem to be a direct replacement for RNNs.

The RNN layer allows us to input an arbitrary length sequence and outputs a label which could depend on the length of the input sequence. You can imagine that this could be used for regression or classification.  $\hat{y}$  would be a scalar. Or you could take the output from an RNN layer into an MLP to get a class.

### 14.3.1 Generative RNNs

An interesting use case for an RNN is in unsupervised generative models, where we try to predict new examples. This means that we're trying to learn  $P(\mathbf{X})$  [SKTW18]. With a generative RNN, we predict the sequence one symbol at a time by conditioning on a growing sequence. This is called **autoregressive** generation.

$$P(\mathbf{X}) = \prod P(\vec{x}_L | \vec{x}_{L-1}, \vec{x}_{L-2}, \dots, \vec{x}_0) \dots P(\vec{x}_1 | \vec{x}_0) P(\vec{x}_0) \quad (14.2)$$

This is also called **self-supervised** instead of unsupervised learning. The distinction is that we're creating labels by chopping up our training data – so it is supervised. But it's not quite supervised because labels do not need to be supplied.

The RNN is trained to take as input a sequence and output the probability for the next character. Our network is trained to be this conditional probability:  $P(\vec{x}_i | \vec{x}_{L-i}, \vec{x}_{L-i-1}, \dots, \vec{x}_0)$ . What about the  $P(\vec{x}_0)$  term? Typically we just *pick* what the first character should be. Or, we could create an artificial “start” character that marks the beginning of a sequence (typically 0) and always choose that.

We can train the RNN to agree with  $P(\vec{x}_i | \vec{x}_{L-i}, \vec{x}_{L-i-1}, \dots, \vec{x}_0)$  by taking an arbitrary sequence  $\vec{x}$  and choosing a split point  $\vec{x}_i$  and training on the proceeding sequence elements. This is just multi-class classification. The number of classes is the number of available characters and our model should output a probability vector across the classes. Recall the loss for this cross-entropy.

When doing this process with SMILES an obvious way to judge success would be if the generated sequences are valid SMILES strings. This at first seems reasonable and was used as a benchmark for years in this topic. However, this is a low-bar: we can find valid SMILES in much more efficient ways. You can download 77 million SMILES [CGR20] and you can find vendors that will give you a multi-million entry database of purchasable molecules. You can also just use SELFIES and then an untrained RNN will generate only valid strings, since SELFIES is bijective. A more interesting metric is to assess if your generated molecules are in the same region of chemical space as the training data [SKTW18]. I believe though that generative RNNs are relatively poor compared with other generative models in 2021. They are still strong though when composed with other architectures, like VAEs [GomezBWD+18] or encoder/decoder [RZS20].

You can see a worked out example in [Generative RNN in Browser](#).

## 14.4 Masking & Padding

As in our [Graph Neural Networks](#) chapter, we run into issues with variable length inputs. The easiest and most compute efficient way to treat this is to pad (and/or trim) all strings to be the same length, making it easy to batch examples. A memory efficient way is to not batch and either batch gradients as a separate step or trim your sequences into subsequences and save the RNN hidden-state between them. Due to the way that NVIDIA has written RNN kernels, padding should always be done on the right (sequences all begin at index 0). The character used for padding is typically 0. Don't forget, we will always first convert our string characters to integers corresponding to indices of our vocabulary (see [Standard Layers](#)). Thus, remember to make sure that the index 0 should be reserved for padding.

Masking is used for two things. Masking is used to ensure that the padded values are not accidentally considered in training. This is framework dependent and you can read about [Keras here](#), which is what we'll use. The second use for masking is to do element-by-element training like the generative RNN. We train each time with a shorter mask, enabling it to see more of the sequence. This prevents you from needing to slice-up the training examples into many shorter sequences. This idea of a right-mask that prevents the model for using characters farther in the sequence is sometimes called **causal masking** because we're preventing characters from the "future" affecting the model.

## 14.5 RNN Solubility Example

Let's revisit our solubility example from before. We'll use a GRU to *encode* the SMILES string into a vector and then apply a dense layer to get a scalar value for solubility. Let's revisit the solubility AqSolDB[SKE19] dataset from [Regression & Model Assessment](#). Recall it has about 10,000 unique compounds with measured solubility in water (label) and their SMILES strings. Many of the steps below are explained in the [Standard Layers](#) chapter that introduces Keras and the principles of building a deep model.

I've hidden the cell below which sets-up our imports and shown a few rows of the dataset.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import dmol

soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/master/data/curated-solubility-dataset.
    ↪CSV"
)
features_start_at = list(soldata.columns).index("MolWt")
np.random.seed(0)
```

```
soldata.head()
```

	ID	Name \
0	A-3	N,N,N-trimethyloctadecan-1-aminium bromide
1	A-4	Benzo[cd]indol-2(1H)-one
2	A-5	4-chlorobenzaldehyde
3	A-8	zinc bis[2-hydroxy-3,5-bis(1-phenylethyl)benzo...
4	A-9	4-({4-[bis(oxiran-2-ylmethyl)amino]phenyl}meth...

	InChI \
0	InChI=1S/C21H46N.BrH/c1-5-6-7-8-9-10-11-12-13-...
1	InChI=1S/C11H7NO/c13-11-8-5-1-3-7-4-2-6-9(12-1...
2	InChI=1S/C7H5ClO/c8-7-3-1-6(5-9)2-4-7/h1-5H
3	InChI=1S/2C23H22O3.Zn/c2*1-15(17-9-5-3-6-10-17...
4	InChI=1S/C25H30N2O4/c1-5-20(26(10-22-14-28-22)...

	InChIKey \
0	SZEMGTQCPRXEG-UHFFFAOYSA-M
1	GPYLCFQEKPUWLD-UHFFFAOYSA-N
2	AVPYQKSLYISFPO-UHFFFAOYSA-N
3	XTUPUYCJWKHGSW-UHFFFAOYSA-L
4	FAUAZXVRLVIARB-UHFFFAOYSA-N

(continues on next page)

(continued from previous page)

	SMILES	Solubility	SD	\
0	[Br-].CCCCCCCCCCCCCCCC[N+](C)(C)C	-3.616127	0.0	
1	O=C1Nc2cccc3cccc1c23	-3.254767	0.0	
2	Clc1ccc(C=O)cc1	-2.177078	0.0	
3	[Zn++].CC(c1cccc1)c2cc(C(C)c3cccc3)c(O)c(c2)...	-3.924409	0.0	
4	C1OC1CN(CC2CO2)c3ccc(Cc4ccc(cc4)N(CC5CO5)CC6CO...	-4.662065	0.0	

	Ocurrences	Group	MolWt	...	NumRotatableBonds	NumValenceElectrons	\
0	1	G1	392.510	...	17.0	142.0	
1	1	G1	169.183	...	0.0	62.0	
2	1	G1	140.569	...	1.0	46.0	
3	1	G1	756.226	...	10.0	264.0	
4	1	G1	422.525	...	12.0	164.0	

	NumAromaticRings	NumSaturatedRings	NumAliphaticRings	RingCount	TPSA	\
0	0.0	0.0	0.0	0.0	0.00	
1	2.0	0.0	1.0	3.0	29.10	
2	1.0	0.0	0.0	1.0	17.07	
3	6.0	0.0	0.0	6.0	120.72	
4	2.0	4.0	4.0	6.0	56.60	

	LabuteASA	BalabanJ	BertzCT
0	158.520601	0.000000e+00	210.377334
1	75.183563	2.582996e+00	511.229248
2	58.261134	3.009782e+00	202.661065
3	323.755434	2.322963e-07	1964.648666
4	183.183268	1.084427e+00	769.899934

[5 rows x 26 columns]

We'll extract our labels and convert SMILES into padded characters. We make use of a **tokenizer**, which is essentially a look-up table for how to go from the characters in a SMILES string to integers. To make our model run faster, I will filter out very long SMILES strings.

```
# filter out long smiles
smask = [len(s) <= 96 for s in soldata.SMILES]
print(f"Removed {soldata.shape[0] - sum(smask)} long SMILES strings")
filtered_soldata = soldata[smask]

# make tokenizer with 128 size vocab and
# have it examine all text in dataset
vocab_size = 128
tokenizer = tf.keras.preprocessing.text.Tokenizer(
    vocab_size, filters="", char_level=True
)
tokenizer.fit_on_texts(filtered_soldata.SMILES)
```

Removed 285 long SMILES strings

```
# now get padded sequences
seqs = tokenizer.texts_to_sequences(filtered_soldata.SMILES)
padded_seqs = tf.keras.preprocessing.sequence.pad_sequences(seqs, padding="post")

# Now build dataset
```

(continues on next page)

(continued from previous page)

```

data = tf.data.Dataset.from_tensor_slices((padded_seqs, filtered_soldata.Solubility))
# now split into val, test, train and batch
N = soldata.shape[0]
split = int(0.1 * N)
test_data = data.take(split).batch(16)
nontest = data.skip(split)
val_data, train_data = nontest.take(split).batch(16), nontest.skip(split).shuffle(
    1000
).batch(16)

```

We're now ready to build our model. We will just use an embedding then RNN and some dense layers to get to a final predicted solubility.

```

model = tf.keras.Sequential()

# make embedding and indicate that 0 should be treated as padding mask
model.add(
    tf.keras.layers.Embedding(input_dim=vocab_size, output_dim=16, mask_zero=True)
)

# RNN layer
model.add(tf.keras.layers.GRU(32))
# a dense hidden layer
model.add(tf.keras.layers.Dense(32, activation="relu"))
# regression, so no activation
model.add(tf.keras.layers.Dense(1))

model.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
embedding (Embedding)	(None, None, 16)	2048
gru (GRU)	(None, 32)	4800
dense (Dense)	(None, 32)	1056

dense_1 (Dense)	(None, 1)	33
-----------------	-----------	----

```
=====
```

```
Total params: 7,937
```

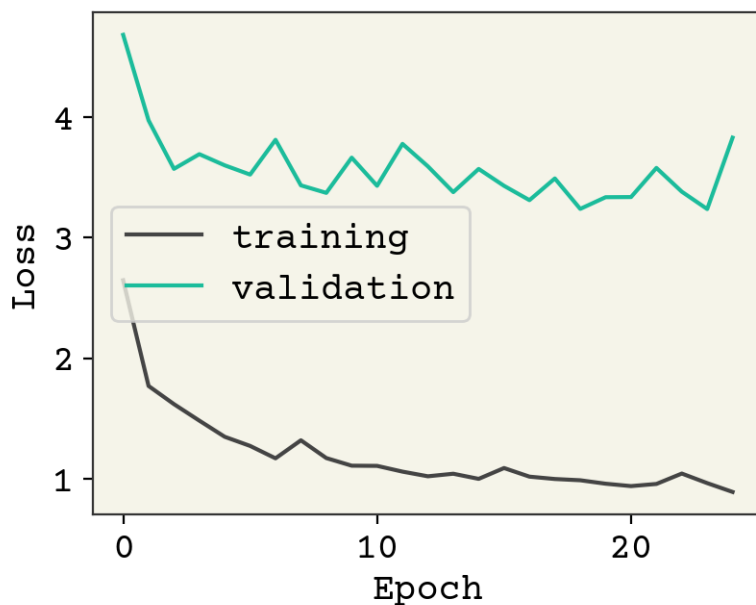
```
Trainable params: 7,937
```

```
Non-trainable params: 0
```

Now we'll compile our model and train it. This is a regression problem, so we use mean squared error for our loss.

```
model.compile(tf.optimizers.Adam(1e-2), loss="mean_squared_error")
result = model.fit(train_data, validation_data=val_data, epochs=25, verbose=0)
```

```
plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



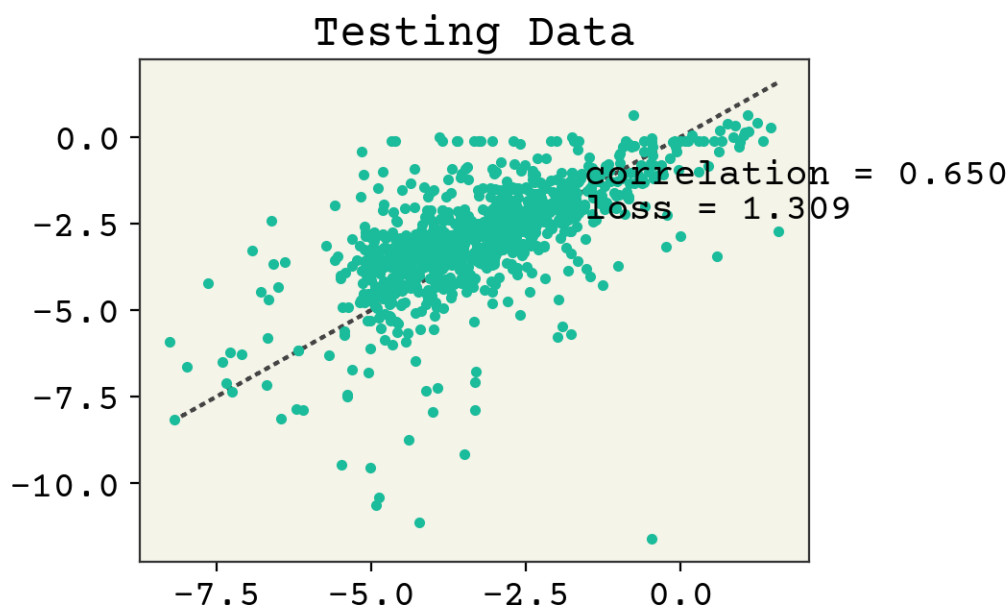
As usual, we could keep training and I encourage you to explore adding regularization or modifying the architecture. Let's now see how the test data looks.

```
# evaluate on test data
```

```
yhat = []
test_y = []
for x, y in test_data:
    yhat.extend(model(x).numpy().flatten())
    test_y.extend(y.numpy().flatten())
yhat = np.array(yhat)
test_y = np.array(test_y)
```

```
# plot test data
```

```
plt.plot(test_y, test_y, ":")
plt.plot(test_y, yhat, ".")
plt.text(min(y) + 1, max(y) - 2, f"correlation = {np.corrcoef(test_y, yhat)[0,1]:.3f}"
        + "\n")
plt.text(min(y) + 1, max(y) - 3, f"loss = {np.sqrt(np.mean((test_y - yhat)**2)):.3f}")
plt.title("Testing Data")
plt.show()
```



Linear regression from *Regression & Model Assessment* still wins, but this demonstrates the use of an RNN for this task.

## 14.6 Transformers

Transformers have been well-established now as the current state of the art for language modeling tasks. The transformer architecture is actually just self-attention repeated in multiple layers. The paper describing the architecture was quite a breakthrough. At the time, the best models used convolutions, recurrence, attention and encoder/decoder. The paper title was “attention is all you need” and that is basically the conclusion [VSP+17]. They found that multi-head attention (including self-attention) was what mattered and this led to **transformers**. Transformers are simple and *scalable* because each layer is nearly the same operation. This has led to simple “scaling-up the language model” resulting in things like GPT-3, which has billions of parameters and cost millions of dollars to train. GPT-3 is also surprisingly good and versatile. The single model is able to answer questions, describe computer code, translate languages, and infer recipe instructions for cookies. I highly recommend reading the paper, it’s quite interesting[BMR+20].

Considering the whole sequence simultaneously is also possible with bi-directional RNNs that read a sequence simultaneously from both ends – meaning context before and after a missing character can be used for training.

There are two principles from the transformer that interest us. One is of course that it is a simple and effective replacement for RNNs. The second is that the transformer considers the whole sequence simultaneously. This has a few consequences. The first is that it is again input size dependent. However, we can pad and mask to get around that. The second consequence is that the self-supervised/unsupervised training can be more interesting than just predict the next character in the string. Instead, we can randomly delete characters and ask the transformer to infer the missing character. This is how transformers are typically “pre-trained” – by feeding a bunch of masked sequences to teach the transformer the language. Then, if desired, the transformer can be refined with labels on your specific task. Transformers and their pre-training training procedure have led to pre-trained chemistry specific models that can be downloaded and used immediately on chemistry data, like ChemBERTa [CGR20]. These pre-trained models have been trained on 77 million molecules and so should already have some “intuition” about molecular structures and they indeed do well on supervised learning tasks.

### 14.6.1 Architecture

The transformer is fundamentally made-up of layers of multi-head attention blocks as discussed in *Attention Layers*. You can get a detailed overview of the [transformer architecture here](#). The overall architecture is an encoder/decoder like seen in *Variational Autoencoder*. Like the variational autoencoder, the decoder portion can be discarded and only the encoder is used for supervised tasks. Thus, you might **pre-train** the encoder/decoder with self-supervised training (strings with withheld characters) on a large dataset without labels and then use only the encoder for a regression tasks with a smaller dataset.

Some have recently argued that convolutions might be as effective as transformers with some tuning. If substantiated, this could upend NLP because convolutions are simpler to understand, parallelize, and interpret. [TDG+21]

What exactly is going in and out of the encoder/decoder? The transformer is an example of a sequence to sequence (seq2seq) model and the most obvious interpretation is translating between two languages like English to French. The encoder takes in English and the decoder produces French. Or maybe SMILES to IUPAC name. However, that requires “labels” (the paired sequence). To do self-supervised training pre-training, we need the input to the encoder to be a sequence missing some values and the decoder output to be the same sequence with probabilities for each position values filled in. This is called **masked** self-supervised training. If you pre-train in this way, you can do two tasks with your pre-trained encoder/decoder. You can use the encoder alone as a way to embed a string into real numbers and then a downstream task like predicting a molecule’s enthalpy of formation from its SMILES string. The other way to use a model trained this way is for autoregressive generation. The input might be a few characters or a *prompt* [RM21] specifically crafted like a question. This is similar the generative RNN, although it allows more flexibility.

Self-supervised training is not just for transformers! It has been successfully applied to graph neural networks as well [WWCF21].

There are many details to transformers and “hand-tuned” hyperparameters. Examples in modern transformers are layer normalizations (similar to batch normalization), embeddings, dropout, weight decay, learning rate decay, and position information encoding [LOG+19]. Position information is quite an interesting topic – you need to include the location of a token (character) in its embedding. Was it the first character or last character? This is key because when you compute

the attention between tokens, the relative location is probably important. Some recent promising work proposed a kind of phase/amplitude split, where the position is the phase and the amplitude is the embedding[SLP+21].

If you would like to see how to implement a real transformer with most of these details, take a look at this [Keras tutorial](#). Because transformers are so tightly coupled with pre-training, there has been a great deal of effort in pre-training models. Aside from [GPT-3](#), a general model pre-trained on an enormous corpus of billions of sequences from multiple languages, there are many language specific pre-trained models. [Hugging Face](#) is a company and API that hosts pre-trained transformers for specific language models like Chinese language, XML, SMILES, or question and answer format. These can be quickly downloaded and utilized, enabling rapid use of state-of-the art language models.

## 14.7 Transformer Example

Let's see a complete Transformer example using the `transformer` library. There are three steps we'll take:

1. Tokenization
2. Training
3. Inference

The tokenization is a more principled way of converting SMILES or other text into integers. The embedding layer we had above converted each character into a single integer, but it may be better to put (OH) – an alcohol group – into its own token.

## 14.8 Using the Latent Space for Design

One of the most interesting applications of these encoder/decoder seq2seq models in chemistry is their use for doing optimal design of a molecule. We pre-train an encoder/decoder pair with masking. The encoder brings our molecule to a continuous representation (seq2vec). Then we can do regression in this vector space for whatever property we would like (e.g., solubility). Then we can optimize this regressed model, finding an input vector that is a minimum or maximum, and finally convert that input vector into a molecule using the decoder [GomezBWD+18]. The vector space output by the encoder is called the **latent space** like we saw in *Variational Autoencoder*. Of course, this works for RNN seq2seq models, transformers, or convolutions.

## 14.9 Representing Materials as Text

Materials are an interesting problem for deep learning because they are not defined by a single molecule. There can be information like the symmetry group or components/phases for a composite material. This creates a challenge for modeling, especially for real materials that have complexities like annealing temperature, additives, and age. From a philosophical point of view, a material is defined by how it was constructed. Practically that means a material is defined by the text describing its synthesis [BDC+18]. This is an idea taken to its extreme in Tshitoyan et al. [TDW+19] who found success in representing thermoelectrics via the text describing their synthesis [SC16]. This work is amazing to me because they had to manually collect papers (publishers do not allow ML/bulk download on articles) and annotate the synthesis methods. Their seq2vec model is relatively old (2 years!) and yet there has not been much progress in this area. I think this is a promising direction but challenging due to the data access limitations. For example, recent progress by Friedrich et al. [FAT+20] built a pre-trained transformer for solid oxide fuel cells materials but their corpus was limited to open access articles (45) over a 7 year period. This is one critical line of research that is limited due to copyright issues. Text can be copyrighted, not data, but maybe someday a court can be convinced that they are interchangeable.



## 14.10 Applications

As discussed above, molecular design has been one of the most popular areas for sequence models in chemistry [SKTW18, GomezBWD+18, MFGS18]. Transformers have been found to be excellent at predicting chemical reactions. Schwaller et al. [SPZ+20] have shown how to do retrosynthetic pathway analysis with transformers. The transformers take as input just the reactants and reagents and can predict the products. The models can be calibrated to include uncertainty estimates [SLG+19] and predict synthetic yield [SVLR20]. Beyond taking molecules as input, Vaucher et al. trained a seq2seq transformer that can translate the unstructured methods section of a scientific paper into a set of structured synthetic steps [VZG+20]. Finally, Schwaller et al. [SPV+21] trained a transformer to classify reactions into organic reaction classes leading to a [fascinating map of chemical reactions](#).

## 14.11 Summary

- Text is a natural representation of both molecules and materials
- SMILES and SELFIES are ways to convert molecules into strings
- Recurrent neural networks (RNNs) are an input-length independent method of converting strings into vectors for regression or classification
- RNNs can be trained in seq2seq (encoder/decoder) setting by having it predict the next character in a sequence. This yields a model that can autoregressively generate new sequences/molecules
- Withholding or masking sequences for training is called self-supervised training and is a pre-training step for seq2seq models to enable them to learn the properties of a language like English or SMILES
- Transformers are currently the best seq2seq models
- The latent space of seq2seq models can be used for molecular design
- Materials can be represented as text which is a complete representation for many materials

## 14.12 Cited References



## VARIATIONAL AUTOENCODER

A variational autoencoder (VAE) is a kind of **generative** deep learning model that is capable of **unsupervised learning** [KW13]. Unsupervised learning is the process of fitting models to unlabeled data. A generative model is a specific kind of unsupervised learning model that is capable of *generating* new data points that were not seen in training. Generative models can be viewed as a trained probability distribution over that data:  $\hat{P}(x)$ . You can then draw samples from this distribution. It is generally too difficult to construct  $\hat{P}(x)$  directly, and so most generative models make some changes to the structure.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Input Data & Equivariances*. It also assumes a good knowledge of probability theory, including conditional probabilities. You can read [my notes](#) or any introductory probability text to get an overview. After completing this chapter, you should be able to

- Understand the derivation for the loss function of a VAE
  - Construct an encoder/decoder pair in JAX and train it with the VAE loss function
  - Sample from the decoder
  - Rebalance VAE loss for reconstruction or disentangling
- 

A VAE approaches this problem by introducing a dummy random variable  $z$ , which we define to have a known distribution (e.g., normal). We can then rewrite  $\hat{P}(x)$  as:

$$\hat{P}(x) = \int \hat{P}(x|z) P(z) dz \quad (15.1)$$

using the definition of a marginal and conditional probability. Training  $\hat{P}(x|z)$  directly is not really possible either, but we can create a symmetric distribution  $\hat{P}(z|x)$  and train both simultaneously. This symmetric distribution only is created to help us train; our end goal is to find  $\hat{P}(x|z)$  so that we can obtain  $\hat{P}(x)$ . VAEs were first introduced in [KW13].

A VAE is thus a set of two trained conditional probability distributions that operate on the data  $x$  and latent variables  $z$ . The first conditional is  $p_{\theta}(x|z)$ , where  $\theta$  indicates the trainable parameters that we will be fitting.  $p_{\theta}(x|z)$  is known as the “decoder” because it goes from the latent variable  $z$  to  $x$ . The decoder analogy is because you can view  $z$  as a kind of encoded compression of  $x$ . The other conditional is  $q_{\phi}(z|x)$  and is known as the encoder.

Remember we always know  $p(z)$  because we chose it to be a defined distribution — that is the key idea. We’re grounding our encoder/decoder by having them communicate through  $p(z)$ , which we know. For the rest of this chapter we’ll take  $p(z)$  to be a **standard normal distribution**.  $p(z)$  can be other distributions though. It can even be trained using the techniques from the *Normalizing Flows*.

## 15.1 VAE Loss function

To see how  $q_\phi(z|x)$  enables us to train, let's construct our loss. The loss function should only take in a value  $x_i$  and trainable parameters. There are no labels. Our goal is to make our VAE model be able to generate  $x_i$ , so the loss is the log likelihood that we saw  $x_i$ :  $\log [\hat{P}(x_i)]$ .

Log likelihood is the loss of choice for fitting distributions to data. It is a likelihood, not a probability, because the distribution parameters are changing, not the random variables (which are set to be the data). We take a log so that we can sum/average over data to aggregate multiple points due to properties of logs.

### 15.1.1 Derivation

**Note:** The derivation below is a little unusual. Most derivations rely on Bayes' theorem following a principle of evidence lower bound (ELBO). I thought I'd give a different derivation since you can readily find examples of [the ELBO in many places](#).

Remember we do not have an expression for  $\hat{P}(x_i)$ . We have  $p_\theta(x_i|z)$ . To connect them we'll use the following expression:

$$\log [\hat{P}(x_i)] = \log \left[ \int p_\theta(x_i|z) P(z) dz \right] = \log E_z [p_\theta(x_i|z)] \quad (15.2)$$

where we have rewritten the integral more compactly by using the definition of expectation. This expression requires integrating over the latent variable, which is not easy since as you can guess  $p_\theta(x|z)$  is a neural network and it's not straightforward to integrate over the input ( $z$ ) of a neural network. Instead, we can approximate this integral by sampling some  $z$ s from  $P(z)$

We actually could just integrate over the latent variables. This is called a normalizing flow and is a class of generative models we'll see later.

$$\log E_z [p_\theta(x_i|z)] \approx \log \left[ \frac{1}{N} \sum_j^N p_\theta(x_i|z_j) \right], z_j \sim P(z_j) \quad (15.3)$$

You'll find though that grabbing  $z$ 's from  $P(z)$  is not so efficient at approximating this integral, because you need the  $z$ 's to be likely to have led to the observed  $x_i$ . The integral is dominated by the  $p_\theta(x_i|z_j)$  terms. This is where we use  $q(z|x)$ : it can provide efficient guesses for  $z_j$ . To approximate  $\log E_z [p_\theta(x_i|z)]$  with samples from  $q(z|x_i)$ , we need to account for the fact that sampling from  $q(z|x_i)$  is not identical to sampling from  $P(z)$  by adding their ratio to the expression ([importance sampling](#)).

$$\log E_z [p_\theta(x_i|z)] \approx \log \left[ \frac{1}{N} \sum_j^N p_\theta(x_i|z_j) \frac{P(z_j)}{q_\phi(z_j|x_i)} \right], z_j \sim q_\phi(z_j|x_i) \quad (15.4)$$

The ratio of  $P(z)/q_\phi(z|x)$  enables our numerical approximation of the expectation. For notational purposes though I'll go back to the exact expression, with the understanding that when we go to implementation we'll use the numerical approximation:

$$\log E_z [p_\theta(x_i|z)] = \log E_{z \sim q_\phi(z|x_i)} \left[ p_\theta(x_i|z) \frac{P(z)}{q_\phi(z|x_i)} \right] \quad (15.5)$$

Notice how the expectation now is wrt  $z \sim q_\phi(z|x_i)$  since we have that importance sampling ratio in the expression.

Now if the log was on the inside of our expectation, we could simplify this. We can actually swap the order of expectation and the log using Jensen's Inequality for the concave log function. The consequence is that our loss is no longer an exact estimate of the log likelihood, but a lower bound.

$$\log \mathbb{E} [\dots] \geq \mathbb{E} [\log \dots] \quad (15.6)$$

We'll use that and can now separate into two terms by properties of the log

$$\mathbb{E}_{z \sim q_\phi(z|x_i)} \left[ \log \left( p_\theta(x_i|z) \frac{P(z)}{q_\phi(z|x_i)} \right) \right] = \mathbb{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \mathbb{E}_{z \sim q_\phi(z|x_i)} \left[ \log \left( \frac{P(z)}{q_\phi(z|x_i)} \right) \right] \quad (15.7)$$

Remember we always planned to re-introduce numerically approximate the expectation. However, the right-hand side does not involve  $p_\theta(x|z)$ , so we do not need to integrate over a neural network input. We just need to integrate over the output of  $q_\phi(z|x)$  and  $P(z)$ , which is a standard normal distribution. We'll see later on that we can make the output of  $q_\phi(z|x)$  specifically be a normal distribution to make sure we can easily compute the integral. Finally, we can use an identity that relates the Kullback–Leibler divergence (KL divergence) (a binary functional of two probabilities) to the right-hand side term:

$$\mathbb{E}_{p(x)} \left[ \ln \left( \frac{q(x)}{p(x)} \right) \right] = -\text{KL} [p(x)||q(x)] \quad (15.8)$$

arriving at our final result:

### 15.1.2 Log-Likelihood Approximation

$$\log [\hat{P}(x_i)] \geq \mathbb{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] - \text{KL} [q_\phi(z|x_i)||P(z)] \quad (15.9)$$

The left term is called the **reconstruction loss** and assess how close we come after going from  $x \rightarrow z \rightarrow x$  in expectation. The right-hand term is the **KL-divergence** and measures how close our encoder is to our defined  $P(z)$  (normal distribution). The right-hand term involves an integral that can be computed analytically and no sampling is required to estimate it. Remember, in the derivation the KL-divergence term appeared as a correction term to account for the fact that our loss doesn't use  $P(z)$  directly, but instead uses the encoder  $q_\phi(z|x_i)$  which generates  $z$ 's from our training data point  $x_i$ . The last step is that we want to minimize our loss, so we need to add a minus sign.

---

**Note:** The log-likelihood equation we've derived for VAE training is also sometimes called the evidence lower bound (ELBO). ELBO is a general equation used in Bayesian modeling, which usually has nothing to do with VAEs.

---

$$\mathcal{L}(x_i, \phi, \theta) = -\mathbb{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \text{KL} [q_\phi(z|x_i)||P(z)] \quad (15.10)$$

Remember that in practice, we will approximate the expectation in the reconstruction loss by sampling  $z$ 's from the decoder  $q_\phi(z|x)$ . We'll only use a single sample.

## 15.2 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

## 15.3 VAE for Discrete Data

The features are classes; we *are not* trying to make a classifier that takes in features and outputs classes. VAEs are for unlabeled data.

Our first example will be to generate new example classes from a distribution of possible classes. An application for this might be to sample conditions of an experiment. Our features  $x$  are one-hot vectors indicating class and our goal is to learn the distribution  $P(x)$  so that we can sample new  $x$ 's. Learning the latent space can also provide a way to embed your features into low dimensional continuous vectors, allowing you to do things like optimization because you've moved from discrete classes to continuous vectors. That is an extra benefit, our loss and training goal are to create a new  $P(x)$ .

Let's think for a moment about our encoder and decoder.  $q_\phi(z|x)$ , the encoder, should give out a *probability distribution* for vectors of real numbers  $z$  and take an input of a one-hot vector  $x$ . This sounds difficult; we've never seen a neural network output a probability distribution over real number vectors. We can simplify though. We defined  $P(z)$  to be normally distributed, let's assume that the form of  $q_\phi(z|x)$  should be normal. Then our neural network could output the parameters to a normal distribution (mean/variance) for  $z$ , rather than trying to output a probability at every possible  $z$  value. It's up to you if you want to have  $q_\phi(z|x)$  output a D-dimensional Gaussian distribution with a covariance matrix or just output D independent normal distributions. Having  $q_\phi(z|x)$  output a normal distribution also allows us to analytically simplify the expectation/integral in the KL-divergence term.

The decoder  $p_\theta(x|z)$  should output a probability distribution over classes given a real vector  $z$ . We can use the same form we used for classification: softmax activation. Just remember that we're not trying to output a specific  $x$ , just a probability distribution of  $x$ 's.

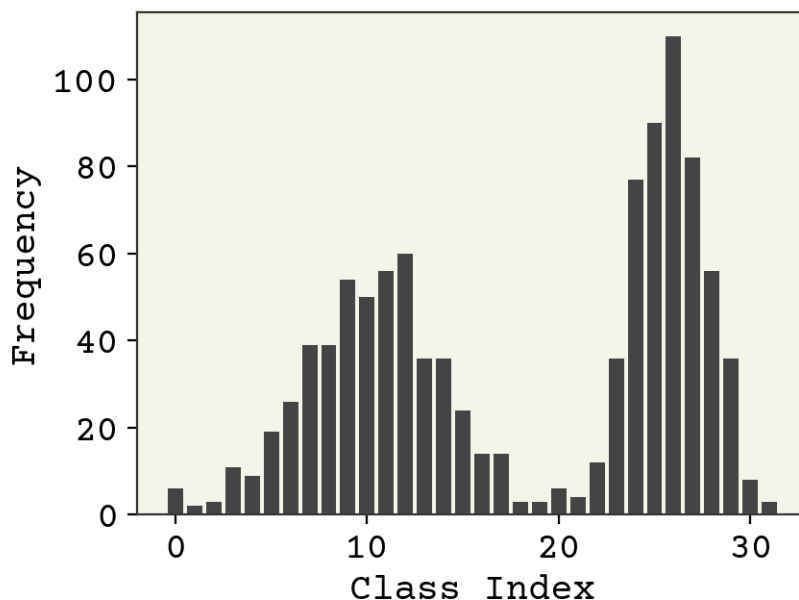
Choices we have to make are the hyperparameters of the encoder and decoder and the size of  $z$ . It makes sense to have the encoder and decoder share as many hyperparameters as possible, since they're somewhat symmetric. Just remember that the encoder in our example is outputting a mean and variance, which means using regression, and the decoder is outputting a normalized probability vector, which means using softmax. Let's get started!

### 15.3.1 The Data

The data is 1024 points  $\vec{x}_i$  where each  $\vec{x}_i$  is a 32 dimensional one-hot vector indicating class. We won't define the classes – the data is synthetic. Since a VAE is unsupervised learning, there are no labels. Let's start by examining the data. We'll sum the occurrences of each class to see what the distribution of classes looks like. *The hidden cells show how the data was generated*

```
import numpy as np
import matplotlib.pyplot as plt
import urllib
import seaborn as sns
import dmol
```

```
plt.bar(np.arange(nclasses), height=np.sum(class_data, axis=0))
plt.xlabel("Class Index")
plt.ylabel("Frequency")
plt.show()
```



### 15.3.2 The encoder

Our encoder will be a basic two hidden layer network. We will output a  $D \times 2$  matrix, where the first column is means and the second is standard deviations for independent normal distributions that make up our guess for  $q(z|x)$ . Outputting a mean is simple, just use no activation. Outputting a standard deviation is unusual because they should be on  $(0, \infty)$ . `jax.nn.softplus` can accomplish this.

```
import jax.numpy as jnp
from jax.example_libraries import optimizers
import jax
import funtools

def random_vec(size):
    return np.random.normal(size=size, scale=1)
```

```
latent_dim = 1
hidden_dim = 16
input_dim = nclasses
```

```
def encoder(x, theta):
    """The encoder takes as input x and gives out probability of z,
    expressed as normal distribution parameters. Assuming each z dim is independent,
    output |z| x 2 matrix"""
    w1, w2, w3, b1, b2, b3 = theta
    hx = jax.nn.relu(w1 @ x + b1)
    hx = jax.nn.relu(w2 @ hx + b2)
    out = w3 @ hx + b3
    # slice out stddeviation and make it positive
    reshaped = out.reshape((-1, 2))
    # we slice with ':' to keep rank same
    std = jax.nn.softplus(reshaped[:, 1:])
```

(continues on next page)

(continued from previous page)

```

mu = reshaped[:, 0:1]
return jnp.concatenate((mu, std), axis=1)

def init_theta(input_dim, hidden_units, latent_dim):
    """Create initial theta parameters"""
    w1 = random_vec(size=(hidden_units, input_dim))
    b1 = np.zeros(hidden_units)
    w2 = random_vec(size=(hidden_units, hidden_units))
    b2 = np.zeros(hidden_units)
    # need to params per dim (mean, std)
    w3 = random_vec(size=(latent_dim * 2, hidden_units))
    b3 = np.zeros(latent_dim * 2)
    return [w1, w2, w3, b1, b2, b3]

# test them
theta = init_theta(input_dim, hidden_dim, latent_dim)
encoder(class_data[0], theta)

```

```

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
↳and rerun for more info.)

```

```

DeviceArray([[0.67200387, 0.00897247]], dtype=float32)

```

The decoder should output a vector of probabilities for  $\vec{x}$ . This can be achieved by just adding a softmax to the output. The rest is nearly identical to the encoder.

```

def decoder(z, phi):
    """decoder takes as input the latent variable z and gives out probability of x.
    Decoder outputs a real number, then we use softmax activation to get probability
    ↳across
    possible values of x.
    """
    w1, w2, w3, b1, b2, b3 = phi
    hz = jax.nn.relu(w1 @ z + b1)
    hz = jax.nn.relu(w2 @ hz + b2)
    out = jax.nn.softmax(w3 @ hz + b3)
    return out

def init_phi(input_dim, hidden_units, latent_dim):
    """Create initial phi parameters"""
    w1 = random_vec(size=(hidden_units, latent_dim))
    b1 = np.zeros(hidden_units)
    w2 = random_vec(size=(hidden_units, hidden_units))
    b2 = np.zeros(hidden_units)
    w3 = random_vec(size=(input_dim, hidden_units))
    b3 = np.zeros(input_dim)
    return [w1, w2, w3, b1, b2, b3]

# test it out
phi = init_phi(input_dim, hidden_dim, latent_dim)
decoder(np.array([1.2] * latent_dim), phi)

```



```
DeviceArray([1.4369683e-11, 2.8801292e-29, 3.2273541e-20, 7.1850895e-22,
             1.0793007e-22, 2.8247908e-20, 8.5903684e-09, 5.0419994e-28,
             3.8993281e-25, 1.9217204e-23, 1.3062071e-12, 2.6221546e-16,
             4.2119552e-23, 1.1967079e-20, 4.3358453e-27, 3.8699083e-20,
             1.3168897e-22, 3.3939088e-20, 5.1175348e-27, 3.9091000e-24,
             1.0000000e+00, 1.6622006e-19, 2.5878642e-29, 3.6575650e-17,
             5.0655268e-25, 3.9531148e-23, 5.9112239e-20, 3.3607102e-19,
             6.1983621e-12, 2.7988031e-19, 9.9489904e-13, 3.6622517e-27],
            dtype=float32)
```

## 15.4 Training

We use ELBO equation for training:

$$l = -\mathbb{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \text{KL} [(q_\phi(z|x)) || P(z)]$$

where  $P(z)$  is the standard normal distribution and we approximate expectations using a single sample from the encoder. We need to expand the KL-divergence term to implement. Both  $P(z)$  and  $q_\theta(z|x)$  are normal. You can look-up the KL-divergence between two normal distributions:

$$\text{KL}(q, p) = \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2} \quad (15.11)$$

we can simplify because  $P(z)$  is standard normal ( $\sigma = 1, \mu = 0$ )

$$\text{KL} [(q_\theta(z|x_i)) || P(z)] = -\log \sigma_i + \frac{\sigma_i^2}{2} + \frac{\mu_i^2}{2} - \frac{1}{2} \quad (15.12)$$

where  $\mu_i, \sigma_i$  are the output from  $q_\phi(z|x_i)$

```
@jax.jit
def loss(x, theta, phi, rng_key):
    """VAE Loss"""
    # reconstruction loss
    sampled_z_params = encoder(x, theta)
    # reparameterization trick
    # we use standard normal sample and multiply by parameters
    # to ensure derivatives correctly propagate to encoder
    sampled_z = (
        jax.random.normal(rng_key, shape=(latent_dim,)) * sampled_z_params[:, 1]
        + sampled_z_params[:, 0]
    )
    # log of prob
    rloss = -jnp.log(decoder(sampled_z, phi) @ x.T + 1e-8)
    # KL loss
    klloss = (
        -0.5
        - jnp.log(sampled_z_params[:, 1])
        + 0.5 * sampled_z_params[:, 0] ** 2
        + 0.5 * sampled_z_params[:, 1] ** 2
    )
    # combined
    return jnp.array([rloss, jnp.mean(klloss)])
```

(continues on next page)

(continued from previous page)

```
# test it out
loss(class_data[0], theta, phi, jax.random.PRNGKey(0))
```

```
DeviceArray([18.420681,  4.439429], dtype=float32)
```

Our loss works! Now we need to make it batched so we can train in batches. Luckily this is easy with `vmap`.

```
batched_loss = jax.vmap(loss, in_axes=(0, None, None, None), out_axes=0)
batched_decoder = jax.vmap(decoder, in_axes=(0, None), out_axes=0)
batched_encoder = jax.vmap(encoder, in_axes=(0, None), out_axes=0)
```

```
# test batched loss
batched_loss(class_data[:4], theta, phi, jax.random.PRNGKey(0))
```

```
DeviceArray([[18.420681 ,  4.439429 ],
             [18.420681 , 32.165703 ],
             [ 0.1743061, 73.976494 ],
             [18.420681 ,  4.439429 ]], dtype=float32)
```

We'll make our gradient take the average over the batch

```
grad = jax.grad(
    lambda x, theta, phi, rng_key: jnp.mean(batched_loss(x, theta, phi, rng_key)),
    (1, 2),
)
fast_grad = jax.jit(grad)
fast_loss = jax.jit(batched_loss)
```

Alright, great! An important detail we've skipped so far is that when using `jax` to generate random numbers, we must step our random number generator forward. You can do that using `jax.random.split`. Otherwise, you'll get the same random numbers at each draw.

We're going to use a `jax` optimizer here. This is to simplify parameter updates. We have a lot of parameters and they are nested, which will be complex for treating with python for loops.

```
batch_size = 32
epochs = 16

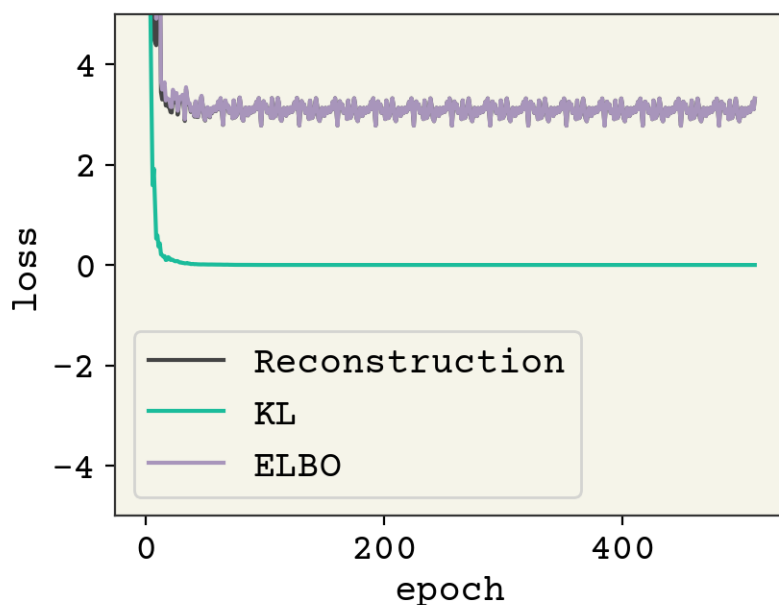
key = jax.random.PRNGKey(0)
opt_init, opt_update, get_params = optimizers.adam(step_size=1e-1)
theta0 = init_theta(input_dim, hidden_dim, latent_dim)
phi0 = init_phi(input_dim, hidden_dim, latent_dim)
opt_state = opt_init((theta0, phi0))
losses = []
for e in range(epochs):
    for bi, i in enumerate(range(0, len(data), batch_size)):
        # make a batch into shape B x 1
        batch = class_data[i : (i + batch_size)]
        # update random number key
        key, subkey = jax.random.split(key)
        # get current parameter values from optimizer
        theta, phi = get_params(opt_state)
        last_state = opt_state
```

(continues on next page)

(continued from previous page)

```
# compute gradient and update
grad = fast_grad(batch, theta, phi, key)
opt_state = opt_update(bi, grad, opt_state)
lvalue = jnp.mean(fast_loss(batch, theta, phi, subkey), axis=0)
losses.append(lvalue)
```

```
plt.plot([l[0] for l in losses], label="Reconstruction")
plt.plot([l[1] for l in losses], label="KL")
plt.plot([l[1] + l[0] for l in losses], label="ELBO")
plt.legend()
plt.ylim(-5, 5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



### 15.4.1 Evaluating the VAE

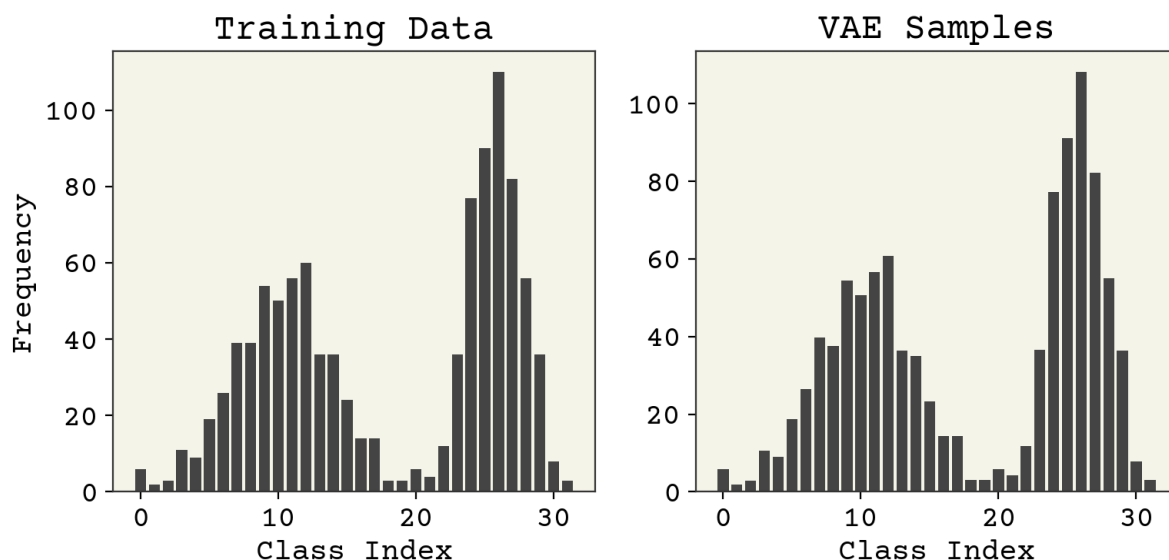
Remember our goal with the VAE is to reproduce  $P(x)$ . We can sample from our VAE using the chosen  $P(z)$  and our decoder. Let's compare that distribution with our training distribution.

```
zs = np.random.normal(size=(1024, 1))
sampled_x = batched_decoder(zs, phi)
fig, axs = plt.subplots(ncols=2, figsize=(8, 4))
axs[0].set_title("Training Data")
axs[0].bar(np.arange(nbins), height=np.sum(class_data, axis=0))
axs[0].set_xlabel("Class Index")
axs[0].set_ylabel("Frequency")
axs[1].set_title("VAE Samples")
axs[1].bar(np.arange(nbins), height=np.sum(sampled_x, axis=0))
axs[1].set_xlabel("Class Index")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



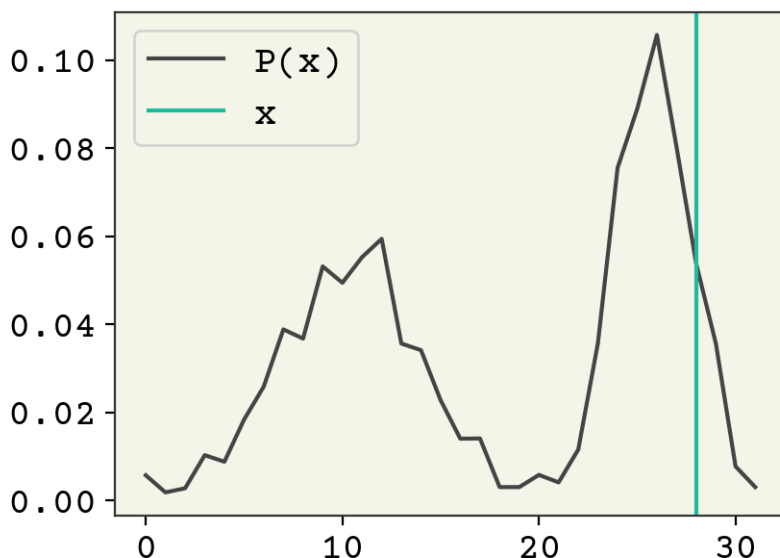
It appears we have succeeded! There were two more goals of the VAE model: making the encoder give output similar to  $P(z)$  and be able to reconstruct. These goals are often opposed and they represent the two terms in the loss: reconstruction and KL-divergence. Let's examine the KL-divergence term, which causes the encoder to give output similar to a standard normal. We'll sample from our training data in histogram look at the resulting average mean and std dev.

```
d = batched_encoder(class_data, theta)
print("Average mu = ", np.mean(d[:, 0]), "Average std dev = ", np.mean(d[:, 1]))
```

```
Average mu = 0.0003880005 Average std dev = 0.9998034
```

Wow! Very close to a standard normal. So our model satisfied the match between the decoder and the  $P(z)$ . The last thing to check is reconstruction. These are distributions, so I'll only look at the maximum  $z$  value to do the reconstruction.

```
plt.plot(decoder(encoder(class_data[2], theta)[0:1, 0], phi), label="P(x)")
plt.axvline(np.argmax(class_data[2]), color="C1", label="x")
plt.legend()
plt.show()
```



The reconstruction is not great, it puts a lot of probability mass on other points. In fact, the reconstruction seems to not use the encoder's information at all – it looks like  $P(x)$ . The reason for this is that our KL-divergence term dominates. It has a very good fit.

## 15.5 Re-balancing VAE Reconstruction and KL-Divergence

Often we desire more reconstruction at the cost of making the latent space less normal. This can be done by adding a term that adjusts the balance between the reconstruction loss and the KL-divergence. You would choose to do this if you want to use the latent space for something and are not just interested in creating a model  $\hat{P}(x)$ . Here is the modified ELBO equation for training:

$$l = -\mathbb{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \beta \cdot \text{KL} [q_\phi(z|x) || P(z)]$$

where  $\beta > 1$  emphasizes the encoder distribution matching chosen latent distribution (standard normal) and  $\beta < 1$  emphasizes reconstruction accuracy.

```
def modified_loss(x, theta, phi, rng_key, beta):
    """This loss allows you to vary which term is more important
    with beta. Beta = 0 - all reconstruction, beta = 1 - ELBO"""
    bl = batched_loss(x, theta, phi, rng_key)
    l = bl @ jnp.array([1.0, beta])
    return jnp.mean(l)
```

```
new_grad = jax.grad(modified_loss, (1, 2))
fast_grad = jax.jit(new_grad)
```

```
# note we used a lower step size for this loss
# and more epochs
opt_init, opt_update, get_params = optimizers.adam(step_size=5e-2)
epochs = 32
theta0 = init_theta(input_dim, hidden_dim, latent_dim)
phi0 = init_phi(input_dim, hidden_dim, latent_dim)
```

(continues on next page)

(continued from previous page)

```

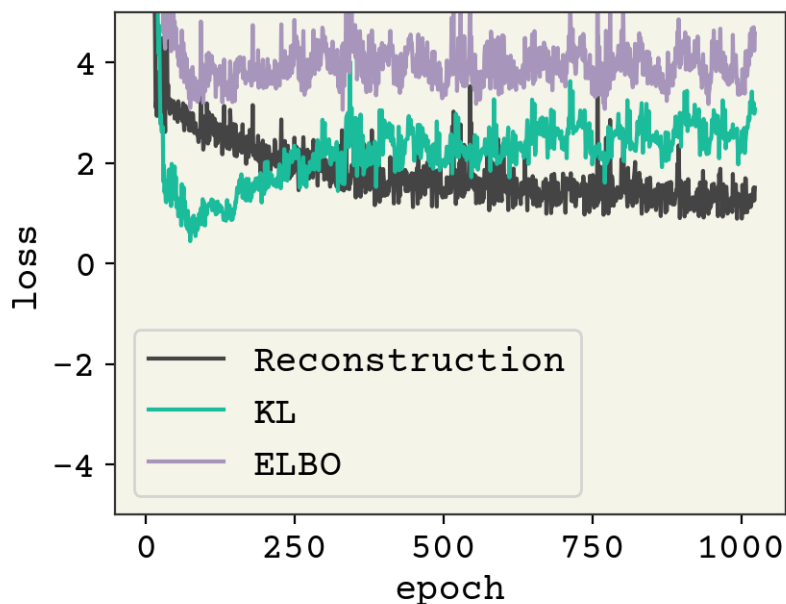
opt_state = opt_init((theta0, phi0))
beta = 0.2
losses = []
for e in range(epochs):
    for bi, i in enumerate(range(0, len(data), batch_size)):
        # make a batch into shape B x 1
        batch = class_data[i : (i + batch_size)]
        # update random number key
        key, subkey = jax.random.split(key)
        # get current parameter values from optimizer
        theta, phi = get_params(opt_state)
        last_state = opt_state
        # compute gradient and update
        grad = fast_grad(batch, theta, phi, key, beta)
        opt_state = opt_update(bi, grad, opt_state)
        lvalue = jnp.mean(fast_loss(batch, theta, phi, subkey), axis=0)
        losses.append(lvalue)

```

```

plt.plot([l[0] for l in losses], label="Reconstruction")
plt.plot([l[1] for l in losses], label="KL")
plt.plot([l[1] + l[0] for l in losses], label="ELBO")
plt.legend()
plt.ylim(-5, 5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

```



You can see the error is higher, but let's see how it did at our three metrics.

```

zs = np.random.normal(size=(1024, 1))
sampled_x = batched_decoder(zs, phi)
fig, axs = plt.subplots(ncols=2, figsize=(8, 4))
axs[0].set_title("Training Data")

```

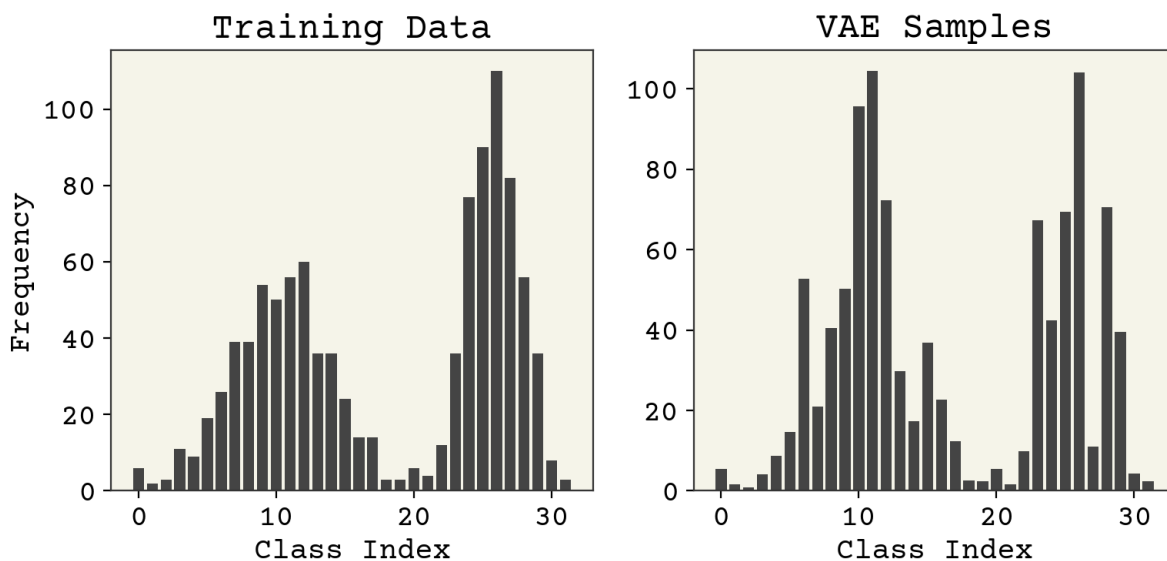
(continues on next page)

(continued from previous page)

```

axs[0].bar(np.arange(nbins), height=np.sum(class_data, axis=0))
axs[0].set_xlabel("Class Index")
axs[0].set_ylabel("Frequency")
axs[1].set_title("VAE Samples")
axs[1].bar(np.arange(nbins), height=np.sum(sampled_x, axis=0))
axs[1].set_xlabel("Class Index")
plt.tight_layout()
plt.show()

```

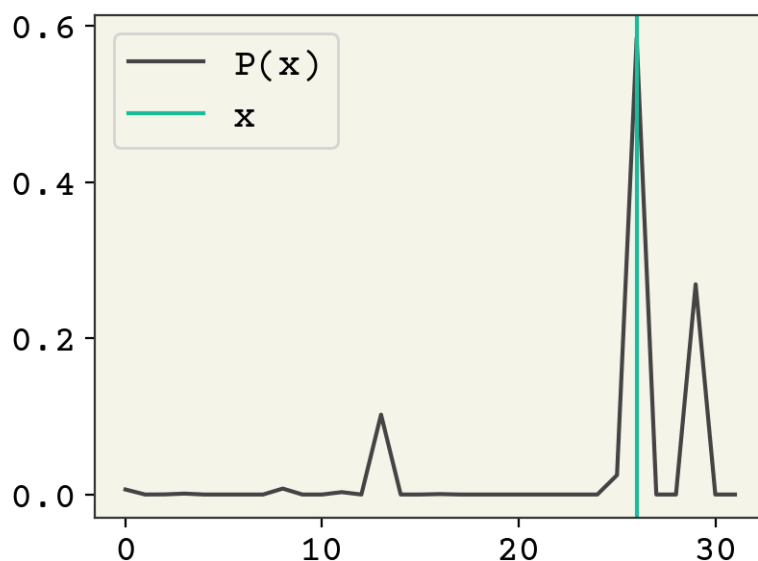


A little bit worse on  $P(x)$ , but overall not bad. What about our goal, the reconstruction?

```

plt.plot(decoder(encoder(class_data[4], theta)[0:1, 0], phi), label="P(x) ")
plt.axvline(np.argmax(class_data[4]), color="C1", label="x")
plt.legend()
plt.show()

```



What about our encoder's agreement with a standard normal?

```
d = batched_encoder(class_data, theta)
print("Average mu = ", np.mean(d[:, 0]), "Average std dev = ", np.mean(d[:, 1]))
```

```
Average mu = 0.13320982 Average std dev = 0.08895968
```

The standard deviation is much smaller! So we squeezed our latent space a little at the cost of better reconstruction.

### 15.5.1 Disentangling $\beta$ -VAE

You can adjust  $\beta$  the opposite direction, to value matching the prior Gaussian distribution more strongly. This can better condition the encoder so that each of the latent dimensions are truly independent. This can be important if you want to disentangle your input features to arrive at an orthogonal projection. This of course comes at the loss of reconstruction accuracy, but can be more important if you're interested in the latent space rather than generating new samples [MRST19].

## 15.6 Regression VAE

We'll now work with continuous features  $x$ . We need to make a few key changes. The encoder will remain the same, but the decoder now must output a  $p_\theta(x|z)$  that gives a probability to all possible  $x$  values. Above, we only had a finite number of classes but now any  $x$  is possible. As we did for the encoder, we'll assume that  $p_\theta(x|z)$  should be normal and we'll output the parameters of the normal distribution from our network. This requires an update to the reconstruction loss to be a log of a normal, but otherwise things will be identical.

One of the mistakes I always make is that the log-likelihood for a normal distribution with a single observation cannot have unknown standard deviation. Our new normal distribution parameters for the decoder will have a single observation for a single  $x$  in training. If you make the standard deviation trainable, it will just pick infinity as the standard deviation since that will for sure capture the point and you only have one point. Thus, I'll make the decoder standard deviation be a hyperparameter fixed at 0.1. We don't see this issue with the encoder, which also outputs a normal distribution, because we training the encoder with the KL-divergence term and not likelihood of observations (reconstruction loss).

```
latent_dim = 1
hidden_dim = 16
input_dim = 1

# make encoder parameters
theta = init_theta(input_dim, hidden_dim, latent_dim)
# test it
encoder(data[0:1], theta)
```

```
DeviceArray([[ -0.48632216,  0.6864413 ]], dtype=float32)
```

```
def decoder(z, phi):
    """decoder takes as input the latent variable z and gives out probability of x.
    Decoder outputs parameters for a normal distribution
    """
    w1, w2, w3, b1, b2, b3 = phi
    hz = jax.nn.relu(w1 @ z + b1)
    hz = jax.nn.relu(w2 @ hz + b2)
    out = w3 @ hz + b3
```

(continues on next page)



(continued from previous page)

```

# slice out stddeviation and make it positive
reshaped = out.reshape((-1, 2))
# we slice with ':' to keep rank same
# std = jax.nn.softplus(reshaped[:,1:])
std = jnp.ones_like(reshaped[:, 1:]) * 0.1
mu = reshaped[:, 0:1]
return jnp.concatenate((mu, std), axis=1)

def init_phi(input_dim, hidden_units, latent_dim):
    """Create initial phi parameters"""
    w1 = random_vec(size=(hidden_units, latent_dim))
    b1 = np.zeros(hidden_units)
    w2 = random_vec(size=(hidden_units, hidden_units))
    b2 = np.zeros(hidden_units)
    w3 = random_vec(size=(input_dim * 2, hidden_units))
    b3 = np.zeros(input_dim * 2)
    return [w1, w2, w3, b1, b2, b3]

# test it out
phi = init_phi(input_dim, hidden_dim, latent_dim)
decoder(np.array([1.2] * latent_dim), phi)

```

```
DeviceArray([[8.568987, 0.1      ]], dtype=float32)
```

```

@jax.jit
def loss(x, theta, phi, rng_key):
    """VAE Loss"""
    # reconstruction loss
    sampled_z_params = encoder(x, theta)
    # reparameterization trick
    # we use standard normal sample and multiply by parameters
    # to ensure derivatives correctly propagate to encoder
    sampled_z = (
        jax.random.normal(rng_key, shape=(latent_dim,)) * sampled_z_params[:, 1]
        + sampled_z_params[:, 0]
    )
    # log of normal dist
    out_params = decoder(sampled_z, phi)
    rloss = (
        -jnp.log(jnp.sqrt(2 * np.pi) * out_params[:, 1] + 1e-10)
        + (x - out_params[:, 0]) ** 2 / out_params[:, 1] ** 2 / 2
    )
    klloss = (
        -0.5
        - jnp.log(sampled_z_params[:, 1])
        + 0.5 * sampled_z_params[:, 0] ** 2
        + 0.5 * sampled_z_params[:, 1] ** 2
    )
    # combined
    return jnp.array([jnp.mean(rloss), jnp.mean(klloss)])

# test it out

```

(continues on next page)

(continued from previous page)

```

loss(data[0:1], theta, phi, jax.random.PRNGKey(0))

# update compiled functions
batched_loss = jax.vmap(loss, in_axes=(0, None, None, None), out_axes=0)
batched_decoder = jax.vmap(decoder, in_axes=(0, None), out_axes=0)
batched_encoder = jax.vmap(encoder, in_axes=(0, None), out_axes=0)
grad = jax.grad(
    lambda x, theta, phi, rng_key: jnp.mean(batched_loss(x, theta, phi, rng_key)),
    (1, 2),
)
fast_grad = jax.jit(grad)
fast_loss = jax.jit(batched_loss)

```

```

batch_size = 32
epochs = 64

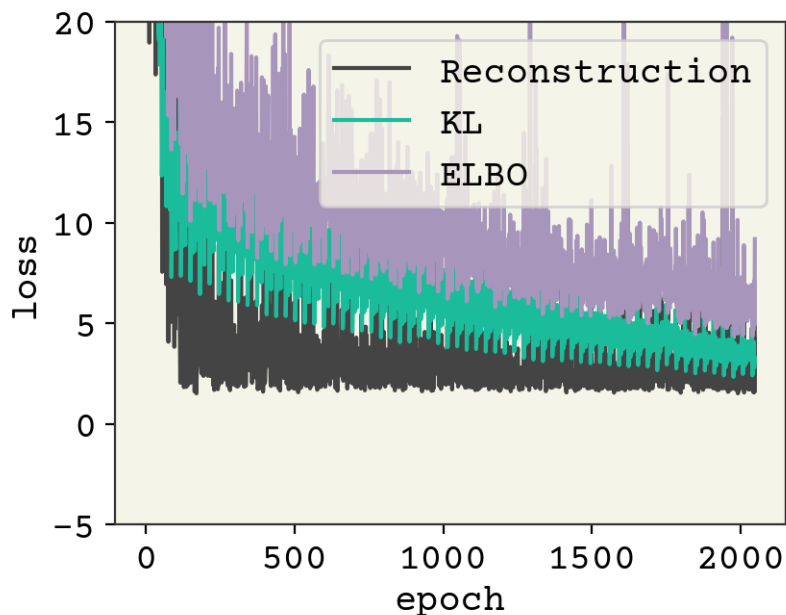
key = jax.random.PRNGKey(0)
opt_init, opt_update, get_params = optimizers.adam(step_size=1e-2)
theta0 = init_theta(input_dim, hidden_dim, latent_dim)
phi0 = init_phi(input_dim, hidden_dim, latent_dim)
opt_state = opt_init((theta0, phi0))
losses = []
for e in range(epochs):
    for bi, i in enumerate(range(0, len(data), batch_size)):
        # make a batch into shape B x 1
        batch = data[i : (i + batch_size)].reshape(-1, 1)
        # update random number key
        key, subkey = jax.random.split(key)
        # get current parameter values from optimizer
        theta, phi = get_params(opt_state)
        last_state = opt_state
        # compute gradient and update
        grad = fast_grad(batch, theta, phi, key)
        opt_state = opt_update(bi, grad, opt_state)
        lvalue = jnp.mean(fast_loss(batch, theta, phi, subkey), axis=0)
        losses.append(lvalue)

```

```

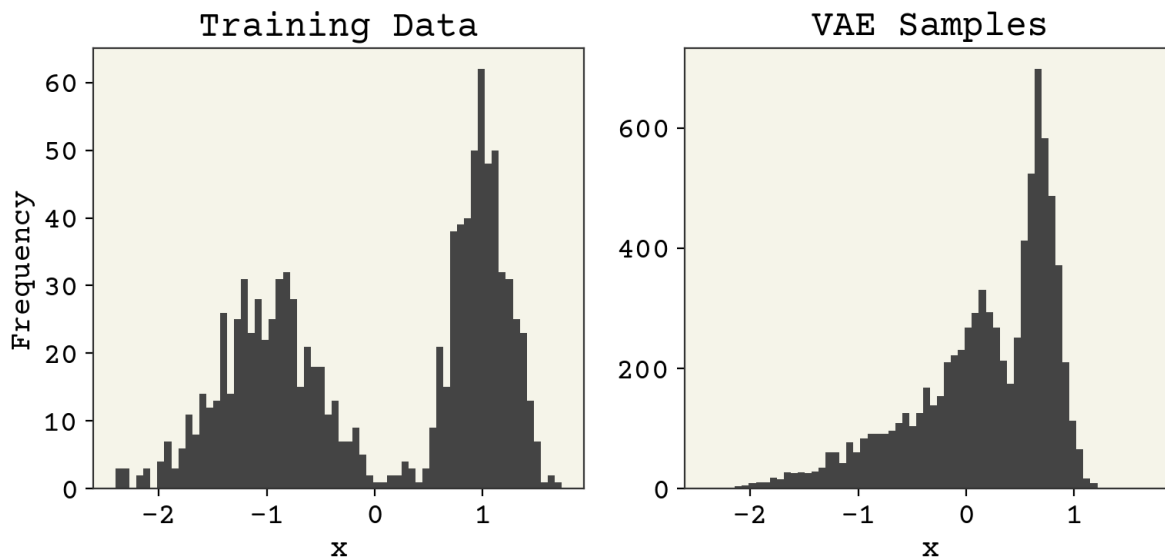
plt.plot([l[0] for l in losses], label="Reconstruction")
plt.plot([l[1] for l in losses], label="KL")
plt.plot([l[1] + l[0] for l in losses], label="ELBO")
plt.legend()
plt.ylim(-5, 20)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

```



This model still has training to be done, but hopefully you get the idea for working with continuous numbers! We can examine the final result below. Note that I must sample from the output parameters to compare with the real training data.

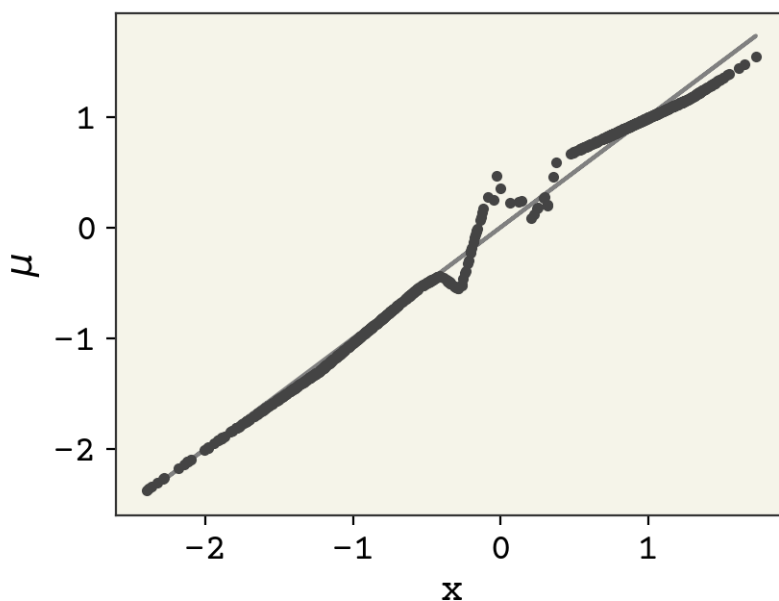
```
bins = 64
zs = np.random.normal(size=(1024, 1))
sampled_x_params = batched_decoder(zs, phi)
fig, axs = plt.subplots(ncols=2, figsize=(8, 4))
axs[0].set_title("Training Data")
_, bins, _ = axs[0].hist(data, bins=bins)
axs[0].set_xlabel("x")
axs[0].set_ylabel("Frequency")
axs[1].set_title("VAE Samples")
# Now we have to sample from output parameters!!
samples = []
for s in sampled_x_params:
    samples.append(np.random.normal(scale=s[:, 1], loc=s[:, 0], size=(8)))
samples = np.array(samples).flatten()
# make them use same bins
axs[1].hist(samples, bins=bins)
axs[1].set_xlabel("x")
plt.tight_layout()
plt.show()
```



The distribution is alright, not great. Comparing reconstruction is a little different because we only compare the mean of the predicted  $P(x)$ . We'll plot our predicted  $\mu$  from the decoder against the real  $x$  values.

```
mus = batched_decoder(batched_encoder(data.reshape(-1, 1), theta)[: , : , 0], phi)[
    : , 0, 0
]

plt.plot(data, mus, ".")
plt.plot(data, data, "-", zorder=-1, color="gray")
plt.xlabel("x")
plt.ylabel(" $\mu$ ")
plt.show()
```



The reconstruction is actually quite good! There is some odd behavior near the top, but otherwise quite reasonable. Finally check how well we did with getting our latent space to be standard normal.

```
d = batched_encoder(data.reshape(-1, 1), theta)
print("Average mu = ", np.mean(d[... , 0]), "Average std dev = ", np.mean(d[... , 1]))
```

```
Average mu = 0.5099522 Average std dev = 0.41120714
```

Surprisingly poor. This gets at one of the issues with VAEs: sometimes your KL will dominate and you have poor reconstruction and other times reconstruction will dominate. It just depends on the variance of your features, dimensions, and hyperparameters. You'll often want to explicitly balance those terms to better agree with your goals for constructing the VAE.

## 15.7 Bead-Spring Polymer VAE

Now we'll move on to a more realistic system. We'll use a bead-spring polymer as shown in the short trajectory snippet below.

This polymer has each bead (atom) joined by a harmonic bond, a harmonic angle between each three, and a Lennard-Jones interaction potential. Knowing these items will not be necessary for the example. We'll construct a VAE that can compress the trajectory to some latent space and generate new conformations.

To begin, we'll use the lessons learned from *Input Data & Equivariances* about how to align points from a trajectory. This will then serve as our training data. The space of our problem will be 12 2D vectors. Our system need not be permutation invariant, so we can flatten these vectors into a 24 dimensional input. The code belows loads and aligns the trajectory

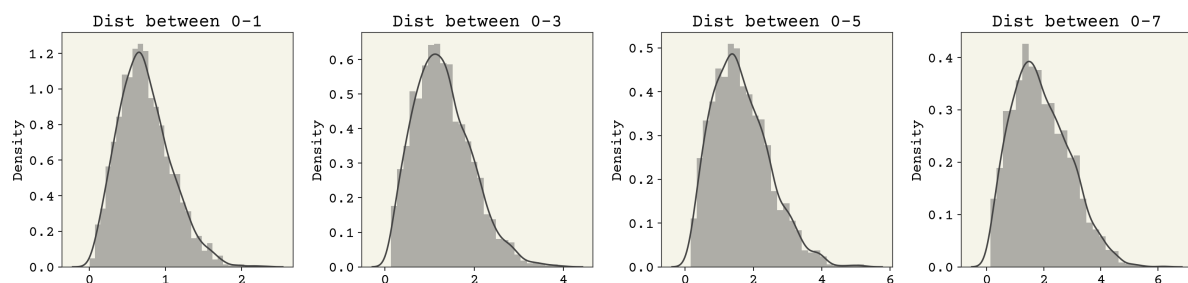
```
urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/master/data/long_paths.npz",
    "long_paths.npz",
)
paths = np.load("long_paths.npz")["arr"]
# transform to be rot/trans invariant
data = align_principle(center_com(paths))
cmap = plt.get_cmap("cool")
for i in range(0, data.shape[0], 16):
    plt.plot(data[i, :, 0], data[i, :, 1], "-", alpha=0.1, color="C2")
plt.title("All Frames")
plt.xticks([])
plt.yticks([])
plt.show()
```

## All Frames



Before training, let's examine some of the **marginals** of the data. Marginals mean we've transformed (by integration) our probability distribution to be a function of only 1-2 variables so that we can plot nicely. We'll look at the pairwise distance between points.

```
fig, axs = plt.subplots(ncols=4, squeeze=True, figsize=(16, 4))
for i, j in enumerate(range(1, 9, 2)):
    axs[i].set_title(f"Dist between 0-{j}")
    sns.distplot(np.linalg.norm(data[:, 0] - data[:, j], axis=1), ax=axs[i])
plt.tight_layout()
```



These look a little like the chi distribution with two degrees of freedom. Notice that the support (x-axis) changes between them though. We'll keep an eye on these when we evaluate the efficacy of our VAE.

### 15.7.1 VAE Model

We'll build the VAE like above. I will make two changes. I will use JAX's random number generator and I will make the number of layers variable. The code is hidden below, but you can expand to see the details. We'll be starting with 4 layers total (3 hidden) with a hidden layer dimension of 256. Another detail is that we flatten the input/output since the order is preserved and thus we do not worry about separating the x,y dimension out.

### 15.7.2 Loss

The loss function is similar to above, but I will not even bother with the Gaussian outputs. You can see the only change is that we drop the output Gaussian standard deviation from the loss, which remember was not trainable anyway.

```
@jax.jit
def loss(x, theta, phi, rng_key):
    """VAE Loss"""
    # reconstruction loss
    sampled_z_params = encoder(x, phi)
    # reparameterization trick
    # we use standard normal sample and multiply by parameters
    # to ensure derivatives correctly propagate to encoder
    sampled_z = (
        jax.random.normal(rng_key, shape=(latent_dim,)) * sampled_z_params[:, 1]
        + sampled_z_params[:, 0]
    )
    # MSE now instead
    xp = decoder(sampled_z, theta)
    rloss = jnp.sum((xp - x) ** 2)
    # LK loss
    klloss = (
        -0.5
        - jnp.log(sampled_z_params[:, 1] + 1e-8)
        + 0.5 * sampled_z_params[:, 0] ** 2
        + 0.5 * sampled_z_params[:, 1] ** 2
    )
    # combined
    return jnp.array([rloss, jnp.mean(klloss)])

# update compiled functions
batched_loss = jax.vmap(loss, in_axes=(0, None, None, None), out_axes=0)
batched_decoder = jax.vmap(decoder, in_axes=(0, None), out_axes=0)
batched_encoder = jax.vmap(encoder, in_axes=(0, None), out_axes=0)
grad = jax.grad(modified_loss, (1, 2))
fast_grad = jax.jit(grad)
fast_loss = jax.jit(batched_loss)
```

### 15.7.3 Training

Finally comes the training. The only changes to this code are to flatten our input data and shuffle to prevent the each batch from having similar conformations.

```
batch_size = 32
epochs = 250
key = jax.random.PRNGKey(0)

flat_data = data.reshape(-1, input_dim)
# scramble it
flat_data = jax.random.shuffle(key, flat_data)

opt_init, opt_update, get_params = optimizers.adam(step_size=1e-2)
theta0, key = init_theta(input_dim, hidden_units, latent_dim, num_layers, key)
```

(continues on next page)

(continued from previous page)

```

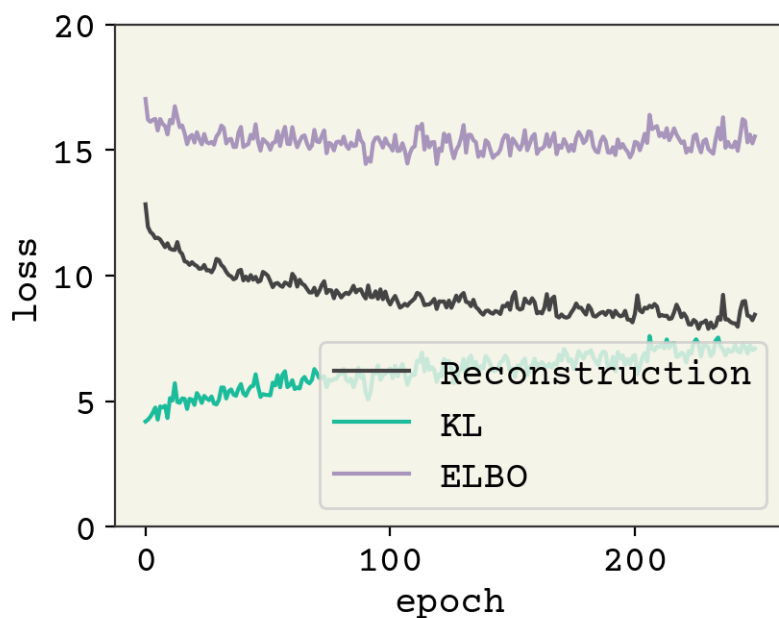
phi0, key = init_phi(input_dim, hidden_units, latent_dim, num_layers, key)
opt_state = opt_init((theta0, phi0))
losses = []
# KL/Reconstruction balance
beta = 0.01
for e in range(epochs):
    for bi, i in enumerate(range(0, len(flat_data), batch_size)):
        # make a batch into shape B x 1
        batch = flat_data[i : (i + batch_size)].reshape(-1, input_dim)
        # update random number key
        key, subkey = jax.random.split(key)
        # get current parameter values from optimizer
        theta, phi = get_params(opt_state)
        last_state = opt_state
        # compute gradient and update
        grad = fast_grad(batch, theta, phi, key, beta)
        opt_state = opt_update(bi, grad, opt_state)
    # use large batch for tracking progress
    lvalue = jnp.mean(fast_loss(flat_data[100], theta, phi, subkey), axis=0)
    losses.append(lvalue)

```

```

plt.plot([l[0] for l in losses], label="Reconstruction")
plt.plot([l[1] for l in losses], label="KL")
plt.plot([l[1] + l[0] for l in losses], label="ELBO")
plt.legend()
plt.ylim(0, 20)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

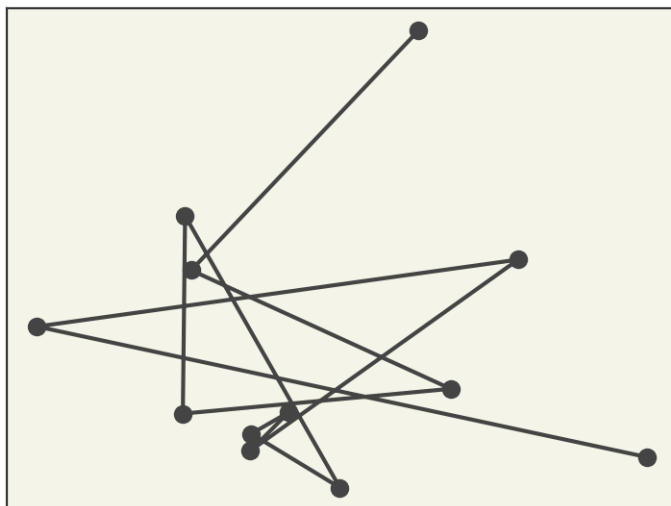
```



As usual, this model is undertrained. A latent space of 2, which we chose for plotting convenience, is also probably a little too compressed. Let's sample a few conformation and see how they look.



```
sampld_data = decoder(jax.random.normal(key, shape=[latent_dim]), theta).reshape(-1, 2)
plt.plot(sampld_data[:, 0], sampld_data[:, 1], "-o", alpha=1)
plt.xticks([])
plt.yticks([])
plt.show()
```



These look reasonable compared with the trajectory video showing the training conformations.

## 15.8 Using VAE on a Trajectory

There are three main things to do with a VAE on a trajectory. The first is to go from a trajectory in the feature dimension to the latent dimension. This can simplify analysis of dynamics or act as a reaction coordinate for free energy methods. The second is to generate new conformations. This could be used to fill-in under sampling or perhaps extrapolate to new regions of latent space. You can also use the VAE to examine marginals that are perhaps under-sampled. Finally, you can do optimization on the latent space. For example, you could try to find the most compact structure. We'll examine these examples but there are many other things you could examine. For a more complete model example with attention and 3D coordinates, see Winter et al. [WNoeC21]. You can find applications of VAEs on trajectories for molecular design [SMS+20], coarse-graining [WGomezB19], and identifying rare-events [RBWT18].

### 15.8.1 Latent Trajectory

Let's start by computing a latent trajectory. I'm going to load a shorter trajectory which has the frames closer together in time.

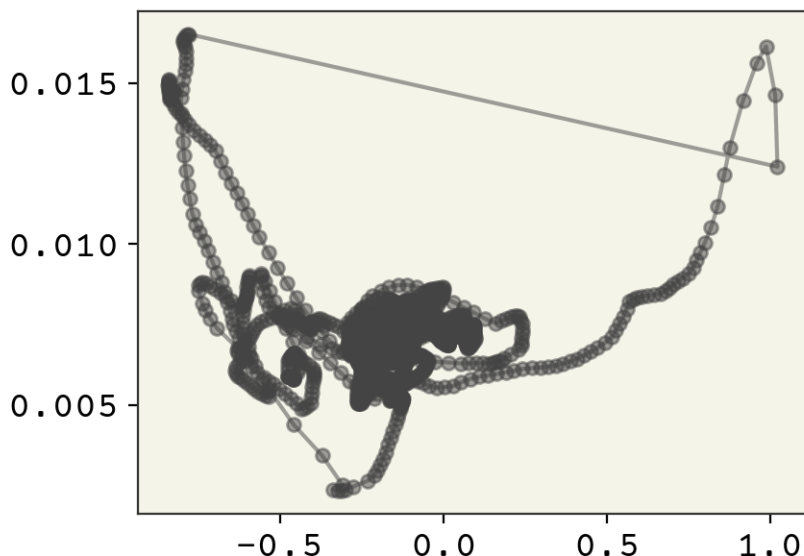
```
urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/master/data/paths.npz", "paths.npz"
)
paths = np.load("paths.npz")["arr"]
short_data = align_principle(center_com(paths))

# get latent params
# throw away standard deviation
```

(continues on next page)

(continued from previous page)

```
latent_traj = batched_encoder(short_data.reshape(-1, input_dim), phi)[: , 0]
plt.plot(latent_traj[:, 0], latent_traj[:, 1], "-o", markersize=5, alpha=0.5)
plt.show()
```



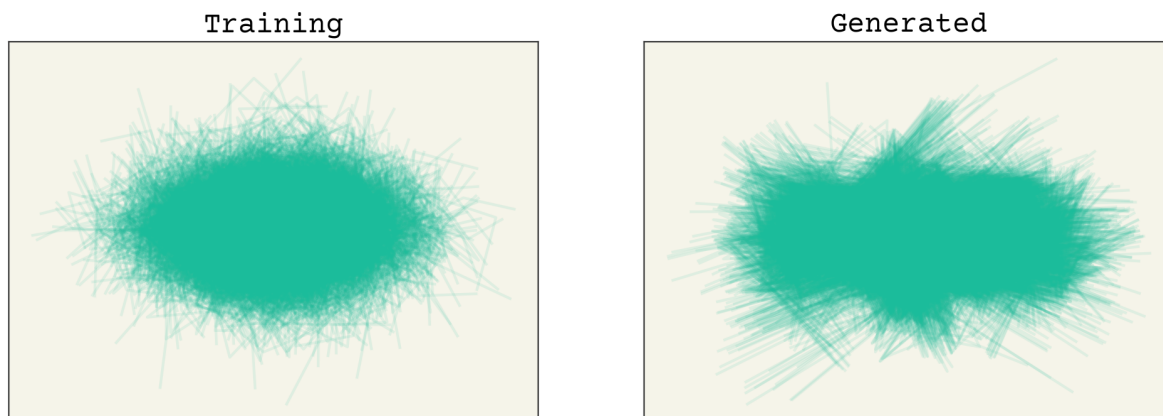
You can see that the trajectory is relatively continuous, except for a few wide jumps. We'll see below that this is because the alignment process can have big jumps as our principle axis rapidly moves when the points rearrange. Let's compare the video and the z-path side-by-side. You can find the code for this movie on the github repo.

You can see the quick change is due to our alignment quickly changing. This is why aligning on the principle axis isn't always perfect: your axis can flip 90 degrees because the internal points change the moment of inertia enough to change.

## 15.8.2 Generate New Samples

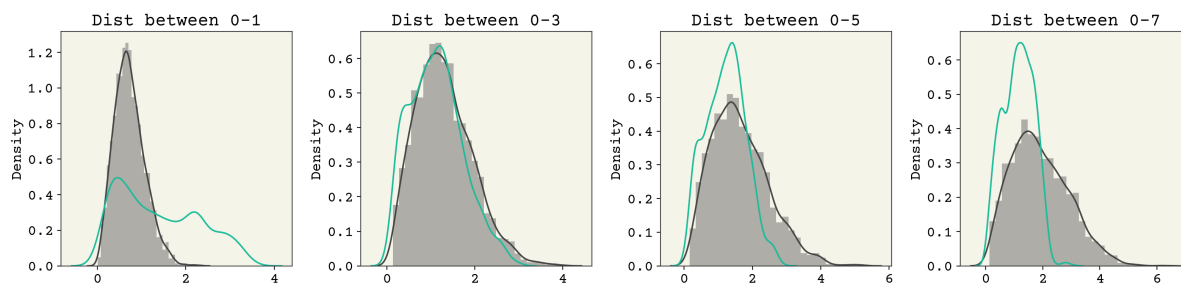
Let's see how our samples look.

```
fig, axs = plt.subplots(ncols=2, figsize=(12, 4))
sampled_data = batched_decoder(
    np.random.normal(size=(data.shape[0], latent_dim)), theta
).reshape(data.shape[0], -1, 2)
for i in range(0, data.shape[0]):
    axs[0].plot(data[i, :, 0], data[i, :, 1], "-", alpha=0.1, color="C1")
    axs[1].plot(
        sampled_data[i, :, 0], sampled_data[i, :, 1], "-", alpha=0.1, color="C1"
    )
axs[0].set_title("Training")
axs[1].set_title("Generated")
for i in range(2):
    axs[i].set_xticks([])
    axs[i].set_yticks([])
plt.show()
```



The samples are not perfect, but we're close. Let's examine the marginals.

```
fig, axs = plt.subplots(ncols=4, squeeze=True, figsize=(16, 4))
for i, j in enumerate(range(1, 9, 2)):
    axs[i].set_title(f"Dist between 0-{j}")
    sns.distplot(np.linalg.norm(data[:, 0] - data[:, j], axis=1), ax=axs[i])
    sns.distplot(
        np.linalg.norm(sampled_data[:, 0] - sampled_data[:, j], axis=1),
        ax=axs[i],
        hist=False,
    )
plt.tight_layout()
```



You can see that there are some issues here as well. Remember that our latent space is quite small: 2D. So we should not be that surprised that we're losing information from our 24D input space.

### 15.8.3 Optimization on Latent Space

Finally, let us examine how we can optimize in the latent space. Let's say I want to find the most compact structure. We'll define our loss function as the radius of gyration and take its derivative with respect to  $z$ . Recall the definition of radius of gyration is

$$R_g = \frac{1}{N} \sum_i r_i^2 \quad (15.13)$$

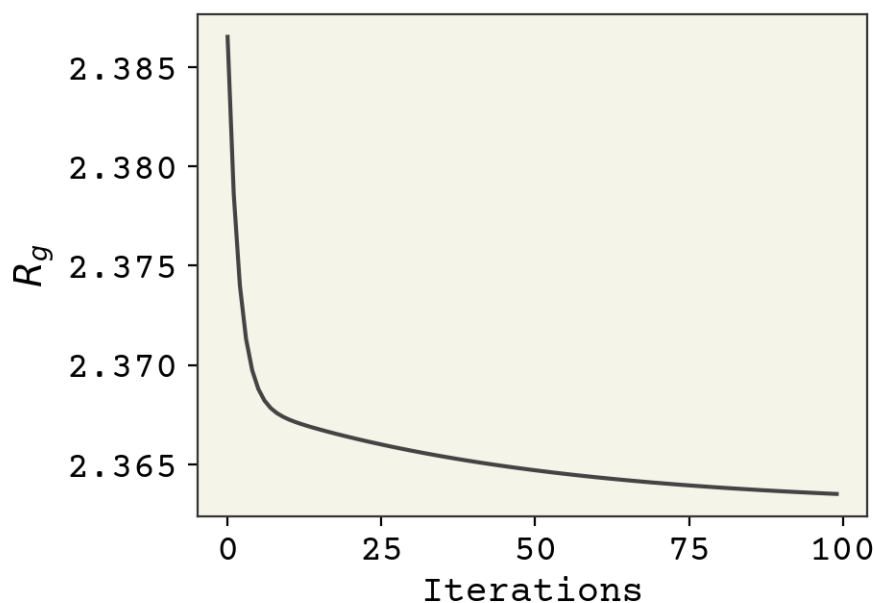
where  $r_i$  is distance to center of mass. Our generated samples are, by definition, centered at the origin though so we do not need to worry about center of mass. We want to take derivatives in  $z$ , but need samples in  $x$  to compute radius of gyration. We use the decoder to get an  $x$  and can propagate derivatives through it, because it is a differentiable neural network.

```
def rg_loss(z):
    x = decoder(z, theta).reshape(-1, 2)
    rg = jnp.sum(x**2)
    return jnp.sqrt(rg)
```

```
rg_grad = jax.jit(jax.grad(rg_loss))
```

Now we will find the  $z$  that minimizes the radius of gyration by using gradient descent with the derivative.

```
z = jax.random.normal(key, shape=[latent_dim])
losses = []
eta = 1e-2
for i in range(100):
    losses.append(rg_loss(z))
    g = rg_grad(z)
    z -= eta * g
plt.plot(losses)
plt.xlabel("Iterations")
plt.ylabel("$R_g$")
plt.show()
```



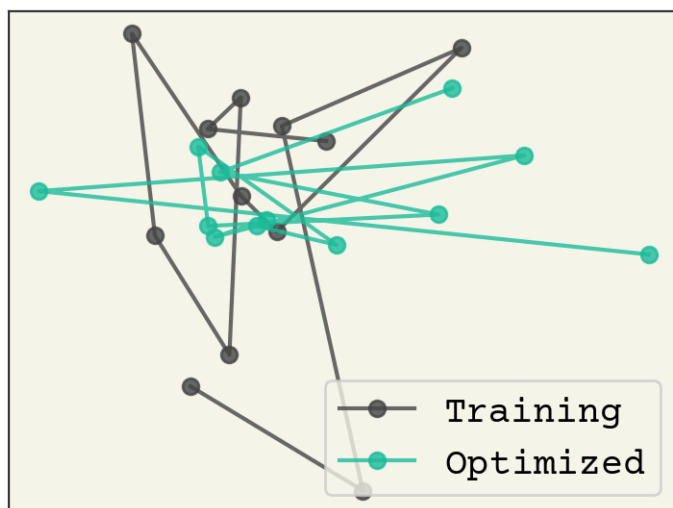
We have a  $z$  with a very low radius of gyration. How good is it? Well, we can also see what was the lowest radius of gyration *observed* structure in our trajectory. We compare them below.

```
# get min from training
train_rgmin = np.argmin(np.sum(data**2, axis=(1, 2)))
# use new z
opt_rgmin = decoder(z, theta).reshape(-1, 2)
plt.plot(
    data[train_rgmin, :, 0], data[train_rgmin, :, 1], "o-", label="Training", alpha=0.8,
    ↵8
)
plt.plot(opt_rgmin[:, 0], opt_rgmin[:, 1], "o-", label="Optimized", alpha=0.8)
```

(continues on next page)

(continued from previous page)

```
plt.xticks([])
plt.yticks([])
plt.legend()
plt.show()
```



What is remarkable about this is that the optimized one has no overlaps and still reasonable bond-lengths. It is also more compact than the lowest radius of gyration found in the training example.

## 15.9 Relevant Videos

### 15.9.1 Using VAE for Coarse-Grained Molecular Simulation

### 15.9.2 Using VAE for Molecular Graph Generation

### 15.9.3 Review of Molecular Graph Generative Models (including VAE)

## 15.10 Chapter Summary

- A variational autoencoder is a generative deep learning model capable of unsupervised learning. It is capable of generating new data points not seen in training.
- A VAE is a set of two trained conditional probability distributions that operate on examples from the data  $x$  and the latent space  $z$ . The encoder goes from data to latent and the decoder goes from latent to data.
- The loss function is the log likelihood that we observed the training point  $x_i$ .
- Taking the log allows us to sum/average over data to aggregate multiple points.
- The VAE can be used for both discrete or continuous features.
- The goal with VAE is to reproduce the probability distribution of  $x$ . Comparing the distribution over  $z$  and that of  $x$  allows us to evaluate how well the VAE operates.
- A bead-spring polymer VAE example shows how VAEs operate on a trajectory.

## 15.11 Cited References

## NORMALIZING FLOWS

The VAE was our first example of a generative model that is capable of sampling from  $P(x)$ . The VAE has two disadvantages though. The first that you cannot numerically evaluate  $P(x)$ , the probability of a single point. The second disadvantage is that training the VAE is difficult, especially because you're assuming the latent space should be normal. **Generative adversarial networks** (GANs) are similar to VAEs and have the same two disadvantages.

A **normalizing flow** is similar to a VAE in that we try to build up  $P(x)$  by starting from a simple known distribution  $P(z)$ . We use functions, like the decoder from a VAE, to go from  $x$  to  $z$ . However, we make sure that the functions we choose keep the probability mass normalized ( $\sum P(x) = 1$ ) and can be used forward (to sample from  $x$ ) and backward (to compute  $P(x)$ ). We call these functions **bijectors** because they are bijective (1 to 1, onto). An example of a bijector is an element-wise cosine  $y_i = \cos x_i$  (assuming  $x_i$  is between 0 and  $\pi$ ) and non-bijective function would be a reduction  $y = \sum_i x_i$ . Any function which changes the number of elements is automatically not bijective. A consequence of using only bijectors in constructing our normalizing flow is that the size of the latent space must be equal to the size of the feature space. Remember the VAE used a smaller latent space than the feature space.

---

### Audience & Objectives

This chapter builds on *Variational Autoencoder* and assumes the same background of probability theory. This chapter is an introduction to the key ideas, but is not fully developed yet. Some knowledge of vector calculus (Jacobians) is assumed as well. After completing it, you should be able to

- Understand the trade-offs between a VAE, GAN, and normalizing flow.
- Identify a bijector and construct a bijector chain
- Construct a normalizing flow using common bijectors types and train it
- Sample from a normalizing flow and compute sample probabilities

---

You can find a recent review of normalizing flows [here](#) [KPB20] and [here](#) [PNR+19]. Although generating images and sound is the most popular application of normalizing flows, some of their biggest scientific impact has been on more efficient sampling from posteriors or likelihoods and other complex probability distributions [PSM19]. You find details on how to do normalizing flows on categorical (discrete) data in Hooeboom et al. [HNJ+21].

## 16.1 Flow Equation

Recall for the VAE decoder, we had an explicit formula for  $p(x|z)$ . This allowed us to compute  $p(x) = \int dz p(x|z)p(z)$  which is the quantity of interest. The VAE decoder is a conditional probability density function. In the normalizing flow, we do not use probability density functions. We use bijective functions. So we cannot just compute an integral to change variables. We can use the change of variable formula. Consider our normalizing flow to be defined by our bijector  $x = f(z)$ , its inverse  $z = g(x)$ , and the starting probability distribution  $P_z(z)$ . Then the formula for probability of  $x$  is

$$P(x) = P_z(g(x)) |\det [\mathbf{J}_g]| \quad (16.1)$$

where the term on the right is the absolute value of the determinant of the Jacobian of  $g$ . **Jacobians** are matrices that describe how infinitesimal changes in each domain dimension change each range dimension. This term corrects for the volume change of the distribution. For example, if  $f(z) = 2z$ , then  $g(x) = x/2$ , and the Jacobian determinant is  $1/2$ . The intuition is that we are stretching out  $z$  by 2, so we need to account for the increase in volume to keep the probability normalized. You can read more about the change of variable formula for [probability distributions here](#)

## 16.2 Bijectors

A bijector is a function that is **injective** (1 to 1) and **surjective** (onto). An equivalent way to view a bijective function is if it has an inverse. For example, a sum reduction has no inverse and is thus not bijective.  $\sum[1, 0] = 1$  and  $\sum[-1, 2] = 1$ . Multiplying by a matrix which has an inverse is bijective.  $y = x^2$  is not bijective, since  $y = 4$  has two solutions.

Remember that we must compute the determinant of the bijector Jacobian. If the Jacobian is dense (all output elements depend on all input elements), computing this quantity will be  $O(|x|_0^3)$  where  $|x|_0$  is the number of dimensions of  $x$  because a determinant scales by  $O(n^3)$ . This would make computing normalizing flows impractical in high-dimensions. However, in practice we restrict ourselves to bijectors that have easy to calculate Jacobians. For example, if the bijector is  $x_i = \cos z_i$  then the Jacobian will be diagonal. Typically, the trick that is done is to make the Jacobian triangular. Then  $x_0$  only depends on  $z_0$ ,  $z_1$  depends on  $z_0, z_1$ , and  $x_2$  depends on  $z_0, z_1, z_2$ , etc. The matrix determinant is then computed in linear time with respect to the number of dimensions.

### 16.2.1 Bijector Chains

Just like in deep neural networks, multiple bijectors are chained together to increase how complex of the final fit distribution  $\hat{P}(x)$  can be. The change of variable equation can be repeatedly applied:

$$P(x) = P_z[g_1(g_0(x))] |\det [\mathbf{J}_{g_1}]| |\det [\mathbf{J}_{g_0}]| \quad (16.2)$$

where we would compute  $x$  with  $f_0(f_1(z))$ . One critical point is that you should also include a **permute bijector** that swaps the order of dimensions. Since the bijectors typically have triangular Jacobians, certain output dimensions will depend on many input dimensions and others will only depend on a single one. By applying a permutation, you allow each dimension to influence each other.

## 16.3 Training

At this point, you may be wondering how you could possibly train a normalizing flow. The trainable parameters appear in the bijectors. They have adjustable parameters. The loss equation is quite simple: the negative log-likelihood (negative to make it minimization). Explicitly:

$$l = -\log P_z[g_1(g_0(x))] - \sum_i \log |\det [\mathbf{J}_{g_i}]| \quad (16.3)$$

where  $x$  is the training point and when you take the gradient of the loss, it is with respect to the parameters of the bijectors.



## 16.4 Common Bijectors

The choice of bijector functions is a fast changing area. I will thus only mention a few. You can of course use any bijective function or matrix, but these become inefficient at high-dimension due to the Jacobian calculation. One class of efficient bijectors are autoregressive bijectors. These have triangular Jacobians because each output dimension can only depend on the dimensions with a lower index. There are two variants: masked autoregressive flows (MAF)[PPM17] and inverse autoregressive flows (IAF) [KSJ+16]. MAFs are efficient at training and computing probabilities, but are slow for sampling from  $P(x)$ . IAFs are slow at training and computing probabilities but efficient for sampling. Wavenets combine the advantages of both [KLS+18]. I'll mention one other common bijector which is not autoregressive: real non-volume preserving (RealNVPs) [DSDB16]. RealNVPs are less expressive than IAFs/MAFs, meaning they have trouble replicating complex distributions, but are efficient at all three tasks: training, sampling, and computing probabilities. Another interesting variant is the Glow bijector, which is able to expand the rank of the normalizing flow, for example going from a matrix to an RGB image [DAS19]. What are the equations for all these bijectors? Most are variants of standard neural network layers but with special rules about which outputs depend on which inputs.

**Warning:** Remember to add permute bijectors between autoregressive bijectors to ensure the dependence between dimensions is well-mixed.

## 16.5 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

The hidden code below imports the tensorflow probability package and other necessary packages. Note that the tensorflow probability package (`tfp`) is further broken into distributions (`tfd`) and bijectors (`tfb`).

```
import tensorflow as tf
import tensorflow_probability as tfp
import matplotlib.pyplot as plt
import sklearn.datasets as datasets
import numpy as np
import dmol
```

```
np.random.seed(0)
tf.random.set_seed(0)

tfd = tfp.distributions
tfb = tfp.bijectors
```

## 16.6 Moon Example

We'll start with a basic 2D example to learn the two moons distribution with a normalizing flow. Two moons is a common example dataset that is hard to cluster and model as a probability distribution.

When doing normalizing flows you have two options to implement them. You can do all the Jacobians, inverses, and likelihood calculations analytically and implement them in a normal ML framework like Jax, PyTorch, or TensorFlow. This is actually most common. The second option is to utilize a probability library that knows how to use bijectors and distributions. The packages for that are PYMC, TensorFlow Probability (which has a non-tensorflow JAX version confusingly), and Pyro (Pytorch). We'll use TensorFlow Probability for this work.

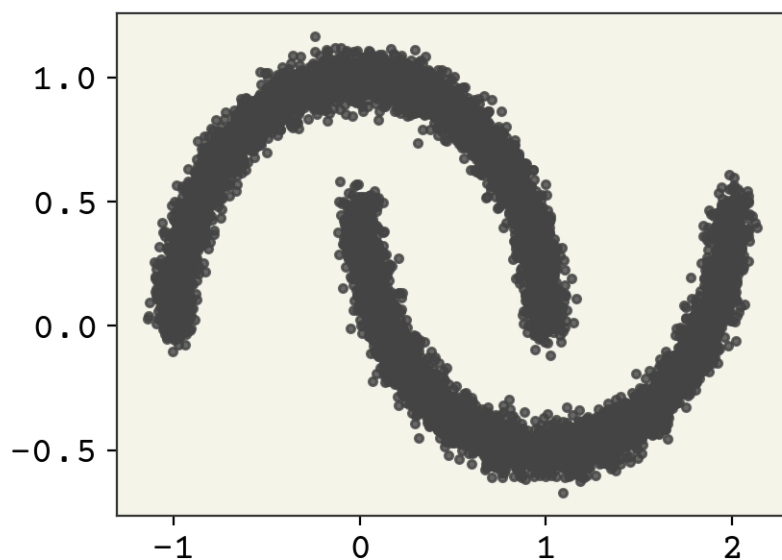
### 16.6.1 Generating Data

In the code below, I set-up my imports and sample points which will be used for training. Remember, this code has nothing to do with normalizing flows – it's just to generate data.

```
moon_n = 10000
ndim = 2
data, _ = datasets.make_moons(moon_n, noise=0.05)
```

```
plt.plot(data[:, 0], data[:, 1], ".", alpha=0.8)
```

```
[<matplotlib.lines.Line2D at 0x7fc91db35880>]
```



## 16.6.2 Z Distribution

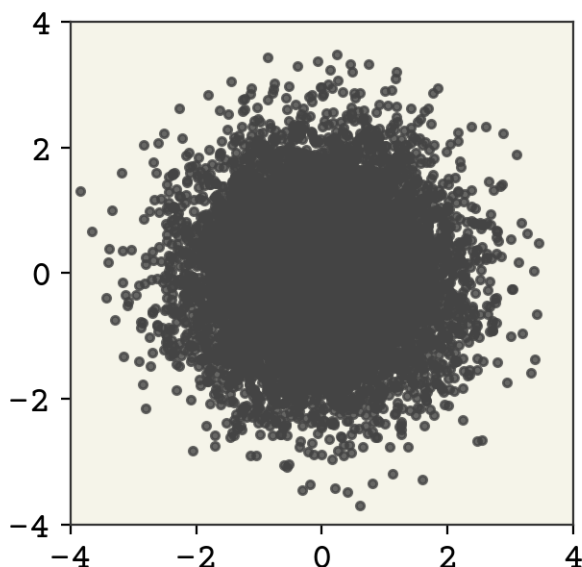
Our Z distribution should always be as simple as possible. I'll create a 2D Gaussian with unit variance, no covariance, and centered at 0. I'll be using the tensorflow probability package for this example. The key new concept is that we organize our tensors that were *sampled* from a probability distribution in a specific way. We, by convention, make the first axes be the **sample** shape, the second axes be the **batch** shape, and the final axes be the **event** shape. The sample shape is the number of times we sampled from our distribution. It is a *shape* and not a single dimension because occasionally you'll want to organize your samples into some shape. The batch shape is a result of possibly multiple distributions batched together. For example, you might have 2 Gaussians, instead of a single 2D Gaussian. Finally, the event shape is the shape of a single sample from the distribution. This is overly complicated for our example, but you should be able to read information about the distribution now by understanding this nomenclature. You can find a tutorial on these [shapes here](#) and more tutorials on [tensorflow probability here](#).

```
zdist = tfd.MultivariateNormalDiag(loc=[0.0] * ndim)
zdist
```

```
<tfd.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[]
event_shape=[2] dtype=float32>
```

With our new understanding of shapes, you can see that this distribution has no `batch_shape` because there is only one set of parameters and the `event_shape` is [2] because it's a 2D Gaussian. Let's now sample from this distribution and view it

```
zsamples = zdist.sample(moon_n)
plt.plot(zsamples[:, 0], zsamples[:, 1], ".", alpha=0.8)
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.gca().set_aspect("equal")
```



As expected, our starting distribution looks nothing like are target distribution. Let's demonstrate a bijector now. We'll implement the following bijector:

$$x = \bar{z} \times (1, 0.5)^T + (0.5, 0.25)$$

This is bijective because the operations are element-wise and invertible. Rather than just write this out using operations

like @ or \*, we'll use the built-in bijectors from TensorFlow probability. The reason we do this is that they have their inverses and Jacobian determinants already defined. We first create a bijector that scales by  $[1, 0.5]$  and one then one that shifts by  $[0.5, 0.25]$ .

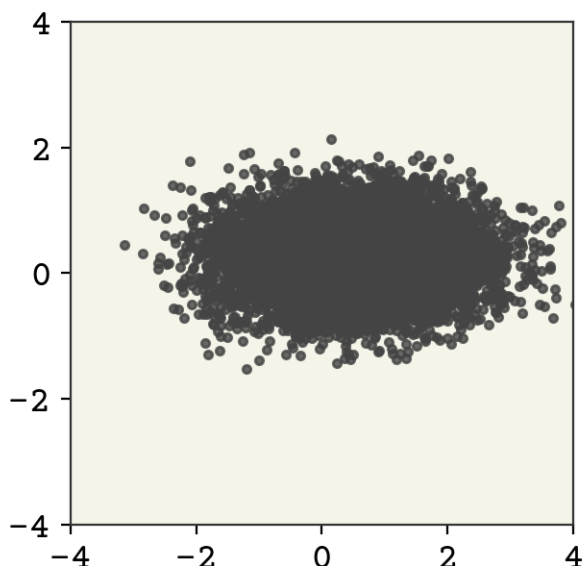
```
shift_bij = tfb.Shift([0.5, 0.25])
scale_bij = tfb.Scale([1, 0.5])
# make composite via function convolution
b = shift_bij(scale_bij)
```

To now apply the change of variable formula, we create a **transformed distribution**. What is important about this choice is that we can compute likelihoods, probabilities, and sample from it.

```
td = tfd.TransformedDistribution(zdist, bijector=b)
td
```

```
<tfp.distributions.TransformedDistribution 'chain_of_shift_of_
scaleMultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype=float32>
```

```
zsamples = td.sample(moon_n)
plt.plot(zsamples[:, 0], zsamples[:, 1], ".", alpha=0.8)
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.gca().set_aspect("equal")
```



We show above the sampling from this new distribution. We can also plot its probability, which is impossible for a VAE-like model!

```
# make points for grid
zpoints = np.linspace(-4, 4, 150)
(
    z1,
    z2,
) = np.meshgrid(zpoints, zpoints)
zgrid = np.concatenate((z1.reshape(-1, 1), z2.reshape(-1, 1)), axis=1)
```

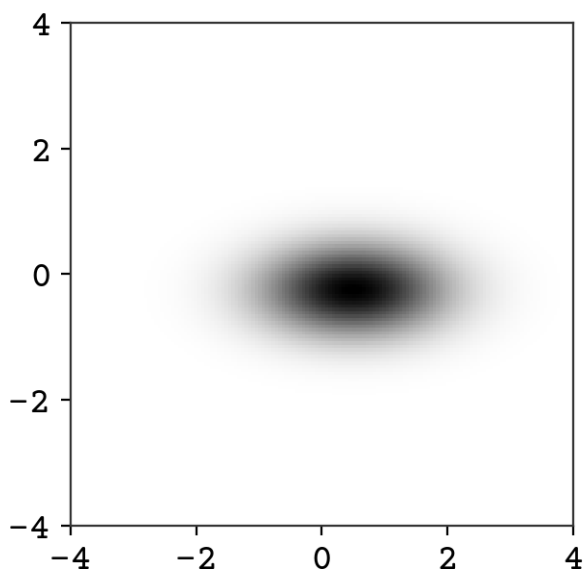
(continues on next page)

(continued from previous page)

```

# compute P(x)
p = np.exp(td.log_prob(zgrid))
fig = plt.figure()
# plot and set axes limits
plt.imshow(p.reshape(z1.shape), aspect="equal", extent=[-4, 4, -4, 4])
plt.show()

```



### 16.6.3 The Normalizing Flow

Now we will build bijectors that are expressive enough to capture the moon distribution. I will use 3 sets of a MAF and permutation for 6 total bijectors. MAF's have dense neural network layers in them, so I will also set the usual parameters for a neural network: dimension of hidden layer and activation.

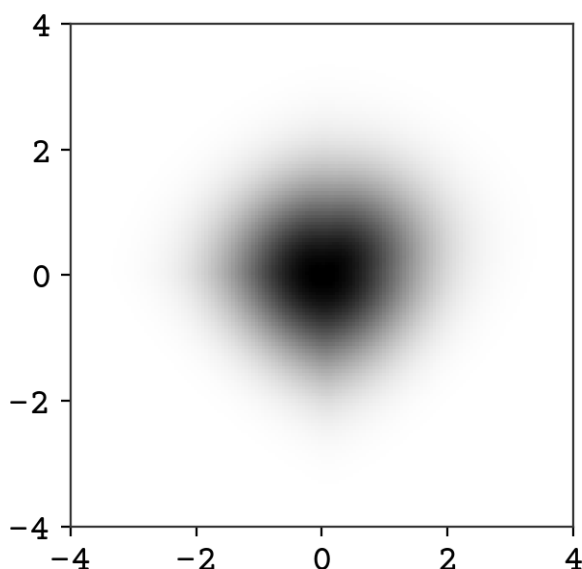
```

num_layers = 3
my_bijectors = []
# loop over desired bijectors and put into list
for i in range(num_layers):
    # Syntax to make a MAF
    anet = tfb.MaskedAutoregressiveNetwork(
        params=ndim, hidden_units=[128, 128], activation="relu"
    )
    ab = tfb.MaskedAutoregressiveFlow(anet)
    # Add bijector to list
    my_bijectors.append(ab)
    # Now permuate (!important!)
    permute = tfb.Permute([1, 0])
    my_bijectors.append(permute)
# put all bijectors into one "chain bijector"
# that looks like one
big_bijector = tfb.Chain(my_bijectors)
# make transformed dist
td = tfd.TransformedDistribution(zdist, bijector=big_bijector)

```

At this point, we have not actually trained but we can still view our distribution.

```
zpoints = np.linspace(-4, 4, 150)
(
    z1,
    z2,
) = np.meshgrid(zpoints, zpoints)
zgrid = np.concatenate((z1.reshape(-1, 1), z2.reshape(-1, 1)), axis=1)
p = np.exp(td.log_prob(zgrid))
fig = plt.figure()
plt.imshow(p.reshape(z1.shape), aspect="equal", extent=[-4, 4, -4, 4])
plt.show()
```



You can already see that the distribution looks more complex than a Gaussian.

### 16.6.4 Training

To train, we'll use TensorFlow Keras, which just handles computing derivatives and the optimizer.

```
# declare the feature dimension
x = tf.keras.Input(shape=(2,), dtype=tf.float32)
# create a "placeholder" function
# that will be model output
log_prob = td.log_prob(x)
# use input (feature) and output (log prob)
# to make model
model = tf.keras.Model(x, log_prob)
# define a loss
def neg_loglik(yhat, log_prob):
    # losses always take in label, prediction
    # in keras. We do not have labels,
    # but we still need to accept the arg
    # to comply with Keras format
    return -log_prob
```

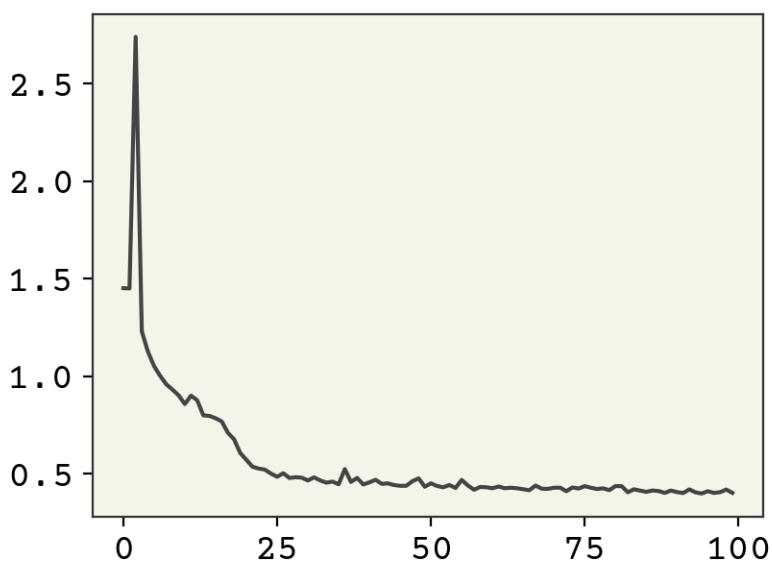
(continues on next page)

(continued from previous page)

```
# now we prepare model for training
model.compile(optimizer=tf.optimizers.Adam(1e-3), loss=neg_loglik)
```

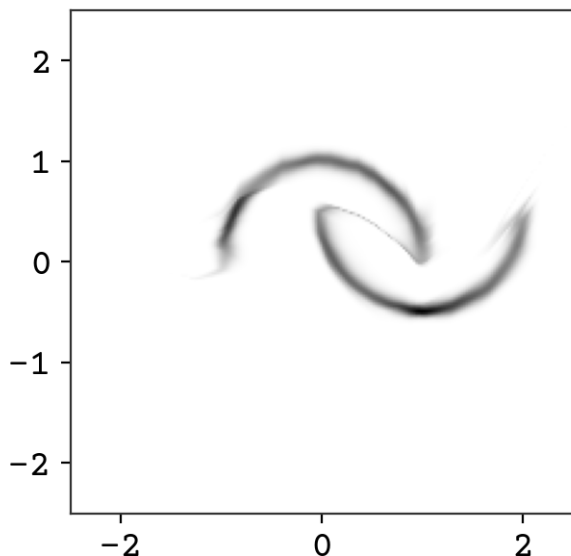
One detail is that we have to create fake labels (zeros) because Keras expects there to always be training labels. Thus our loss we defined above (negative log-likelihood) takes in the labels but does nothing with them.

```
result = model.fit(x=data, y=np.zeros(moon_n), epochs=100, verbose=0)
plt.plot(result.history["loss"])
plt.show()
```



Training looks reasonable. Let's now see our distribution.

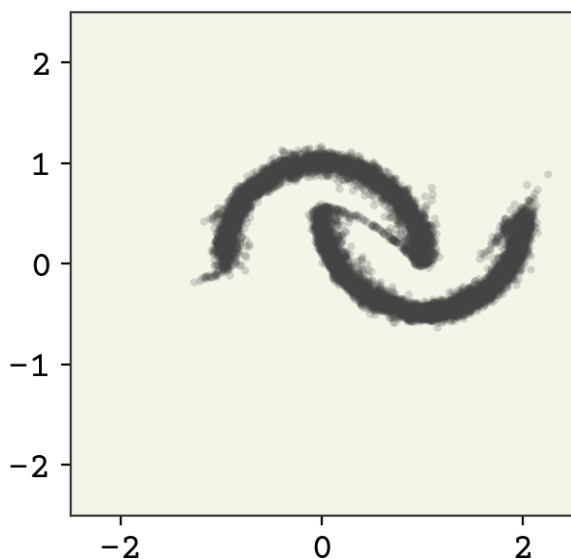
```
zpoints = np.linspace(-2.5, 2.5, 200)
(
    z1,
    z2,
) = np.meshgrid(zpoints, zpoints)
zgrid = np.concatenate((z1.reshape(-1, 1), z2.reshape(-1, 1)), axis=1)
p = np.exp(td.log_prob(zgrid))
fig = plt.figure()
plt.imshow(
    p.reshape(z1.shape), aspect="equal", origin="lower", extent=[-2.5, 2.5, -2.5, 2.5]
)
plt.show()
```



Wow! We now can compute the probability of any point in this distribution. You can see there are some oddities that could be fixed with further training. One issue that cannot be overcome is the connection between the two curves – it is not possible to get fully disconnected densities. This is because of our requirement that the bijectors are invertible and volume preserving – you can only squeeze volume so far but cannot completely disconnect. Some work has been done on addressing this issue by adding sampling to the flow and this gives more expressive normalizing flows [WKohlerNoe20].

Finally, we'll sample from our model just to show that indeed it is generative.

```
zsamples = td.sample(moon_n)
plt.plot(zsamples[:, 0], zsamples[:, 1], ".", alpha=0.2, markeredgewidth=0.0)
plt.xlim(-2.5, 2.5)
plt.ylim(-2.5, 2.5)
plt.gca().set_aspect("equal")
```





## 16.7 Relevant Videos

### 16.7.1 Normalizing Flow for Molecular Conformation

## 16.8 Chapter Summary

- A normalizing flow builds up a probability distribution of  $x$  by starting from a known distribution on  $z$ . Bijective functions are used to go from  $z$  to  $x$ .
- Bijectors are functions that keep the probability mass normalized and are used to go forward and backward (because they have well-defined inverses).
- To find the probability distribution of  $x$  we use the change of variable formula, which requires a function inverse and Jacobian.
- The bijector function has trainable parameters, which can be trained using a negative log-likelihood function.
- Multiple bijectors can be chained together, but typically must include a permute bijector to swap the order of dimensions.

## 16.9 Cited References



## **Part IV**

### **D. Applications**



## PREDICTING DFT ENERGIES WITH GNNS

QM9 is a dataset of 134,000 molecules consisting of 9 heavy atoms drawn from the elements C, H, O, N, F[RDRVL14]. The features are the xyz coordinates ( $\mathbf{X}$ ) and elements ( $\bar{e}$ ) of the molecule. The coordinates are determined from B3LYP/6-31G(2df,p) level DFT geometry optimization. There are multiple labels (see table below), but we'll be interested specifically in the energy of formation (Enthalpy at 298.15 K). The goal in this chapter is to regress a graph neural network to predict the energy of formation given the coordinates of a molecule. We will build upon ideas in the following chapters:

1. *Regression & Model Assessment*
2. *Graph Neural Networks*
3. *Input Data & Equivariances*

QM9 is one of the most popular dataset for machine learning and deep learning since it came out in 2014. The first papers could achieve about 10 kcal/mol on this regression problem and now are down to ~1 kcal/mol and lower. Any model on this dataset must be translation, rotation, and permutation invariant.

### 17.1 Label Description

Index	Name	Units	Description
0	index	-	Consecutive, 1-based integer identifier of molecule
1	A	GHz	Rotational constant A
2	B	GHz	Rotational constant B
3	C	GHz	Rotational constant C
4	mu	Debye	Dipole moment
5	alpha	Bohr <sup>3</sup>	Isotropic polarizability
6	homo	Hartree	Energy of Highest occupied molecular orbital (HOMO)
7	lumo	Hartree	Energy of Lowest unoccupied molecular orbital (LUMO)
8	gap	Hartree	Gap, difference between LUMO and HOMO
9	r2	Bohr <sup>2</sup>	Electronic spatial extent
10	zpve	Hartree	Zero point vibrational energy
11	U0	Hartree	Internal energy at 0 K
12	U	Hartree	Internal energy at 298.15 K
13	H	Hartree	Enthalpy at 298.15 K
14	G	Hartree	Free energy at 298.15 K
15	Cv	cal/(mol K)	Heat capacity at 298.15 K

## 17.2 Data

I have written some helper code in the `fetch_data.py` file. It downloads the data and converts into a format easily used in Python. The data returned from this function is broken into the features  $\mathbf{X}$  and  $\vec{e}$ .  $\mathbf{X}$  is an  $N \times 4$  matrix of atom positions + partial charge of the atom.  $\vec{e}$  is vector of atomic numbers for each atom in the molecule. Remember to slice the specific label you want from the label vector.

## 17.3 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

```
import tensorflow as tf
import numpy as np
from fetch_data import qm9_parse, qm9_fetch
import dmol
```

Let's load the data. This step will take a few minutes as it is downloaded and processed.

```
qm9_records = qm9_fetch()
data = qm9_parse(qm9_records)
```

`data` is an iterable containing the data for the 133k molecules. Let's examine the first one.

```
for d in data:
    print(d)
    break
```

```
((<tf.Tensor: shape=(5,), dtype=int64, numpy=array([6, 1, 1, 1, 1])>, <tf.Tensor: shape=(5, 4), dtype=float32, numpy=
array([[ -1.2698136e-02,  1.0858041e+00,  8.0009960e-03, -5.3568900e-01],
       [ 2.1504159e-03, -6.0313176e-03,  1.9761203e-03,  1.3392100e-01],
       [ 1.0117308e+00,  1.4637512e+00,  2.7657481e-04,  1.3392200e-01],
       [-5.4081506e-01,  1.4475266e+00, -8.7664372e-01,  1.3392299e-01],
       [-5.2381361e-01,  1.4379326e+00,  9.0639728e-01,  1.3392299e-01]]),
  dtype=float32)>, <tf.Tensor: shape=(16,), dtype=float32, numpy=
array([ 1.0000000e+00,  1.5771181e+02,  1.5770998e+02,  1.5770699e+02,
        0.0000000e+00,  1.3210000e+01, -3.8769999e-01,  1.1710000e-01,
        5.0480002e-01,  3.5364101e+01,  4.4748999e-02, -4.0478931e+01,
       -4.0476063e+01, -4.0475117e+01, -4.0498596e+01,  6.4689999e+00],
  dtype=float32)>)
```

These are Tensorflow Tensors. They can be converted to numpy arrays via `x.numpy()`. The first item is the element vector `6, 1, 1, 1, 1`. Do you recognize the elements? It's C, H, H, H, H. The positions come next. Note that there is an extra column containing the atom partial charges, which we will not use as a feature. Finally, the last tensor is the label vector.

Now we will do some processing of the data to get into a more usable format. Let's convert to numpy arrays, remove the partial charges, and convert the elements into one-hot vectors.

```
def convert_record(d):
    # break up record
    (e, x), y = d
    #
    e = e.numpy()
    x = x.numpy()
    r = x[:, :3]
    # make ohc size larger
    # so use same node feature
    # shape later
    ohc = np.zeros((len(e), 16))
    ohc[np.arange(len(e)), e - 1] = 1
    return (ohc, r), y.numpy()[13]

for d in data:
    (e, x), y = convert_record(d)
    print("Element one hots\n", e)
    print("Coordinates\n", x)
    print("Label:", y)
    break
```

```
Element one hots
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Coordinates
[[-1.2698136e-02  1.0858041e+00  8.0009960e-03]
 [ 2.1504159e-03 -6.0313176e-03  1.9761203e-03]
 [ 1.0117308e+00  1.4637512e+00  2.7657481e-04]
 [-5.4081506e-01  1.4475266e+00 -8.7664372e-01]
 [-5.2381361e-01  1.4379326e+00  9.0639728e-01]]
Label: -40.475117
```

## 17.4 Baseline Model

Before we get too far into modeling, let's see what a simple model can do for accuracy. This will help establish a baseline model which any more sophisticated implementation should exceed in accuracy. You can make many choices for this, but I'll just make a linear regression based on number of atom types.

```
import jax.numpy as jnp
import jax.experimental.optimizers as optimizers
import jax
import warnings
import matplotlib.pyplot as plt

warnings.filterwarnings("ignore")
```

(continues on next page)

(continued from previous page)

```

@jax.jit
def baseline_model(nodes, w, b):
    # get sum of each element type
    atom_count = jnp.sum(nodes, axis=0)
    yhat = atom_count @ w + b
    return yhat

def baseline_loss(nodes, y, w, b):
    return (baseline_model(nodes, w, b) - y) ** 2

baseline_loss_grad = jax.grad(baseline_loss, (2, 3))
w = np.ones(16)
b = 0.0

```

We've set-up our simple regression model. One complexity is that we cannot batch the molecules like normal because each molecule contains different shaped tensors.

```

# we'll just train on 5,000 and use 1,000 for test
# shuffle
shuffled_data = data.shuffle(7000)
test_set = shuffled_data.take(1000)
valid_set = shuffled_data.skip(1000).take(1000)
train_set = shuffled_data.skip(2000).take(5000)

```

The labels in this data are quite large, so we're going to make a transform on them to make our learning rates and training going more smoothly.

```

ys = [convert_record(d)[1] for d in train_set]
train_ym = np.mean(ys)
train_ys = np.std(ys)
print("Mean = ", train_ym, "Std = ", train_ys)

```

```
Mean = -360.42856 Std = 44.142815
```

Now we'll just use this transform when training:  $y_s = \frac{y - \mu_y}{\sigma_y}$  and then our predictions will be transformed by  $\hat{y} = \hat{f}(e, x) \cdot \sigma_y + \mu_y$ . This just helps standardize our range of outputs.

```

def transform_label(y):
    return (y - train_ym) / train_ys

def transform_prediction(y):
    return y * train_ys + train_ym

```

```

epochs = 16
eta = 1e-3
baseline_val_loss = [0.0 for _ in range(epochs)]
for epoch in range(epochs):
    for d in train_set:
        (e, x), y_raw = convert_record(d)
        y = transform_label(y_raw)

```

(continues on next page)



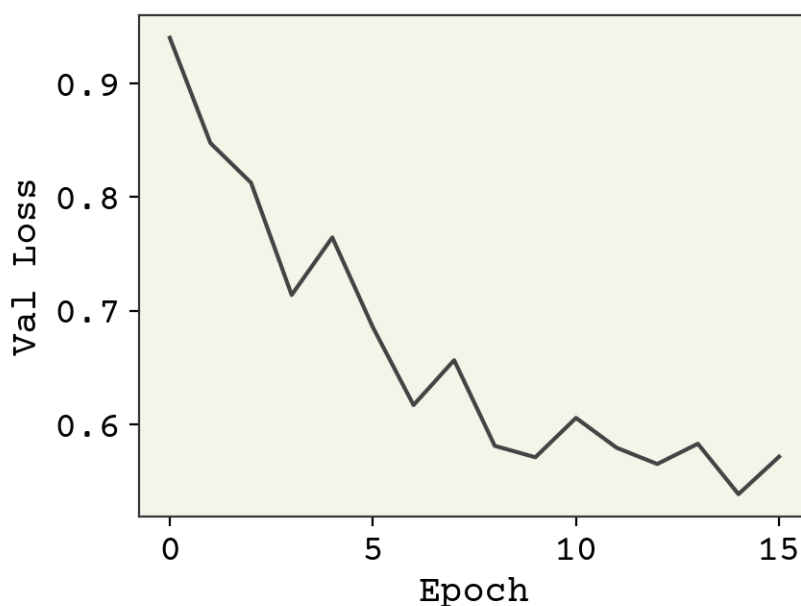
(continued from previous page)

```

grad_est = baseline_loss_grad(e, y, w, b)
# update regression weights
w -= eta * grad_est[0]
b -= eta * grad_est[1]
# compute validation loss
for v in valid_set:
    (e, x), y_raw = convert_record(v)
    y = transform_label(y_raw)
    # convert SE to RMSE
    baseline_val_loss[epoch] += baseline_loss(e, y, w, b)
baseline_val_loss[epoch] = jnp.sqrt(baseline_val_loss[epoch] / 1000)
eta *= 0.9
plt.plot(baseline_val_loss)
plt.xlabel("Epoch")
plt.ylabel("Val Loss")
plt.show()

```

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF\_CPP\_MIN\_LOG\_LEVEL=0 and rerun for more info.)



This is poor performance, but it gives us a baseline value of what we can expect. One unusual detail I did in this training was to slowly reduce the learning rate. This is because our features and labels are all in different magnitudes. Our weights need to move far to get into the right order of magnitude and then need to fine-tune a little. Thus, we start at high learning rate and decrease.

```

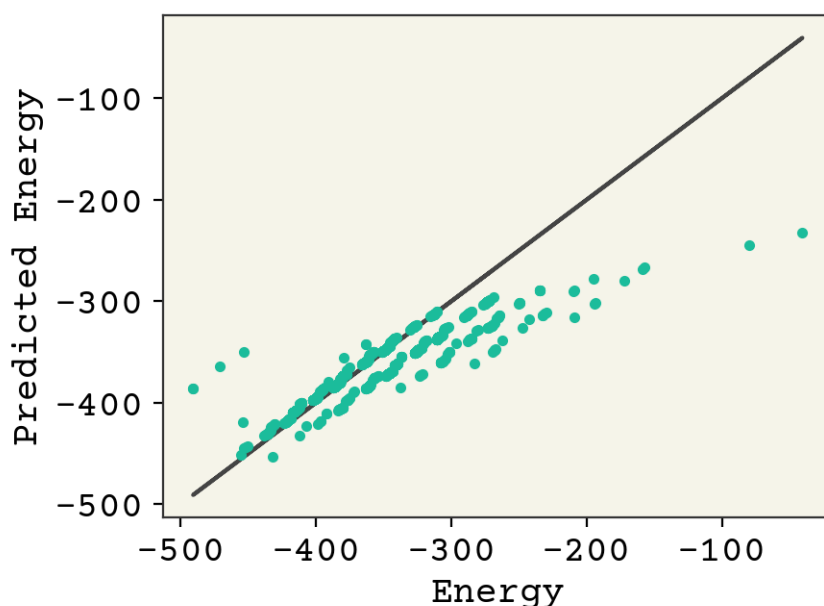
ys = []
yhats = []
for v in valid_set:
    (e, x), y = convert_record(v)
    ys.append(y)
    yhat_raw = baseline_model(e, w, b)
    yhat = transform_prediction(yhat_raw)
    yhats.append(yhat)

```

(continues on next page)

(continued from previous page)

```
plt.plot(ys, ys, "-")
plt.plot(ys, yhats, ".")
plt.xlabel("Energy")
plt.ylabel("Predicted Energy")
plt.show()
```



You can see that the broad trends about molecule size capture a lot of variance, but more work needs to be done.

## 17.5 Example GNN Model

We now can work with this data to build a model. Let's build a simple model that can model energy and obeys the invariances required of the problem. We will use a graph neural network (GNN) because it obeys permutation invariance. We will create a *graph* from the coordinates/element vector by joining all atoms to all other atoms and using their inverse pairwise distance as the edge weight. The choice of pairwise distance gives us translation and rotation invariance. The choice of inverse distance means that atoms which are far away naturally have low edge weights.

I will now define our model using the Battaglia equations [BHB+18]. As opposed to most examples we've seen in class, I will use the graph level feature vector  $\vec{u}$  which will ultimately be our estimate of energy. The edge update will only consider the sender and the edge weight with trainable parameters:

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) = \sigma(\vec{v}_{sk} \vec{w}_e e_k + \vec{b}_e) \quad (17.1)$$

where the input edge  $e_k$  will be a single number (inverse pairwise distance) and  $\vec{b}_e$  is a trainable bias vector. We will use a sum aggregation for edges (not shown).  $\sigma$  is a leaky ReLU. The leaky just prevents vanishing gradients, which I found empirically to reduce performance here. The node update will be

$$\vec{v}'_i = \phi^v(\vec{e}'_i, \vec{v}_i, \vec{u}) = \sigma(\mathbf{W}_v \vec{e}'_i) + \vec{v}_i \quad (17.2)$$

The global node aggregation will also be a sum. Finally, we have our graph feature vector update:

$$\vec{u}' = \phi^u(\vec{e}', \vec{v}', \vec{u}) = \sigma(\mathbf{W}_u \vec{v}') + \vec{u} \quad (17.3)$$

To compute the final energy, we'll use our regression equation:

$$\hat{E} = \vec{w} \cdot \vec{u} + b \quad (17.4)$$

One final detail is that we will pass on  $\vec{u}$  and the nodes, but we will keep the edges the same at each GNN layer. Remember this is an example model: there are many changes that could be made to the above. Also, it is not kernel learning which is the favorite for this domain. Let's implement it though and see if it works.

### 17.5.1 JAX Model Implementation

```
def x2e(x):
    """convert xyz coordinates to inverse pairwise distance"""
    r2 = jnp.sum((x - x[:, jnp.newaxis, :]) ** 2, axis=-1)
    e = jnp.where(r2 != 0, 1 / r2, 0.0)
    return e

def gnn_layer(nodes, edges, features, we, web, wv, wu):
    """Implementation of the GNN"""
    # make nodes be N x N so we can just multiply directly
    # ek is now shaped N x N x features
    ek = jax.nn.leaky_relu(
        web
        + jnp.repeat(nodes[jnp.newaxis, ...], nodes.shape[0], axis=0)
        @ we
        * edges[..., jnp.newaxis]
    )
    # sum over neighbors to get N x features
    ebar = jnp.sum(ek, axis=1)
    # dense layer for new nodes to get N x features
    new_nodes = jax.nn.leaky_relu(ebar @ wv) + nodes
    # sum over nodes to get shape features
    global_node_features = jnp.sum(new_nodes, axis=0)
    # dense layer for new features
    new_features = jax.nn.leaky_relu(global_node_features @ wu) + features
    # just return features for ease of use
    return new_nodes, edges, new_features
```

We have implemented the code to convert coordinates into inverse pairwise distance and the GNN equations above. Let's test them out.

```
graph_feature_len = 8
node_feature_len = 16
msg_feature_len = 16

# make our weights
def init_weights(g, n, m):
    we = np.random.normal(size=(n, m), scale=1e-1)
    wb = np.random.normal(size=(m), scale=1e-1)
    wv = np.random.normal(size=(m, n), scale=1e-1)
    wu = np.random.normal(size=(n, g), scale=1e-1)
    return [we, wb, wv, wu]

# make a graph
nodes = e
```

(continues on next page)

(continued from previous page)

```

edges = x2e(x)
features = jnp.zeros(graph_feature_len)

# eval
out = gnn_layer(
    nodes,
    edges,
    features,
    *init_weights(graph_feature_len, node_feature_len, msg_feature_len),
)
print("input feautres", features)
print("output features", out[2])

```

```

input feautres [0. 0. 0. 0. 0. 0. 0. 0. 0.]
output features [-0.02791678  0.58952665 -0.02129295 -0.00340767  0.4783953  -0.
 0.01959314
 -0.00684738 -0.00327933]

```

Great! Our model can update the graph features. Now we need to convert this into callable and loss. We'll stack two GNN layers.

```

# get weights for both layers
w1 = init_weights(graph_feature_len, node_feature_len, msg_feature_len)
w2 = init_weights(graph_feature_len, node_feature_len, msg_feature_len)
w3 = np.random.normal(size=(graph_feature_len))
b = 0.0

@jax.jit
def model(nodes, coords, w1, w2, w3, b):
    f0 = jnp.zeros(graph_feature_len)
    e0 = x2e(coords)
    n0 = nodes
    n1, e1, f1 = gnn_layer(n0, e0, f0, *w1)
    n2, e2, f2 = gnn_layer(n1, e1, f1, *w2)
    yhat = f2 @ w3 + b
    return yhat

def loss(nodes, coords, y, w1, w2, w3, b):
    return (model(nodes, coords, w1, w2, w3, b) - y) ** 2

loss_grad = jax.grad(loss, (3, 4, 5, 6))

```

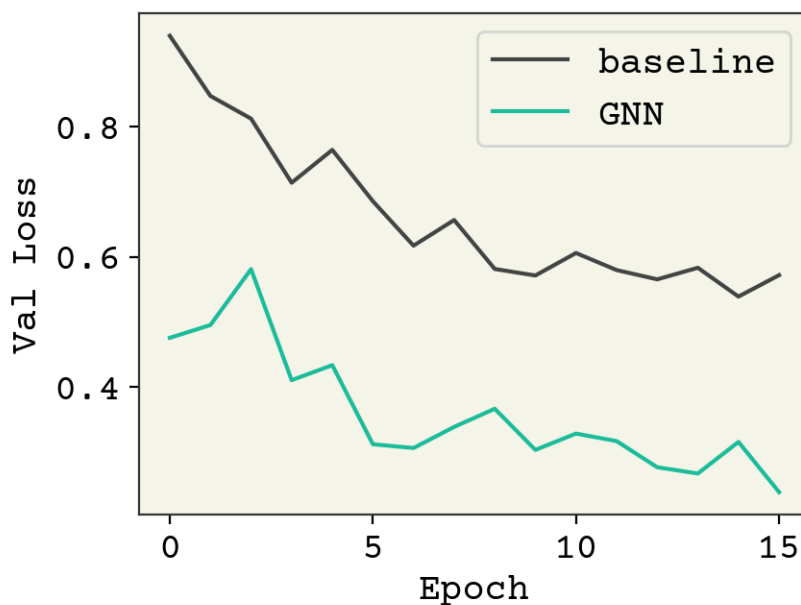
You could pad the molecules to all be the same shape. This is a common strategy. We will skip this though for simplicity.

One small change we've made below is that we scale the learning rate for the GNN to be 1/10 of the rate for the regression parameters. This is because the GNN parameters need to vary slower based on trial and error.

```

eta = 1e-3
val_loss = [0.0 for _ in range(epochs)]
for epoch in range(epochs):
    for d in train_set:
        (e, x), y_raw = convert_record(d)
        y = transform_label(y_raw)
        grad = loss_grad(e, x, y, w1, w2, w3, b)
        # update regression weights
        w3 -= eta * grad[2]
        b -= eta * grad[3]
        # update GNN weights
        for i, w in [(0, w1), (1, w2)]:
            for j in range(len(w)):
                w[j] -= eta * grad[i][j] / 10
    # compute validation loss
    for v in valid_set:
        (e, x), y_raw = convert_record(v)
        y = transform_label(y_raw)
        # convert SE to RMSE
        val_loss[epoch] += loss(e, x, y, w1, w2, w3, b)
    val_loss[epoch] = jnp.sqrt(val_loss[epoch] / 1000)
    eta *= 0.9
plt.plot(baseline_val_loss, label="baseline")
plt.plot(val_loss, label="GNN")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Val Loss")
plt.show()

```



This is a large dataset and we're under training, but hopefully you get the principles of this process! Finally, we'll examine our parity plot.

```

ys = []
yhats = []

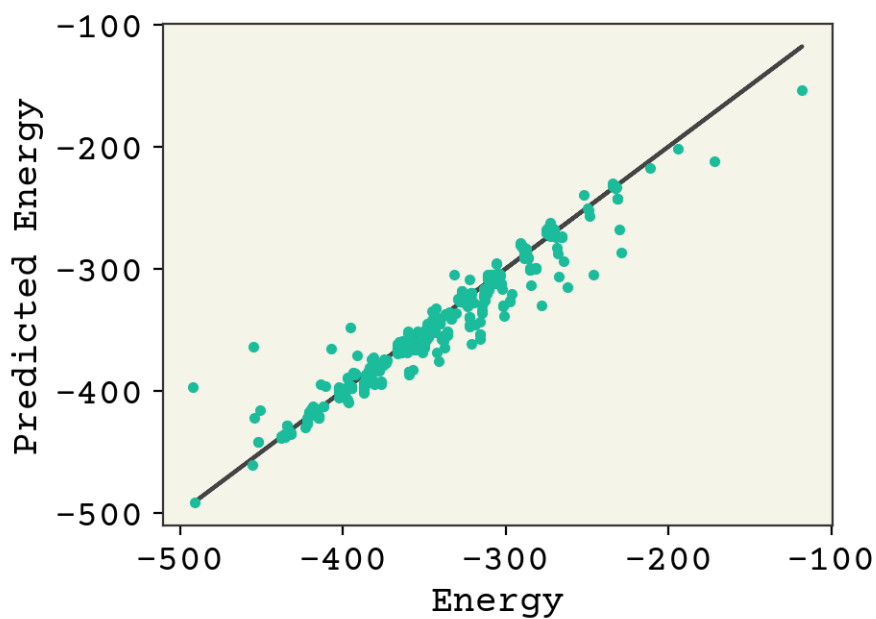
```

(continues on next page)

(continued from previous page)

```
for v in valid_set:
    (e, x), y = convert_record(v)
    ys.append(y)
    yhat_raw = model(e, x, w1, w2, w3, b)
    yhats.append(transform_prediction(yhat_raw))
```

```
plt.plot(ys, ys, "-")
plt.plot(ys, yhats, ".")
plt.xlabel("Energy")
plt.ylabel("Predicted Energy")
plt.show()
```



The clusters are molecule types/sizes. You can see we're starting to get the correct trend within the clusters, but a lot of work needs to be done to move some of them. Additional learning required!

## 17.6 Relevant Videos About Modeling QM9

## 17.7 Cited References

## GENERATIVE RNN IN BROWSER

This chapter builds on the model and information presented in *Deep Learning on Sequences*. We'll see how to develop, train, and deploy a model to a website. The goal is to develop a model that can propose new molecules. A "molecule" is a broad term, so we'll restrict ourselves to molecules like those that might be used for a small molecule drug. These are called drug-like. There is an existing database for this task called **ZINC** (recursive acronym: *ZINC is not commercial*). The ZINC database contains 750 million molecules (as SMILES) that **can be purchased directly**. By training with ZINC, you are restricting your training distribution to molecules that can be synthesized.<sup>[SI15]</sup>

750 million molecules is often enough. Instead of generating new molecules with a model, you can just sample from ZINC and be assured that each molecule is purchasable. Sometimes that is not possible because we're more interested in building a representation (a vector for each molecule), rather than actually sampling like in self-supervised learning. My recommendation in projects is to strongly consider just using ZINC directly if possible. We'll continue on for our approach, because we may go on to use the RNN for "downstream tasks" like predicting solubility using the self-supervised trained RNN.

You can see the final idea of what we're trying to accomplish in this chapter at [whitead.github.io/molecule-dream/](https://whitead.github.io/molecule-dream/). This model was generated using our final code at the bottom.

### 18.1 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

---

**Tip:** To install packages, execute this code in a new cell.

```
!pip install dmol-book
```

If you find install problems, you can get the latest working versions of packages used in [this book here](#)

---

### 18.2 Approach 1: Token Sampling

At first, it may seem simple to generate new molecules. Any sequence of SELFIES is a valid molecule <sup>[KHaseN+20]</sup>. Let's start by trying this approach – making random sequences. Recall that a sequence is an array of tokens, like characters or words. In SELFIES, the tokens are very clear because they are enclosed by square brackets. For example, `[C] [C]` is a SELFIES sequence with two tokens.

We want to generate new molecules similar to ZINC remember, so we'll start by extracting all unique tokens from that database. To save everyone a little time, we'll work with a subset of ZINC that has about 250,000 molecules that comes from the SELFIES repo.

```
import selfies as sf
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import rdkit, rdkit.Chem, rdkit.Chem.Draw
from dataclasses import dataclass
import dmol
```

```
data_url = "https://github.com/aspuru-guzik-group/selfies/raw/
↳16a489afa70882428bc194b2b24a2d33573f1651/examples/vae_example/datasets/dataJ_250k_
↳rdm_zinc_drugs_clean.txt"
pd_data = pd.read_csv(data_url)
print("Total data size", len(pd_data))
```

```
Total data size 249455
```

We now have our starting dataset. To extract the SELFIES tokens, we need to convert the dataset (which is in SMILES) into SELFIES. This might take a while...

```
selfies_list = [sf.encoder(s) for s in pd_data.iloc[:, 0]]
```

Here's what one of the examples look like:

```
print(selfies_list[0])
```

```
[C][C@@Hexpl][C][C][Branch2_1][Ring1][Ring2][N][C][=C][N][=C][C][Branch1_
↳1][Branch2_2][C][=N][N][=C][N][Ring1][Branch1_
↳1][C][=C][Ring1][N][C][C@@Hexpl][Branch1_1][C][C][C][Ring2][Ring1][Ring2]
```

Let's extract now all the possible tokens and count them. I'll initialize my list with the '[nop]' token, which is the SELFIES null token we'll use later for padding.

```
selfies_symbol_counts = {"[nop]": 0}

def parse(s):
    for si in s.split("[")[1:]:
        token = "[" + si
        if token in selfies_symbol_counts:
            selfies_symbol_counts[token] += 1
        else:
            selfies_symbol_counts[token] = 0

[parse(s) for s in selfies_list]

# print out topic tokens
sorted_token_counts = list(sorted(selfies_symbol_counts.items(), key=lambda i: -i[1]))
for p in sorted_token_counts[:10]:
    print(*p)
```



```
[C] 3535514
[=C] 1122876
[Ring1] 857002
[Branch1_1] 621378
[Branch1_2] 523333
[N] 510053
[=O] 318168
[O] 279513
[Branch2_1] 202227
[Ring2] 176665
```

We'll finish up parsing by creating a dictionary to go back from string to index and a list for our vocab.

```
vocab = list(selfies_symbol_counts.keys())
vocab_stoi = {o: i for o, i in zip(vocab, range(len(vocab)))}

def selfies2ints(s):
    result = []
    for si in s.split("[")[1:]:
        result.append(vocab_stoi["[" + si])
    return result

def ints2selfies(v):
    return "".join([vocab[i] for i in v])

# test them out
s = selfies_list[0]
print("selfies:", s)
v = selfies2ints(s)
print("selfies2ints:", v)
so = ints2selfies(v)
print("ints2selfies:", so)
assert so == s
```

```
selfies: [C][C@@Hexpl][C][C][Branch2_1][Ring1][Ring2][N][C][=C][N][=C][C][Branch1_
↵1][Branch2_2][C][=N][N][=C][N][Ring1][Branch1_
↵1][C][=C][Ring1][N][C][C@@Hexpl][Branch1_1][C][C][C][Ring2][Ring1][Ring2]
selfies2ints: [1, 2, 1, 1, 3, 4, 5, 6, 1, 7, 6, 7, 1, 8, 9, 1, 10, 6, 7, 6, 4, 8,
↵1, 7, 4, 6, 1, 2, 8, 1, 1, 1, 5, 4, 5]
ints2selfies: [C][C@@Hexpl][C][C][Branch2_
↵1][Ring1][Ring2][N][C][=C][N][=C][C][Branch1_1][Branch2_
↵2][C][=N][N][=C][N][Ring1][Branch1_1][C][=C][Ring1][N][C][C@@Hexpl][Branch1_
↵1][C][C][C][Ring2][Ring1][Ring2]
```

Now we'll try sampling directly from the tokens. I'm not very good at inferring if a SELFIES string is a reasonable molecule, so we'll render a few using rdkit.

```
def model1():
    # pick random length
    length = np.random.randint(1, 100)
    seq = np.random.choice(len(vocab), size=length)
    return seq
```

```
def draw_examples(model, count=10):
    examples = [ints2selfies(model()) for _ in range(9)]
    examples_smiles = [sf.decoder(s) for s in examples]
    from rdkit.Chem import rdDepictor

    examples_mols = [rdkit.Chem.MolFromSmiles(s) for s in examples_smiles]
    return rdkit.Chem.Draw.MolsToGridImage(
        examples_mols, molsPerRow=3, subImgSize=(250, 250)
    )

draw_examples(model1)
```

<IPython.core.display.SVG object>

As you can see, these molecules are a bit extreme – we see bizarre valences and highly unusual combinations of elements. We can try to sample now according to the frequency of tokens we saw in the corpus (ZINC smiles).

```
# get sorted token counts to be order
# same as vocab
token_counts = [
    x[1] for x in sorted(sorted_token_counts, key=lambda i: vocab_stoi[i[0]])
]
token_counts = np.array(token_counts)

def model2():
    # pick random length
    length = np.random.randint(1, 100)
    seq = np.argmax(
        np.random.multinomial(1, token_counts / np.sum(token_counts), size=length),
        axis=-1,
    )
    return seq

# now draw
draw_examples(model2)
```

<IPython.core.display.SVG object>

These are much better! Still, they are very exotic – if they could even be synthesized they would probably explode if you sneezed on them. The advantage of SELFIES over SMILES is that none of these were invalid. Just a bit unreasonable.

## 18.3 Approach 2: Token RNN

Our next approach will be to use a recurrent neural network as we did in *Deep Learning on Sequences*. We will make one major change. The RNN will be trained to predict a whole sequence instead of one value (like solubility). This training process is called self-supervised. We create labels, from splitting up the training data, so it is not exactly unsupervised.

The way we implement self-supervised training with an RNN is that we feed it a sequence up to position  $i - 1$  and then ask it to predict the sequence value at position  $i$ . Then at inference time, we feed its output at  $i - 1$  as the input to  $i$ . There are a few tricks to make this efficient. The first is that if we split our data into all possible pairs of training labels (sequence up to  $i - 1$ ) and labels (value at  $i$ ), we'll have  $N \times L$  examples which in our case is about 100 million. To treat this, we'll consider the predicted label to be the whole sequence  $\vec{\hat{y}}$  and that will include each individual prediction challenge. For example, predicting  $\hat{y}$  given  $x_0, x_1, x_2$  will be the value of  $\vec{\hat{y}}$  at index 3.

You'll notice there is a bit of a length mismatch with this approach. What would  $\hat{y}_0$  correspond to? You might say it is the predicted label without any input features, but RNNs require an input to have an output. Instead, we'll prefix every sequence with the special [nop] token which means do nothing. Then 0 will be the prediction given a [nop] token. We just need to remember at inference time we need to always begin with the [nop] token to match the training conditions.

Let's start by building our training set using the splitting process. I'm going to define all my hyperparameters in one place as well, so I can easily view them. These hyperparameters are mostly first guesses built from previous experience with GRUs for sequence modeling.

```
@dataclass
class Config:
    vocab_size: int
    example_number: int
    batch_size: int
    buffer_size: int
    embedding_dim: int
    rnn_units: int

config = Config(
    vocab_size=len(vocab),
    example_number=len(selfies_list),
    batch_size=64,
    buffer_size=10000,
    embedding_dim=256,
    rnn_units=128,
)
```

### 18.3.1 Data Construction

```
# now get sequences
encoded = [selfies2ints(s) for s in selfies_list]
# Keras pads with 0s - [nop] in our vocab
padded_seqs = tf.keras.preprocessing.sequence.pad_sequences(encoded, padding="post")

# Now build dataset
seqs_data = tf.data.Dataset.from_tensor_slices((padded_seqs,))

def split_input_target(sequence):
    # remove last input (since no label exists)
```

(continues on next page)

(continued from previous page)

```
# prefix with [nop]
input_text = tf.concat([0], sequence[:-1]), 0)
target_text = sequence
return input_text, target_text
```

```
data = seqs_data.map(split_input_target)
data = (
    data.shuffle(config.buffer_size)
    .batch(config.batch_size, drop_remainder=True)
    .prefetch(tf.data.experimental.AUTOTUNE)
)
# grab examples
for d in data:
    example = d[0]
    example_y = d[1]
    break
```

### 18.3.2 Model Construction

Our model will be the same as *Deep Learning on Sequences*: an embedding, one RNN (GRU variant), and one dense layer. The dense layer could be omitted if an even simpler model was desired. We do not add a softmax to the dense layer because we'll be working with logits instead of probability in training and inference. One change is that we specify the `return_sequences` since we're training across sequences. The other flags to `tf.keras.layers.GRU` are to set-up for deploying the model to javascript, which we'll see below.

```
x = tf.keras.Input(shape=(None,))
ex = tf.keras.layers.Embedding(
    input_dim=config.vocab_size, output_dim=config.embedding_dim, mask_zero=True
)(x)
# reset_after - TFJS requires this as false
h = tf.keras.layers.GRU(
    config.rnn_units, return_sequences=True, reset_after=False, stateful=False
)(ex)
yhat = tf.keras.layers.Dense(config.vocab_size)(h)
train_model = tf.keras.Model(inputs=x, outputs=yhat)
```

Now we'll call it once to make sure it works and to enable Keras to set all the unknown dimensions.

```
yhat = train_model(example)
train_model.summary()
```

Model: "model"

Layer (type)

Output Shape

Param #

=====

input_1 (InputLayer)	[(None, None)]	0
embedding (Embedding)	(None, None, 256)	27648
gru (GRU)	(None, None, 128)	147840
dense (Dense)	(None, None, 108)	13932
=====		
Total params: 189,420		
Trainable params: 189,420		
Non-trainable params: 0		

### 18.3.3 Training

Now we will train the model. We use a special loss function `tf.losses.SparseCategoricalCrossentropy` that works on a sequence of logits for computing cross-entropy in the multi-class setting (multi-class because we have multiple possible tokens). I will train for only 2 epochs to reduce the runtime of this notebook, but I've found ~4 is a reasonable balance of time and loss.

```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
train_model.compile(tf.optimizers.Adam(1e-2), loss=loss)
result = train_model.fit(data, epochs=2, verbose=0)
```

### 18.3.4 Inference

In the inference setting, we need to feed the output from each step back into the model to generate the next token. RNNs are defined by the input and their **state vector** (or state vectors for LSTM). To ensure our model is not forgetting about the tokens it has seen so far, we could keep the state and last output token to use as input to the model. However, an easier approach is to make the underlying model **stateful**. That means its state will be stored as a kind of internal weight and we do not need to bother with passing around the state vector.

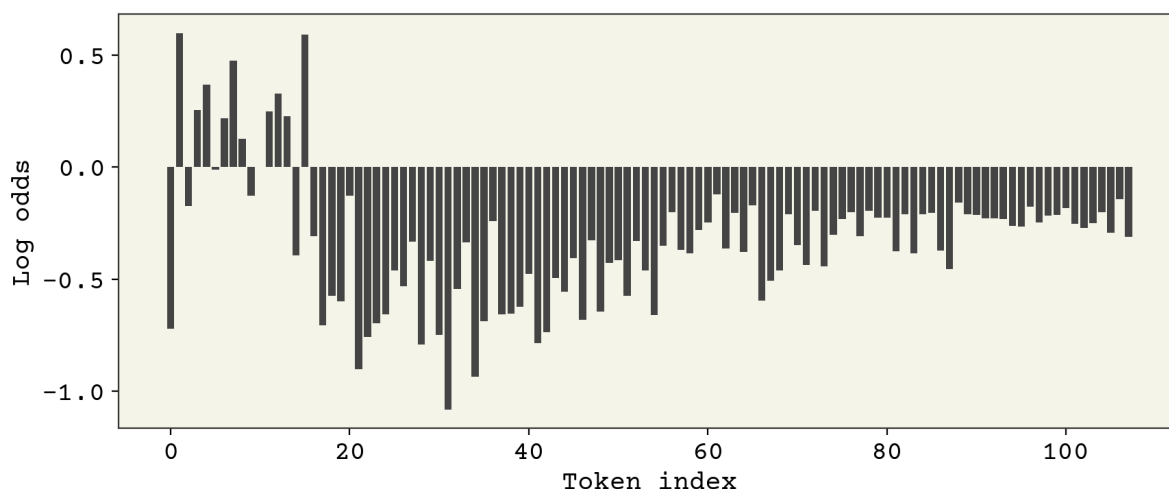
We'll construct a second model that is stateful and make its weights equal to the one we trained. Another change is that we do not need the model to output the whole sequence since we're generating one token at a time.

```
x = tf.keras.Input(shape=(None,), batch_size=1)
ex = tf.keras.layers.Embedding(
    input_dim=config.vocab_size, output_dim=config.embedding_dim, mask_zero=True
)(x)
h = tf.keras.layers.GRU(
    config.rnn_units, return_sequences=False, reset_after=False, stateful=True
)(ex)
yhat = tf.keras.layers.Dense(config.vocab_size)(h)
inference_model = tf.keras.Model(inputs=x, outputs=yhat)

# now copy over weights
inference_model.set_weights(train_model.get_weights())
```

To try the model, remember we start with the [nop] token which is zero.

```
plt.figure(figsize=(10, 4))
x = np.array([0]).reshape(1, 1)
v = inference_model.predict(x, verbose=0)
plt.bar(x=np.arange(config.vocab_size), height=np.squeeze(v))
plt.xlabel("Token index")
plt.ylabel("Log odds")
plt.show()
```



This is the logit probability for each token. We need to sample from this to get our prediction for the first token. We'll sample it with an adjustable parameter called **temperature**. Temperature controls how close we are to sampling the maximum.  $T = 0$  means we sampling the max only,  $T = 1$  means we sample according to the logits, and  $T = \infty$  means we sample randomly.

```
def sample_token(x, T=1):
    return tf.random.categorical(x / T, 1)
```

```
t = sample_token(v)
print(t)
```

```
tf.Tensor([[22]], shape=(1, 1), dtype=int64)
```

Now to continue we feed back into the stateful model. Let's wrap this into a function.

```
def sample_model(T=1):
    length = np.random.randint(1, 100)
    seq = []
    x = tf.zeros((1, 1))
    # reset stateful model
    inference_model.reset_states()
    for _ in range(length):
        v = inference_model.predict(x, verbose=0)
        x = sample_token(v, T)
        seq.append(int(np.squeeze(x.numpy())))
    return seq
```

We'll draw the output for  $T = 1$

```
draw_examples(sample_model)
```

```
<IPython.core.display.SVG object>
```

These are better still! Now we can try adjusting the temperature to see more unusual or more common molecules

$T = 0.5$

```
draw_examples(lambda: sample_model(T=0.5))
```

```
<IPython.core.display.SVG object>
```

$T = 100$

```
draw_examples(lambda: sample_model(T=100))
```

```
<IPython.core.display.SVG object>
```

You can see that the  $T = 0.5$  case seems to work the best of the three.

## 18.4 Model Deployment

Finally, to get the model into javascript as shown at [Molecule Dream](#) we can export using tensorflowjs. It is relatively simple, but you may notice when you load the model into javascript that console errors may indicate you need to adjust flags in model construction.

```
import tensorflowjs as tfjs

tfjs.converters.save_keras_model(inference_model, "tfjs_model")
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to
be built. `model.compile_metrics` will be empty until you train or evaluate the
model.
```

I also like to save the config in case I need to know things like the vocab or dimension of the vectors. JSON files can be easily loaded into javascript objects.

```
import json
from dataclasses import asdict

model_info = asdict(config)
model_info["stoi"] = vocab_stoi
model_info["vocab"] = vocab

with open("tfjs_model/config.json", "w") as f:
    json.dump(model_info, f)
```

```
import * as tf from '@tensorflow/tfjs';
import config from 'tfjs_model/config.json';

const rnn_mod = {}

const loader = tf.loadLayersModel('/tfjs_model/model.json');
loader.then((model) => {
  rnn_mod.model = (t) => {
    return model.predict(t);
  }
  rnn_mod.resetStates = () => {
    model.resetStates();
  }
});
```

Next we need a few helper functions that we used for inference. These are sampling and for converting from integers to the SELFIES tokens.

```
rnn_mod.sample = (x, seed, T = 0.1, k = 1) => {
  return tf.multinomial(
    tf.mul(tf.scalar(2.0), x), k, seed
  );
}

rnn_mod.selfie2vec = (s) => {
  const vec = tf.tensor(s.split('').slice(1).map((e, i) => {
```

(continues on next page)



(continued from previous page)

```
        if (e)
            parseInt(config.stoi['[' + e]);
    });
    return vec;
}

rnn_mod.initVec = () => {
    return tf.tensor([0]);
}

rnn_mod.vec2selfie = (v) => {
    const out = v.array().then((x) => {
        if (Array.isArray(x)) {
            return x.map((e, i) => {
                return config.vocab[parseInt(e)];
            });
        } else {
            return [config.vocab[parseInt(x)]];
        }
    });
    return out;
}

export default rnn_mod;
```

This code gives us a javascript module that can then be used for sampling from our trained model. See the [Molecular Dream repo](#) for the complete code.

## 18.5 Cited References



## **Part V**

### **E. Contributed Chapters**



## HYPERPARAMETER TUNING

---

### Authors:

Mehrad Ansari

---

As you have learned so far, there are number of optimization algorithms to train deep learning models. Training a complex deep learning model can take hours, days or even weeks. Therefore, random guesses for model's hyperparameters might not be very practical and some deeper knowledge is required. These hyperparameters include but are not limited to learning rate, number of hidden layers, dropout rate, batch size and number of epochs to train for. Notably, not all these hyperparameters contribute in the same way to the model's performance, which makes finding the best configurations of these variables in such high dimensional space a nontrivial challenge (searching is expensive!). In this chapter, we look at different strategies to tackle this searching problem.

---

### Audience & Objectives

This chapter builds on *Standard Layers* and *Classification*. After completing this chapter, you should be able to

- Distinguish between training and model design-related hyperparameters
  - Understand the importance of validation data in hyperparameter tuning
  - Understand how each hyperparameter can affect model's performance
- 

Hyperparameters can be categorized into two groups: those used for training and those related to model structure and design.

## 19.1 Training Hyperparameters

### 19.1.1 Learning rate

Gradient descent algorithms multiply the gradient by a scalar known as learning rate to determine the next point in the weights' space. Learning rate is a hyperparameter that controls the step size to move in the direction of lower loss function, with the goal of minimizing it. In most cases, learning rate is manually adjusted during model training. Large learning rates ( $\alpha$ ) make the model learn faster but at the same time it may cause us to miss the minimum loss function and only reach the surrounding of it. In cases where the learning rate is too large, the optimizer overshoots the minimum and the loss updates will lead to divergent behaviours. On the other hand, choosing lower  $\alpha$  values gives a better chance of finding the local minima with the trade-off of needing larger number of epochs and more time.

Note that we can almost never plot the loss as a function of weight's space (as shown in [Fig. 19.1](#)), and this makes finding the reasonable  $\alpha$  tricky. With a proper constant  $\alpha$ , the model can be trained to a passable yet still unsatisfactory accuracy,

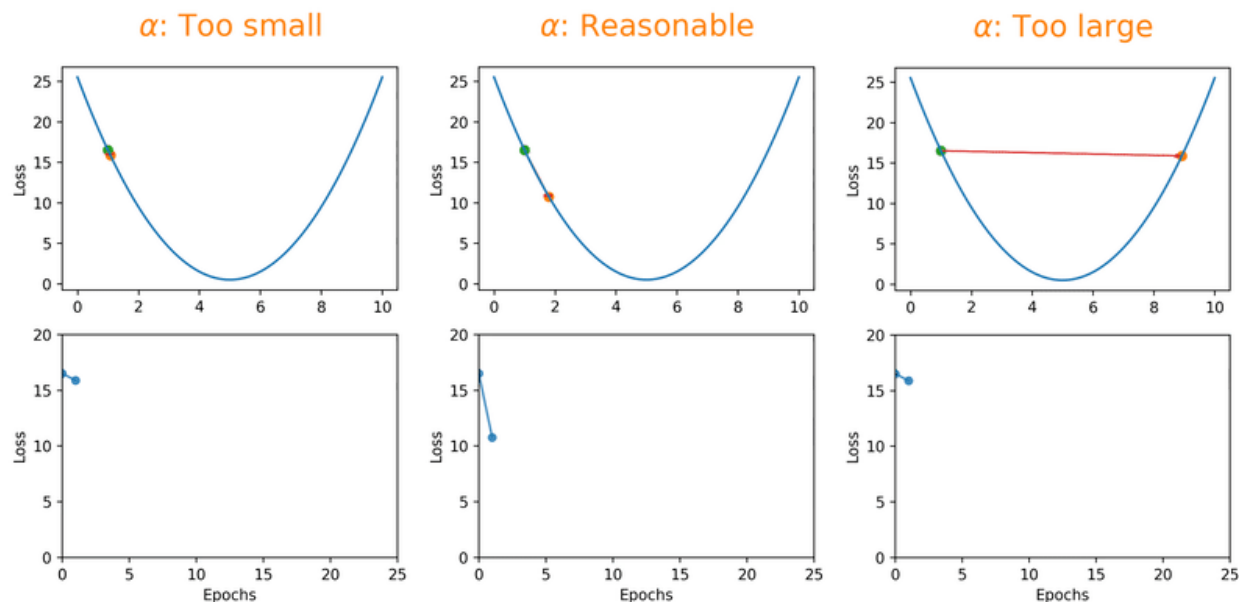


Fig. 19.1: Effect of learning rate on loss.

because the constant  $\alpha$  can be overlarge, especially in the last few epochs. Alternatively,  $\alpha$  can be adaptively adjusted in response to the performance of the model. This is also known as learning rate decay schedule. Some commonly applied decay schedules include linear (step), exponential, polynomial and cyclic. By starting at a larger learning rate, we achieve the rarely discussed benefit of allowing our model to escape the local minima that overfits, and find a broader minimum as learning rate decreases over the number of epochs. If you are using `Keras`, you can either define your own scheduler or use any from `tf.keras.optimizers.schedules` and add it as a callback when you train your model. Some other adaptive learning rate schedulers that might help with reducing tuning efforts can be found in [KDVK21, YLSZ20, VMKS21].

### 19.1.2 Momentum

Another tweak that can help escaping the local minima is the addition of the history to the weight update. “The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction” [GBC17]. Simply speaking, this means that rather than using only the gradient of the current step to guide the search in the weights’ space, momentum also accumulates the gradient of the past steps to determine which direction to go to. Momentum has the effect of smoothing the optimization process, allowing for slower updates to continue in the previous directions instead of oscillating or getting stuck. Momentum has a value greater than zero and less than one, but common values used in practice range between 0.9 and 0.99. Note that momentum is rarely changed in deep learning as a hyperparameter, as it does not really help with configuring the proper learning rate. Instead, it can increase the speed of the optimization process by increasing the likelihood of obtaining a better set of weights in fewer training epochs.

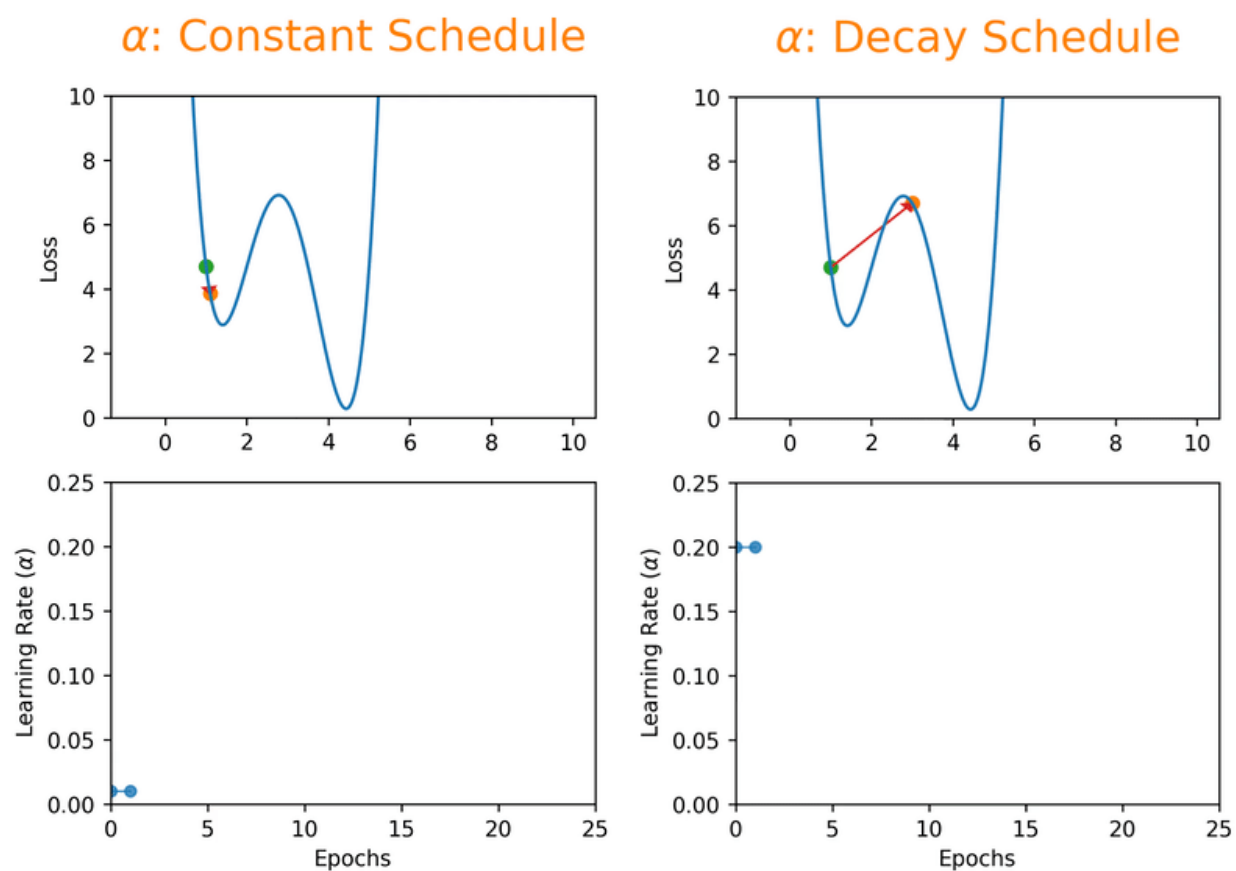


Fig. 19.2: Decay schedules on learning rate can possibly help escaping the local minima.

### 19.1.3 Batch Size

The batch size is always a trade-off between computational efficiency and accuracy. By reducing the batch size, you are essentially reducing the number of samples based on which loss is calculated at each training iteration. Considering model evaluation metrics, smaller batch sizes generalize well on the unobserved data in validation and test set. But why do large batch sizes result a poorer generalization? [This thread](#) on Stack Exchange has some great hypotheses:

- Gradient descent-based optimization makes linear approximation of the loss function, and this approximation will not be the best for highly nonlinear loss functions, thus, having a smaller batch size helps.
- Large-batch training methods are shown to converge to sharp minimizers of the training and testing functions, and that sharp minima results poorer generalization [KMN+16, HHS17, LKSJ20].
- Since smaller samples can have more variations from one another, they can add more noise to convergence. Thus, by keeping the batch size small, we are more likely to escape the local minima and find a more broader one.

As mentioned in the previous section, decaying the learning rate is a common practice in training ML frameworks. But, it has actually been shown that we can obtain the same benefits and learning curve by scaling up the batch size during training instead [SKYL17].

## 19.2 Model Design-Related Hyperparameters

### 19.2.1 Number of hidden layers

The overall structure of neural networks are determined based on the number of hidden layers ( $d$ ). Before describing this, let us talk about the traditional disagreement on how the total number of layers are counted. This disagreement centers around whether or not the input layer is counted. There is an argument that suggests it should not be counted since inputs are not active. We go by this convention, which is also recommended in [RM99]. A single-layer can only be used to represent linearly separable functions, in very simple problems. On the other hand, deep learning models with more layers are more likely to capture more complex features and obtain a relative higher accuracy. As a common sense, you can keep adding layers until the test error does not improve anymore.

### 19.2.2 Number of nodes in each hidden layer

The number of nodes ( $w$ ) in each layer should be carefully considered to avoid overfitting and underfitting. Small ( $w$ ) may result underfitting as the model lacks complexity. On the other hand, too many nodes (large  $w$ ) can cause overfitting and increase training time. Here are some [rules-of-thumb](#) that can be a good start for tuning the number of nodes:

1.  $w_{input} < w < w_{output}$
2.  $w = \frac{2}{3}w_{input} + w_{output}$
3.  $w < 2w_{output}$



### 19.2.3 Regularization

Regularization is applied to counteract the additional model complexity that comes as a result of adding more nodes in deep neural networks. The most common regularization methods ( $L_1$  and  $L_2$ ) are explained in *Regression & Model Assessment*. Hyperparameter  $\lambda$  determines the magnitude of the regularization term in the loss function. Too large  $\lambda$  pushes the weights closer to zero and oversimplifies the structure of the deep learning model, where as undersized  $\lambda$  is not strong enough to reduce the weights. Thanks to its computational efficiency  $L_2$  regularization is more widely used, however,  $L_1$  regularization prevails over  $L_2$  on sparse properties. A brief comparison between  $L_1$  and  $L_2$  are presented in the table below.

$L_1$	$L_2$
Penalizes the sum of the absolute value of weights	Penalizes the sum of square weights
Unable to learn complex patterns	Able to learn complex data patterns
Robust to outliers	Not robust to outliers
Built-in feature selection	No feature selection
Multiple solution	One solution
Sparse solution	Nonsparse solution

## 19.3 Hyperparameter Optimization

Mathematically, hyperparameter optimization (HPO) is the process of finding the set of hyperparameters to either achieve minimum loss or maximum accuracy of the objective model. The general philosophy is the same for all HPO algorithms: determine which hyperparameters to tune and their corresponding search space, adjust them from coarse to fine and obtain the optimal combination.

The state-of-the-art HPO algorithms are classified into two categories: search algorithms and trial schedulers. In general, search algorithms are applied for sampling and trial schedulers deal with the early stopping methods for model evaluation.

### 19.3.1 Search Algorithms

#### Grid Search

As long as you have sufficient computational resources, grid search is a straightforward method for HPO. It performs an exhaustive search on the hyperparameters sets defined by the user. Grid search is applicable for cases where we have limited number of hyperparameters with limited search space.

#### Random Search

Random search is a more effective version of grid search but still computationally exhaustive. It performs a randomized search over hyperparameters from certain distributions over possible parameter values. The searching process continues until the desired accuracy is reached or until the predetermined computational budget is exhausted. Random search works better than grid search considering two benefits: First, a budget can be assigned independently based on the distribution of the search space, whereas in grid search the budget for each hyperparameter set is a fixed value. This makes random search perform better than grid search, especially in search patterns where some hyperparameters are not uniformly distributed. Secondly, a larger time consumption of random search quite certainly will lead to a larger probability finding the best hyperparameter set (this is known as Monte Carlo techniques).

### Bayesian Optimization

Bayesian optimization (BO) is a sequential model-based optimization that aims at becoming less wrong with more data by finding the global optimum by balancing exploration and exploitation that minimizes the number of trials. BO outperforms random and grid search in two aspects: 1. There is no need to have some preliminary knowledge of the distribution of hyperparameters. 2. Unlike random and grid search, the posterior probability is obtained based on a relevant search space, meaning that the algorithm discards the hyperparameter ranges that will most likely not deliver promising solutions according to the previous trials. 3. Another remarkable advantage is that BO is applicable to different settings, where the derivative of the objective function is unknown or expensive to calculate, whether it is stochastic or discrete, or convex or non-convex.

BO consists of two key ingredients: 1. A Bayesian probability surrogate model to model the expensive objective function. A **surrogate** mother is a woman who agrees to bear a child for another person, so in context, a surrogate function is a less expensive approximation of the objective function. A popular surrogate model for BO are Gaussian processes (GPs). 2. An acquisition function that acts as a metric function to determine the next optimal sampling point. This is where BO provides a balanced trade-off between exploitation and exploration. Exploitation means sampling where the surrogate model predicts a high objective, given the current available solutions. Exploration means sampling at locations where the prediction uncertainty is high. These both correspond to high acquisition function values, and the goal is to determine the next sampling point by maximizing the acquisition function.

### 19.3.2 Trial Schedulers

HPO is a time-consuming process and in realistic scenarios, it is necessary to obtain the best hyperparameter with limited available resources. When it comes to training the hyperparameters by hand, by experience we can narrow the search space down, evaluate the model during training and decide whether to stop the training or keep going. An early stopping strategy tries to mimic this behavior and maximize the computational resource budget for promising hyperparameter sets.

#### Median Stopping

Median stopping is a straightforward early termination policy that makes the stopping decision based on the average primary metrics, such as accuracy or loss reported by previous runs. A trial  $X$  is halted at step  $S$  if the best objective value by step  $S$  is strictly worse than the median value of the running average of all completed trials objective values reported at step  $S$  [GSM+17].

#### Curve Fitting

Curve Fitting is another early stopping algorithm rule that predicts the final accuracy or loss using a performance curve regressed from a set of completed or partially completed trials [DSH15]. A trial  $X$  will be stopped at step  $S$  if the extrapolation of the learning curves is worse than the tolerant value of the optimal in the trial history. Unlike median stopping, where we don't have any hyperparameters, curve fitting is a model with parameters and it also requires a training process.

## Successive Halving


Successive Halving (SHA) converts the hyperparameter optimization problem into a non-stochastic best-arm identification and tries to allocate more resources only to those hyperparameter sets that are more promising [JT16]. In SHA, user defines a fixed budget ( $B$ ) and a fixed number of trials ( $n$ ). The hyperparameter sets are uniformly queried for a portion of the initial budget and the corresponding model performances are evaluated for all trials. The worst promising half is dropped while the budget is doubled for the other half and this is done successively until one trial remains. One drawback of SHA is how resources are allocated. There is a trade-off between the total budget ( $B$ ) and number of trials ( $n$ ). If  $n$  is too large, each trial may result in premature termination, whereas too small  $n$  would not provide enough optional choices.

Compared with Bayesian optimization, SHA is easier to understand and it is more computationally efficient as it evaluates the intermediate model results and determines whether to terminate it or not.

## HyperBand

HyperBand is an extension of SHA that tries to solve the resource allocation problem by considering several possible  $n$  values and fixing  $B$  [LJD+18].

## 19.4 Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages, either on your own environment or on Google Colab

**Tip:** To install packages, execute this code in a new cell

```
!pip install dmol-book
```

## 19.5 Hyperparameter Tuning

Now that we know more about different HPO techniques, we develop a deep learning model to predict hemolysis in peptides and tune its hyperparameters. Hemolysis is defined as the disruption of erythrocyte membranes that decrease the life span of red blood cells and causes the release of Hemoglobin. Identifying hemolytic antimicrobial is critical to their applications as non-toxic and safe measurements against bacterial infections. However, distinguishing between hemolytic and non-hemolytic peptides is complicated, as they primarily exert their activity at the charged surface of the bacterial plasma membrane. Timmons and Hewage [TH20] differentiate between the two whether they are active at the zwitterionic eukaryotic membrane, as well as the anionic prokaryotic membrane. The model for hemolytic prediction is trained using data from the Database of Antimicrobial Activity and Structure of Peptides (DBAASP v3 [PAG+21]). The activity is defined by extrapolating a measurement assuming a dose response curves to the point at which 50% of red blood cells (RBC) are lysed. If the activity is below  $100 \frac{\mu g}{ml}$ , it is considered hemolytic. Each measurement is treated independently, so sequences can appear multiple times. The training data contains 9,316 positive and negative sequences of only L- and canonical amino acids.

```
urllib.request.urlretrieve(
    "https://github.com/ur-whitelab/peptide-dashboard/raw/master/ml/data/hemo-
    positive.npz",
    "positive.npz",
```

(continues on next page)

(continued from previous page)

```

)
urllib.request.urlretrieve(
    "https://github.com/ur-whitelab/peptide-dashboard/raw/master/ml/data/hemo-
    ↪negative.npz",
    "negative.npz",
)
with np.load("positive.npz") as r:
    pos_data = r[list(r.keys())[0]]
with np.load("negative.npz") as r:
    neg_data = r[list(r.keys())[0]]

# create labels and stich it all into one
# tensor
labels = np.concatenate(
    (
        np.ones((pos_data.shape[0], 1), dtype=pos_data.dtype),
        np.zeros((neg_data.shape[0], 1), dtype=pos_data.dtype),
    ),
    axis=0,
)
features = np.concatenate((pos_data, neg_data), axis=0)
# we now need to shuffle before creating TF dataset
# so that our train/test/val splits are random
i = np.arange(len(labels))
np.random.shuffle(i)
labels = labels[i]
features = features[i]
full_data = tf.data.Dataset.from_tensor_slices((features, labels))

```

```

def build_model(reg=0.1, add_dropout=False):
    model = tf.keras.Sequential()
    # make embedding and indicate that 0 should be treated specially
    model.add(
        tf.keras.layers.Embedding(
            input_dim=21, output_dim=16, mask_zero=True, input_length=pos_data.shape[
            ↪1]
        )
    )

    # now we move to convolutions and pooling
    model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=5, activation="relu"))
    model.add(tf.keras.layers.MaxPooling1D(pool_size=4))

    model.add(
        tf.keras.layers.Conv1D(
            filters=16,
            kernel_size=3,
            activation="relu",
        )
    )
    model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

    model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=3, activation="relu"))
    model.add(tf.keras.layers.MaxPooling1D(pool_size=2))

```

(continues on next page)

(continued from previous page)

```

# now we flatten to move to hidden dense layers.
# Flattening just removes all axes except 1 (and implicit batch is still in there_
→as always!)

model.add(tf.keras.layers.Flatten())
if add_dropout:
    model.add(tf.keras.layers.Dropout(0.3))
model.add(
    tf.keras.layers.Dense(
        256, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(reg)
    )
)
if add_dropout:
    model.add(tf.keras.layers.Dropout(0.3))
model.add(
    tf.keras.layers.Dense(
        64, activation="tanh", kernel_regularizer=tf.keras.regularizers.l2(reg)
    )
)
if add_dropout:
    model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
return model

```

```

model = build_model(reg=0, add_dropout=False)
# now split into val, test, train
N = pos_data.shape[0] + neg_data.shape[0]
print(N, "examples")
split = int(0.1 * N)
test_data = full_data.take(split).batch(64)
nontest = full_data.skip(split)
val_data, train_data = nontest.take(split).batch(64), nontest.skip(split).shuffle(
    1000
).batch(64)

```

9316 examples

```

def train_model(
    model, lr=1e-3, Reduced_LR=False, Early_stop=False, batch_size=32, epochs=20
):
    tf.keras.backend.clear_session()
    callbacks = []

    if Early_stop:
        early_stopping = tf.keras.callbacks.EarlyStopping(
            monitor="val_auc",
            mode="max",
            patience=5,
            min_delta=1e-2,
            restore_best_weights=True,
        )
        callbacks.append(early_stopping)
    opt = tf.optimizers.Adam(lr)

```

(continues on next page)

(continued from previous page)

```

if Reduced_LR:
    # decay learning rate on plateau
    reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_loss", factor=0.9, patience=5, min_lr=1e-5
    )
    # add a callback to print lr at the beginning of every epoch
    callbacks.append(reduce_lr)
model.compile(
    opt,
    loss="binary_crossentropy",
    metrics=[
        tf.keras.metrics.AUC(from_logits=False),
        tf.keras.metrics.BinaryAccuracy(threshold=0.5),
    ],
)
history = model.fit(
    train_data,
    validation_data=val_data,
    epochs=epochs,
    batch_size=batch_size,
    verbose=0,
    callbacks=callbacks,
)
# print(
#     f"Train Loss: {history.history['loss'][-1]:.3f}, Test Loss: {history.
history['val_loss'][-1]:.3f}"
# )
return history

def plot_losses(history, test_data):
    plt.figure(dpi=100)
    plt.plot(history.history["loss"], label="training")
    plt.plot(history.history["val_loss"], label="validation")
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.show()
    result = model.evaluate(test_data, verbose=0)
    print(f" Test AUC: {result[1]:.3f} Test Accuracy: {result[2]:.3f}")

```

So if you take a look at the `train_model` function defined above, we are using the validation data **exclusively** for the hyperparameter search. The test data is only used in `plot_losses` and gives us an estimation on model's generalization error.

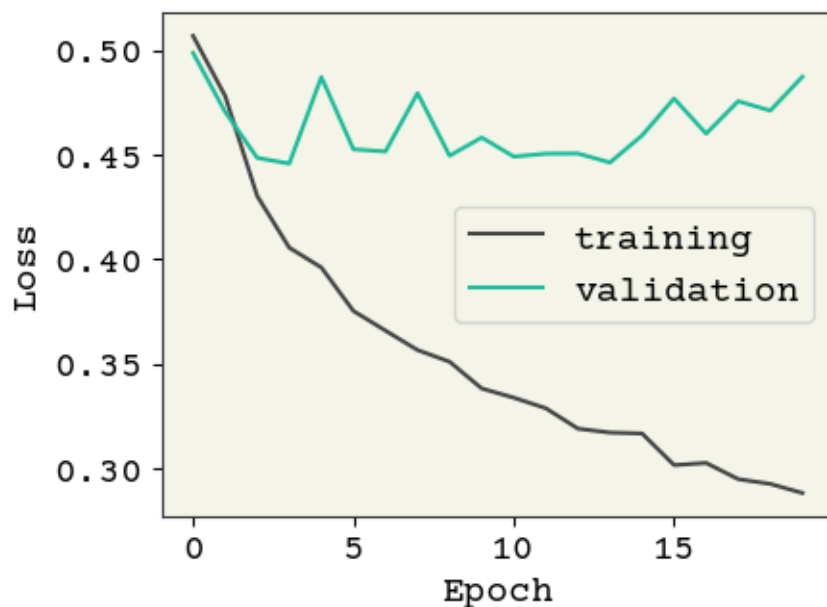
### 19.5.1 Baseline Model

We first start off with a baseline model with similar structure to the model used in *Standard Layers*.

```

history = train_model(model, Reduced_LR=False)
plot_losses(history, test_data)

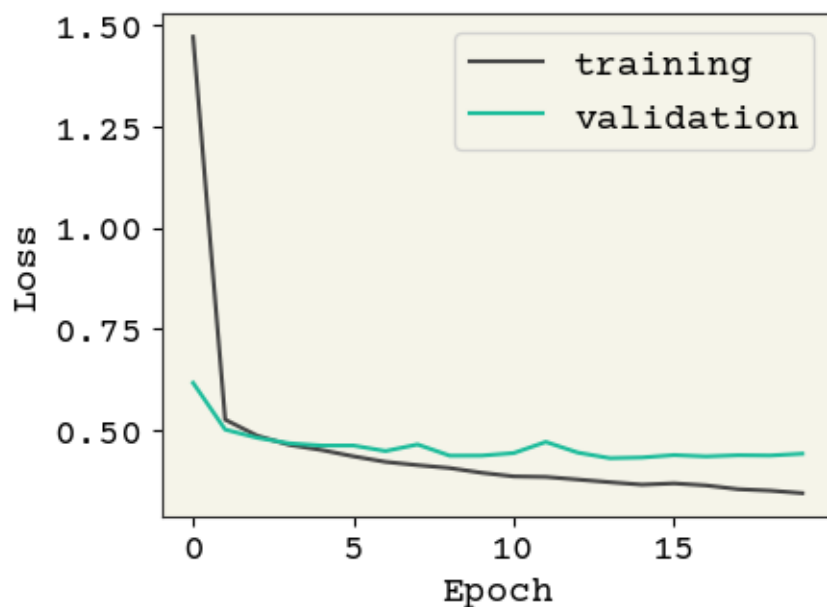
```



Test AUC: 0.789 Test Accuracy: 0.829

So with the default hyperparameters, the **baseline** model above is clearly overfitting to the training data. We first try adding l2 regularization:

```
model = build_model(reg=0.01, add_dropout=False)
history = train_model(model, Reduced_LR=False, Early_stop=False)
plot_losses(history, test_data)
```



Test AUC: 0.813 Test Accuracy: 0.826

Great! Now We have a lower test loss and better AUC.

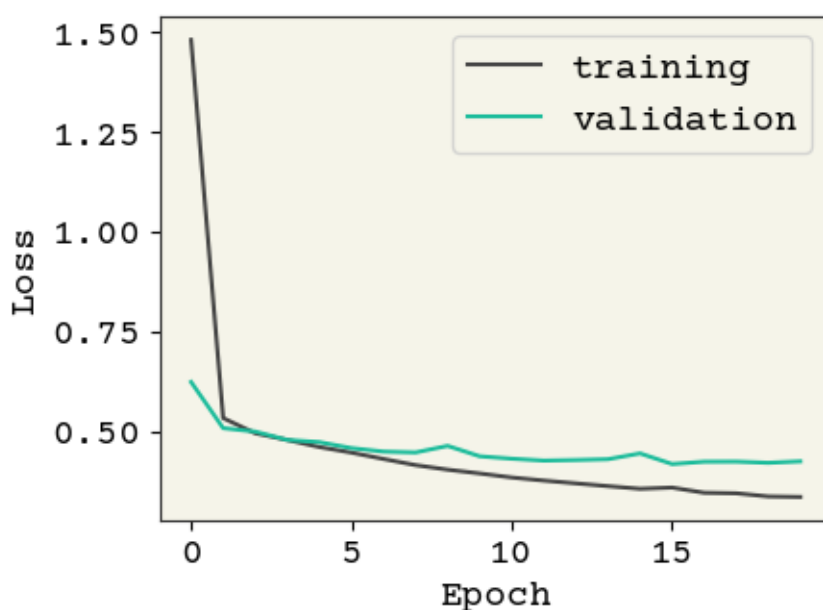
### 19.5.2 Early Stopping

We can use early stopping regularization and return best weights based on maximum obtained AUC value for the **validation data**. This is done by adding the early stopping callback, when compiling the model:

```
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor="val_auc", mode="max", patience=5, min_delta=1e-2, restore_best_
    ←weights=True
)
```

Let's see how the model performs with early stopping:

```
model = build_model(reg=0.01, add_dropout=False)
history = train_model(model, Reduced_LR=False, Early_stop=True)
plot_losses(history, test_data)
```



Test AUC: 0.796 Test Accuracy: 0.816

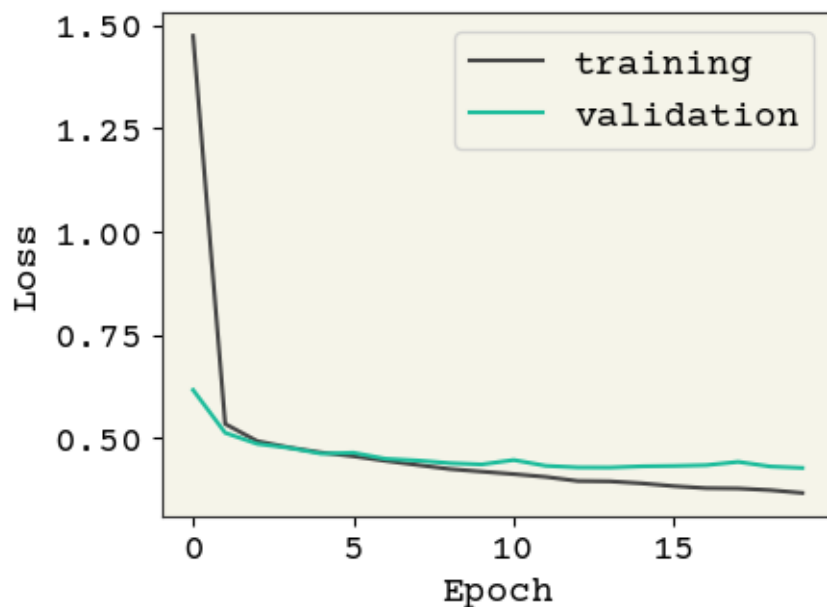
We have almost the same performance but in **fewer** number of epochs. Note that for learning purposes, we have limited the number of epochs to 20 in this example. Early stopping regularization becomes more relevant when we typically have a large number of epochs, as it halts the training and saves computational budget, unless there is gain in more training.

### 19.5.3 Reduced Learning Rate on Plateau and Dropout

Now let's try reducing the learning rate and dropout. Since the training epochs is already limited to 20, we don't use early stopping here:

```
model = build_model(reg=0.01, add_dropout=True)
history = train_model(model, Reduced_LR=True, Early_stop=False, epochs=20)
plot_losses(history, test_data)
```





Test AUC: 0.795    Test Accuracy: 0.824

## 19.6 Discussion

In this chapter, we explored means of reducing overfitting and enhancing feature selection in deep learning models. Techniques suggested can give you a good head start on tuning your model's hyperparameters. What is important is that, you need to experiment to find a good set of hyperparameters. This can be time-consuming, so use your own judgment and intuition to come up with a smart search strategy. Before you start hypertuning, make sure you obtain a baseline model and slowly add more pieces to the puzzle, based on training and validation loss, AUC or other metrics.

There are also some toolkits for hyperparameter optimization that might be handy:

- Ray Tune for PyTorch [LLN+18]
- Keras-Tuner for Keras []
- Optuna [ASY+19]
- Hyperopt [BYC13]
- Scikit-Optimize [Lou17]
- Microsoft's Neural Network Intelligence
- Google's Vizer [GSM+17]

## 19.7 Cited References

## **Part VI**

### **F. Appendix**



## STYLE GUIDE

These are some notes for me and contributors on some style choice specific to this book.

### 20.1 Plots

Import `dmol` package. It will set-up the style for all plots.

### 20.2 Citations

The goal of this book is instructional and not a comprehensive review of the literature. Citations should be added as needed for explaining concepts and justifying statements. Citations should be added as:

### 20.3 Links to Other Resources

Some topics are already covered well in other books and do not need to be repeated here. These should be linked and it is clear what can be found in that resource. For example:

### 20.4 Tips and Warnings

Tips (bordered boxes in text) should be used sparingly to convey critical notes beyond bolding. Examples include shuffling data, not using testing data for validation, etc. A warning is a possible mistake or pitfall. A tip is an important thing to remember and not necessarily a potential mistake.

### 20.5 Right-column Notes

Notes on the right column are meant to be extra information that adds context, reminders, or additional details about the main text. It should not be necessary to read these for understanding the main text. Should be equivalent to footnotes in-text.



## CHANGELOG

### 21.1 Version 0.5.1 (2022-06-20)

- Added SymPy
- Added description of nonlinearity to equiv chapter
- Added example implementation to equiv chapter

### 21.2 Version 0.5.0 (2022-06-17)

- Added Merhad's Chapter on Hyperparameter optimization
- Removed permutation equivariance from equivariance chapter
- Finished irrep discussion in that chapter
- Started new chapter on modern molecule nets
- Removed pygraphviz
- Switched to rdkit

### 21.3 Version 0.4.3 (2022-05-31)

- Increased standard figure size a bit
- Revised GNN intro

### 21.4 Version 0.4.2 (2022-05-30)

- Doubled DPI on figures
- Revised GNN chapter to have discussion of future section
- Fixed long-standing emlp bug
- Improved logic on branch deploy

## 21.5 Version 0.4.1 (2022-05-29)

- Add SchNet implementation
- Switched to three version scheme

## 21.6 Version 0.3 (2022-05-28)

- Centered output figures
- Made github button open on github
- Add SchNet description to GNN

## 21.7 Version 0.2 (2022-05-27)

- Put package onto pypi
- Revised fonts

## 21.8 Version 0.1 (2022-05-26)

- First versioned released
- Added consistent style to plots
- Simplified header
- Replaced logo
- Bumped to latest dependencies



## BIBLIOGRAPHY

- [Str20] J Straub. Mathematical methods for molecular science. 2020.
- [Alp20] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.
- [FZJS21] Victor Fung, Jiaxin Zhang, Eric Juarez, and Bobby G. Sumpter. Benchmarking graph neural networks for materials chemistry. *npj Computational Materials*, June 2021. URL: <https://doi.org/10.1038/s41524-021-00554-0>, doi:10.1038/s41524-021-00554-0.
- [Bal19] Prasanna V Balachandran. Machine learning guided design of functional materials with targeted properties. *Computational Materials Science*, 164:82–90, 2019.
- [GomezBAG20] Rafael Gómez-Bombarelli and Alán Aspuru-Guzik. Machine learning and big-data in computational chemistry. *Handbook of Materials Modeling: Methods: Theory and Modeling*, pages 1939–1962, 2020.
- [NDJ+18] Aditya Nandy, Chenru Duan, Jon Paul Janet, Stefan Gugler, and Heather J Kulik. Strategies and software for machine learning accelerated discovery in transition metal chemistry. *Industrial & Engineering Chemistry Research*, 57(42):13973–13986, 2018.
- [Wei88] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.
- [SKE19] Murat Cihan Sorkun, Abhishek Khetan, and Süleyman Er. AqSolDB, a curated reference set of aqueous solubility and 2D descriptors for a diverse set of compounds. *Sci. Data*, 6(1):143, 2019. doi:10.1038/s41597-019-0151-1.
- [PDN05] Duc Truong Pham, Stefan S Dimov, and Chi D Nguyen. Selection of k in k-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 219(1):103–119, 2005.
- [SKE19] Murat Cihan Sorkun, Abhishek Khetan, and Süleyman Er. AqSolDB, a curated reference set of aqueous solubility and 2D descriptors for a diverse set of compounds. *Sci. Data*, 6(1):143, 2019. doi:10.1038/s41597-019-0151-1.
- [Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
- [NMB+18] Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. A modern take on the bias-variance tradeoff in neural networks. *arXiv preprint arXiv:1810.08591*, 2018.
- [BCRT19] Rina Foygel Barber, Emmanuel J Candes, Aaditya Ramdas, and Ryan J Tibshirani. Predictive inference with the jackknife+. *arXiv preprint arXiv:1905.02928*, 2019.
- [SBG+20] Christopher Sutton, Mario Boley, Luca M Ghiringhelli, Matthias Rupp, Jilles Vreeken, and Matthias Scheffler. Identifying domains of applicability of machine learning models for materials science. *Nature Communications*, 11(1):1–9, 2020.

- [Ras18] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.
- [SZY+19] Wenbo Sun, Yujie Zheng, Ke Yang, Qi Zhang, Akeel A Shah, Zhou Wu, Yuyang Sun, Liang Feng, Dongyang Chen, Zeyun Xiao, and others. Machine learning–assisted molecular design and efficiency prediction for high-performance organic photovoltaic materials. *Science advances*, 5(11):eaay4275, 2019.
- [BJW18] Rainier Barrett, Shaoyi Jiang, and Andrew D White. Classifying antimicrobial and multifunctional peptides with bayesian network models. *Peptide Science*, 110(4):e24079, 2018.
- [WRF+18] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.
- [MTKT18] Hiroto Moriaki, Yu-Shi Tian, Norihito Kawashita, and Tatsuya Takagi. Mordred: a molecular descriptor calculator. *Journal of cheminformatics*, 10(1):4, 2018.
- [CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
- [YIAPLP21] Badr Youbi Idrissi, Martin Arjovsky, Mohammad Pezeshki, and David Lopez-Paz. Simple data balancing achieves competitive worst-group-accuracy. *arXiv e-prints*, pages arXiv–2110, 2021.
- [HG09] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.
- [BL19] Jonathon Byrd and Zachary Lipton. What is the effect of importance weighting in deep learning? In *International Conference on Machine Learning*, 872–881. PMLR, 2019.
- [SBH+21] Hyebin Song, Bennett J Bremer, Emily C Hinds, Garvesh Raskutti, and Philip A Romero. Inferring protein sequence-function relationships with large-scale positive-unlabeled learning. *Cell Systems*, 12(1):92–101, 2021.
- [SKE19] Murat Cihan Sorkun, Abhishek Khetan, and Süleyman Er. AqSolDB, a curated reference set of aqueous solubility and 2D descriptors for a diverse set of compounds. *Sci. Data*, 6(1):143, 2019. doi:10.1038/s41597-019-0151-1.
- [SSAB20] Christoph Scherer, René Scheid, Denis Andrienko, and Tristan Bereau. Kernel-based machine learning for efficient simulations of molecular liquids. *Journal of Chemical Theory and Computation*, 16(5):3194–3204, 2020.
- [RTMullerVL12] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O Anatole Von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical review letters*, 108(5):058301, 2012.
- [LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: a view from the width. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6232–6240. 2017.
- [NMB+18] Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. A modern take on the bias-variance tradeoff in neural networks. *arXiv preprint arXiv:1810.08591*, 2018.
- [GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249–256. 2010.
- [CirecsanMGS10] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.
- [FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

- [ZL17] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2017. URL: <https://arxiv.org/abs/1611.01578>.
- [LJD+18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL: <http://jmlr.org/papers/v18/16-558.html>.
- [CSTR14] Catherine Ching Han Chang, Jiangning Song, Beng Ti Tey, and Ramakrishnan Nagasundara Ramanan. Bioinformatics approaches for improved recombinant protein production in escherichia coli: protein solubility prediction. *Briefings in bioinformatics*, 15(6):953–962, 2014.
- [LecunBottouBengioHaffner98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [KRK+18] Sameer Khurana, Reda Rawi, Khalid Kunji, Gwo-Yu Chuang, Halima Bensmail, and Raghvendra Mall. DeepSol: a deep learning framework for sequence-based protein solubility prediction. *Bioinformatics*, 34(15):2605–2613, 03 2018. URL: <https://doi.org/10.1093/bioinformatics/bty166>, doi:10.1093/bioinformatics/bty166.
- [BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreychuk Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.
- [STIMkadry18] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Mądry. How does batch normalization help optimization? In *Proceedings of the 32nd international conference on neural information processing systems*, 2488–2498. 2018.
- [GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: representing model uncertainty in deep learning. In *international conference on machine learning*, 1050–1059. 2016.
- [EYG19] Steffen Eger, Paul Youssef, and Iryna Gurevych. Is it time to swish? comparing deep learning activation functions across nlp tasks. *arXiv preprint arXiv:1901.02671*, 2019.
- [HG16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [DJI+20] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.
- [BBL+17] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.
- [WPC+20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
- [LWC+20] Zhiheng Li, Geemi P Wellawatte, Maghesree Chakraborty, Heta A Gandhi, Chenliang Xu, and Andrew D White. Graph neural network based coarse-grained mapping prediction. *Chemical Science*, 11(35):9524–9531, 2020.
- [YCW20] Ziyue Yang, Maghesree Chakraborty, and Andrew D White. Predicting chemical shifts with graph neural networks. *bioRxiv*, 2020.
- [XFLW+19] Tian Xie, Arthur France-Lanord, Yanming Wang, Yang Shao-Horn, and Jeffrey C Grossman. Graph dynamical networks for unsupervised learning of atomic scale dynamics in materials. *Nature communications*, 10(1):1–9, 2019.
- [SLRPW21] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alex Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. <https://distill.pub/2021/gnn-intro>. doi:10.23915/distill.00033.
- [XG18] Tian Xie and Jeffrey C. Grossman. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Phys. Rev. Lett.*, 120:145301, Apr 2018. URL: <https://link.aps.org/doi/10.1103/PhysRevLett.120.145301>, doi:10.1103/PhysRevLett.120.145301.

- [KW16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [GSR+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.
- [LTBZ15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.
- [CGCB14] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
- [XHLJ18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*. 2018.
- [LDLio19] Enxhell Luzhnica, Ben Day, and Pietro Liò. On graph classification networks, datasets and baselines. *arXiv preprint arXiv:1905.04682*, 2019.
- [MSK20] Diego Mesquita, Amauri Souza, and Samuel Kaski. Rethinking pooling in graph neural networks. *Advances in Neural Information Processing Systems*, 2020.
- [GZBA21] Daniele Grattarola, Daniele Zambon, Filippo Maria Bianchi, and Cesare Alippi. Understanding pooling in graph neural networks. *arXiv preprint arXiv:2110.05292*, 2021.
- [DRA21] Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. Understanding convolutions on graphs. *Distill*, 2021. <https://distill.pub/2021/understanding-gnns>. doi:10.23915/distill.00032.
- [ZKR+17] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, 3391–3401. 2017.
- [BHB+18] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, and others. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
- [SchuttSK+18] Kristof T Schütt, Huziel E Sauceda, P-J Kindermans, Alexandre Tkatchenko, and K-R Müller. Schnet—a deep learning architecture for molecules and materials. *The Journal of Chemical Physics*, 148(24):241722, 2018.
- [CW22] Sam Cox and Andrew D White. Symmetric molecular dynamics. *arXiv preprint arXiv:2204.01114*, 2022.
- [ZSX+18] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.
- [HYL17] Will Hamilton, Zhitaoying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, 1024–1034. 2017.
- [EPBM19] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations*. 2019.
- [SMBGunnemann18] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.
- [KGrossGunnemann20] Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *International Conference on Learning Representations*. 2020.
- [JES+20] Bowen Jing, Stephan Eismann, Patricia Suriana, Raphael JL Townshend, and Ron Dror. Learning from protein structure with geometric vector perceptrons. *arXiv preprint arXiv:2009.01411*, 2020.
- [Velivckovic22] Petar Veličković. Message passing all the way up. *arXiv preprint arXiv:2202.11097*, 2022.
- [BFO+21] Cristian Bodnar, Fabrizio Frasca, Nina Otter, Yuguang Wang, Pietro Lio, Guido F Montufar, and Michael Bronstein. Weisfeiler and leman go cellular: cw networks. *Advances in Neural Information Processing Systems*, 34:2625–2640, 2021.

- [TZK21] Erik Thiede, Wenda Zhou, and Risi Kondor. Autobahn: automorphism-based graph neural nets. *Advances in Neural Information Processing Systems*, 2021.
- [RTMullerVL12] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O Anatole Von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical review letters*, 108(5):058301, 2012.
- [GSR+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.
- [ZKR+17] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, 3391–3401. 2017.
- [TSK+18] Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. Tensor field networks: rotation-and translation-equivariant neural networks for 3d point clouds. *arXiv preprint arXiv:1802.08219*, 2018.
- [WGW+18] Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco S Cohen. 3d steerable cnns: learning rotationally equivariant features in volumetric data. In *Advances in Neural Information Processing Systems*, 10381–10392. 2018.
- [Est20] Carlos Esteves. Theoretical aspects of group equivariant neural networks. *arXiv preprint arXiv:2004.05154*, 2020.
- [MGSNoe20] Benjamin Kurt Miller, Mario Geiger, Tess E Smidt, and Frank Noé. Relevance of rotationally equivariant convolutions for predicting molecular properties. *arXiv preprint arXiv:2008.08461*, 2020.
- [MGB+21] Felix Musil, Andrea Grisafi, Albert P. Bartók, Christoph Ortner, Gábor Csányi, and Michele Ceriotti. Physics-inspired structural representations for molecules and materials. *arXiv preprint arXiv:2101.04673*, 2021.
- [CJZ+20] Alex K Chew, Shengli Jiang, Weiqi Zhang, Victor M Zavala, and Reid C Van Lehn. Fast predictions of liquid-phase acid-catalyzed reaction rates using molecular dynamics simulations and convolutional neural networks. *Chemical Science*, 11(46):12464–12476, 2020.
- [Beh11] Jörg Behler. Atom-centered symmetry functions for constructing high-dimensional neural network potentials. *The Journal of chemical physics*, 134(7):074106, 2011.
- [BartokKCsanyi13] Albert P. Bartók, Risi Kondor, and Gábor Csányi. On representing chemical environments. *Phys. Rev. B*, 87:184115, May 2013. URL: <https://link.aps.org/doi/10.1103/PhysRevB.87.184115>, doi:10.1103/PhysRevB.87.184115.
- [RSPoczos17] Siamak Ravanbakhsh, Jeff Schneider, and Barnabás Póczos. Equivariance through parameter-sharing. In *International Conference on Machine Learning*, 2892–2901. 2017.
- [FR00] Jefferson Foote and Anandi Raman. A relation between the principal axes of inertia and ligand binding. *Proceedings of the National Academy of Sciences*, 97(3):978–983, 2000.
- [Pow77] Michael James David Powell. Restart procedures for the conjugate gradient method. *Mathematical programming*, 12(1):241–254, 1977.
- [TSK+18] Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. Tensor field networks: rotation-and translation-equivariant neural networks for 3d point clouds. *arXiv preprint arXiv:1802.08219*, 2018.
- [WGW+18] Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco S Cohen. 3d steerable cnns: learning rotationally equivariant features in volumetric data. In *Advances in Neural Information Processing Systems*, 10381–10392. 2018.
- [FSIW20] Marc Finzi, Samuel Stanton, Pavel Izmailov, and Andrew Gordon Wilson. Generalizing convolutional neural networks for equivariance to lie groups on arbitrary continuous data. *arXiv preprint arXiv:2002.12880*, 2020.



- [CGW19] Taco S Cohen, Mario Geiger, and Maurice Weiler. A general theory of equivariant cnns on homogeneous spaces. *Advances in neural information processing systems*, 32:9145–9156, 2019.
- [KT18] Risi Kondor and Shubhendu Trivedi. On the generalization of equivariance and convolution in neural networks to the action of compact groups. In *International Conference on Machine Learning*, 2747–2755. 2018.
- [LW20] Leon Lang and Maurice Weiler. A wigner-eckart theorem for group equivariant convolution kernels. *arXiv preprint arXiv:2010.10952*, 2020.
- [FWW21] Marc Finzi, Max Welling, and Andrew Gordon Wilson. A practical method for constructing equivariant multilayer perceptrons for arbitrary matrix groups. *Arxiv*, 2021.
- [WAR20] Renhao Wang, Marjan Albooyeh, and Siamak Ravanbakhsh. Equivariant maps for hierarchical structures. *arXiv preprint arXiv:2006.03627*, 2020.
- [BSS+21] Simon Batzner, Tess E. Smidt, Lixin Sun, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, and Boris Kozinsky. Se(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *arXiv preprint arXiv:2101.03164*, 2021.
- [KGrossGunnemann20] Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. *arXiv preprint arXiv:2003.03123*, 2020.
- [SHF+21] Victor Garcia Satorras, Emiel Hooeboom, Fabian B. Fuchs, Ingmar Posner, and Max Welling. E(n) equivariant normalizing flows for molecule generation in 3d. *arXiv preprint arXiv:2105.09016*, 2021.
- [SK19] Connor Shorten and Taghi M Khoshgohar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.
- [Zee16] Anthony Zee. *Group theory in a nutshell for physicists*. Princeton University Press, 2016.
- [RBTH20] David W Romero, Erik J Bekkers, Jakub M Tomczak, and Mark Hoogendoorn. Attentive group equivariant convolutional networks. *arXiv*, pages arXiv–2002, 2020.
- [CW16] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, 2990–2999. 2016.
- [Ser77] Jean-Pierre Serre. *Linear representations of finite groups*. Volume 42. Springer, 1977.
- [KLT18] Risi Kondor, Zhen Lin, and Shubhendu Trivedi. Clebsch-gordan nets: a fully fourier space spherical convolutional neural network. *arXiv preprint arXiv:1806.09231*, 2018.
- [WRF+18] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.
- [LS04] John D Lee and Katrina A See. Trust in automation: designing for appropriate reliance. *Human factors*, 46(1):50–80, 2004.
- [DVK17] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.
- [GF17] Bryce Goodman and Seth Flaxman. European Union regulations on algorithmic decision-making and a “right to explanation”. *AI Magazine*, 38(3):50–57, 2017.
- [Dev19] Organisation for Economic Co-operation and Development. Recommendation of the Council on Artificial Intelligence. 2019. URL: <https://legalinstruments.oecd.org/en/instruments/OECD-LEGAL-0449>.
- [CLG+15] Rich Caruana, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1721–1730. ACM, 2015.
- [Mil19] Tim Miller. Explanation in artificial intelligence: insights from the social sciences. *Artificial intelligence*, 267:1–38, 2019.

- [MSK+19] James W Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Interpretable machine learning: definitions, methods, and applications. *eprint arXiv*, pages 1–11, 2019. URL: <http://arxiv.org/abs/1901.04592>.
- [MSMuller18] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018.
- [BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [AGFW21] Mehrad Ansari, Heta A Gandhi, David G Foster, and Andrew D White. Iterative symbolic regression for learning transport equations. *arXiv preprint arXiv:2108.03293*, 2021.
- [BD00] Lynne Billard and Edwin Diday. Regression analysis for interval-valued data. In *Data analysis, classification, and related methods*, pages 369–374. Springer, 2000.
- [UT20] Silviu-Marian Udrescu and Max Tegmark. Ai feynman: a physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.
- [CSGB+20] Miles Cranmer, Alvaro Sanchez Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. *Advances in Neural Information Processing Systems*, 33:17429–17442, 2020.
- [WSW22] Geemi P Wellawatte, Aditi Seshadri, and Andrew D White. Model agnostic generation of counterfactual explanations for molecules. *Chem. Sci.*, pages –, 2022. URL: <http://dx.doi.org/10.1039/D1SC05259D>, doi:10.1039/D1SC05259D.
- [RSG16a] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 1135–1144. 2016.
- [RSG16b] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386*, 2016.
- [WMR17] Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: automated decisions and the gdpr. *Harv. JL & Tech.*, 31:841, 2017.
- [KL17] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, 1885–1894. PMLR, 2017.
- [SML+21] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J. Anders, and Klaus-Robert Müller. Explaining deep neural networks and beyond: a review of methods and applications. *Proceedings of the IEEE*, 109(3):247–278, 2021. doi:10.1109/JPROC.2021.3060483.
- [Mol19] Christoph Molnar. *Interpretable Machine Learning*. Lulu.com, 2019. <https://christophm.github.io/interpretable-ml-book/>.
- [BFL+17] David Balduzzi, Marcus Frean, Lennox Leary, J. P. Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: if resnets are the answer, then what is the question? In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, 342–350. PMLR, 06–11 Aug 2017. URL: <http://proceedings.mlr.press/v70/balduzzi17b.html>.
- [STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, 3319–3328. PMLR, 2017.
- [STK+17] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.
- [MBL+19] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. *Layer-Wise Relevance Propagation: An Overview*, pages 193–209. Springer International Publishing, Cham, 2019. URL: [https://link.springer.com/chapter/10.1007%2F978-3-030-28954-6\\_10](https://link.springer.com/chapter/10.1007%2F978-3-030-28954-6_10).

- [vStrumbeljK14] Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems*, 41(3):647–665, 2014.
- [BW21] Rainier Barrett and Andrew D. White. Investigating active learning and meta-learning for iterative peptide design. *Journal of Chemical Information and Modeling*, 61(1):95–105, 2021. URL: <https://doi.org/10.1021/acs.jcim.0c00946>, doi:10.1021/acs.jcim.0c00946.
- [SLL20] Pascal Sturmfels, Scott Lundberg, and Su-In Lee. Visualizing the impact of feature attribution baselines. *Distill*, 2020. <https://distill.pub/2020/attribution-baselines>. doi:10.23915/distill.00022.
- [CK18] Kangway V Chuang and Michael J Keiser. Comment on “predicting reaction performance in c–n cross-coupling using machine learning”. *Science*, 362(6416):eaat8603, 2018.
- [Lip18] Zachary C Lipton. The mythos of model interpretability: in machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, 2018.
- [KWG+18] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and others. Interpretability beyond feature attribution: quantitative testing with concept activation vectors (tcav). In *International conference on machine learning*, 2668–2677. PMLR, 2018.
- [NB20] Danilo Numeroso and Davide Bacciu. Explaining deep graph networks with molecular counterfactuals. *arXiv preprint arXiv:2011.05134*, 2020.
- [RH10] David Rogers and Mathew Hahn. Extended-connectivity fingerprints. *Journal of chemical information and modeling*, 50(5):742–754, 2010.
- [AZL21] Chirag Agarwal, Marinka Zitnik, and Himabindu Lakkaraju. Towards a rigorous theoretical analysis and evaluation of gnn explanations. *arXiv preprint arXiv:2106.09078*, 2021.
- [YYGJ20] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in graph neural networks: a taxonomic survey. *arXiv preprint arXiv:2012.15445*, 2020.
- [MRC21] Andreas Madsen, Siva Reddy, and Sarath Chandar. Post-hoc interpretability for neural nlp: a survey. *arXiv preprint arXiv:2108.04840*, 2021.
- [NPK+21] AkshatKumar Nigam, Robert Pollice, Mario Krenn, Gabriel dos Passos Gomes, and Alán Aspuru-Guzik. Beyond generative models: superfast traversal, optimization, novelty, exploration and discovery (stoned) algorithm for molecules using selfies. *Chem. Sci.*, 12:7079–7090, 2021. URL: <http://dx.doi.org/10.1039/D1SC00231G>, doi:10.1039/D1SC00231G.
- [BHB+18] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, and others. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
- [ZSX+18] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.
- [BP97] Shumeet Baluja and Dean A. Pomerleau. Expectation-based selective attention for visual monitoring and control of a robot vehicle. *Robotics and Autonomous Systems*, 22(3):329–344, 1997. Robot Learning: The New Wave. URL: <http://www.sciencedirect.com/science/article/pii/S0921889097000468>, doi:[https://doi.org/10.1016/S0921-8890\(97\)00046-8](https://doi.org/10.1016/S0921-8890(97)00046-8).
- [TG80] Anne M Treisman and Garry Gelade. A feature-integration theory of attention. *Cognitive psychology*, 12(1):97–136, 1980.
- [LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
- [VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008. 2017.



- [MDM+20] Łukasz Maziarka, Tomasz Danel, Sławomir Mucha, Krzysztof Rataj, Jacek Tabor, and Stanisław Jastrzębski. Molecule attention transformer. *arXiv preprint arXiv:2002.08264*, 2020.
- [Wei88] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.
- [SKE19] Murat Cihan Sorkun, Abhishek Khetan, and Süleyman Er. AqSolDB, a curated reference set of aqueous solubility and 2D descriptors for a diverse set of compounds. *Sci. Data*, 6(1):143, 2019. doi:10.1038/s41597-019-0151-1.
- [YCW20] Ziyue Yang, Maghesree Chakraborty, and Andrew D White. Predicting chemical shifts with graph neural networks. *bioRxiv*, 2020.
- [KGrossGunnemann20] Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *International Conference on Learning Representations*. 2020.
- [VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008, 2017.
- [KHN+20] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alan Aspuru-Guzik. Self-referencing embedded strings (SELFIES): a 100% robust molecular string representation. *Machine Learning: Science and Technology*, 1(4):045024, nov 2020. URL: <https://doi.org/10.1088/2632-2153/aba947>, doi:10.1088/2632-2153/aba947.
- [BFSV19] Nathan Brown, Marco Fiscato, Marwin HS Segler, and Alain C Vaucher. Guacamol: benchmarking models for de novo molecular design. *Journal of chemical information and modeling*, 59(3):1096–1108, 2019.
- [OBoyle12] Noel M O’Boyle. Towards a universal smiles representation—a standard method to generate canonical smiles based on the inchi. *Journal of cheminformatics*, 4(1):1–14, 2012.
- [RZS20] Kohulan Rajan, Achim Zielesny, and Christoph Steinbeck. Decimer: towards deep learning for chemical image recognition. *Journal of Cheminformatics*, 12(1):1–9, 2020.
- [CGR20] Seyone Chithrananda, Gabe Grand, and Bharath Ramsundar. Chemberta: large-scale self-supervised pre-training for molecular property prediction. *arXiv preprint arXiv:2010.09885*, 2020.
- [Bal11] Philip Ball. Beyond the bond. *Nature*, 469(7328):26–28, 2011.
- [SKTW18] Marwin HS Segler, Thierry Kogej, Christian Tyrchan, and Mark P Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4(1):120–131, 2018.
- [GomezBWD+18] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.
- [BMR+20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and others. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [TDG+21] Yi Tay, Mostafa Dehghani, Jai Gupta, Dara Bahri, Vamsi Aribandi, Zhen Qin, and Donald Metzler. Are pre-trained convolutions better than pre-trained transformers? *arXiv preprint arXiv:2105.03322*, 2021.
- [RM21] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: beyond the few-shot paradigm. *arXiv preprint arXiv:2102.07350*, 2021.
- [WWCF21] Yuyang Wang, Jianren Wang, Zhonglin Cao, and Amir Barati Farimani. Molclr: molecular contrastive learning of representations via graph neural networks. *arXiv preprint arXiv:2102.10056*, 2021.
- [LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: a robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

- [SLP+21] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.
- [BDC+18] Keith T Butler, Daniel W Davies, Hugh Cartwright, Olexandr Isayev, and Aron Walsh. Machine learning for molecular and materials science. *Nature*, 559(7715):547–555, 2018.
- [TDW+19] Vahe Tshitoyan, John Dagdelen, Leigh Weston, Alexander Dunn, Ziqin Rong, Olga Kononova, Kristin A Persson, Gerbrand Ceder, and Anubhav Jain. Unsupervised word embeddings capture latent knowledge from materials science literature. *Nature*, 571(7763):95–98, 2019.
- [SC16] Matthew C Swain and Jacqueline M Cole. Chemdataextractor: a toolkit for automated extraction of chemical information from the scientific literature. *Journal of chemical information and modeling*, 56(10):1894–1904, 2016.
- [FAT+20] Annemarie Friedrich, Heike Adel, Federico Tomazic, Johannes Hingerl, Renou Benteau, Anika Maruscyk, and Lukas Lange. The sofc-exp corpus and neural approaches to information extraction in the materials science domain. *arXiv preprint arXiv:2006.03039*, 2020.
- [MFGS18] Daniel Merk, Lukas Friedrich, Francesca Grisoni, and Gisbert Schneider. De novo design of bioactive small molecules by artificial intelligence. *Molecular informatics*, 37(1-2):1700153, 2018.
- [SPZ+20] Philippe Schwaller, Riccardo Petraglia, Valerio Zullo, Vishnu H Nair, Rico Andreas Haeuselmann, Riccardo Pisoni, Costas Bekas, Anna Iuliano, and Teodoro Laino. Predicting retrosynthetic pathways using transformer-based models and a hyper-graph exploration strategy. *Chemical Science*, 11(12):3316–3325, 2020.
- [SLG+19] Philippe Schwaller, Teodoro Laino, Théophile Gaudin, Peter Bolgar, Christopher A Hunter, Costas Bekas, and Alpha A Lee. Molecular transformer: a model for uncertainty-calibrated chemical reaction prediction. *ACS central science*, 5(9):1572–1583, 2019.
- [SVLR20] Philippe Schwaller, Alain C Vaucher, Teodoro Laino, and Jean-Louis Reymond. Prediction of chemical reaction yields using deep learning. *ChemRxiv Preprint*, 2020. URL: <https://doi.org/10.26434/chemrxiv.12758474.v2>.
- [VZG+20] Alain C Vaucher, Federico Zipoli, Joppe Geluykens, Vishnu H Nair, Philippe Schwaller, and Teodoro Laino. Automated extraction of chemical synthesis actions from experimental procedures. *Nature communications*, 11(1):1–11, 2020.
- [SPV+21] Philippe Schwaller, Daniel Probst, Alain C Vaucher, Vishnu H Nair, David Kreutter, Teodoro Laino, and Jean-Louis Reymond. Mapping the space of chemical reactions using attention-based neural networks. *Nature Machine Intelligence*, pages 1–9, 2021.
- [KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- [MRST19] Emile Mathieu, Tom Rainforth, N Siddharth, and Yee Whye Teh. Disentangling disentanglement in variational autoencoders. In *International Conference on Machine Learning*, 4402–4412. PMLR, 2019.
- [WNoeC21] Robin Winter, Frank Noé, and Djork-Arné Clevert. Auto-encoding molecular conformations. *arXiv preprint arXiv:2101.01618*, 2021.
- [SMS+20] Kirill Shmilovich, Rachael A Mansbach, Hythem Sidky, Olivia E Dunne, Sayak Subhra Panda, John D Tovar, and Andrew L Ferguson. Discovery of self-assembling  $\pi$ -conjugated peptides by active learning-directed coarse-grained molecular simulation. *The Journal of Physical Chemistry B*, 124(19):3873–3891, 2020.
- [WGomezB19] Wujie Wang and Rafael Gómez-Bombarelli. Coarse-graining auto-encoders for molecular dynamics. *npj Computational Materials*, 5(1):1–9, 2019.
- [RBWT18] João Marcelo Lamim Ribeiro, Pablo Bravo, Yihang Wang, and Pratyush Tiwary. Reweighted autoencoded variational bayes for enhanced sampling (rave). *The Journal of chemical physics*, 149(7):072301, 2018.

- [KPB20] Ivan Kobyzev, Simon Prince, and Marcus Brubaker. Normalizing flows: an introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.
- [PNR+19] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*, 2019.
- [PSM19] George Papamakarios, David Sterratt, and Iain Murray. Sequential neural likelihood: fast likelihood-free inference with autoregressive flows. In *The 22nd International Conference on Artificial Intelligence and Statistics*, 837–848. PMLR, 2019.
- [HNJ+21] Emiel Hoogetboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows: learning categorical distributions with normalizing flows. In *Third Symposium on Advances in Approximate Bayesian Inference*. 2021. URL: <https://openreview.net/forum?id=fdSxhAy5Cp>.
- [PPM17] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, 2338–2347. 2017.
- [KSJ+16] Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, 4743–4751. 2016.
- [KLS+18] Sungwon Kim, Sang-gil Lee, Jongyoon Song, Jaehyeon Kim, and Sungroh Yoon. Flowavenet: a generative flow for raw audio. *arXiv preprint arXiv:1811.02155*, 2018.
- [DSDB16] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.
- [DAS19] Hari Prasanna Das, Pieter Abbeel, and Costas J Spanos. Dimensionality reduction flows. *arXiv preprint arXiv:1908.01686*, 2019.
- [WKohlerNoe20] Hao Wu, Jonas Köhler, and Frank Noé. Stochastic normalizing flows. *arXiv preprint arXiv:2002.06707*, 2020.
- [BHB+18] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, and others. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
- [RDRVL14] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole Von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1(1):1–7, 2014.
- [SI15] Teague Sterling and John J Irwin. Zinc 15–ligand discovery for everyone. *Journal of chemical information and modeling*, 55(11):2324–2337, 2015.
- [KHaseN+20] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alan Aspuru-Guzik. Self-referencing embedded strings (selfies): a 100% robust molecular string representation. *Machine Learning: Science and Technology*, 1(4):045024, 2020.
- [LJD+18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL: <http://jmlr.org/papers/v18/16-558.html>.
- [KDVK21] Alireza Khodamoradi, Kristof Denolf, Kees Vissers, and Ryan C Kastner. Aslr: an adaptive scheduler for learning rate. In *2021 International Joint Conference on Neural Networks (IJCNN)*, 1–8. IEEE, 2021.
- [YLSZ20] Haixu Yang, Jihong Liu, Hongwei Sun, and Henggui Zhang. Pacl: piecewise arc cotangent decay learning rate for deep neural network training. *IEEE Access*, 8:112805–112813, 2020.
- [VMKS21] D Vidyabharathi, V Mohanraj, J Senthil Kumar, and Y Suresh. Achieving generalization of deep learning models in a quick way by adapting t-htr learning rate scheduler. *Personal and Ubiquitous Computing*, pages 1–19, 2021.

- [GBC17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning (adaptive computation and machine learning series). *Cambridge Massachusetts*, pages 321–359, 2017.
- [KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.
- [HHS17] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in neural information processing systems*, 2017.
- [LKSJ20] Tao Lin, Lingjing Kong, Sebastian Stich, and Martin Jaggi. Extrapolation for large-batch training in deep learning. In *International Conference on Machine Learning*, 6094–6104. PMLR, 2020.
- [SKYL17] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.
- [RM99] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.
- [GSM+17] Daniel Golovin, Benjamin Solnik, Subhdeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: a service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 1487–1495. 2017.
- [DSH15] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*. 2015.
- [JT16] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*, 240–248. PMLR, 2016.
- [TH20] P. Brendan Timmons and Chandralal M. Hewage. Happenn is a novel tool for hemolytic activity prediction for therapeutic peptides which employs neural networks. *Scientific reports*, 10(1):1–18, 2020.
- [PAG+21] Malak Pirtskhalava, Anthony A Amstrong, Maia Grigolava, Mindia Chubinidze, Evgenia Alimbarashvili, Boris Vishnepolsky, Andrei Gabrielian, Alex Rosenthal, Darrell E Hurt, and Michael Tartakovsky. Dbaasp v3: database of antimicrobial/cytotoxic activity and structure of peptides as a resource for development of new therapeutics. *Nucleic acids research*, 49(D1):D288–D297, 2021.
- [LLN+18] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: a research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.
- [ASY+19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: a next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.
- [BYC13] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, 115–123. PMLR, 2013.
- [Lou17] Gilles Louppe. Bayesian optimisation with scikit-optimize. In *PyData Amsterdam*. 2017.