SANJEEV ARORA, SIMON PARK, DENNIS JACOB, DANQI CHEN

INTRODUCTION TO MACHINE LEARNING

LECTURE NOTES FOR COS 324 AT PRINCETON UNIVERSITY

PUBLISHED BY

TUFTE-LATEX.GOOGLECODE.COM

September 2022

Contents

Preface

Introduction

These lecture notes accompany a junior-level machine learning course (COS 324) at Princeton University. This course provides a broad introduction to machine learning paradigms including supervised, unsupervised, deep learning, and reinforcement learning as a foundation for further study or independent work in ML, AI, and data science. Topics include linear models for classification and regression, clustering, low rank representations (PCA), n-gram language models, matrix factorization, feedforward neural nets and convolutional neural nets, Markov decision process, and reinforcement learning. Interesting applications are presented for all these models.

The course design was shaped by some constraints that may not exist at other universities.

Background assumed: The formal prerequisites are the following courses, which all our majors have taken through sophomore year: An introduction to computer science course (COS 126), *Data Structures and Algorithms* (COS 226), *Single-variable Calculus* (MAT 103, 104) and *Linear Algebra* (MAT 202/204/217). While many majors also take multi-variable calculus and a probability course, not all do. Hence we do not include them in the list of prerequisites, although we do assume some exposure to elementary probability at high-school level. All our majors take a course on proof-based reasoning (COS 240) but many don't take it before junior year. Hence that course is not a prerequisite and our course doesn't rely on formal proofs *per se*.

A side benefit of assuming minimal math and programming prerequisites is that this also makes our course accessible to hundreds of non-majors, many of whom only take introductory CS courses.

Should provide a broad introduction to today's AI and machine learning: Since AI and Machine Learning has been transformed by deep learning and related methods in the past decade, we wanted to provide a reasonable competence in deep learning as well as

reinforcement learning. Thus students who don't take another AI/ML course still leave with a skill set appropriate for applying ML techniques in their careers. Of course, devoting the second half of our course to these "advanced" topics required leaving out several topics that are classically taught in introductory ML. Essentially, the classical topics are condensed into the first half of the term.

Provide a taste of today's programming environments: Students can start with zero python background, and slowly graduate to comfort in python as well as dipping their toes into deep learning on Google CoLab and RL on OpenAI gym.

Taste of interesting applications + discussion of broader issues of societal interest. An introductory course needs to introduce students to these issues, given machine learning's ubiquitous role in research across disciplines, as well as throughout our economy and our society. Sometimes we invite guest lecturers to provide such perspectives.

Finally, note that all the above material has to fit in Princeton's 12-week term. We do not have 13-15 weeks as at other universities.

Structure of the Notes

These notes are divided into six main parts.

Part I introduces supervised learning. Topics include linear regression, linear classification, and gradient descent.

Part II is about unsupervised learning. Topics include clustering, dimensionality reduction, n-gram language models, and matrix factorization.

Part III covers the basics of deep learning. Topics include feedforward neural networks and convolutional neural networks.

Part IV presents reinforcement learning. Topics include Markov decision process and Q-learning.

Part V introduces some advanced applications of machine learning. Topics include ethics of machine learning and deep learning for natural language processing.

Part VI provides some mathematical background that is useful for machine learning. Topics include probability, calculus, and linear algebra.

Basic Ingredients of Machine Learning

Machine learning is the discipline of creating decision-making programs that improve themselves automatically, based on data or repeated experience. The entire setup consists of the following elements.

Data: We have a set of *data* to learn from. The data may have an additional field called a *label*. If the data is labeled, the goal of the learning is to predict the label of newly seen data. Such learning is called *supervised learning*; examples include regression (Chapter 1, Chapter 5) or classification (Chapter 4). If the data is unlabeled, the goal of the learning is to extract the structure of the current data. Such learning is called *unsupervised learning*; examples include clustering (Chapter 6) or dimensionality reduction (Chapter 7).

Model: We have a *model* that we want to learn. A model is a mapping from a data point to a desired answer or output. For supervised learning, the answer will be the *label* of the data; for unsupervised learning, the output will be the structure of the data.

Model parameters: Each model is defined by a number of constants, or internal *parameters*. The goal of the learning is to find the values of these parameters that yield the best model. Throughout the training process, the values of the parameters will be updated or reset.

Model fitting: The process of finding the best (or good enough) values of model parameters, based on the provided data, is called *fitting* the model. There are many different ways to assess how "good" a model is. In many cases, a *loss function* is defined to measure how far the predictions of the model are from the actual output. In Chapter 4, we see how different loss functions are defined for different models.

Testing: In many cases, especially in supervised learning, we want to predict how "good" the model will be for a newly seen data. This process is called *testing* the model. For this purpose, it is customary to use only a fraction (*e.g.*, 80%) of the data to fit the model, and use the rest (*e.g.*, 20%) to test the model. The portion of the data that is used for testing and not for fitting the model is called the *held-out* data and is assumed to be a good sample of the population data.

Part I

Supervised Learning

1
Linear Regression: An Introduction

This chapter introduces *least squares linear regression*, one of the simplest and and most popular model in data science. Several of you may have seen it in high school. In particular, we focus on understanding linear regression in the context of machine learning. Using linear regression as an example, we will introduce the terminologies and ideas (some of them were mentioned in the Preface) that are widely applicable to more complicated models in ML.

1.1 A Warm-up Example

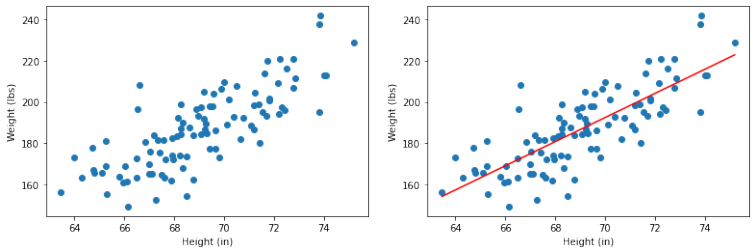


Figure 1.1: A dataset of heights and weights of some male adults. The figure on the right shows the least squares regression line that fits the data. Data from https://gist.github.com/nstokoe/7d4717e96c21b8ad04ec91f361b000cb

Suppose we have a dataset of heights and weights of some male individuals randomly chosen from the population. We wish to determine a relationship between heights and weights. The simplest relationship would be a linear relationship; namely:

$$w = a_0 + a_1 h \tag{1.1}$$

where w is the weight, h is the height, and a_0, a_1 are constant coefficients. We can think of this as a *predictor* that maps height h to a predicted weight $a_0 + a_1 h$, and we want this value to be similar to the actual weight w. Obviously, a linear relationship won't describe the data exactly but we hope it is a reasonable fit to data. [1] In a ML setting, this relationship between h and w is called a *model* — a linear model to be more specific.

[1] Similar linear models are used in many disciplines. For instance, the famous Philips model in economics suggests a linear relationship between inflation rate and unemployment rate, at least when inflation rate is high.

Based on the values of a_0 and a_1, there are infinitely many different choices of this linear model. Therefore, it is natural that we want to find the values of a_0, a_1 that yield the "best" model. In a ML setting, finding these optimal values of a_0, a_1 is known as *fitting* the model. One can posit different criteria for defining "goodness" of the model.

Here we use classic *least squares fit*, invented by Gauss. Given a dataset $\{(h_1, w_1), (h_2, w_2), \ldots, (h_n, w_n)\}$ of n pairs of heights and weights, the "goodness" of the model in (1.1) is

$$\frac{1}{n}\sum_{i=1}^{n}(w_i - a_0 - a_1 h_i)^2 \tag{1.2}$$

Notice that $w_i - a_0 - a_1 h_i$ is the difference between the actual weight w_i and the predicted weight $a_0 + a_1 h_i$. This difference is called the *residual* for the data point (h_i, w_i), and the full term in (1.2) is called the *average squared residuals*, or equivalently the *mean squared error* (MSE), of the dataset. The smaller the MSE, the closer the model's predictions are to actual weights, and the more accurate the model is. Therefore, the "best" model according to the least squares method would be the one defined by the values of a_0, a_1 that minimize (1.2). In a ML setting, a mathematical expression like (1.2) that captures the "goodness" of the model is called a *loss function*. In general, we find the "best" model by minimizing the loss function.

Example 1.1.1. *If the data points (h, w) are given as $\{(65, 130), (58, 120), (73, 160)\}$, the least squares fit will find the values of a_0, a_1 that minimize*

$$\frac{1}{3}((130 - a_0 - 65a_1)^2 + (120 - a_0 - 58a_1)^2 + (160 - a_0 - 73a_1)^2)$$

which are $a_0 = -\frac{510}{13}$, $a_1 = \frac{35}{13}$.

Problem 1.1.2. *Between the two lines in Figure 1.2, which is more preferred by the least squares regression method?*

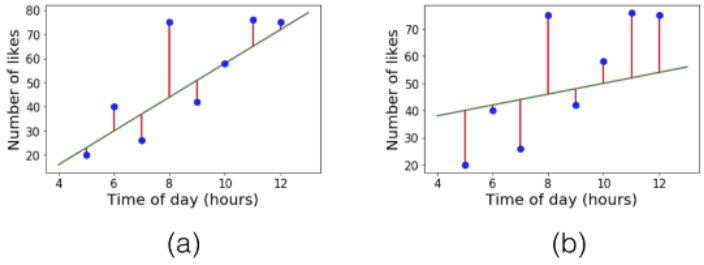


Figure 1.2: Two lines that describe the relationship of the same dataset.

Problem 1.1.3. *Using calculus, give an exact expression for a_0, a_1 that minimize (1.2). (Hint: (1.2) is quadratic in both a_0 and a_1. Fix the value of*

a_1 and minimize for a_0. Then minimize for a_1. Completing the square may be useful.) [2]

[2] A more general calculus based approach will be introduced in a later chapter.

1.1.1 Multivariate Linear Regression

One can generalize the above example to multi-variable settings. In general, we have k *predictor variables* and one *effect* variable. [3] The data points consist of $k+1$ coordinates, where the last coordinate is the value y of the effect variable and the first k coordinates contain values of the predictor variables $x_1, x_2, \ldots, x_k$. Then the relationship we are trying to fit has the form

[3] In the above example $k = 1$. The predictor variable was height and effect variable was weight.

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_k x_k \tag{1.3}$$

and the least squares fit method will find the values of $a_0, a_1, \cdots, a_k$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - a_0 - a_1 x_1^i - a_2 x_2^i - \cdots - a_k x_k^i)^2 \tag{1.4}$$

where $(x_1^i, x_2^i, \ldots x_k^i, y^i)$ is the i-th data point.

We can simplify the notation by rewriting everything above in a vectorized notation. If we set $\vec{\mathbf{x}} = (1, x_1, x_2, \cdots, x_k)$ [4] and $\vec{\mathbf{a}} = (a_0, a_1, \cdots, a_k)$, then the relationship we are trying to fit has the form

[4] The 1 in the first coordinate is a dummy variable to naturally include the constant term into the vectorized notation.

$$y = \vec{\mathbf{a}} \cdot \vec{\mathbf{x}} \tag{1.5}$$

and the least squares fit method will find $\vec{\mathbf{a}} \in \mathbb{R}^{k+1}$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - \vec{\mathbf{a}} \cdot \vec{\mathbf{x}}^i)^2 \tag{1.6}$$

where $(\vec{\mathbf{x}}^i, y^i)$ is the i-th data point. We discuss how to find the best values of $a_0, a_1, \ldots, a_k$ later in Chapter 3; for now just assume that the solution can be found.

1.1.2 Testing a model (Held-out Data)

A crucial step in machine learning is to *test* the trained/fitted model on *newly seen data*, or *held-out data*, that was not used during training. If we were to test the model in the above example, we would hold out a portion of the data points (say 20%) — *i.e.*, not use them during training — and check the average squared residual of the model on the held-out data points.

We can think of the average squared residual of the held-out data as an *estimate* of the average squared residual of the fitted model on the entire population of male adults. [5] The reason is that if the training data points were a random sample of the adult male population,

[5] If later in life you ever write up the results of a regression study, be sure to report the *RMSE error*, which is the square root of the average square residual on held-out data. Also report the R^2 value, which is closely related.

then so is the set of held-out data points. This is quite analogous to opinion polls, where the opinions of a few thousand randomly sampled individuals can be a reasonable estimate for the average opinion across the US. The math for such sampling estimates is covered in Chapter 18.

1.1.3 *More about Linear Regression*

In the above example, we used the least squares method, which uses the average squared residual to assess the model. The least squares fit is very common but other notions of fit may also be used. For instance, instead of taking the sum of squares of residuals, one could consider the sum of absolute values, or expressions using logarithms, etc. We see some of these examples in Chapter 4.

It is important to note that the relationship learnt via regression — and machine learning in general — is (a) approximate and (b) only holds for the population that the data was drawn from. Therefore, we cannot use the relationship to predict the output of a data that is not from the same distribution. Additionally, if the distribution of the data is shifted, the relationship no longer holds. We will discuss more about this in depth in Chapter 2.

1.2 *Using Linear Regression for Sentiment Prediction*

1.2.1 *Introduction*

While you might have seen linear regression as early as in high school, you probably did not see this cool application. In *sentiment classification*, we are given a piece of text and have to label it with $+1$ if it expresses positive sentiment and -1 otherwise.

Example 1.2.1. *Consider the following dataset, collected by showing snippets of text to humans and asking them to label them as positive ($+1$) or negative (-1)*

The film's performances are thrilling.	$+1$
It's not a great monster movie.	-1
It is definitely worth seeing.	$+1$
Unflinchingly bleak and desperate.	-1

Table 1.1: Data from Stanford Sentiment Treebank (SST). https://nlp.stanford.edu/sentiment/treebank.html

How can we train a model to label text snippets with the correct sentiment value, given a dataset of training examples? Here is an idea to try to solve it using a linear regression model. We first enumerate all English words and assign a real-valued score to each word, where the score for the i-th word is denoted by w_i. These scores will

be the *parameters* of the model. The output of the model, given a training example, is defined as $\sum_{j \in S} w_j$ where S is the multiset of words in the text. [6] Then the least squares method needs to solve the following optimization problem for a dataset of (text, sentiment) pairs

$$\text{minimize} \sum_i \left(y^i - \sum_{j \in S^i} w_j \right)^2 \tag{1.7}$$

where S^i is the multiset of words in the i-th piece of text. Each of the values $\left(y^i - \sum_{j \in S^i} w_j \right)^2$ is called a *least squares error* or more generally the *loss* for that particular training example. The full summation is called a *training loss* of the dataset.

[6] Unlike in a set, an element can appear multiple times in a multiset. For example, if the word *good* appears twice in a text, then S contains two copies of *good*.

Example 1.2.2. *Assume we are training a sentiment prediction model on a dataset. Table 1.1 shows some of the model parameter values. Then the output of the model on the sentence "I like this movie" from the training data will be* $0.15 + 0.55 + 0.03 - 0.07 = 0.66$. *The output for "I dislike this movie" from the training data will be* $0.15 - 0.74 + 0.03 - 0.07 = -0.63$

i	word	w_i
1	I	0.15
2	like	0.55
3	dislike	−0.74
4	this	0.03
5	movie	−0.07
6	a	0

Table 1.2: Some of the parameter values of a sentiment prediction model.

We can also cast this in the standard formulation of linear regression as follows. The *bag of words (BoW)* representation of a piece of text is a vector in $\mathbb{R}^N$ where N is the number of dictionary words. The i-th coordinate is the number of times the i-th word appears in the piece of text. This represents the text as a very long vector, one coordinate per one English word in the dictionary. The vector usually contains a lot of zeros, since most words probably do not appear in this piece of text. If we denote the BoW vector as $\vec{\mathbf{x}}$, the output of the model is seen to be

$$\sum_{j \in S} w_j = \sum_i w_i x_i$$

which shows that the linear model we have proposed for sentiment prediction is just a subcase of linear regression (see (1.5)).

Example 1.2.3. *Consider the same model in Example 1.2.2. The BoW representation for the sentence "I like this movie" is* $(1, 1, 0, 1, 1, 0 \cdots)$. *The BoW representation for the sentence "I dislike this movie" is* $(1, 0, 1, 1, 1, 0 \cdots)$.

1.2.2 *Testing the Model*

Here we use the model from Example 1.2.2 to illustrate the training and testing process of a model. Assume that the following four sentences were a part of the training dataset.

I like this movie.	$+1$
I dislike this movie.	-1
I like this.	$+1$
I dislike this.	-1

Table 1.3: A portion of the training data for a sentiment prediction model.

Assuming that the model parameters are the same as reported in Table 1.2, we can calculate the training loss of the sentence "I like this movie" as $(+1 - 0.66)^2 \simeq 0.12$. Similarly, the squared residual for each of the four training sentences in Table 1.3 can be calculated as

I like this movie.	0.12
I dislike this movie.	0.14
I like this.	0.07
I dislike this.	0.19

Table 1.4: The squared residual for four training examples.

Now it is time to test the model. Assume that the sentence "I like a movie" is provided to the model as a test data. The *test loss* can be calculated in a way similar to the training loss as $(+1 - 0.63)^2 \simeq 0.14$. But to actually test if the model produces the correct sentiment label for this newly seen data, we now wish the model to output either $+1$ or -1, the only two labels that exist in the population. An easy fix is to change the output of the model at test time to be $\text{sign}(\sum_{j \in S} w_j)$. For this test data, the model will output $\text{sign}(0.63) = +1$.

On the Stanford Standard Treebank, this approach of training a least squares model yields a success rate of 78% [7]. By contrast, the state-of-the-art deep learning methods yield success rate exceeding 96%!

[7] To be more exact, this result is from a model called *ridge regression* model, which is linear regression model augmented by an ℓ_2 regularizer, which will be explained in Chapter 3

One thing to note is that while the training loss is calculated and explicitly used in the training process, the test loss is only a statistics that is generated after the training is over. It is a metric to assess if the model fitted on the training data also performs well for a more general data.

1.2.3 *Test Loss, Generalization, and Test accuracy*

As mentioned already, the goal of training a model is that it should make good predictions on new, previously-unseen data. Most models will exhibit a low training loss, but not all of them show a low test loss. This observation motivates the following definition:

$$\boxed{\text{Generalization Error} = |\text{training loss} - \text{test loss}|}$$

A trained model is said to *generalize well* if the generalization error is small. In our case, the loss is the average squared residual. Thus good generalization means that the average squared residual on test data points is similar to that on the training data.

Let us see what happens on our sentiment model when it is fitted and tested on the SST dataset.

Train MSE	0.0727
Test MSE	0.7523
Training accuracy	99.55%
Test accuracy	78.09%

Table 1.5: *Accuracy* refers to the classification accuracy when we make the model to output only ± 1 labels.

Example 1.2.4. *The **generalization error** above is the difference between MSE on test points and the MSE on training points, namely* $0.75 - 0.07 = 0.68$.

Let's try to understand the relationship between low test loss (the squared residual) and high test accuracy (for what fraction of test data points the sentiment was correct). Heuristically, the test loss (average squared residual) being 0.75 means that the the absolute value of the residual on a typical point is $\sqrt{0.75} \approx 0.87$. This means that for a data point with an actual positive sentiment (*i.e.*, label $+1$), the output of the model is roughly expected to lie in the interval $[1 - 0.87, 1 + 0.87]$, and similarly, for a data point with an actual negative sentiment (*i.e.*, label -1), the output of the model is roughly expected to lie in the interval $[-1 - 0.87, -1 + 0.87]$. Once we take the sign $\text{sign}(\sum_{j \in S} w_j)$ of the output of the model, the output is thus likely to be rounded off to the correct label. We also note that the training accuracy is almost 100%. This usually happens in settings where the number of parameters (*i.e.*, number of predictor variables) exceeds the number of training data points (or is close to it). The following problem explores this.

Problem 1.2.5. *An expert on TV claims to have a formula to predict outcome of presidential election. It uses* 31 *measurements of various economic and societal quantities (inflation, divorce rate, etc). The formula correctly predicts the winner of all elections* 1928-2020. *Should you believe the formula's prediction for the* 2024 *election? (Hint: Under fairly general conditions,* $T + 1$ *completely nonsense variables — i.e., having nothing to do with presidential politics — can be used to perfectly fit (via linear regression) the outcomes for* T *past presidential elections.* [8]*)*

[8] If a model does not generalize well, then it is said to *overfit* the training data.

1.2.4 Interpreting the Model

In many settings (*e.g.*, medicine), an important purpose of regression modeling is to understand the data or the phenomenon a bit

better. In this case, the phenomenon is "sentiment" and we are naturally curious about what positive or negative sentiment amounts to. Specifically, what caused the model's output to be $+1$ or -1 given a specific sentence?

Figure 1.3 shows a histogram of the values of w_i, the parameters of a sentiment prediction model that was trained on the Stanford Sentiment Treebank. Positive values of w_i imply that the words carry a positive sentiment, while negative values of w_i imply that the words carry a negative sentiment. Also, the greater the absolute value of w_i is, the stronger the sentiment. Notice that most words have a value of w_i close to zero, meaning the model views most words as neutral. The model "pays attention" to only a tiny set of words.

Words with high positive w_i values (*i.e.*, positive words) include *enjoyable*, *fun*, and *remarkable*. Words with high negative values (*i.e.*, negative words) include *suffers*, *dull*, and *worst*. Words with w_i values close to 0 (*i.e.*, neutral words) include *duty* and *desire*.

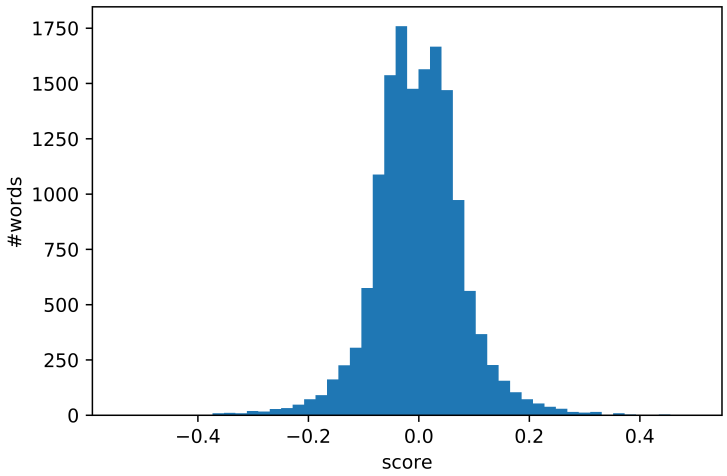


Figure 1.3: A histogram of the learned parameters w_i of a sentiment prediction model trained on the Stanford Sentiment Treebank.

1.3 *Importance of Featurization*

In the sentiment model, we chose a particular method to represent a piece of text with a vector. The coordinates of this vector are often referred to as *features* and this process of converting data into vectors is called *featurization*. One can conceive of other choices for featurizing text. For example, *bigram* featurization consists of the following: the coordinates of the vector correspond to *pairs* of words and the coordinate contains the number of times this pair of words appeared consecutively in the piece of text. In contrast, the choice of featurization from the earlier example matches each coordinate with a single word, and is called a *unigram* featuraization.

Bigram features allow the model to access information about phrases that were present in the text. For instance, in isolation "pretty" is a positive word and "bad" is a negative word. If they both occur in text one would imagine that they cancel each other out as far as overall sentiment is concerned. But the phrase "pretty bad" is more negative than "bad." Thus bigram features can improve the model's ability to capture sentiment.

The required number of dimension for bigram representations can get rather large. If the number of words is N, then the number of coordinates is $N(N-1)$. Realize that the number of model parameters in linear regression is the same as the number of coordinates. Thus if N is $30,000$ then the number of coordinates in bigram feature vector (and hence number of model parameters) is close to a billion, which a rather large number. In practice one might throw away information for all pairs except say the $10,000$ most common ones in the dataset. Usually models that incorporate bigram features do better than unigram-only models.

If one is trying to do studies of medical treatment with regression, there can be many potential featurizations of patient data. Doctors' annotations, test results, x-ray scans, etc. all have to be converted somehow into real-valued features, and the experimenter uses their prior knowledge and intuitions while featurizing the data.

Example 1.3.1. *Patients' raw data might include height and weight. If we use linear regression, the effect variable can only depend upon a linear combination of height and weight. But it is known that several health outcomes are better modeled using* Body Mass Index*, defined as weight/height*^{2}*. Thus the experimenter may include a separate coordinate for BMI, even though it duplicates information already present in the other coordinates.*

Example 1.3.2. *In the above dataset, the weight in pounds may span a range of* $[90, 350]$*, whereas cholesterol ratio may span a range of* $[1, 10]$*. It is often a good idea to* normalize *the coordinate, which means to replace* x *with* $(x-\mu)/\sigma$ *where* μ *is the mean of the coordinate values in the dataset and* σ *is the standard deviation.*

Thus the same raw dataset can have multiple featurizations, with different number of coordinates. Problem 1.2.5 may make us wary of using featurizations with too many coordinates. We will learn a technique called *regularization* in Chapter 3, which helps mitigate the issue identified in Problem 1.2.5.

1.4 *Linear Regression in Programming*

In this section, we briefly discuss how to write the Python code to perform linear regression (*e.g.*, sentiment prediction). Python is often the language of choice for many machine learning applications due to its relative ease of use and the large variety of external packages available to automate the process. Here, we introduce a few of these packages:

- *numpy*: This package is ubiquitous throughout the machine learning community. It provides access to specialized array data structures which are implemented in highly optimized C code. Linear algebra computations and array restructuring operations are significantly faster with *numpy* compared to using Python directly. [9]

- *matplotlib*: This package enables Python programmers to create high quality plots and graphs. Visualizations are highly configurable and interoperable with several other Python packages. [10]

- *sklearn*: This package provides a potpourri of machine learning and data science models through an easy to use object-oriented API. In addition to linear regression, *sklearn* makes it possible to implement SVMs, clustering, neural networks, and much more; you will learn about a some of these models later in the course. [11]

[9] Documentation is available at https://numpy.org/

[10] Documentation is available at https://matplotlib.org/

[11] Documentation is available at https://scikit-learn.org/stable/index.html

Throughout this course, you will be asked to make use of functions defined in some of these external packages. You may not always be familiar with the usage of these functions. It is important to check the official documentation to learn about the usage and the signature of the functions.

The code snippet below uses these the three aforementioned packages to perform linear regression on any given dataset.

```
# import necessary packages
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt

# prepare train, test data
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# perform linear regression on train data
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = np.sign(linreg.predict(X_test))
```

```
# print train results
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
print('Test Acc: ', '{0:.2f}'.format(100*(pred_test==y_test).mean()))

# plot gold vs predicted value
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

For readers who are not familiar with Python, we discuss some key details. In the first section of the code, we import the relevant packages

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt
```

As seen in this example, there are two ways to load a package. The first option is to import the full package with the *import* keyword

```
import numpy as np
```

Notice that the we can assign the imported package a customized name with the *as* keyword. In this case, we decided refer to the package *numpy* with the name *np* throughout the rest of the code. This is indeed the case when we call

```
np.sign()
```

Here we refer to the method *sign()* of the *numpy* package with the customized name *np*. Alternatively, we can selectively import particular methods or classes with the *from* keyword

```
from sklearn.model_selection import train_test_split
```

The next part of the code is preparing the train, test data.

```
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

X will have to be an array of arrays, and y will have to be an array of values, with the same length as X. These arrays can be defined directly by specifying each of their entries, or they could be read from some external data (most commonly a csv file). Here, we present an example dataset where $\vec{\mathbf{x}} \in \mathbb{R}^2$:

```
X = [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
y = [1, 1, 1, -1, -1]
```

Then we call the *train_test_split()* method to split the dataset into data for model training and testing. Alternatively, we can split the dataset by manually slicing the data arrays. [12] In general, slicing a Python array involves the *:* operator along with start and end indices. For instance, consider an arbitrary array *a*. Then, the output of *a[i:j]* will be a subarray of *a* from the index *i* (inclusive) to the index *j* (exclusive). In the following code sample, we slice the data by specifying the number of training data points

[12] In Python, the term *slicing* refers to the process of creating a subarray of an array.

```
train_size = ...
X_train = X[:train_size]
X_test = X[train_size:]
y_train = y[:train_size]
y_test = y[train_size:]
```

Note that we have omitted some of the bounding indices. If the start index is omitted, Python assumes it to be 0 (so that the subarray is from the start of the array); for example, *X[:train_size]* is the first *train_size* entries of *X*. If the end index is omitted, Python assumes it to be *n*, the length of the array (so that the subarray ends at the end of the array); for instance, *X[train_size:]* is the remaining entries of *X*, once we remove the first *train_size* entries. Another way to slice the arrays is by specifying the number of test data points

```
test_size = ...
X_train = X[:-test_size]
X_test = X[-test_size:]
y_train = y[:-test_size]
y_test = y[-test_size:]
```

Here, notice that the index *-test_size* is a negative number. In this case, Python interprets this as *n - test_size*, where *n* is the size of the array. In other words, it is the index of the *test_size*-th element from the back of the array.

The third part of the code is fitting the linear regression model.

```
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = np.sign(linreg.predict(X_test))
```

The first line will generate the least squares fit model based on the train data. Then we can have the model make predictions on the train, test data. Notice that we changed the output of the model to be the sign of the predicted values, so that we can compare them with the gold values.

Next, we print out the mean squared loss and the accuracy for the train, test data.

```
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
```

```
print('Test Acc: ', '{0:.2f}'.format(100*(pred_test==y_test).mean()))
```

Notice that we use the *mse()* method that we imported from the *sklearn* package. In many cases, there are packages that perform these elementary operations for machine learning.

Finally, we plot the actual and predicted values using the *matplotlib* package.

```
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

The first line draws a scatter plot with the *y_test* in the x-axis and *pred_test* in the y-axis. Notice that you can specify the color of the data points by specifying the value of the parameter *c*. In general, *parameters* are optional values you can provide to Python functions. If the values to parameters are omitted, the function will use their default values. The second and third lines specify the labels that will be written next to the axes. The final line specifies the title of the plot.

2
Statistical Learning: What It Means to Learn

Students often get confused about the meaning and significance of a relationship learnt via fitting a model to data. Some of them think such relationships are analogous to, say, a law of nature like $F = ma$, which applies every time force is applied to a mass anywhere in the universe. The main goal of this chapter is to explain the statistical nature of machine learning — models are fitted on a *particular* distribution of data points, and its predictions are valid only for data points from the same distribution. (See Chapter 18.)

2.1 *A Warm-up Example*

We work through a concrete example [1] before enunciating the general properties of statistical learning. Suppose we are studying the relationship between the following quantities for the population of Princeton: *height (H), number of exercise hours per week (E), amount of calories consumed per week (C), and weight (W)*. After collecting information from 200 randomly sampled residents, and using a $80:20$ train/test split, we perform a linear regression on the training dataset to come up with the following relationship:

$$W = 50 + H + 0.1C - 4E \tag{2.1}$$

Let's also say that the average squared residual on train and test data were both 100. This means that the relationship (2.1) holds with an error of 10 lbs on a typical test data point. [2]

Question 2.1.1. *Alice was one of the Princeton residents in the study, but the prediction of the model is very off of her actual value (squared residual is 300). Does this prove the model wrong?*

The answer is no. The least squares linear regression finds the model that minimizes the *average* squared residual across all training data points. The residual could be large for a particular individual.

[1] This example is purely hypothetical, and all numbers in this section are made up.

[2] Also, the trained model exhibits *perfect* generalization: test loss is the same as training loss!

Question 2.1.2. *There was a follow-up research for every Princeton resident who is taller than 7 feet. All of them reported squared residual of* 500*. Does this prove the model wrong?*

The answer is still no. People who are taller than 7 feet make up a tiny fraction of the entire population. Their residuals have very small effect on the *average* squared residual. The residual could be large for a small subset of the population.

Question 2.1.3. *There was a follow-up survey that tested the model on every single Princeton resident. Is it possible that the average squared residue is* 200 *for the entire population?*

The answer is yes, although it is unlikely. Consider the distribution of 4-tuples (H, E, C, W) over the entire Princeton population. This is some distribution over a finite set of 4-dimensional vectors. [3] The 200 residents we surveyed were randomly drawn from this distribution. Out of these 200 data points, 40 were randomly chosen to be held-out as test data, while the remaining 160 were used as training data. We can also say that these 40 data points were chosen at random from the distribution over the entire population of Princeton. Thus when we test the model in (2.1) on held-out data, we're testing this relationship over a random sample of 40 data points drawn from the population. 40 is a large enough number to give us some confidence that the average squared residual of the test data is a good estimate of the squared residual in the population, but just as polling errors happen during elections, there is some chance that this estimate is off. In this case, we would say that the 40 test samples were *unrepresentative* of the full population.

[3] $31,000$ vectors to be more exact. The population of Princeton is $31,000$.

It is important to remember that the training and test data are sampled from the same distribution as the population. Therefore, the average squared residual of the training and test data are only a good estimate of the squared residual of the distribution they were sampled from. This also means that the relationship found from the training data only holds (with small residue) for that *particular* distribution. If the population is different, or if the distribution shifts within the same population, the relationship is not guaranteed to hold. For example, the relationship in (2.1) is not expected to hold for people from Timbuktu, Mali (a different population), or for residents of Princeton who are taller than 7 feet (a tiny subpopulation that is likely unrepresentative of the population). Now consider the following situation:

Question 2.1.4. *It becomes fashionable in Princeton to try to gain weight. Based on the relationship in* (2.1)*, everyone decides to increase their value of C and reduce their value of E. Does the model predict that many of them will gain weight?*

The answer is no. The model was fitted to and tested on the distribution obtained before everyone tried to gain weight. It has not been fitted on the distribution of data points from people who changed their values of C and E. In particular, note that if everyone reduces their E and increases their C, then the distribution has definitely changed — the average value of the E coordinate in this distribution has decreased, whereas the average value of the C coordinate has increased.

In general, a relationship learned from a fitted model illustrates *correlation* and need not imply *causation*. The values of H, C, E in (2.1) do not *cause* W to take a specific value. The equation only shows that the values are connected via this linear relationship on average (with some bounded square residuals).

2.2 *Summary of Statistical Learning*

The above discussion leads us to summarize properties of Statistical Learning. Note that these apply to most methods of machine learning, not just linear regression.

Training/test data points are sampled from some distribution $\mathcal{D}$: In the above example, 200 residents were randomly sampled from the entire population of Princeton residents.

The learnt relationship holds only for the distribution $\mathcal{D}$ that the data was sampled from. The performance of the model on test data is an estimate of the performance of the model on the full distribution $\mathcal{D}$.

There is a small probability that the estimate using test data is off. This is analogous to polling errors in opinion polls. The computation of "confidence bounds" is discussed in Chapter 18.

2.3 *Implications for Applications of Machine Learning*

The above framework and its limitations have real-life implications.

1. *Results of medical studies may not apply to minority populations.* This can happen if the minority population is genetically distinct and constitutes only a small fraction of the population. Then test error could be large on the minority population even if it is small on average. In fact, there have been classic studies about heart disease in the 1960s whose conclusions and recommendations fail to apply well to even a group that is half of the population: females! In those days heart disease was thought to largely strike males (which subsequently turned out to be quite false) and so

the studies were done primarily on males. It turns out that heart diseases in female patients behave differently. Many practices that came out of those studies turned out to be harmful to female patients. [4]

2. *Classifiers released by tech companies in the recent past were found to have high error rates on certain minority populations.* It was quickly recognized that relying on test error alone can lead to adverse outcomes on subpopulations. [5]

3. *Creating interactive agents is difficult.* In an interactive setting (*e.g.*, an online game), a decision-making program is often called an *agent*. When an agent has to enter an extended number of interactions [6] with a human (or another agent designed by a different group of researchers, as happens in Robocup soccer [7]), then statistical learning requires that the agent to have been exposed to similar situations/interactions during training (*i.e.*, from a fixed distribution). It is quite unclear if this is true.

[4] See https://www.theatlantic.com/health/archive/2015/10/heart-disease-women/412495/.

[5] See https://time.com/5520558/artificial-intelligence-racial-gender-bias/.

[6] Later in the book we encounter Reinforcement Learning, which deals with such settings.

[7] See https://www.robocup.org/.

3
Optimization via Gradient Descent

This chapter discusses how to train model parameters through *optimization* techniques that help find the best (or fairly good) model that has low training loss. We assume that you have seen simple root-finding techniques in high school or in calculus. Optimization in machine learning often uses a procedure called *gradient descent*. This chapter assumes your knowledge of basis multivariable calculus. If you have not taken a course in multivariable calculus, read Chapter 19 to familiarize yourself with the basic definitions.

3.1 Gradient Descent

In general, a ML model has an associated loss function. The "best" model is the one that minimizes the training loss. In most cases, it is impossible or difficult to find the minimum analytically; instead, we use a numerical method called the *gradient descent algorithm* to find the (approximate) optimum.

3.1.1 Univariate Example

Let's start with an univariate example to motivate the topic. Let $f(w) = 4w^2 - 6w - 9$ be a quadratic function. Figure 3.1 shows the graph of this function.

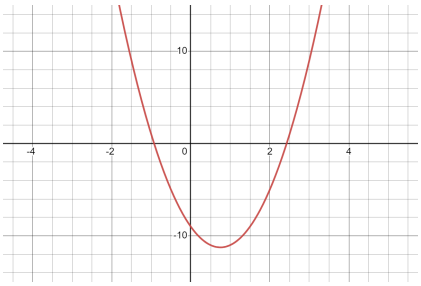


Figure 3.1: The graph of $f(w) = 4w^2 - 6w - 9$

Let's say that f attains its minimum at some point $w = w^*$. How

should we find the value of w^*? Here is an idea. Let's start from some random point on the curve and "walk down" the curve.

Notice from the graph that $f'(w^*) = 0$. Also, f is decreasing (*i.e.*, $f'(w) < 0$) when $w < w^*$ and increasing (*i.e.*, $f'(w) > 0$) when $w > w^*$. So if we examine a point w and find that $f'(w) = 0$, then we have arrived at our minimum. If $f'(w) > 0$, then we are currently on the right side of the minimum, so we need to *decrease* w. On the other hand, if $f'(w) < 0$, then we need to *increase* w.

For example, we start with the point $w = 0$. Since $f'(w) = -6 < 0$, we know that we are on the left side of the minimum, so we update $w \leftarrow 1$. Since $f'(w) = 2 > 0$, we are now on the right side of the minimum, so we update $w \leftarrow \frac{1}{2}$. When we iterate this process, we hope that we eventually slide down to the bottom of the curve. Observe that the change of value of w has the opposite sign from $f'(w)$ at that point. That is, for each step of this iteration, we can always find a $\eta > 0$ such that

$$w \leftarrow w - \eta f'(w)$$

This is not a mere coincidence — a similar result holds for a multivariate function.

3.1.2 *Gradient Descent (GD)*

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a multivariate function. If we want to "walk down" the curve of f as in the univariate case, we need to find a direction from the current point $\vec{\mathbf{w}}$ that *decreases* f.

A generalization of the Taylor expansion in the multivariable setting shows that the value of f in a small neighborhood around $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_d)$ can be approximated as a linear function in terms of the gradient.

$$f(\vec{\mathbf{w}} + \vec{\mathbf{h}}) \approx f(\vec{\mathbf{w}}) + \nabla f(\vec{\mathbf{w}}) \cdot \vec{\mathbf{h}}$$

where $\vec{\mathbf{h}} \in \mathbb{R}^d$ is small enough (*i.e.*, $\left\|\vec{\mathbf{h}}\right\| \approx 0$).

If ∇f is nonzero and we choose $\vec{\mathbf{h}} = -\eta \nabla f$ where η is a sufficiently small positive number, then

$$f(\vec{\mathbf{w}} - \eta \nabla f) \approx f(\vec{\mathbf{w}}) - \eta \|\nabla f\|_2^2$$

Since $\|\nabla f\|_2^2$ is positive, being the squared length of the vector ∇f, we conclude that the update $\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f$ causes a decrease in value of f. [1] This discussion motivates the *gradient descent algorithm*, which iteratively decreases the value of f until $\nabla f = 0$.

[1] In fact, the gradient ∇f is known as the direction of steepest increase of f. Hence, the opposite direction $-\nabla f$ is the direction of steepest decrease of f.

Definition 3.1.1 (Gradient Descent)**.** *Gradient descent is an iterative algorithm that updates the weight vector* $\vec{\mathbf{w}}$ *with the following rule:*

$$\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f(\vec{\mathbf{w}}) \tag{3.1}$$

where $\eta > 0$ is a sufficiently small positive constant, called the ***learning rate*** *or* ***step size****.*

We illustrate with an example.

Example 3.1.2. *Let $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. From Figure 3.2, we see that it attains a global minimum at $(0, 0)$. The partial derivatives of f can be calculated as:*

$$\frac{\partial f}{\partial w_1} = 2w_1(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$
$$\frac{\partial f}{\partial w_2} = 2w_2(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$

Now imagine initiating the gradient descent algorithm from the point $(0.5, 1)$ where the gradient vector is $(7.5, 15)$. One iteration of gradient descent with $\eta = 0.01$ would move from $(0.5, 1)$ to $(0.425, 0.85)$. The gradient vector at $(0.425, 0.85)$ is $(7.90, 15.81)$ and the next iteration of GD will move the point from $(0.425, 0.85)$ to $(0.35, 0.69)$. After 200 iterations, the algorithm moves the point to $(0.03, 0.06)$, which is very close to the global minimum of f.

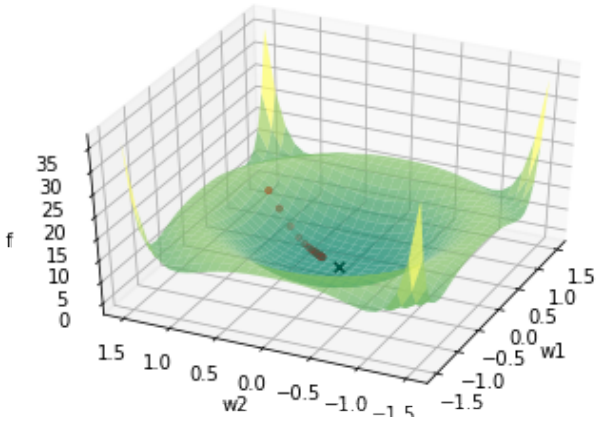


Figure 3.2: The graph of $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. The function attains a global minimum at $(0, 0)$.

3.1.3 *Learning Rate (LR)*

Choosing an appropriate learning rate is crucial for GD. Figure 3.3 shows the result of two iterations of gradient descent with a different learning rate. On the left, we see the result when λ is too small. The change of w is too small, and the loss function converges to the minimum very slowly. On the right, we see the result when λ is too big. The change of w is too large that the algorithm "shoots past" the minimum. If λ is even larger, the algorithm may even fail to converge to the minimum.

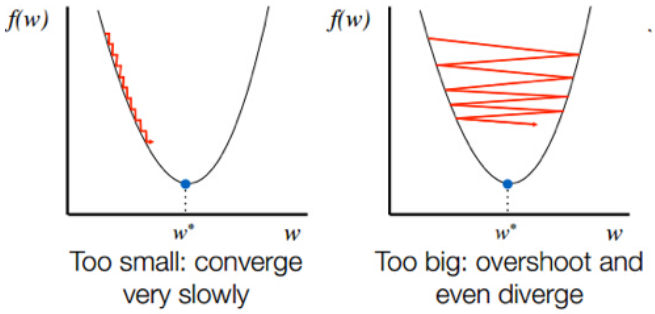


Figure 3.3: Two iterations of gradient descent with a different learning rate.

The natural question to ask is: what is the appropriate learning rate? There is some theory, and the best setting is known in some cases. But in general, it has to be set by trial and error, especially for non-convex loss functions. For instance, we start with some learning rate, say 0.5 and decrease η by $\frac{1}{2}$ if we do not observe a steady decrease in the training loss. Such heuristics are called *training schedules* and they are derived via trial and error on that *particular* dataset and model. [2]

[2] Constants whose values are decided by trial and error based on dataset and model are called *hyperparameters*. Modern ML models have several hyperparameters. Often optimization packages will suggest a default value and a fine-tuning method.

3.1.4 *Non-convex Functions*

For convex functions that are "bowl shaped," gradient descent with a small enough learning rate provably converges to the minimum solution. But for non-convex functions, the best we can hope for is converging to a point where $\nabla f = 0$. [3] Finding the global minimum of a non-convex function is NP-hard in the worst case.

In practice, loss functions are non-convex and have multiple local minima. Then, the gradient descent algorithm may converge to a different local minimum based on the initialization of the parameter vector $\vec{\mathbf{w}}$.

[3] Points where the gradient is zero are called *stationary points*, which include local minima, local maxima, and saddle points. It is possible for a GD algorithm to terminate at a saddle point, instead of the intended local minimum. There is advanced theory on how to escape saddle points, which will not be covered in this course.

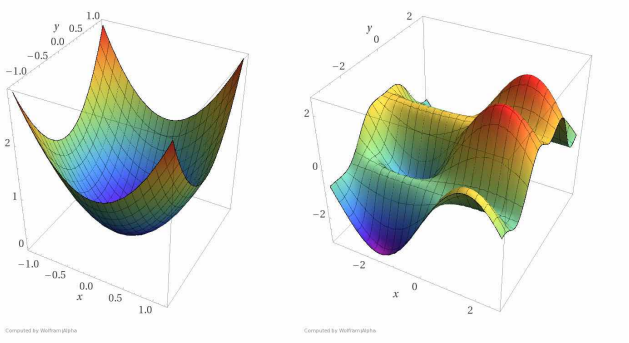


Figure 3.4: An example of a convex and a non-convex function in two variables. For non-convex functions, GD will reach a stationary point, where gradient is zero. Figure from https://www.kdnuggets.com/2016/12/hard-thing-about-deep-learning.html.

Example 3.1.3. *Consider the function* $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$*, which has two local minima at* $(-1, -1)$ *and* $(2, -1)$*. As seen in Figure 3.5, the*

local minimum that the gradient descent algorithm outputs depends on the initial point.

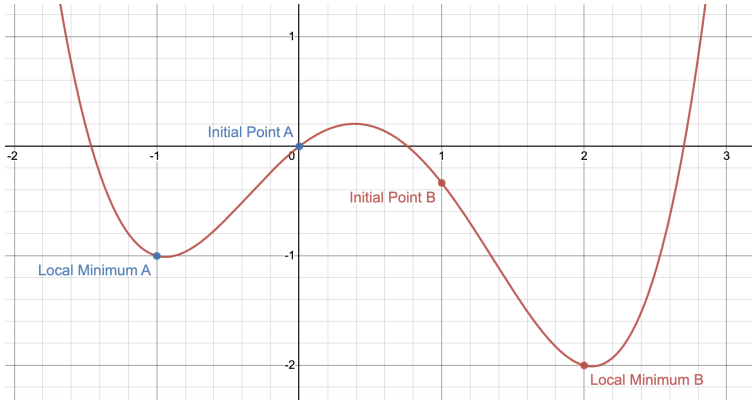


Figure 3.5: The graph of $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$ with two local minima.

3.2 *Implications of Linearity of Gradient*

The fact that gradient is a linear operator (*i.e.*, $\nabla(f_1 + f_2) = \nabla f_1 + \nabla f_2$) has great practical importance in machine learning.

Just like in (1.4), the training loss of a machine learning model is usually defined as the average (or the sum) of the loss on individual training data point. By the linearity of gradient, the gradient of the entire loss can be found by taking the sum of the gradient of the loss on individual data points.

3.2.1 *Stochastic Gradient Descent*

Since computing the gradient of the loss involves some computation on each of the data points, the computation can be quite slow for today's large data sets, which can contain millions of data points. A simple workaround is to estimate the gradient at each step by randomly sampling k data points and averaging the corresponding loss gradients. This is very analogous to opinion polls, which can also be seen as sampling from a distribution on vectors and using the average of the sample as a substitute for the population average. This algorithm is called *Stochastic Gradient Descent (SGD)*. [4] This technique works for two reasons: (1) all training data points are assumed to be sampled from the same distribution; (2) the overall training loss is just the sum/average of loss for individual data points.

[4] Some authors call this the *Batch SGD* and use the name *SGD* only for the case where $k = 1$.

3.2.2 *Mini-batch Stochastic Gradient Descent*

Today, large scale machine learning is done using special-purpose processors called *Graphical Processing Units (GPUs)*. [5] These highly

[5] As the name suggests, these were originally developed for computer graphics operations needed in computer games. Around 2012, deep learning experts realized their usefulness for deep learning. At the time writing code for GPUs was extremely difficult, but today's environments have made this much easier.

specialized architectures have the ability to perform fast parallel computations. To exploit these special capabilities, a special variant of SGD — Mini-batch SGD — can be used. Here the dataset is randomly partitioned into mini batches whose size is dictated by the degree of parallelism available in the GPU, usually a power of 2, such as 256. The members of each batch are loaded onto a different processor. Together the processors compute the gradient for one mini-batch in one go, add up the gradients to perform a single iteration for the gradient descent. Then they move on to the next batch, perform another update step, and so on.

3.2.3 *Federated Learning*

This is a conceptual framework for training a ML model on data belonging to different parties, who do not wish to hand the data over to a central server. Consider the following two examples:

1. Hospitals who wish to train an ML model on their pooled data, but who are forbidden by privacy laws to hand the data to other organizations.

2. Owners of Internet of Things (IoT) devices, who wish to benefit from training on their data but do not wish to submit the data.

In Federated Learning, the model is trained at a central server, whereas data remains with the data owners, who actively participate in the training. Users retrieve the current model parameters from the server and calculate the gradients *locally*. They send only the gradients, but not the data, to the server, and the overall gradient is calculated at the server as the weighted sum (or average) of the user gradients.

3.3 *Regularizers*

This section describes *regularization*, a useful idea that often improves generalization of the model. The main idea is that instead of minimizing the training loss function $\ell(\vec{\mathbf{w}})$, we minimize the function

$$\ell(\vec{\mathbf{w}}) + \lambda R(\vec{\mathbf{w}}) \tag{3.2}$$

where $\lambda > 0$ is a constant and $R(\vec{\mathbf{w}})$ is some non-negative function. $R(\vec{\mathbf{w}})$ is called a *regularizer* or sometimes *penalty*. We refer to (3.2) as the *regularized loss function*.

The most commonly used regularizer is the ℓ_2 regularizer where the squared ℓ_2 norm $R(\vec{\mathbf{w}}) = \|\vec{\mathbf{w}}\|_2^2$ of the weight vector is used.

Example 3.3.1. *Recall the sentiment prediction model using least squares loss. Suppose the training data consists of two data points:* $(\vec{\mathbf{x}}^1, y^1) = ((1,0,1),-1)$ *and* $(\vec{\mathbf{x}}^2, y^2) = ((1,1,0),+1)$. *Then the least squares loss, without any regularizer, can be written as*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2) \tag{3.3}$$

A little thought suggests that the minimum value of this loss is 0 *provided there exists* (w_0, w_1, w_2) *such that*

$$(-1-(w_0+w_2))^2 = 0 = (1-(w_0+w_1))^2.$$

You can verify that infinitely many solutions exist: all $\vec{\mathbf{w}}^* = (w_0, w_1, w_2)$ *that lie on the line* $(0,1,-1)+t(1,-1,-1)$ *where* $t \in \mathbb{R}$. *In other words, the loss has infinitely many minimizers.*

Now if impose an ℓ_2 *regularizer, the loss becomes*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2)+\lambda(w_0^2+w_1^2+w_2^2) \tag{3.4}$$

Any minimizer of this loss must make the gradient zero. In other words, the minimizer will satisfy the following system of linear equations:

$$\begin{cases} (2+2\lambda)w_0 + w_1 + w_2 = 0 \\ w_0 + (1+2\lambda)w_1 = 1 \\ w_0 + (1+2\lambda)w_2 = -1 \end{cases}$$

You can verify that $\vec{\mathbf{w}}^{**} = \left(0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda}\right)$ *is the unique minimizer for any* $\lambda > 0$. *For a sufficiently small value of* λ, *the corresponding* $\vec{\mathbf{w}}^{**}$ *is close enough to the line* $(0,1,-1)+t(1,-1,-1)$. *That is, it has a non-zero training loss, but the value is very close to zero. Combined with the fact it has a small norm,* $\vec{\mathbf{w}}^{**}$ *becomes the minimizer for the regularized loss.*

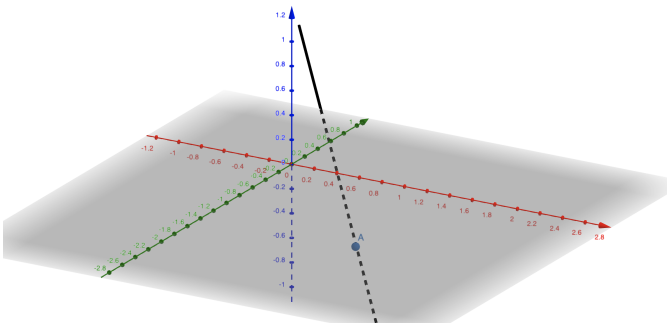


Figure 3.6: The graph of the line $(0,1,-1)+t(1,-1,-1)$ and the point $\vec{\mathbf{w}}^{**} = (0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda})$ when $\lambda = 0.01$

Note that if $\vec{\mathbf{w}}^*$ is the minimizer of $\ell(\vec{\mathbf{w}})$ and $\vec{\mathbf{w}}^{**}$ the minimizer of the regularized loss, then by definition of a minimizer, it always

holds that $\ell(\vec{\mathbf{w}}^*) \leq \ell(\vec{\mathbf{w}}^{**})$. In general, regularization ends up leading to training models with a *higher* value of $\ell(\vec{\mathbf{w}})$. This is considered acceptable because the models often generalize better. In other words, a slightly higher training loss is considered a price worth paying for a significantly lower test loss. This is illustrated by the example of sentiment prediction from Chapter 1. As hinted there, the results shown used a model trained with an ℓ_2 regularizer. The dataset involves 15k distinct words, so that is the number of model variables. There are 8k data points. Recall from Problem 1.2.5 that in such settings, there usually will exist a linear model that perfectly fits the data points. Indeed, we see in Table 3.1 that this is the case when we don't use a regularizer. However, using a regularizer prevents the model from perfectly fitting the training data. But the test loss drops tenfold with regularization.

	No regularizer	With ℓ_2-regularizer
Train MSE	0.0000	0.0727
Test MSE	7.9469	0.7523
Training accuracy	100.00%	99.55%
Test accuracy	61.67%	78.07%

Table 3.1: Training sentiment model on the SST with and without ℓ_2 regularizer.

3.3.1 *Effects of Regularization*

Here we briefly list some benefits of regularization.

1. Regularizers often help improve generalization. Above we saw a concrete example with the sentiment prediction model.

2. Adding a scalar multiple of $\|\vec{\mathbf{w}}\|_2^2$ to a function can speed up optimization by slightly reshaping the optimization landscape. The mathematical treatment of this is beyond the scope of this course.

3. Without a regularizer term, models such as logistic regression and soft-margin SVMs begin to lose their power. This will be explained when we discuss these models in Chapter 4.

3.3.2 *Why Does Regularization Help?*

The simplest answer is that we do not fully understand this concept yet. In this section, we present some intuitions derived from simple models, but keep in mind that these ideas might be misleading in more complicated models.

The usual explanation given is that the norm of the parameter vector controls the *expressiveness* or *complexity* of the model. Here "complexity" is being used in the sense of "complicatedness". By

trying to minimize loss as well as the norm of the parameter vector, the learned model tends to stay simple. [6] Whereas this discussion can be made fairly rigorous for linear models, it does not seem to apply to more complicated models: for instance regularization often helps a lot in deep learning, but the rigorous explanation appear to be at best incomplete and at worst incorrect there. [7]

Another explanation [8] is that a regularizer serves as a penalty for large weights and forces the model to choose smaller absolute values of parameters. According to this explanation, adding regularizers to a model penalizes higher-order terms or unnecessary variables and is able to avoid overfitting. Indeed, Figure 3.7 shows that the weights of the parameters in our sentiment model is significantly smaller when trained with a regularizer. But one lingering question with this explanation is: *How come attaching the **same** penalty to all variables forces the model to identify variables that are needed, and those that are not? What causes this disparate treatment of the variables?*

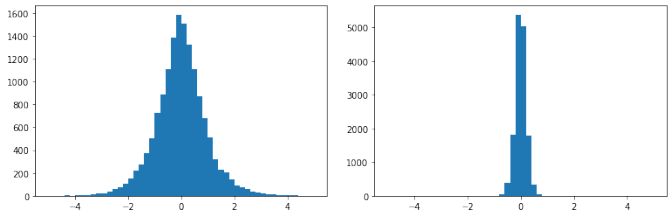


Figure 3.7: The histogram of weights of the parameters in the sentiment prediction model with (right) or without (left) an ℓ_2 regularizer.

Now consider this explanation — ℓ_2 regularization introduces a new dynamic to gradient descent, whereby gradient updates have to constantly battle against a rescaling that is always trying to whittle *all* variables down to zero. The effort succeeds only for variables where gradient updates are pushing hardest to make them nonzero. Therefore, the weights for "necessary" variables survive, while "unnecessary" variables are thrown away. To say this more precisely, consider the regularized loss $\ell(\vec{\mathbf{w}}) + \lambda \|\vec{\mathbf{w}}\|_2^2$ whose gradient is

$$\nabla \ell + 2\lambda \vec{\mathbf{w}}$$

Thus the update rule in gradient descent can be written as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t - \eta(\nabla \ell + 2\lambda \vec{\mathbf{w}}^t)$$

where $\vec{\mathbf{w}}^t$ denotes the weight vector at the t-th time step. This update rule can be rewritten as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t(1 - 2\eta\lambda) - \eta \nabla \ell \tag{3.5}$$

The first term is *down-scaling*: if for example $\eta = \lambda = 0.1$, this amounts to multiplying the current vector by 0.98, and this of course will make $\vec{\mathbf{w}}$ very small in a few hundred iterations.

[6] Recall the famous *Occam's Razor* for judging goodness of scientific theories: The simpler the theory that explains the known facts, the more likely it is to be correct. An ML model can be seen as a "theory" about relationships in the data, and thus the simplest theory is to be preferred.

[7] See the blog https://www.offconvex.org for posts about generalization and deep learning. They also discuss how other ideas such as VC dimension, which we did not cover in this course, also do not apply in deep learning.

[8] See the online lecture video by Andrew Ng. https://www.youtube.com/watch?v=KvtGD37Rm5I&ab_channel=ArtificialIntelligence-AllinOne

The second term is the gradient update. It can counteract the down-scaling by making the variables larger. But notice that the amount of change is based on how much each of the coordinates contribute to reducing the loss. Variables that are not useful will tend not to get increased by the gradient update and thus will keep getting down-scaled to low values. [9] The choice of λ mediates between these two processes.

[9] It is one of those "use it or lose it" situations!

3.4 *Gradient Descent in Programming*

In this section, we briefly discuss how to implement the Gradient Descent algorithm in Python. It is customary to use the *numpy* package to speed up computation and the *matplotlib* package for visualization.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

# initialize variables
num_iter = ...
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
x[0], y[0], eta = ...

# define functions to calculate f and grad_f
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f

# run Gradient Descent
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y

# plot the surface
xmin, xmax, ymin, ymax, n = ...
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)

ax = plt.figure(figsize=(12, 10)).gca(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)

# plot the trajectory of Gradient Descent
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

We first start off by importing necessary packages and initializing variables. The following code initializes *numpy* arrays of length *num_iter* + 1, with all entries initialized to 0

```
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
```

Sometimes, it is useful to make use of *np.ones()*, which will generate arrays filled with entries equal to 1.

We then define functions that will calculate the values of f and ∇f given an array of data points (x, y).

```
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f
```

This allows us to run the Gradient Descent algorithm as in

```
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y
```

Here we iteratively update the value of (x, y) using $\nabla f(x, y)$ and store each of the points in the array x and y.

We next plot the surface of the function $f(x, y)$. To start, we first create a grid of (x, y) points to evaluate $f(x, y)$ at.

```
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)
```

The function call *np.linspace(min, max, n)* generates an array of n equally spaced values from *min* to *max*. For example, the code

```
np.linspace(-2, 2, 5)
```

will create an array $[-2, -1, 0, 1, 2]$. Then *np.meshgrid(x, y)* will create a grid from the array of x values and the array of y values. We can now perform the $3D$ plotting with the following code.

```
ax = plt.figure(figsize=(12, 10)).gca(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)
```

Feel free to change the values of the optional parameters to understand their purpose. Unlike the code for plotting a scatter plot of linear regression in Chapter 1, here we create an object of the *Axes* class with the function *plt.figure().gca()*[10]. Then we call its instance

[10] You can read more about the differences between these two *matplotlib* interfaces at https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

methods to add features to it (*e.g.*, x-, y-, z-labels).

Finally, we can plot the trajectory of the Gradient Descent algorithm with the code

```
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

You can alternatively call

```
ax.scatter(x, y, f(x, y))
```

but the names of optional parameters might be slightly different.

3.4.1 *Using Machine Learning Packages*

When the function f is simple and it is possible to calculate ∇f by hand, we can implement the Gradient Descent algorithm by hand as in the previous subsection. However, in most ML programs, the loss function f is very high-dimensional, and it is difficult to write a single function to directly compute the gradient ∇f. Instead, we can make use of functions defined in popular ML packages. Here, we introduce one such package called PyTorch:

- *torch*: This is a popular package used for designing and training deep learning models. PyTorch uses an object-oriented interface for user convenience and provides access to optimized array data structures called *tensors* to make computations faster and more efficient. The package also provides support for GPU training. [11]

[11] Documentation is available at https://pytorch.org/docs/stable/index.html

Using PyTorch, Gradient Descent can be implemented in just a few lines:

```
import torch
model = ...
opt = torch.optim.SGD(model.parameters(), lr=0.1)
```

The code above will create an instance of the *Optimzer* class, which has pre-defined methods that will compute the gradients and automate the Gradient Descent process.

3.4.2 *Beyond Vanilla Gradient Descent*

If you visit the documentation for the *torch.optim*, [12] you may notice that there are other algorithms listed as an alternative to the Stochastic Gradient Descent. A lot of these algorithms are extensions of the GD algorithm we explained throughout this chapter, which have proven to be more effective than the vanilla GD algorithm in certain cases (*e.g.*, Adam, Adagrad, Nesterov momentum). For example, these algorithms may choose to add a *momentum* to the gradient, so that the rate of change of f will be accelerated if it has been updating

[12] https://pytorch.org/docs/stable/optim.html

in the same direction in the recent few steps. These algorithms may also choose to use a different learning rate for each of the model parameters. In particular, the appropriate learning rate can be computed based on the mean and the variance of the gradient values from the recent few steps.

4
Linear Classification

Multi-way Classification is a task of learning to predict a label on newly seen data out of k possible labels. In *binary classification*, there are only two possible labels, say ± 1. Sentiment prediction in Chapter 1 was an example of a binary classification task. In this chapter, we introduce two other linear models that perform binary classification: logistic regression and Support Vector Machines (SVMs). From these two models, we learn more about the thought process of designing loss functions that are appropriate to the task. [1]

In this chapter, we are interested in using linear models to perform classification. In a binary classification problem, the training dataset consists of (point, label) pairs $(\vec{\mathbf{x}}, y)$ where y can take two values (*e.g.*, $\{\pm 1\}$ or $\{0, 1\}$). In a more general multi-class classification problem, the data has one of k labels, drawn from $\{0, 1, \ldots, k-1\}$.

[1] All the linear models we will study fall under an all-encompassing framework called *Generalized Linear Models*. If you ever are faced with a new situation where none of the models below are an exact match, try looking up this general framework.

4.1 General Form of a Linear Model

You already encountered a linear model in Chapter 1 — the least squares regression model for sentiment prediction. Given an input $\vec{\mathbf{x}}$, we learned a parameter vector $\vec{\mathbf{w}}$ that minimizes the loss $\sum_i (y^i - \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)^2$. The model can be seen as mapping an input vector $\vec{\mathbf{x}}$ to a real value $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$. For sentiment classification, we changed this real-valued output at test time to ± 1 by outputting $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$.

You probably wondered there: *Why don't we simply use sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$ directly as the output of the model while training?* In other words, why not do training on the following loss:

$$\sum_i (y^i - \text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i))^2 \tag{4.1}$$

The answer is that using the $\text{sign}(z)$ function in the loss makes gradient-based optimization ill-behaved. The derivative of $\text{sign}(z)$ is 0 except at $z = 0$ (where the derivative is discontinuous) and thus the gradient is uninformative about how to update the weight vector.

So the work-around in Chapter 1 (primarily for ease of exposition) was to train the sentiment classification model using the least squares loss $\sum_i (y^i - \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)^2$, which in practice is used more often in settings where the desired output y^i is real-valued output as opposed to binary. This gave OK results, but in practice one would use either of the two linear models [2] introduced in this chapter: *Logistic Regression* and *Support Vector Machines*. These are similar in spirit to the linear regression model — (1) given an input $\vec{\mathbf{x}}$, the models learn a parameter vector $\vec{\mathbf{w}}$ that minimizes a loss, defined as a differentiable function on $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$; (2) at test-time, the model outputs $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$. [3] The main difference, however, is that the loss for the linear models introduced in this chapter is designed specifically for the binary classification task. Pay close attention to our "story" for why the loss makes sense. This will prepare you to understand any new loss functions you come across in your future explorations.

[2] They are called *linear* because they use the mapping $\vec{\mathbf{x}} \mapsto \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$.

[3] There are other ways to output a discrete ± 1 label, but using the sign function is the most canonical way. We will discuss the behavior of the models at test-time later in the chapter.

4.2 Logistic Regression

The logistic regression model arises from thinking of the answer as being probabilistic: the model assigns a "probability" to each of the two labels, with the sum of the two probabilities being 1. [4] This paradigm of a probabilistic answer is a popular way to design loss functions in a host of ML settings, including deep learning.

[4] This "probability" is what is called *subjective probability*, analogous to what we mean when say things like "I am 99 percent sure my friend X will like movie Y." There is only one person X and one movie Y and they are not drawn from some probability space. Instead we're expressing a subjective feeling of near-certainty based upon past observations of person X.

Definition 4.2.1 (Logistic model)**.** *Given the input* $\vec{\mathbf{x}}$, [5] *the model assigns the "Probability that the output is* $+1$*" to be*

$$\sigma(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) = \frac{1}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} \tag{4.2}$$

where σ *is the sigmoid function (see Chapter 19). This implies that "Probability that the output is* -1*" is given by*

$$1 - \frac{1}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} = \frac{\exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} = \frac{1}{1 + \exp(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} \tag{4.3}$$

[5] As in Chapter 1 we assume $\vec{\mathbf{x}}$ contains a dummy coordinate x_0 that is 1 at all points: this allows us to include a constant bias term when we take the dot product $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ with the weight vector.

See Figure 4.1. Note that "the probability that the output is $+1$" is greater than $\frac{1}{2}$ precisely if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 0$. Furthermore, increasing the value of $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ causes the probability to rise towards 1. Conversely, if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < 0$, then "the probability of label -1" is greater than $\frac{1}{2}$. When $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$, the probability of label $+1$ and -1 are both equal to $\frac{1}{2}$. In this way, the logistic model can be seen as a continuous version of the $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$.

Example 4.2.2. *If* $\vec{\mathbf{x}} = (1, -3)$ *and* $\vec{\mathbf{w}} = (0.2, -0.1)$*, then the probability of label* $+1$ *is*

$$\frac{1}{1 + \exp(-0.2 - 0.3)} = \frac{1}{1 + e^{-0.5}} \simeq 0.62$$

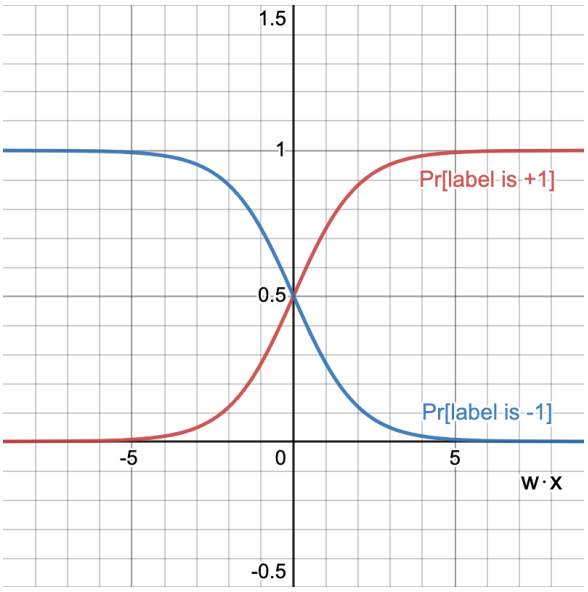


Figure 4.1: The graph of the probability that the output of a logistic model is +1 (red) or -1 (blue) given $\vec{w} \cdot \vec{x}$.

4.2.1 *Defining Goodness of Probabilistic Predictions*

Thus far, we explained how the logistic model generates its output given an input vector $\vec{x}$ and the current weight vector $\vec{w}$. But we have not yet talked about how to train the model. To define a loss function, we need to decide what are the "good" values for $\vec{w}$. Specifically, we formulate a definition of "quality" of probabilistic predictions.

Definition 4.2.3 (Maximum Likelihood Principle)**.** *Given a set of observed events, the* ***goodness*** *of a probabilistic prediction model* [6] *is the probability it assigned to the observed events.*

[6] This is a *definition* of goodness, not the consequence of some theory.

We illustrate with an example.

Example 4.2.4. *You often see weather predictions that include an estimate of the probability of rain. Table 4.1 shows the predictions by two models at the start of each day of the week. After the week is over, we have observed if it actually rained on each of the days. Based on these observations, which model made better predictions this week?*

	M	T	W	Th	F
Model 1	60%	20%	90%	50%	40%
Model 2	70%	50%	80%	20%	60%
Rained?	Y	N	Y	N	N

Table 4.1: Weather predictions by Model 1 and Model 2.

We can answer this question by seeing which model assigns higher likelihood to the events that were actually observed (i.e., whether or not it rained). For instance, the likelihood of the observed sequence according to Model 1 *is*

$$0.6 \times (1 - 0.2) \times 0.9 \times (1 - 0.5) \times (1 - 0.4) = 0.1296$$

The corresponding number for Model 2 *is* 0.0896 *(check this!). So Model* 1 *was a "better" model for this week.*

4.2.2 Loss Function for Logistic Regression

We employ the Maximum Likelihood Principle from the previous part to define the loss function for the logistic model. Suppose we are provided the labeled dataset $\{(\vec{\mathbf{x}}^1, y^1), (\vec{\mathbf{x}}^2, y^2), \ldots, (\vec{\mathbf{x}}^N, y^N)\}$ for training where y^i is a ± 1 label for the input $\vec{\mathbf{x}}^i$. By the description given in Definition 4.2.1, the probability assigned by the model with the weights $\vec{\mathbf{w}}$ to the i-th labeled data point is

$$\frac{1}{1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)}$$

which means that the total probability ("likelihood") assigned to the dataset is

$$P = \prod_{i=1}^{N} \frac{1}{1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)} \tag{4.4}$$

We desire the model $\vec{\mathbf{w}}$ that maximizes P. Since $\log(x)$ is an increasing function, the best model is also the one that maximizes $\log P$, hence the one that minimizes $-\log P = \log \frac{1}{P}$. This leads to the logistic loss function:

$$\log\left(\prod_{i=1}^{N} (1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i))\right) = \sum_{i=1}^{N} \log(1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)) \tag{4.5}$$

Note that this expression involves a sum over training data points, which as discussed in Section 3.2, is a very desirable and practical property of loss in machine learning.

Problem 4.2.5. *Verify that the gradient for the logistic loss function is*

$$\nabla \ell = \sum_{i=1}^{N} \frac{-y^i \vec{\mathbf{x}}^i}{1+\exp(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)} \tag{4.6}$$

4.2.3 Using Logistic Regression for Roommate Matching

In this part, we use the following example to illustrate some of the material covered in the previous parts.

Example 4.2.6. *Suppose Princeton University decides to pair up newly admitted undergraduate students as roommates. All students are asked to fill a questionnaire about their sleep schedule and their music taste. The questionnaire is used to generate a compatibility score in* $[0, 1]$ *for each of the two attributes, for each pair of students. Table* 4.2 *shows the calculated*

Sleep (S)	Music (M)	Compatible?
1	0.5	+1
0.75	1	+1
0.25	0	−1
0	1	−1

Table 4.2: Sample data of compatibility scores for four pairs of students.

compatibility scores for four pairs of roommates from previous years and whether or not they turned out to be compatible (+1 for compatible, −1 for incompatible).

We wish to train a logistic model to predict if a pair of students will be compatible based on their sleep and music compatibility scores. To do this, we first convert the data in Table 4.2 into a vector form.

$$\begin{aligned} \vec{\mathbf{x}}^1 &= (1, 1, 0.5) \quad & y^1 &= +1 \\ \vec{\mathbf{x}}^2 &= (1, 0.75, 1) \quad & y^2 &= +1 \\ \vec{\mathbf{x}}^3 &= (1, 0.25, 0) \quad & y^3 &= -1 \\ \vec{\mathbf{x}}^4 &= (1, 0, 1) \quad & y^4 &= -1 \end{aligned} \tag{4.7}$$

where the first coordinate x_0^i of $\vec{\mathbf{x}}^i$ is a dummy variable to introduce a constant bias term, and the second and third coordinates are respectively for sleep and music compatibility scores.

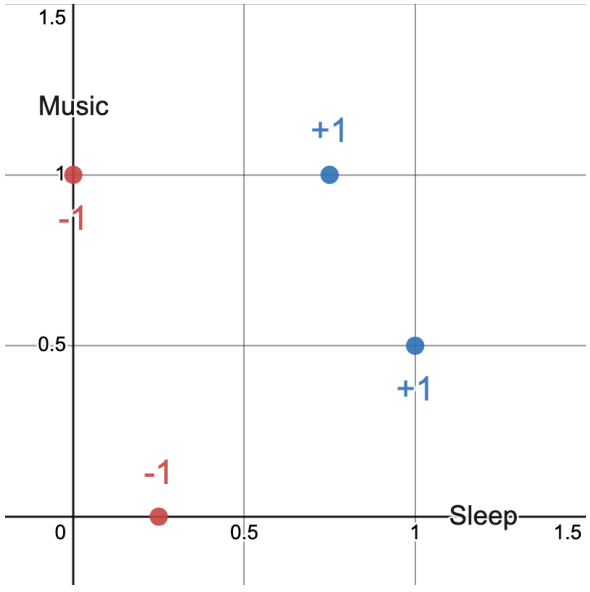


Figure 4.2: Graph representing the points in Table 4.2. The x-, y-axis in the graph correspond to the Sleep and Music compatibility scores, or the second and third coordinates in (4.7).

Consider two models — Model 1 with the weight vector $\vec{\mathbf{w}}^1 = (0, 1, 0)$ and Model 2 with the weight vector $\vec{\mathbf{w}}^2 = (0, 0, 1)$. Model 1 only looks at the sleep compatibility score to calculate the probability that a pair of students will be compatible as roommates, whereas

Model 2 only uses the music compatibility score. For example, Model 1 assigns the probability that the first pair of students are compatible as

$$\sigma(\vec{\mathbf{w}}^1 \cdot \vec{\mathbf{x}}^1) = \frac{1}{1+\exp(-1)} \simeq 0.73$$

We can calculate the probability for the other pairs and for Model 2 and fill out the following Table 4.3:

	Pair 1	Pair 2	Pair 3	Pair 4
Model 1	0.73	0.68	0.56	0.50
Model 2	0.62	0.73	0.50	0.73
Compatible?	Y	Y	N	N

Table 4.3: Roommate compatibility predictions by Model 1 and Model 2.

Then the likelihood of the observations (YYNN) according to Model 1 can be calculated as

$$0.73 \times 0.68 \times (1-0.56) \times (1-0.50) \simeq 0.11$$

where as the likelihood of the observations according to Model 2 is

$$0.62 \times 0.73 \times (1-0.50) \times (1-0.73) \simeq 0.06$$

Therefore, the Maximum Likelihood Principle tells us that Model 1 is a "better" model than Model 2.

The full logistic loss for this training data can be written as

$$\begin{aligned}\sum_{i=1}^{4} \log(1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)) = &\log(1+\exp(-(w_0 \cdot 1 + w_1 \cdot 1 + w_2 \cdot 0.5))+ \\ &\log(1+\exp(-(w_0 \cdot 1 + w_1 \cdot 0.75 + w_2 \cdot 1))+ \\ &\log(1+\exp(w_0 \cdot 1 + w_1 \cdot 0.25 + w_2 \cdot 0)+ \\ &\log(1+\exp(w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 1)\end{aligned}$$

and the values that minimize this loss can be found as $w_0 = -21, w_1 = 32, w_2 = 8.9$.

4.2.4 *Testing the Model*

After training the model on the training data, we can use it to define label probabilities on any new data point. However, the probabilities do not explicitly tell us what label to output on a new data point. There are two options:

1. (Probabilistic) If p is the probability of the label $+1$ according to (4.2), then use a random number generator to output $+1$ with probability p and -1 with probability $1-p$.
2. (Deterministic) Output the label with a higher probability.

Recall from an earlier discussion that $\Pr[+1] \geq \Pr[-1]$ if and only if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} \geq 0$. In other words, the second deterministic option is equivalent to the sign(z) function: sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$!

We conclude that logistic regression is quite analogous to what we did in Chapter 1, except instead of least squares loss, we are using logistic loss to train the model. The logistic loss is explicitly designed with binary classification in mind. [7]

[7] Using logistic loss (and ℓ_2 regularizer) instead of least squares in our sentiment dataset boosts test accuracy from 78.1% to 80.7%.

4.3 *Support Vector Machines*

A *Support Vector Machine (SVM)* [8] is also a linear model. It comes in several variants, including a more powerful *kernel SVM* that we will not study here. But this rich set of variants made it an interesting family of models, and it is fair to say that in the 1990s its popularity was somewhat analogous to the popularity of deep nets today. It remains a very useful model for your toolkit. The version we are describing is a so-called *soft margin SVM*.

[8] From *An optimal algorithm for training maximum margin classifiers.* by Boser, Guyon, and Vapnik in *COLT* 1992. The name *Support Vector Machine* comes from a theorem that characterizes the optimum model in terms of "support vectors." We will not cover that theorem here.

As in the least squares regression, the main idea in designing the loss is that the label should be $+1$ or -1 according to sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$. But we want to design a loss with a well-behaved gradient that provides a clearer direction of improvement. To be more specific, we want the model to have more "confident" answers, and we will penalize the model if it comes up with a correct answer but with a low degree of "confidence."

For $z \in \mathbb{R}$, let us define

$$\text{Hinge}(z) = \max\{0, 1 - z\} \tag{4.8}$$

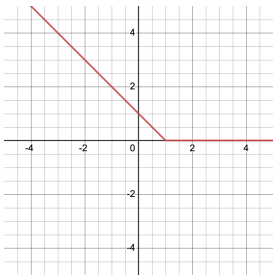


Figure 4.3: The graph of the hinge function.

Note that this function is always at least zero, and strictly positive for $z < 1$. When z decreases to negative infinity, there is no finite upper bound to the value. The derivative is zero for $z > 1$ and 1 for $z < 1$. The derivative is currently undefined at $z = 1$, but we can arbitrarily choose between 0 or 1 as the newly defined value.

For a single labeled data point $(\vec{\mathbf{x}}, y)$ where $y \in \{-1, 1\}$, the SVM loss is defined as

$$\ell = \text{Hinge}(y\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) \tag{4.9}$$

and its gradient is

$$\nabla \ell = \begin{cases} -y\vec{\mathbf{x}} & \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < 1 \\ 0 & \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1 \end{cases}$$

The SVM loss for the entire training dataset can be defined as

$$\sum_i \text{Hinge}(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i) \tag{4.10}$$

that is, the sum of the SVM loss on each of the training data points.

Suppose $y = +1$. Then this loss is 0 only when $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$. In other words, making loss zero not only requires $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ to be positive, but also be comfortably above 0. If $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ dips below 1, the loss is positive and increases towards $+\infty$ as $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} \to -\infty$. (Likewise if the label $y = -1$, then the loss is 0 only when $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < -1$.)

Recall that the goal of a gradient-based optimization algorithm is to minimize the loss. Therefore, the loss gives a clear indication of the direction of improvement until the data point has been classified correctly with a comfortable *margin* away from 0, out of the *zone of confusion*.

Example 4.3.1. *Recall the roommate compatibility data from Table 4.2. Consider the soft-margin SVM with the weight vector* $\vec{\mathbf{w}} = (-1.5, 3, 0)$. *This means the decision boundary — the set of points where* $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$ *— is drawn at Sleep* $= \frac{1}{2}$, *and the margins — the set of points where* $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = \pm 1$ *— are drawn at Sleep* $= \frac{5}{6}$ *and Sleep* $= \frac{1}{6}$. *Figure 4.4 shows the decision boundary and the two margin lines of the model. The SVM loss is zero for the point* $(1, 0.5)$ *because it is labeled +1 and away from the decision boundary with enough margin. Similarly, the loss is zero for the point* $(0, 1)$. *The loss for the point* $(0.75, 1)$, *however, can be calculated as*

$$Hinge(1 \cdot (-1.5 \cdot 1 + 3 \cdot 0.75)) = 0.25$$

and similarly, the loss for the point $(0.25, 0)$ *is 0.25.*

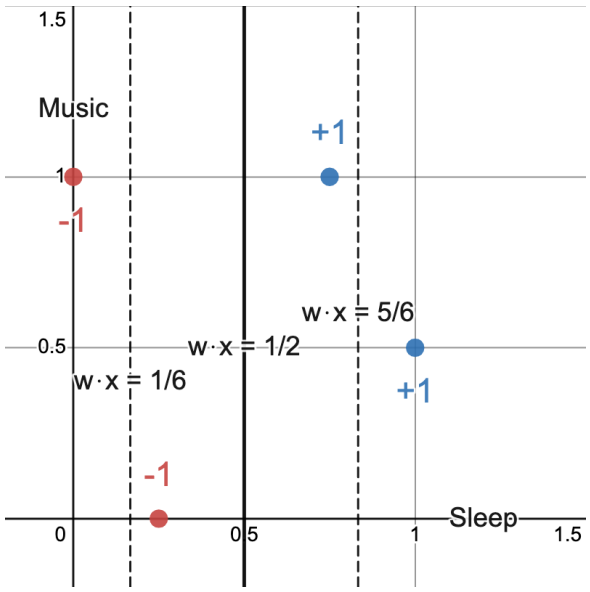


Figure 4.4: The decision boundary of a soft-margin SVM on the roommate matching example. The region to the left of the two dotted lines is where the model confidently classifies as -1; the region to the right is where it confidently classifies as $+1$; and the region between the two dotted lines is the zone of confusion.

The gradient of the loss at the point $(0.75, 1)$ *is*

$$-y\vec{\mathbf{x}} = (-1, -0.75, -1)$$

and the update rule for a gradient descent algorithm will be written as

$$\vec{\mathbf{w}} \leftarrow (-1.5, 3, 0) - 0.1(-1, -0.75, -1) = (-1.4, 3.075, 0.1)$$

where $\eta = 0.1$*, and the new SVM loss will be*

$$Hinge(1 \cdot (-1.4, 3.075, 0.1) \cdot (1, 0.75, 1)) = 0$$

which is now lower than the SVM loss before the update.

4.4 Multi-class Classification (Multinomial Regression)

So far, we have only seen problems where the model has to classify using two labels ± 1. In many settings there are k possible labels for each data point [9] and the model has to assign one of them. The conceptual framework is similar to logistic regression, except the model defines a nonzero probability for each label as follows. The notation assumes data is d-dimensional and the model parameters consist of k vectors $\vec{\theta}^1, \vec{\theta}^2, \ldots, \vec{\theta}^k \in \mathbb{R}^d$. We define a new vector $\vec{\mathbf{z}} \in \mathbb{R}^k$ where each coordinate z_i satisfies $z_i = \vec{\theta}^i \cdot \vec{\mathbf{x}}$. Then the probability of a particular label is defined through the *softmax function* (see Chapter 19):

[9] This is the case in most settings in modern machine learning. For instance in the famous ImageNet challenge, each image belongs to one of 1000 classes.

$$\begin{aligned} \Pr[\text{label } i \text{ on input } \vec{\mathbf{x}}] &= softmax(\vec{\mathbf{z}}) \\ &= \frac{\exp(\vec{\theta}^i \cdot \vec{\mathbf{x}})}{\sum_{j=1}^{k} \exp(\vec{\theta}^j \cdot \vec{\mathbf{x}})} \end{aligned} \tag{4.11}$$

This distribution can be understood as assigning a probability to label i such that it is *exponentially proportional* to the value of $\vec{\theta}^i \cdot \vec{\mathbf{x}}$.

Problem 4.4.1. *Using the result of Problem 19.2.4, verify that the definition of logistic regression as in (4.2), (4.3) are equivalent to the definition of multi-class regression as in (4.11).*

Problem 4.4.2. *Reasoning analogously as in logistic regression, derive a training loss for this model using Maximum Likelihood Principle.*

Since $\exp(z) > 0$ for every real number z the model above assigns a nonzero probability to every label. In some settings that may be appropriate. But as in case of logistic regression, at test time we also have the option of extracting a deterministic answer out of the model: the $i \in \{1, 2, \ldots, k\}$ that has the largest value of $\vec{\theta}^j \cdot \vec{\mathbf{x}}$.

4.5 Regularization with SVM

It is customary to use a regularizer, typically ℓ_2, with logistic regression models and SVMs. When a ℓ_2 regularizer is applied, the full

SVM loss is rewritten as

$$\sum_i \text{Hinge}(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i) + \lambda \|\vec{\mathbf{w}}\|_2^2 \tag{4.12}$$

Let's see why regularization is sensible for SVMs, and even needed. The Hinge function (4.8) treats the point $z = 1$ as special. In terms of the SVM loss, this translates to the thought that having $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$ is a more "confident" classification of $\vec{\mathbf{x}}$ than just having $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$ to be correct (*i.e.*, $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 0$). But this choice is arbitrary because we have not specified the scale of $\vec{\mathbf{w}}$. If $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 1/10$ then scaling $\vec{\mathbf{w}}$ by a factor 10 ensures $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$. Thus the training algorithm has cheap and meaningless ways of reducing the training loss. By applying an ℓ_2 regularizer, we are able to prevent this easy route for the model, and instead, force the training to find optimal weights $\vec{\mathbf{w}}$ with a small norm.

Problem 4.5.1. *Write a justification for why it makes sense to limit the ℓ_2 norm of the classifier during logistic regression. How can large norm lead to false confidence (i.e., unrealistically low training loss)?*

4.6 Linear Classification in Programming

In this section, we briefly discuss how to implement the logistic regression model in Python. It is customary to use the *numpy* package to speed up computation and the *matplotlib* package for visualization.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt

# prepare dataset
X = ... # array of shape (n, d), each row is a d-dimensional data point
y = ... # array of shape (n), each value = -1 or +1
w = ... # array of shape (d), each value is a weight for each dimension
X_train, X_test, y_train, y_test, eta = ...

# define functions
def loss(X, y, w):
    # returns the logistic loss
    # return sum log(1 + exp(-y*w*x))
    ...

def grad_loss(X, y, w):
    # returns the gradient of the logistic loss
    # return sum (-y*x)/(1 + exp(y*w*x))
    ...

def gradient_descent(X, y, w0, eta)
    ...
    return w

# run Gradient Descent
w = gradient_descent(X_train, y_train, w, eta)
```

```
# plot the learned classifier
# assuming data is 2-dimensional
colors = {1: 'blue', -1: 'red'}
xmin, xmax, ymin, ymax = ...
plt.scatter(X[:,0], X[:,1], c=np.array([colors[y_i] for y_i in y]))
plt.plot([xmin, xmax], [ymin, ymax], c='black')
```

We have already discussed how to implement the majority of the code sample above in previous chapters. The only parts that are new are the functions to calculate the logistic loss and its gradient. This is consistent with the theme of this chapter — to discuss how to design loss functions that are appropriate for the task. Nevertheless, while the content of this code sample is familiar, some sections of the code introduce new Python functionality and syntax. We first consider the logistic loss and gradient functions:

```
def loss(X, y, w):
    # returns the logistic loss
    # return sum log(1 + exp(-y*w*x))
    ...

def grad_loss(X, y, w):
    # returns the gradient of the logistic loss
    # return sum (-y*x)/(1 + exp(y*w*x))
    ...
```

In Java, the programming language you learned in earlier programming classes, you would have to rely on a *for* loop to account for the array inputs in the *loss()* and *grad_loss()* functions. However, Python and *numpy* support many *vectorized operations*, including matrix multiplication and element-wise multiplication. These operations are far more concise to read and will also improve the runtime of the program by a great margin. Note that the code snippet above does not contain these operations; it is simply pseudo-code for your intuition. You will be introduced to these vectorized operations during the precept, and you will be expected to implement the loss function with these new tools in your programming assignments.

Next, we use a Python *dictionary* to store information corresponding to the plot's coloring scheme:

```
colors = {1: 'blue', -1: 'red'}
```

This is equivalent to a hash table from Java. Here, 1 and -1 are the *keys* and "blue" and "red" are respectively their *values*.

We will now discuss multi-dimensional arrays in Python. There are multiple ways to perform array indexing. For example, if X is a 2-dimensional array, both *X[i][j]* and *X[i, j]* can be used to extract the entry at the i-th row, j-th column. It is also possible to provide a set of rows or a set of columns to extract. The following code snippet generates an array of shape (2, 2), where each entry is from the row 0

or 1 and column 0 or 2:

```
X[[0, 1], [0, 2]]
```

Note that similar to the $1D$ case, the *:* operator is used to perform array slicing. Bounding indices can be omitted as shown in the following code snippet:

```
X[:,0]
```

This extracts the full set of rows and the column 0, or in other words, the first column of X.

Finally, we use a *list comprehension* to specify the plotting color for each data point:

```
[colors[y_i] for y_i in y]
```

This is Python syntactic sugar that allows the user to create an array while iterating over the elements of an iterator. The code snippet here is equivalent to the following code.

```
list = []
for y_i in y:
    list.append(colors[y_i])
```

5
Exploring "Data Science" via Linear Regression

So far, our treatment of machine learning has been from the perspective of a computer scientist. It is important to note, however, that models such as linear regression are useful in a variety of other fields including the physical sciences, social sciences, etc. In this chapter, we present case studies from different fields. Here, the inputs x_i are considered to be *explanatory variables*, the output y is considered to be the *effect variable*, and the weights w_i quantify the causal significance of the associated inputs x_i on the output y. The interpretation of weights as a type of causality is crucial; often, the ideal method of determining causality through a set of rigorous randomized control trials is too expensive.

5.1 Boston Housing: Machine Learning in Economics

Our first case study comes from the field of economics. In 1978, Harrison and Rubinfeld released a classic study on the willingness to pay for clean air in the Boston metropolitan area. Their methodology involved an economic model called *hedonic pricing*, [1] which essentially estimates the value of a good by breaking it down into "constituent characteristics." It turns out we can use linear regression can help determine which of these attributes are most important. Specifically, suppose we have a dataset of house sales where y represents the price of the house and $\vec{\mathbf{x}} \in \mathbb{R}^{15}$ represents a set of house attributes. [2] Then, we aim to find an optimum set of weights $\vec{\mathbf{w}}$ for the linear model:

$$y \approx \sum_{i=0}^{14} w_i x_i \tag{5.1}$$

Table 5.1 lists all 14 attributes that were used in the linear regression model. Before fitting the model with these attributes, it is useful to intuitively reason about some of the attributes. For instance, we expect the weight w_5 corresponding to RM, the number of bedrooms,

[1] This definition is paraphrased from the following Wikipedia article: https://en.wikipedia.org/wiki/Hedonic_regression

[2] x_0 is a dummy variable, and the remaining 14 coordinates $x_1, \ldots, x_{14}$ each correspond to an attribute.

Index	Code	Description
1	ZN	proportion of residential land zoned for lots over $25,000$ ft^2
2	INDUS	proportion of non-retail business acres per town
3	CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
4	NOX	nitric oxides concenteration (parts per 10 million)
5	RM	average number of rooms per dwelling
6	AGE	proportion of owner-occupied units built prior to 1940
7	DIS	weighted distances to five Boston employment centres
8	RAD	index of accessibility to radial highways
9	TAX	full-value property-tax rate per \$10,000
10	MEDV	Median value of owner-occupied homes (in \$1,000s)
11	CRIM	per capita crime rate in town
12	PTRATIO	pupil-teacher ratio by town
13	LSTAT	% lower status of the population
14	B	$1000(Bk - 0.63)^2$ where Bk is the proportion of black population in town

Table 5.1: 14 attributes used in the Boston housing regression model. The attributes are presented in a different order from the paper.

to be positive because larger houses typically sell for more. Conversely, we expect the weight w_4 corresponding to NOX, the amount of air pollution, to be negative as people would prefer not to live in a polluted environment. After running the regression, it indeed turns out that these intuitions are correct. [3] In general, it can be useful to double-check that the calculated weights align with intuition: if they do not, it could be a sign that a modeling assumption is incorrect.

[3] The regression weights can be found on page 100 of the original paper. https://deepblue.lib.umich.edu/bitstream/handle/2027.42/22636/0000186.pdf?sequence=1&isAllowed=y.

5.1.1 *The Strange Math of Feature B*

The headline result of the paper is that the willingness to pay for cleaner air increases both when income level is higher and when the current pollution level is higher. However, if you read the paper closely, you may notice the presence of a curious parameter B, which is defined in terms of Bk, the proportion of black population in the neighborhood. This parameter is meant to represent a social segregation effect present within the Boston housing market. The authors of the paper speculated that (1) at a lower level of Bk, the housing price will decrease as Bk increases since the white population tend to avoid black population, but (2) at a very high level of Bk, the housing price will increase as Bk increases because black population prefer predom-

inantly black neighborhoods. To capture this intuition, they defined the attribute B in the parabolic expression $B = 1000(Bk - 0.63)^2$. It indeed turns out that the weight w_{14} corresponding to B is positive as shown in Figure 5.1.

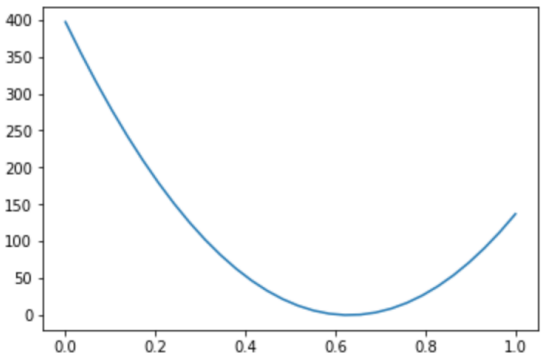


Figure 5.1: The graph of $B = 1000(Bk - 0.63)^2$. This is an example of featurization as discussed in Chapter 1. It encodes prevailing discrimination of that period. The term "black" is not favored today either.

5.1.2 *Ethnic Concerns Behind a Model*

It seems strange to have such a sensitive attribute B have an influence on the model. We might wonder about the social harm that could arise if the model was used by real-life sellers or buyers (*e.g.*, the buyers could demand a house for a lower price based on the proportion of black population in the neighborhood). On the other hand, the fitted model confirms that there is an underlying segregation effect already present in the society. Also, we cannot guarantee that the model would be race-neutral even if we eliminated the parameter B. For instance, maybe one or more of the other variables (*e.g.*, air quality variables) is highly correlated with B. [4]

[4] We will revisit such issues of bias in Chapter 16.

Ultimately, the primary takeaway from this case study is that implementing machine learning models in real life is a challenge itself. At a technical level, the model may make sense and make good predictions of house prices. But one has to consider the social effects of an ML model on the phenomenon being studied: in particular, whether it supports or extends prevailing inequities. The following are some important pointers to keep in mind:

1. If the world has a problem, the data will reflect it and *so will our models*
2. If a problematic model later gets used in real life, it can *exacerbate the existing problem*
3. The choices of attributes when making a model might *bias the outcome*

4. Carelessly using data can later *lead to modeling issues*

5.2 *fMRI Analysis: Machine Learning in Neuroscience*

We next consider an application of ML in a vastly different field. One of the most important tools in contemporary neuroscience is Functional Magnetic Resonance Imaging (fMRI). fMRI has been used successfully to map human functionality (*e.g.*, speech, memory) to brain regions. In a more active role, it can assist with tumor surgery or "decoding" thoughts and emotions.

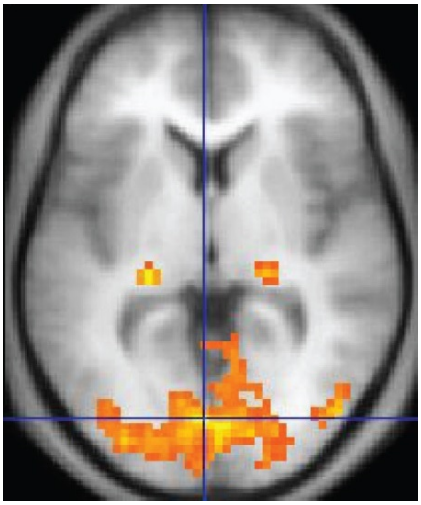


Figure 5.2: A sample image of a fMRI reading. Source: https://en.wikipedia.org/wiki/Functional_magnetic_resonance_imaging

fMRI experiments often involve presenting a set of stimuli (*e.g.*, images of human face) to the subject in order to elicit a neurological response, which is then captured through a fMRI reading. Each reading reveals the concentration of oxygen in the blood stream throughout the brain, which is used as a proxy for brain activity. [5] Through the result of the reading, we are able to conclude if a particular voxel responds to a particular stimulus. The naive way of conducting these experiments is to present one stimulus at a time and wait until we get a reading of the brain response before we move on to the next stimulus.

[5] Formally, this is referred to as the blood-oxygen-level-dependent (BOLD) signal

But if you have previously taken a course in neuroscience, you may recall that fMRI is unfortunately a double-edged sword. It features excellent spatial resolution, with each voxel as small as 1 mm^3. However, it has poor temporal resolution: often, readings require several seconds for blood flow to stabilize! Coupled with the fact that regulations limit the amount of time human subjects can spend in the scanner, it becomes clear that methodologies based on sequential presentation of stimuli are too inefficient. In this section, we explore

how to leverage techniques from linear regression in order to solve this problem.

5.2.1 *Linear Superposition*

The key intuition involves a concept called *linear superposition*: if a subject is shown multiple stimuli in quick succession, the strength of the voxel's response is the sum of the strength of its response to each of the individual stimuli. [6] Instead of waiting until we have the image of one stimulus to move on, considering showing a new stimulus every 1 or 2 seconds. Each fMRI reading will now capture the *composite* brain response to the stimuli from the past few seconds. We will use linear regression to *disentangle* the information, and extract which voxel responded to which stimulus. [7]

Consider the following example.

Example 5.2.1. *See Figure 5.3. The graph on the top left represents a voxel's response when the subject is shown the image of a face. The graph on the top right represents the response when the subject is shown the image of a flower. The bottom graph represents the response when the subject is shown the image of a flower* 1 *second after the image of a face. Notice that the first two graphs have been* ***superposed*** *to create the third graph. In practice, we are interested in the problem of extracting the individual graphs when given the superposed graph.*

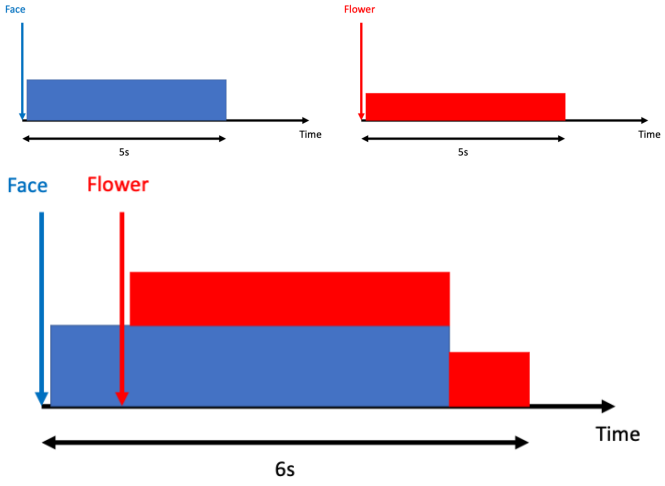


Figure 5.3: Three graphs explaining the effect of linear superposition.

[6] This is exactly like the linear superposition of wave functions in physics.

[7] Note that this is a very simplified version. The actual process is much more complicated.

5.2.2 *Linear Regression*

Now let us describe how to formulate this problem in terms of linear regression. First assume that the subject is shown one of k types of stimuli at each time step t where $t \in \{1, 2, \ldots, T\}$. Let y_t be the

response of a particular voxel at step t. The main assumption is that y_t is the linear superposition of the responses to stimuli from the steps in $[t-10, t]$. We also define a $T \times k$ matrix X with 0/1 entries, where $X_{ts} = 1$ if stimulus type s is shown during $[t-10, t]$ and 0 otherwise. Then we can set up the following linear regression model:

$$y_t \approx \sum_{s=1}^{k} w_s X_{ts}$$

When we find the optimal values of w_s via least squares, $w_s = 1$ means that the particular voxel responds to the stimulus type s.

5.2.3 *Neural Correlates of Thought*

Now we know how to find the values of w_s for a specific voxel. That is, we can test if a particular voxel responds to a particular stimulus. Combining this method with a *spatial smoothing* (*i.e.*, applying the principle that nearby voxels behave similarly), [8] we are able to identify which region of a brain is associated to which stimulus. So far, more than 1,000 regions of the brain have been identified and mapped.

[8] The simplest smoothing method is to take the w_s values for one voxel and replace them with the average of the neighboring voxels.

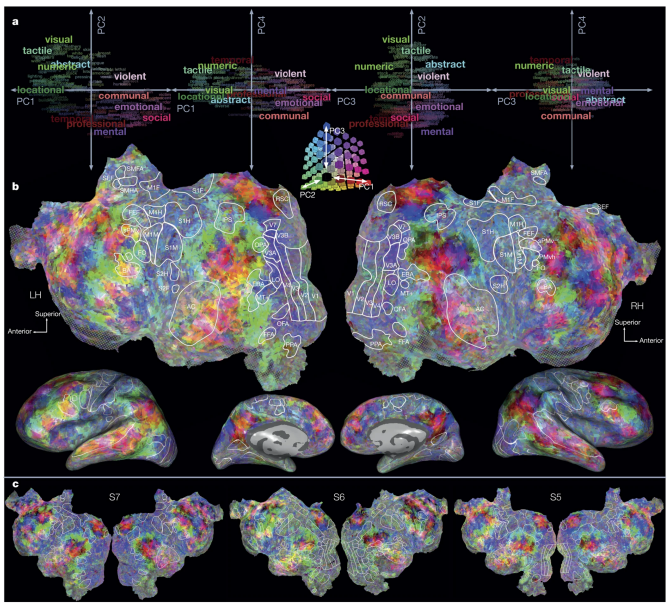


Figure 5.4: A detailed map labeling areas of the brain with corresponding stimuli. https://www.nature.com/articles/nature17637

5.2.4 *Brain-Computer Interface (BCI)*

We finish off with a tangible example of how our studies can help people. Patients who are suffering from Locked-in Syndrome (LIS) are aware of their surroundings and have normal reasoning capacities

but have *no way* of communicating with others through speech or facial movements. Using a combination of a technology called Brain-Computer Interface and a linear regression model, we are able to communicate with these patients.

Brain-Computer Interface is an electode sensor implanted near the motor cortex that can detect the electric signal that LIS patients are trying to send to the motor cortex. We can teach the patients to *visualize* writing with their dominant hand if they want to answer "no" and visualize writing with their non-dominant hand if they want to answer "yes." Since the neural correlates of the two movements are very different, BCI will pick up essentially disjoint signals, and we can use linear regression model to distinguish between them. [9]

[9] Note: training also requires labeled data, which can be produced by asking the patient questions about known facts (*e.g.*, birth date, marital status, etc.). This technique has been used to communicate with patients in deep coma and presumed to be in a vegetative state. See *Science of Mind Reading*, New Yorker, December 6 2021, which also profiles several Princeton researchers.

Part II

Unsupervised Learning

6
Clustering

So far, we have considered ML models which require labeled data in order to learn. However, there is a large class of models which can learn from *unlabeled* data. From this chapter, we will begin to introduce models from this modeling paradigm, called *unsupervised learning*. In this chapter, we focus on one application of unsupervised learning, called *clustering algorithm*.

6.1 Unsupervised Learning

Unsupervised learning is a branch of machine learning which only uses unlabeled data. Examples of unlabeled data include a text corpus containing the works of William Shakespeare (Chapter 8) or a set of unlabeled images (Chapter 7). Some key goals in this setting include:

- *Learn the structure of data:* It is possible to learn if the data consists of clusters, or if it can be represented in a lower dimension.
- *Learn the probability distribution of data:* By learning the probability distribution where the training data came from, it is possible to generate synthetic data which is "similar" to real data.
- *Learn a representation for data:* We can learn a representation that is useful in solving other tasks later. With this new representation, for example, we can reduce the need for labeled examples for classification.

6.2 Clustering

Clustering is one of the main tasks in unsupervised learning. It is the process of detecting *clusters* in the dataset. Often the membership of a cluster can replace the role of a label in the training dataset. In general, clusters reveal a lot of information about the underlying structure of the data.

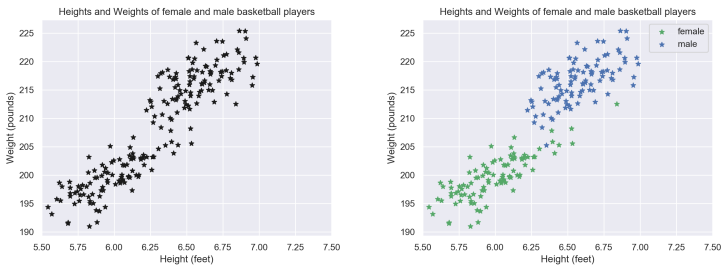


Figure 6.1: Height vs weight scatter plot of basketball players. In the plot on the right, the points in green and blue respectively correspond to female and male players.

In Figure 6.1, we see a scatter plot of measurements of height and weight of basketball players. If you look at the plot on the left, it is easy to conclude that there is a usual linear relationship between the height and the weight of the athletes. However, upon further inspection, it seems like there are two clusters of the data points, separated around the middle of the plot. In fact, this is indeed the case! If we label the dataset with the additional information of whether the data point is from a male or female athlete, the plot on the right shows something more than just the linear relationship. In practice, however, we do not always have access to this additional label. Instead, one uses clustering algorithms to find natural clusterings of the data. This raises the question of what a "clustering" is, in the first place.

Technically, any partition of the dataset $\mathcal{D}$ into k subsets $C_1, C_2, \ldots, C_k$ can be called a clustering. [1] That is,

$$\bigcup_{i=1}^{k} C_i = \mathcal{D} \quad \text{and} \quad \bigcap_{i=1}^{k} C_i = \emptyset$$

But we intuitively understand that not all partitions are a natural clustering of the dataset; our goal therefore will be to define what a "good" clustering is.

[1] Here k, the number of clusters may be given as part of the problem, or k may have to be decided upon after looking at the dataset. We'll revisit this soon.

6.2.1 *Some Attempts to Define a "Good" Cluster*

The sample data in Figure 6.1 suggests that our vision system has evolved to spot natural clusterings in two or three dimensional data. To do machine learning, however, we need a more precise definition in $\mathbb{R}^d$: specifically, for any partition of the dataset into clusters, we try to quantify the "goodness" of the clusters.

Definition 6.2.1 (Cluster: Attempt 1)**.** *A "good" **cluster** is a subset of points which are closer to each other than to all other points in the dataset.*

But this definition does not apply to the clusters in Figure 6.1. The points in the middle of the plot are far away from the points on the top right corner or the bottom left corner. So whichever cluster we assign the middle points to, they will be farther away from some

points in their assigned cluster than to some of the points on the other cluster. Ok, so that did not work. Consider the following definition.

Definition 6.2.2 (Cluster: Attempt 2)**.** *A "good" **cluster** is a subset of points which are closer to the **mean** of their own cluster than to the mean of other clusters.*

Here Mean and Variance are defined as follows:

Definition 6.2.3 (Mean and Variance of Clusters)**.** *Let C_i be one of the clusters for a dataset $\mathcal{D}$. Let $m_i = |C_i|$ denote the **cluster size**. The **mean** of the cluster C_i is*

$$\vec{\mathbf{y}}_i = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \vec{\mathbf{x}}$$

*and the **variance** within the cluster C_i is*

$$\sigma_i^2 = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$$

You may notice that Definition 6.2.2 appears to be using circular reasoning: it defines clusters using the mean of the clusters, but the mean can only be calculated once the clusters have been defined. [2]

[2] Such circular reasoning occurs in most natural formulations of clustering. Look at the Wikipedia page on clustering for some other formulations.

6.3 k-Means Clustering

In this section, we present a particular partition of the dataset called the *k-means clustering*. Given k, the desired number of clusters, the k-means clustering partitions $\mathcal{D}$ into k clusters $C_1, C_2, \ldots, C_k$ so as to minimize the cost function:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2 \tag{6.1}$$

This can be seen as minimizing the average of the individual *cost* of the k clusters, where cost of C_i is $\sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$. [3] This idea is similar in spirit to our earlier attempt in Definition 6.2.2 — we want the distance of each data point to the mean of the cluster to be small. But this method is able to circumvent the problem of circular reasoning.

[3] Notice that each cluster cost is the cluster size times the variance.

The process of finding the optimal solution for (6.1) is called the *k-means clustering problem*.

6.3.1 k-Means Algorithm

Somewhat confusingly, the most famous algorithm that is used to solve the *k-means clustering problem* is also called *k-means*. It is technically a *heuristic*, meaning it makes intuitive sense but it is not

guaranteed to find the optimum solution.[4] The following is the k-means algorithm. It is given some *initial* clustering (we discuss some choices for initialization below) and we repeat the following iteration until we can no longer improve the cost function:

[4] There is extensive research on finding near-optimal solutions to k-means. The problem is known to be NP-complete, so we believe that an algorithm that is guaranteed to produce the optimum solution on *all instances* must require exponential time.

Maintain clusters $C_1, C_2, \ldots, C_k$
For each cluster C_i, find the mean $\vec{\mathbf{y}}_i$
Initialize new clusters $C'_i \leftarrow \emptyset$
for $\vec{\mathbf{x}} \in \mathcal{D}$ **do**
  $i_x = \arg\min_i \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2$
  $C'_{i_x} \leftarrow C'_{i_x} \cup \{\vec{\mathbf{x}}\}$
end for
Update clusters $C_i \leftarrow C'_i$

At each iteration, we find the mean of each current cluster. Then for each data point, we assign it to the cluster whose mean is the closest to the point, without updating the mean of the clusters. In case there are multiple cluster means that the point is closest to, we apply the tie-breaker rule that the point gets assigned to the current cluster if it is among the closest ones; otherwise, it will be randomly assigned to one of them. Once we have assigned all points to the new clusters, we update the current set of clusters, thereby updating the mean of the clusters as well. We repeat this process until there is no point that is mis-assigned.

6.3.2 *Why Does k-Means Algorithm Terminate in Finite time?*

The k-means algorithm is actually quite akin to Gradient Descent, in the sense that the iterations are trying to improve the cost.

Lemma 6.3.1. *Given a set of points $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \ldots, \vec{\mathbf{x}}_m$, their mean $\vec{\mathbf{y}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$ is the point that minimizes the average squared distance to the points.*

Proof. For any vector $\vec{\mathbf{z}}$, let $C(\vec{\mathbf{z}})$ denote the sum of squared distance to the set of points. That is,

$$\begin{aligned} C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} \|\vec{\mathbf{z}} - \vec{\mathbf{x}}_i\|_2^2 &= \sum_{i=1}^{m} ((\vec{\mathbf{z}} - \vec{\mathbf{x}}_i) \cdot (\vec{\mathbf{z}} - \vec{\mathbf{x}}_i)) \\ &= \sum_{i=1}^{m} (\vec{\mathbf{z}} \cdot \vec{\mathbf{z}} - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \vec{\mathbf{x}}_i \cdot \vec{\mathbf{x}}_i) \\ &= \sum_{i=1}^{m} (\|\vec{\mathbf{z}}\|_2^2 - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \|\vec{\mathbf{x}}_i\|_2^2) \end{aligned}$$

To find the optimal $\vec{\mathbf{z}}$, we set the gradient ∇C to 0

$$\nabla C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} (2\vec{\mathbf{z}} - 2\vec{\mathbf{x}}_i) = 0$$

which yields the solution

$$\vec{\mathbf{z}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$$

□

We are ready to prove the main result.

Theorem 6.3.2. *Each iteration of the k-means Algorithm 6.3.1, possibly except for the last iteration before termination, strictly decreases the total cluster cost (6.1).*

Proof. We follow the same notation as in Algorithm 6.3.1. The total cost at the end of one iteration is:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2$$

where $\vec{\mathbf{y}}_i'$ is the mean of the newly defined cluster C_i'. Notice that each of the cluster cost $\sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2$ is the sum of the squared distance between a set of points $\vec{\mathbf{x}} \in C_i'$ and their mean. By Lemma 6.3.1, this sum is smaller than the sum of squared distance between the same set of points to any other point. In particular, we can compare with $\vec{\mathbf{y}}_i$, the mean of C_i before the update. That is,

$$\sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2 \leq \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$$

for any $1 \leq i \leq k$. If we sum over all clusters, we see that

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2 \leq \sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$$

Now notice that the summand $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$ in the right hand side of the inequality is the squared distance between the point $\vec{\mathbf{x}}$ and the mean $\vec{\mathbf{y}}_i$ (before update) of the cluster C_i' that $\vec{\mathbf{x}}$ is newly assigned to. In other words, we can rewrite this term as $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2$ and instead sum over all points $\vec{\mathbf{x}}$ in the dataset. That is,

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2 = \sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2$$

Finally, recall that the index i_x was defined as $i_x = \arg\min_i \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2$. In particular, if j was the index of the cluster that a data point $\vec{\mathbf{x}}$ originally belonged to, then $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2 \leq \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_j \right\|_2^2$. Therefore, we have the following inequality,

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2 \leq \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_j \right\|_2^2$$

The equality holds in the inequality above if and only if when $\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 = \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$ for each point $\vec{\mathbf{x}}$, which means that the original cluster C_j was one of the closest clusters to $\vec{\mathbf{x}}$. By the tie-breaker rule, i_x would have been set to j. This is exactly the case when the algorithm terminates immediately after this iteration since no point is reassigned to a different cluster. In all other cases, we have a strict inequality:

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 < \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$$

Notice that the right hand side of the inequality is the total cost at the beginning of the iteration. □

Now we are ready to prove that the k-means algorithm is guaranteed to terminate in finite time. Since each iteration strictly reduces the cost, we conclude that the current clustering (*i.e.*, partition) will never be considered again, except at the last iteration when the algorithm terminates. Since there is only a finite number of possible partitions of the dataset $\mathcal{D}$, the algorithm must terminate in finite time.

6.3.3 *k-Means Algorithm and Digit Classification*

You might be familiar with the MNIST hand-written digits dataset. Here, each image, which depicts some digit between 0 and 9, is represented as a an 8×8 matrix of pixels and each pixel can take on a different luminosity value from 0 to 15.

We can apply k-means clustering to differentiate between images depicting the digit "1" and the digit "0." After running the model with $k = 2$ on 360 images of the two digits, we achieve the clusters in Figure 6.2. [5] Note the presence of two colored regions: a point is colored red if a hypothetical held-out data point at that location would get assigned a "0;" otherwise it is colored blue. This assignment is based on which cluster center is closer.

[5] This 2D visualization of the clusters is achieved through a technique called low dimensional representation, which is covered in Chapter 7.

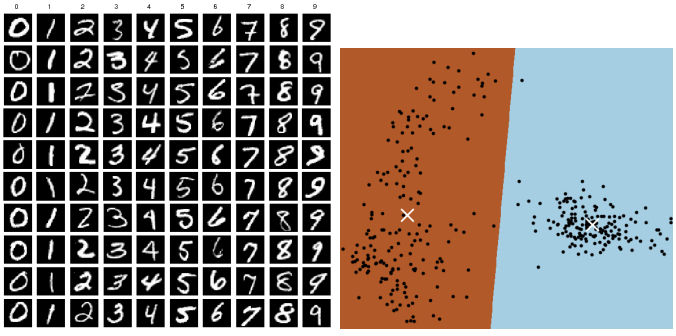


Figure 6.2: Sample images from the MNIST dataset (left) and 2D visualization of the k-means clusters differentiating between the digits "1" and "0" (right). Only two images were misclassified!

This example also shows that clustering into two clusters can be turned into a technique for binary classification — use training data to come up with two clusters; at test time, compute a ± 1 label for each data point according to which of the two cluster centers it is closer to.

6.3.4 *Implementation Detail: How to Pick the Initial Clustering*

The choice of initial clusters greatly influences the quality of the solution found by the k-means algorithm. The most naive method is to pick k data points randomly to serve as the initial cluster centers and create k clusters by assigning each data point to the closest cluster center. However, this approach can be problematic. Suppose there exists some "ground truth" clustering of the dataset. By picking the initial clusters randomly, we may end up splitting one of these ground truth clusters (*e.g.*, two initial centers are drawn from within the same ground truth cluster), and the final clustering ends up being very sub-optimal. Thus one tries to select the initial clustering more intelligently. For instance the popular k-means++ initialization procedure [6] is the following:

[6] It was invented by Arthur and Vassilvitskii in 2007.

1. Choose one center uniformly at random among all data points.
2. For each data point $\vec{\mathbf{x}}$ compute $D(\vec{\mathbf{x}})$, the distance between $\vec{\mathbf{x}}$ and the nearest center which has already been chosen.
3. Choose a new data point at random as a new center, where a point $\vec{\mathbf{x}}$ is chosen with probability proportional to $D(\vec{\mathbf{x}})^2$.
4. Repeat steps 2 and 3 until k centers have been chosen.

In COS 324, we will not expect you to understand why this is a good initialization procedure, but you may be expected to be able to implement this or similar procedures in code.

6.3.5 *Implementation Detail: Choice of k*

Above we assumed that the number of clusters k is given, but in practice you have to choose the appropriate number of clusters k.

Example 6.3.3. *Is there a value of k that guarantees an optimum cost of 0? Yes! Just set $k = n$ (i.e., each point is its own cluster). Of course, this is useless from a modeling standpoint!*

Problem 6.3.4. *Argue that the optimum cost for $k + 1$ clusters is no more than the optimum cost for k clusters.*

Note that Problem 6.3.4 only concerns the optimum cost, which as we discussed may not be attained by the k-means algorithm. Nevertheless, it does suggest that we can try various values of k and see when the cost is low enough to be acceptable.

A frequent heuristic is the *elbow method*: create a plot of the number of clusters vs. the final value of the cost as in Figure 6.3 and look for an "elbow" where the objective tapers off. Note that if the dataset is too complicated for a simple Euclidean distance cost, the data might not be easy to cluster "nicely" meaning there is no "elbow" shown on the plot.

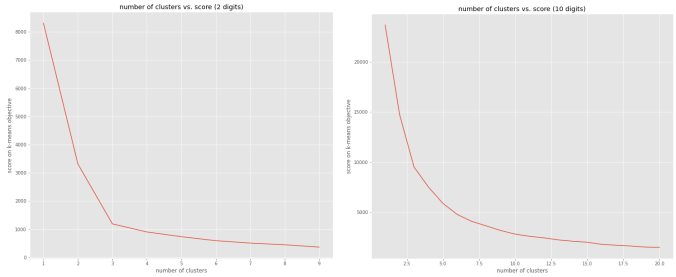


Figure 6.3: Two graphs of number of clusters vs. final value of cost. There is a distinct elbow on the left, but not on the right.

6.4 *Clustering in Programming*

In this section, we briefly discuss how to implement k-means algorithm for digit classification in Python. As usual, we use the *numpy* package to speed up computation and the *matplotlib* package for visualization. Additionally, we use the *sklearn* package to help perform the clustering.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA

# prepare dataset
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...

# define functions
def initialize_cluster_mean(X, k):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # returns Y: array of shape (k, d), each row is the center of a cluster

def assign_cluster(X, Y)
    # X: array of shape (n, d), each row is a d-dimensional data point
    # Y: array of shape (k, d), each row is the center of a cluster
    # returns loss, the sum of squared distance from each point to its
                                        assigned cluster
```

```
    # returns C: array of shape (n), each value is the index of the closest
                                        cluster

def update_cluster_mean(X, k, C):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # C: array of shape (n), each value is the index of the closest cluster
    # returns Y: array of shape (k, d), each row is the center of a cluster

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y

def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()

# run k-means algorithm and plot the result
loss, C, Y = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

We start by importing outside packages.

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
```

The *load_digits()* method loads the MNIST digits dataset, with around 180 data points per digit. The *scale()* method linearly scales each of the data points such that the mean is 0 and variance is 1. The *PCA()* method helps visualize the MNIST digits data points, which are 64-dimensional, in the Cartesian plane (*i.e.*, $\mathbb{R}^2$). See the next Chapter 7 for details on this process.

Next we prepare the dataset by calling the *load_digits()* method.

```
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...
```

Notice that we discard the target array y because we are performing clustering, a type of unsupervised learning. If we were to perform supervised learning instead, we would need to make use of y.

Then we define the functions necessary for the k-means algorithm.

```
def initialize_cluster_mean(X, k):
    return Y

def assign_cluster(X, Y)
    return loss, C

def update_cluster_mean(X, k, C):
    return Y

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y
```

In practice, it is common to limit the number of cluster update iterations (*i.e.*, the parameter *max_iters*) and specify the smallest amount of loss change allowed for one iteration (*i.e.*, the constant ϵ). By terminating the algorithm once either one of the conditions is reached, we can get an approximate solution within a reasonable amount of time.

Next, take a look at the helper function used to plot the result of the k-means algorithm.

```
def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()
```

The *cycle()* method from the *itertools* package lets you iterate through an array indefinitely, with the index wrapping around back to the start, once it reaches the end of the array.

Now, consider the *for* loop section in the helper function above. We first use *Boolean* conditions to concisely generate a new array.

```
idx = (C == i)
```

This generates a *Boolean* array with the same length as C, where each entry is either *True/False* based on whether the corresponding entry in C is equal to i. The following code is equivalent.

```
idx = np.zeros(C.size)
for j in range(C.size):
```

```
    idx[j] = (C[j] == i)
```

We then use a technique called *Boolean masking* to extract a particular subset of rows of X.

```
X[idx, 0]
```

Notice that in place of a list of indices of rows to extract, we are indexing with the *Boolean* array we just defined. The code will extract only the rows where the *Boolean* value is *True*. For example, if the value of *idx* is *[True, False, True]*, then the code above is equivalent to

```
X[[0, 2], 0]
```

Finally, we make use of the helper functions we defined earlier to run the k-means algorithm and plot results.

```
_, C, _ = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

The first line of this code snippet shows how we can use the _ symbol to selectively disregard individual return values of a function call. The second line of code uses the *PCA()* method to transform the 64-dimensional data *X_train* into 2-dimensional data so that we can visualize it with the *scatter_plot()* method. We will learn the details of this process in the next Chapter 7.

7
Low-Dimensional Representation

High-dimensional datasets arise in quite a few settings. This chapter concerns a phenomenon that arises frequently: the data points (*i.e.*, vectors) collectively turn out to be "approximately low rank." A running theme in this chapter is that arrays and matrices, which in introductory courses like COS 126 and COS 226 were thought of as data structures (*i.e.*, an abstraction from programming languages), are treated now as objects that we can pass through some remarkable (but simple) mathematical procedures.

If a large dataset of N vectors in $\mathbb{R}^d$ has rank k, then we can think of a natural compression method. Let U be the k-dimensional subspace spanned by the vectors, and identify k basis vectors for U. For each of the N vectors, find k coefficients of their representation in terms of the basis vectors. Following this method, instead of specifying the N vectors using Nd real numbers, we can represent them using $k(N+d)$ real numbers, which is a big win if d is much larger than k.

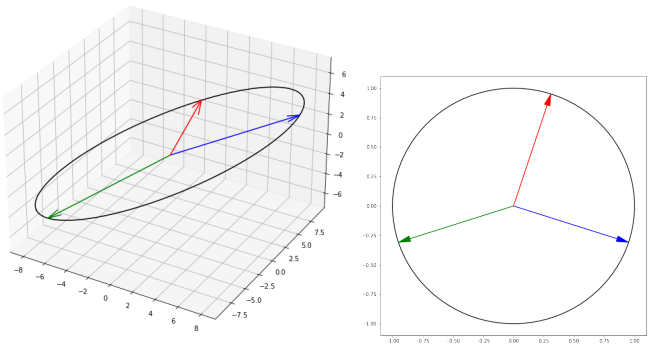


Figure 7.1: $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3 \in \mathbb{R}^3$ (left) and their 2-dimensional representations $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \widehat{\vec{\mathbf{v}}}_3 \in \mathbb{R}^2$.

Example 7.0.1. *Figure 7.1 shows three vectors* $\vec{\mathbf{v}}_1 = (3.42, -1.33, 6.94), \vec{\mathbf{v}}_2 = (7.30, 8.84, 1.95), \vec{\mathbf{v}}_3 = (-7.92, -6.37, -5.66)$ *in* $\mathbb{R}^3$. *The three vectors have rank* 2 *— they are all in the* 2*-dimensional linear subspace generated*

by $\vec{\mathbf{u}}_1 = (8, 8, 4)$ *and* $\vec{\mathbf{u}}_2 = (1, -4, 6)$. *Specifically,*

$$\vec{\mathbf{v}}_1 = 0.31\vec{\mathbf{u}}_1 + 0.95\vec{\mathbf{u}}_2$$
$$\vec{\mathbf{v}}_2 = 0.95\vec{\mathbf{u}}_1 - 0.31\vec{\mathbf{u}}_2$$
$$\vec{\mathbf{v}}_3 = -0.95\vec{\mathbf{u}}_1 - 0.31\vec{\mathbf{u}}_2$$

Therefore, we can represent these vectors in a 2*-dimensional plane, as* $\widehat{\vec{\mathbf{v}}}_1 = (0.31, 0.95), \widehat{\vec{\mathbf{v}}}_2 = (0.95, -0.31), \widehat{\vec{\mathbf{v}}}_3 = (-0.95, -0.31)$

7.1 Low-Dimensional Representation with Error

Of course, in general, high dimensional datasets are not exactly low rank. We're interested in datasets which have low-dimension representations *once we allow some error*.

Definition 7.1.1 (Low-dimensional Representation with Error)**.** *We say a set of vectors* $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_N \in \mathbb{R}^d$ *has* ***rank*** k ***with mean-squared error*** ϵ *if there exist some basis vectors* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k \in \mathbb{R}^d$ *and* N *vectors* $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \ldots, \widehat{\vec{\mathbf{v}}}_N \in \mathit{span}(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k)$ *such that*

$$\frac{1}{N}\sum_i \left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2 \leq \epsilon \tag{7.1}$$

We say that $\widehat{\vec{\mathbf{v}}}_1, \ldots, \widehat{\vec{\mathbf{v}}}_N$ *are the* ***low-rank*** *or* ***low-dimensional*** *approximation of* $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_N$*. Typically we will assume without loss of generality that the basis vectors are orthonormal (i.e., have* ℓ_2 *norm equal to* 1 *and are pairwise orthogonal).*

Definition 7.1.1 can be thought of as a *lossy* compression of the dataset of vectors since the low-dimensional representation of vectors is roughly correct, but with a bound of ϵ on the MSE. This compression view will be used in Section 7.3.

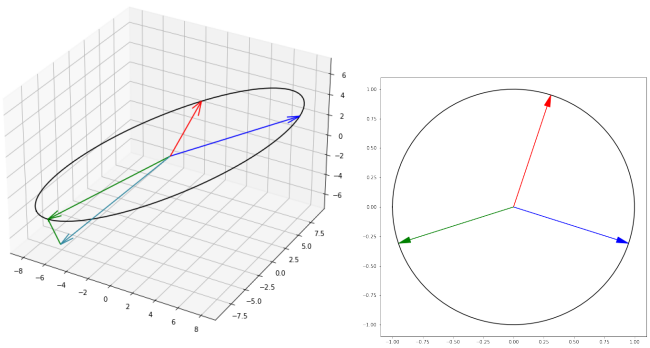


Figure 7.2: $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3 \in \mathbb{R}^3$ (left) and their 2-dimensional approximations $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \widehat{\vec{\mathbf{v}}}_3$ represented in the 2-dimensional subspace spanned by $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2$.

Example 7.1.2. *Figure 7.2 shows three vectors* $\vec{\mathbf{v}}_1 = (3.42, -1.33, 6.94), \vec{\mathbf{v}}_2 = (7.30, 8.84, 1.95), \vec{\mathbf{v}}_3 = (-6.00, -7.69, -6.86)$ *in* $\mathbb{R}^3$*. The three vectors have rank* 2 *with mean-squared error 2.5. If you choose the basis vectors* $\vec{\mathbf{u}}_1 = (8, 8, 4), \vec{\mathbf{u}}_2 = (1, -4, 6)$ *and the low-rank approximations* $\widehat{\vec{\mathbf{v}}}_1 = \vec{\mathbf{v}}_1, \widehat{\vec{\mathbf{v}}}_2 = \vec{\mathbf{v}}_2, \widehat{\vec{\mathbf{v}}}_3 = (-7.92, -6.37, -5.66) \in span(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2)$ *then,*

$$\frac{1}{3}\sum_i \left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2 \simeq 2.28 \leq 2.5$$

Note that the basis vectors in this example are only orthogonal and not orthonormal, but it is easy to set them as orthonormal by normalizing them.

Problem 7.1.3. *Show that if* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k \in \mathbb{R}^d$ *is any set of orthonormal vectors and* $\vec{\mathbf{v}} \in \mathbb{R}^d$ *then the vector* $\widehat{\vec{\mathbf{v}}}$ *in* $span(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k)$ *that minimizes* $\left\|\vec{\mathbf{v}} - \widehat{\vec{\mathbf{v}}}\right\|_2^2$ *is*

$$\sum_{j=1}^{k} (\vec{\mathbf{v}} \cdot \vec{\mathbf{u}}_j)\vec{\mathbf{u}}_j \tag{7.2}$$

(Hint: If $\alpha_1, \alpha_2, \ldots, \alpha_k$ *minimize* $\left\|\vec{\mathbf{v}} - \sum_j \alpha_j \vec{\mathbf{u}}_j\right\|_2^2$ *then the gradient of this expression with respect to* $\alpha_1, \alpha_2, \ldots, \alpha_k$ *must be zero.)*

Problem 7.1.3 illustrates how to find the low-dimensional representation of the vectors, once we specify the k basis vectors. Notice that (7.2) is the *vector projection* of $\vec{\mathbf{v}}$ onto the subspace U spanned by the vectors $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k$. Therefore, the approximation error $\left\|\vec{\mathbf{v}} - \widehat{\vec{\mathbf{v}}}\right\|_2^2$ is the squared norm of the component of $\vec{\mathbf{v}}$ that is orthogonal to U. [1]

[1] Also known as the *vector rejection* of $\vec{\mathbf{v}}$ from U.

Problem 7.1.4 is only for more advanced students but all students should read its statement to understand the main point. It highlights how miraculous it is that real-life datasets have low-rank representations. It shows that generically one would expect ϵ in (7.1) to be $1 - k/n$, which is almost 1 when $k \ll n$. And yet in real life ϵ is small for fairly tiny k.

Problem 7.1.4. *Suppose the* $\vec{\mathbf{v}}_i$*'s are unit vectors* [2] *and the vectors* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k$ *were the basis vectors of a random k-dimensional subspace in* $\mathbb{R}^d$*. (That is, chosen without regard to the* $\vec{\mathbf{v}}_i$*'s.) Heuristically argue that the* ϵ *one would need in* (7.1) *would be* $1 - k/n$*.*

[2] Note: the maximum possible value of ϵ when $\vec{\mathbf{v}}_i$'s are unit vectors is 1. Convince yourself!

7.1.1 Computing the Low-Dimensional Representation with Error

In Problem 7.1.3, we have already seen how to find the low-dimension representation with error, once we are given the basis vectors. All there remains is to identify a suitable value of k and find the corresponding basis vectors that will minimize the error.

There is a simple linear algebraic procedure, the *Singular Value Decomposition (SVD)*. Given a set of vectors $\vec{\mathbf{v}}_i$ and a positive integer

k, SVD can output the best orthonormal basis in sense of Definition 7.1.1 that has the lowest possible value of ϵ. In practice, we treat k as a hyperparameter and use trial and error to find the most suitable k. Problem 7.1.5 shows that the accuracy of the low-dimensional representation will decrease when we choose a smaller number of dimensions. So we are making a choice between the accuracy of the representations against how condensed our compression is.

Problem 7.1.5. *Show that as we decrease k in Definition 7.1.1, the corresponding ϵ can only increase (i.e., cannot decrease).*

Formally, SVD takes a matrix as its input; the rows of this matrix are the vector $\vec{\mathbf{v}}_i$'s. The procedure operates on this matrix to output a low-rank approximation. We discuss details in Section 20.3. To follow the rest of this chapter, you do not need to understand details of the procedure. You just to remember the fact that the best k-dimensional representation is computable for each k. In practice, programming languages have packages that will do the calculations for you. Below is a Python code snippet that will calcuate the SVD.

```
import sklearn.decomposition.TruncatedSVD

# n * n matrix
data = ...
# prepare transform on dataset matrix "data"
svd = TruncatedSVD(n_components=k)
svd.fit(data)
# apply transform to dataset and output a n * k matrix
transformed = svd.transform(data)
```

Now we see some fun applications.

7.2 *Application 1: Stylometry*

In many cases in old literature, the identity of the author is disputed. For instance, the King James Bible (*i.e.*, the canonical English bible from the 17th century) was written by a team whose identities and work divisions are not completely known. Similarly the *Federalist Papers*, an important series of papers explicating finer points of the US government and constitution, were published in the early days of the republic with the team of authors listed as Alexander Hamilton, James Madison, and John Jay. But it was not revealed which paper was written by whom. In such cases, can machine learning help identify who wrote what?

Here we present a fun example about the books in the Wizard of Oz series. [3] L. Frank Baum was the author of the original *Wonderful Wizard of Oz*, which was a best-seller in its day and remains highly popular to this day. The publisher saw a money-making opportunity

[3] Original paper at http://dh.obdurodon.org/Binongo-Chance.pdf. A survey paper by Erica Klarreich in Science News Dec 2003: *Statistical tests are unraveling knotty literary mysteries* at http://web.mit.edu/allanmc/www/stylometrics.pdf

and convinced Baum to also write 15 follow-up books. After Baum's death the publisher managed to pass on the franchise to Ruth Plumly Thompson, who wrote many other books.

However, the last of the Baum books, *Royal Book of Oz* (RBOO), always seemed to Oz readers closer in style to Thompson's books than to Baum's. But with all the principals in the story now dead, there seemed to be no way to confirm the suspicion. Now we describe how simple machine learning showed pretty definitively that this book was indeed written by Ruth Plumly Thompson. The main idea is to represent the books vectors in some way and then find their low-rank representations.

Figure 7.3: *Royal Book of Oz* (1921). Cover image from https://en.wikipedia.org/wiki/The_Royal_Book_of_Oz

The key idea is that different authors use English words at different frequencies. Surprisingly, the greatest difference lies in frequencies of *function words* such as WITH, HOWEVER, UPON, rather than fancy vocabulary words (the ones found on your SAT exam).

Example 7.2.1. *Turns out Alexander Hamilton used* UPON *about* 10 *times more frequently than James Madison. We know this from analyzing their individual writing outside their collaboration on the Federalist Papers. Using these kinds of statistics, it has been determined that Hamilton was the principal author or even the sole author of almost all of the Federalist Papers.*

The statistical analysis of the Oz books consisted of looking at the frequencies of 50 function words. All Oz books except RBOO were divided into text blocks of 5000 words each. For each text block, the frequency (*i.e.*, number of occurrences) of each function word was computed, which allows us to represent the block as a vector in $\mathbb{R}^{50}$. There were 223 text blocks total, so we obtain 223 vectors in $\mathbb{R}^{50}$.

the (6.7%)	with (0.7%)	up (0.3%)	into (0.2%)	just (0.2%)
and (3.7%)	but (0.7%)	no (0.3%)	now (0.2%)	very (0.2%)
to (2.6%)	for (0.7%)	out (0.3%)	down (0.2%)	where (0.2%)
a/an (2.3%)	at (0.6%)	what (0.3%)	over (0.2%)	before (0.2%)
of (2.1%)	this/these (0.5%)	then (0.3%)	back (0.2%)	upon (0.1%)
in (1.3%)	so (0.5%)	if (0.3%)	or (0.2%)	about (0.1%)
that/those (1.0%)	all (0.5%)	there (0.3%)	well (0.2%)	after (0.1%)
it (1.0%)	on (0.5%)	by (0.3%)	which (0.2%)	more (0.1%)
not (0.9%)	from (0.4%)	who (0.3%)	how (0.2%)	why (0.1%)
as (0.7%)	one/ones (0.3%)	when (0.2%)	here (0.2%)	some (0.1%)

Figure 7.4: The top 50 most frequently used function words in the Wizard of Oz series. Their occurrences were counted in 223 text blocks. Figure from Binongo's paper.

Then we compute a rank 2 approximation of these 223 vectors.

Figure 7.5 shows the scatter plot in the 2-dimensional visualization. The two axes correspond to the two basis vectors we found for the rank 2 approximation. It becomes quickly clear that the vectors from the Baum books are in a different part of the space than those from the Thompson books. It is also clear that RBOO vectors fall in the

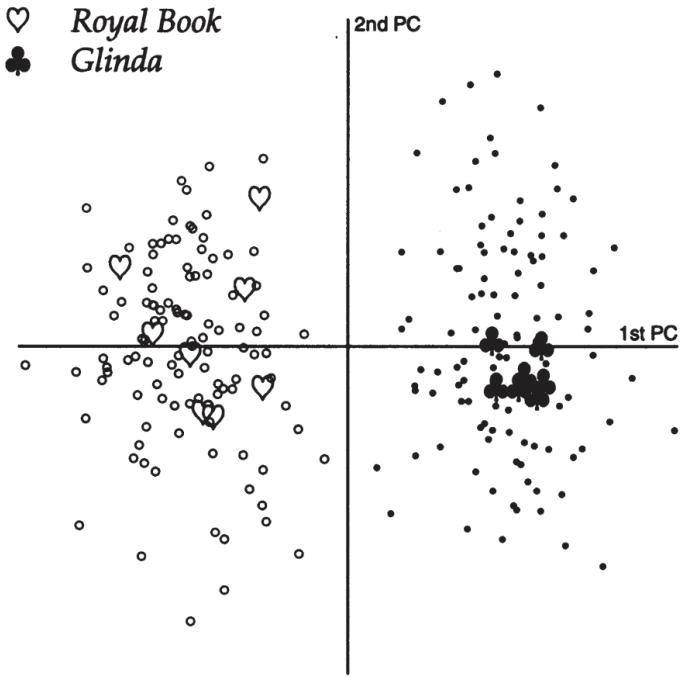


Figure 7.5: Rank-2 visualization of the 223 text block vectors from books of Oz. The dots on the left correspond to vectors from Oz books known to be written by Ruth Plumly Thompson. The hearts on the left correspond to vectors from RBOO. The ones on the right correspond to ones written by L. Frank Baum. Figure from Binongo's paper.

same place as those from other Thompson books. Conclusion: *Ruth Plumly Thompson was the true author of Royal Book of Oz!*

By the way, if one takes the non-Oz writings of Baum and Thompson and plot their vectors in the 2D-visualization in Figure 7.6, they also fall on the appropriate side. So the difference in writing style came across clearly even in non-Oz books!

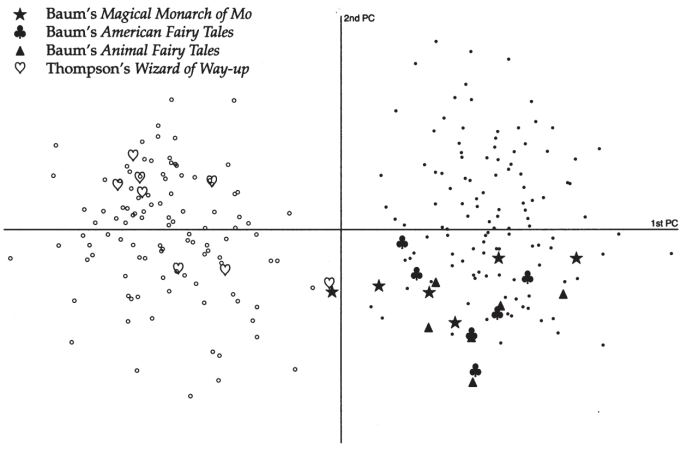


Figure 7.6: Rank-2 visualization of text block vectors from books written by Baum and Thompson outside of the Oz series. Figure from Binongo's paper.

7.3 *Application 2: Eigenfaces*

This section uses the *lossy compression* viewpoint of low-rank representations. As you may remember from earlier computer science courses (*e.g.*, Seam Carver from COS 226), images are vectors of pixel values. In this section, let us only consider grayscale (*i.e.*, B&W) images where each pixel has an integer value in $[-122, 122]$. -122 corresponds to the pixel being pitch black; 0 corresponds to middle gray; and 122 corresponds to total white. We can also reorganize the entries to form a single vector:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \rightarrow (a_{11}, a_{12}, \cdots, a_{1n}, a_{21}, \cdots, a_{2n}, \cdots, a_{mn})$$

Once we have this vectorized form of an image, it is possible to perform linear algebraic operations on the vectors. For example, we can take 0.3 times the first image and add it to -0.8 times the second image, etc. See Figure 7.7 for some of these examples.

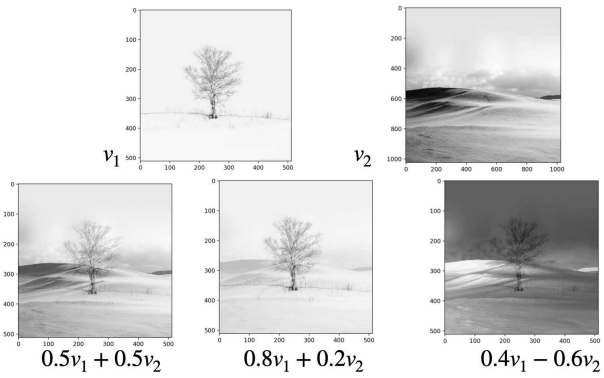


Figure 7.7: Example of linear algebra on images.

Eigenfaces was an idea for face recognition [4]. The dataset in this lecture is from a classic Olivetti dataset from 1990s. Researchers took images of people facing the camera in good light, downsized to 64×64 pixels. This makes them vectors in $\mathbb{R}^{4096}$. Now we can find a 64-rank approximation of the vectors using procedures we will explore more in detail in Section 20.3.

Figure 7.8 shows four basis vectors in the low-rank approximation of the images. The first image looks like a generic human with a ill-defined nose and lips; the second image looks like having glasses and a wider nose; the third image potentially looks like a female face; the fourth image looks like having glasses, a moustache, and a beard. All images in the dataset can be approximated as a linear

[4] L. Sirovich; M. Kirby (1987). *Low-dimensional procedure for the characterization of human faces*. Journal of the Optical Society of America.

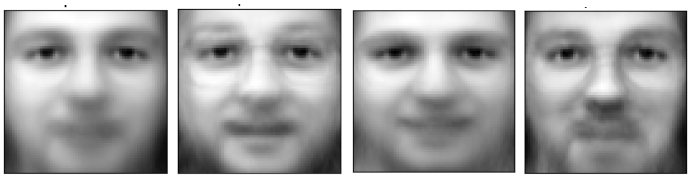


Figure 7.8: Some basis vectors (which turn out to be face-like) in the low-rank approximation of the Olivetti dataset.

combination of 128 of these basis images, and the approximations are surprisingly accurate. Figure 7.9 shows four original images of the dataset, compared with their 64-rank approximations and 128-rank approximations.

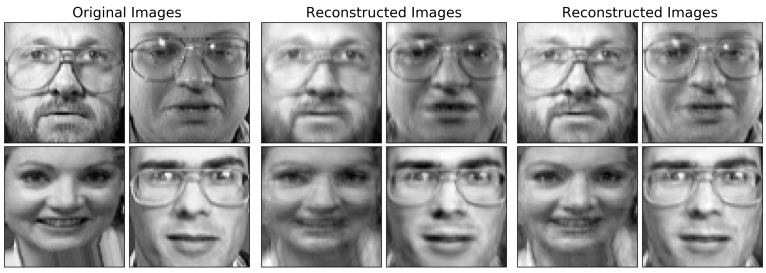


Figure 7.9: 4 original images in the Olivetti dataset (left), compared with their 64-rank approximations (middle) and 128-rank approximations (right).

From Figure 7.9, we also see that the approximations are more accurate when the corresponding value of k is larger. In fact, Figure 7.10 shows the average value of $\frac{\left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2}{\left\|\vec{\mathbf{v}}_i\right\|_2^2}$ as a function of the rank of the approximation. Note that this value roughly represents the *fraction of* $\vec{\mathbf{v}}$ *lost in the approximation*. It can be seen that the error is a decreasing function in terms of k. [5] However, note that doing machine learning — specifically face recognition — on low-rank representations is computationally more efficient particularly because the images are compressed to a lower dimension. With a smaller value of k, we can improve the speed of the learning.

[5] This was also explored in Problem 7.1.5

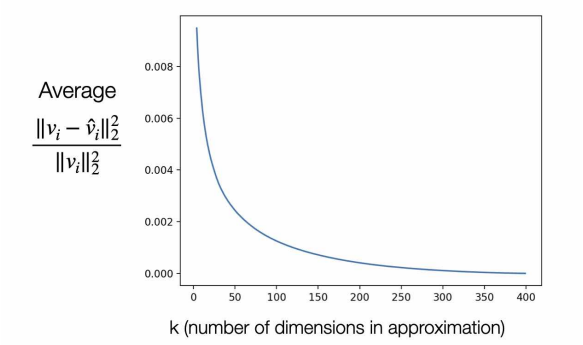


Figure 7.10: What fraction of norm of the image is not captured in the low-dimensional representation, plotted versus the rank k.

8
n-Gram Language Models

In this chapter, we continue our investigation into unsupervised learning techniques and now turn our attention to language models. You may have heard of natural language processing (NLP) and models such as GPT-3 in the news lately. The latter is quite impressive, being able to write and publish its own opinion article on a reputable news website! [1] While most of these models are trained using state-of-the-art deep learning techniques which we will discuss later on in this text, this chapter explores a key idea, which is to view language as the output of a probabilistic process, which leads to an interesting measure of the "goodness" of the model. Specifically, we will investigate the so-called *n-gram language model*.

[1] The full piece can be found at https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3

8.1 Probabilistic Model of Language

Classical linguistics focused on the syntax or the formal grammar of languages. The linguists believed that a language can be modeled by a set of sentences, constructed from a finite set of vocabularies and a finite set of grammatical rules. But this approach in language modeling had limited success in machine learning.

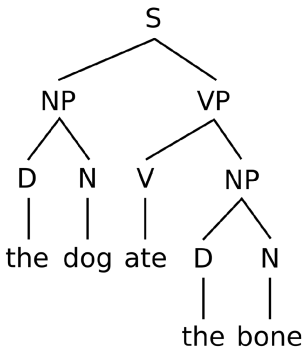


Figure 8.1: An example of a syntax tree of an English sentence.

Instead, the approach of machine learning in language model,

pioneered by Claude Shannon, has been to learn the *distribution* of pieces of text.

In other words, the model assigns a probability to all conceivable finite pieces of English text (even those that have not yet been spoken or written). For example, the sentence "how can I help you" will be assigned some probability, most likely larger than the probability assigned to the sentence "can I how you help." Note that we don't expect to find a "correct" model; all models found to date are approximations. But even an approximate probabilistic model can have interesting uses, such as the following:

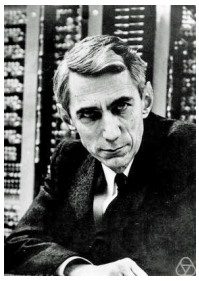


Figure 8.2: Claude Shannon, inventor of the n-gram language model in https://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf. Picture from https://en.wikipedia.org/wiki/Claude_Shannon.

1. *Speech recognition:* A machine processes a recording of a human speech that sounds somewhere between "I ate a cherry" and "eye eight a Jerry." If the model assigns a higher probability score to the former, speech recognition can still work in this instance.

2. *Machine translation:* "High winds tonight" should be considered a better translation than "large winds tonight."

3. *Context sensitive spelling correction:* We can compare the probabilities of sentences that are similar to the following sentence — "Their are problems wit this sentence." — and output the corrected version of the sentence.

4. *Sentence completion:* We can compare the probabilities of sentences that will complete the following phrase — "Please turn off your ..." — and output the one with the highest probability.

8.2 *n-Gram Models*

Say we are in the middle of the process of assigning a probability distribution over all English sentences of length 5. We want to find the probability of the sentence "I love you so much." If we let X_i be the random variable that takes the value of the i-th word, the probability we are looking for is the joint probability

$$\Pr[X_1 = "I", X_2 = "love", X_3 = "you", X_4 = "so", X_5 = "much"] \quad (8.1)$$

By Chain Rule, we can split this joint probability into the product of a marginal probability and four conditional probabilities:

$$\begin{aligned}(8.1) = &\Pr[X_1 = "I"] \\ &\times \Pr[X_2 = "love" \mid X_1 = "I"] \\ &\times \Pr[X_3 = "you" \mid X_1 = "I", X_2 = "love"] \\ &\times \Pr[X_4 = "so" \mid X_1 = "I", X_2 = "love", X_3 = "you"] \\ &\times \Pr[X_5 = "much" \mid X_1 = "I", X_2 = "love", X_3 = "you", X_4 = "so"]\end{aligned} \quad (8.2)$$

If we estimate all components of the product in (8.2), we will be able to estimate the joint probability (8.1).

Now consider the *bigram model*, which has the following two assumptions:

1. The probability of a word is only dependent on the immediately previous word.
2. That probability does not depend on the position of the word in the sentence.

The first assumption says that, for example, the conditional probability

$$\Pr[X_3 = \text{"}you\text{"} \mid X_1 = \text{"}I\text{"}, X_2 = \text{"}love\text{"}]$$

can be simplified as

$$\Pr[X_3 = \text{"}you\text{"} \mid X_2 = \text{"}love\text{"}]$$

The second assumption says that

$$\Pr[X_3 = \text{"}you\text{"} \mid X_2 = \text{"}love\text{"}] = \Pr[X_{i+1} = \text{"}you\text{"} \mid X_i = \text{"}love\text{"}]$$

for any $1 \leq i \leq 5$. We abuse notation and denote any of these probabilities as $\Pr[\text{"}you\text{"} \mid \text{"}love\text{"}]$.

Applying these assumptions to (8.2), we can simplify it as

$$\begin{aligned}(8.1) = \Pr[\text{"}I\text{"}] &\times \Pr[\text{"}love\text{"} \mid \text{"}I\text{"}] \times \Pr[\text{"}you\text{"} \mid \text{"}love\text{"}] \\ &\times \Pr[\text{"}so\text{"} \mid \text{"}you\text{"}] \times \Pr[\text{"}much\text{"} \mid \text{"}so\text{"}] \end{aligned} \tag{8.3}$$

Now we are going to estimate each component of (8.3) from a large corpus of text. The estimation for the marginal probability of the word "I" is given as

$$\Pr[\text{"}I\text{"}] \approx \frac{\text{Count("I")}}{\text{total number of words}} \tag{8.4}$$

where Count refers to the number of occurrences of the substring in the text. In other words, this is the proportion of the occurrence of the word "I" in the entire corpus. Similarly, we can estimate the conditional probability of the word "love" given its previous word is "I" as

$$\Pr[\text{"}love\text{"} \mid \text{"}I\text{"}] \approx \frac{\text{Count("I love")}}{\sum_w \text{Count("I" + } w\text{)}} \tag{8.5}$$

where in the denominator, we sum over all possible vocabularies in the dictionary. This is the proportion of the word "love" occurring immediately after the word "I" out of every time some word w in the dictionary occurring immediately after the word "I." [2] In general, we

[2] Notice that there is no word occurring immediately after the word "I" when "I" is at the end of the sentence in the training corpus. Therefore, the denominator in (8.5) is equal to the Count of "I" minus the Count of "I" at the end of a sentence. This is not necessarily the case when we introduce the sentence stop tokens in Section 8.3.

can estimate the following conditional probability as

$$\Pr[w_{i+1} \mid w_i] \approx \frac{\text{Count}(w_i w_{i+1})}{\sum_w \text{Count}(w_i w)} \tag{8.6}$$

where w_i is the i-th word of the sentence. Once we calculate these estimates from the corpus, we are able to define the probability of the sentence "I love you so much."

8.2.1 *Defining n-Gram Probabilities*

We can extend the example above to a more general setting. Now we want to define the probability distribution over all sentences of length k (grammatical or not). Say we want to find the joint probability of the sentence $w_1 w_2 \dots w_k$ where w_i is the i-th word of the sentence. We will employ a n-gram model which has two assumptions:

1. The probability of a word is only dependent on the immediately previous $n-1$ words. [3]
2. That probability does not depend on the position of the word in the sentence.

[3] If $n = 1$, the model is called a *unigram model*, and the probability is not dependent on any previous word. When $n = 2$ and $n = 3$, the model is respectively called a *bigram* and a *trigram model*.

By a similar logic from the earlier example, we abuse notation and denote the joint probability of the sentence $w_1 w_2 \cdots w_k$ as $\Pr[w_1 w_2 \dots w_k]$; the marginal probability of the first word being w_1 as $\Pr[w_1]$; and so on. We can apply the Chain Rule again to define the n-gram model.

Definition 8.2.1 (n-Gram Model)**.** *An n-gram model assigns the following probability to the sentence $w_1 w_2 \dots w_k$ if $n > 1$:* [4]

$$\begin{aligned}\Pr[w_1 w_2 \dots w_k] &= \Pr[w_1] \Pr[w_2 \mid w_1] \cdots \Pr[w_k \mid w_1 w_2 \dots w_{k-1}] \\ &= \Pr[w_1] \times \prod_{i=2}^{k} \Pr[w_i \mid w_1 \dots w_{i-1}] \\ &= \Pr[w_1] \times \prod_{i=2}^{k} \Pr[w_i \mid w_{\max(1,i-n+1)} \dots w_{i-1}] \end{aligned} \tag{8.7}$$

[4] $\max(1, i-n+1)$ in the third line is to ensure that we access the correct indices for the first $n-1$ words, where there are less than $n-1$ previous words to look at.

and the following probability if $n = 1$:

$$\Pr[w_1 w_2 \dots w_k] = \prod_{i=1}^{k} \Pr[w_i] \tag{8.8}$$

where the n-gram probabilities are estimated from a training corpus as the following

$$\Pr[w_i] \approx \frac{\textit{Count}(w_i)}{\textit{total number of words}}$$

$$\Pr[w_j \mid w_i \dots w_{j-1}] \approx \frac{\textit{Count}(w_i \dots w_{j-1} w_j)}{\sum_w \textit{Count}(w_i \dots w_{j-1} w)}$$

This defines the "best" possible probabilistic model in terms of the Maximum Likelihood Principle from Subsection 4.2.1. [5] We now turn to the following example.

[5] We will prove this for $n = 1$ later.

Example 8.2.2. *We investigate a cowperson language which has two words in the dictionary: $\{Yee, Haw\}$. Suppose the training corpus is given as "Yee Haw Haw Yee Yee Yee Haw Yee." Then the unigram probabilities can be estimated as*

$$\Pr[\text{"}Yee\text{"}] = \frac{5}{8} \qquad \Pr[\text{"}Haw\text{"}] = \frac{3}{8}$$

We can also create the bigram frequency table as in Table 8.1 and we normalize the rows of the bigram frequency table to get the bigram probability table in Table 8.2.

previous \ next	"Yee"	"Haw"	Total
"Yee"	2	2	4
"Haw"	2	1	3

Table 8.1: Bigram frequency table of the cowperson language.

previous \ next	"Yee"	"Haw"	Total
"Yee"	2/4	2/4	1
"Haw"	2/3	1/3	1

Table 8.2: Bigram probabilty table of the cowperson language.

From Table 8.2, we get the following bigram probabilities:

$$\Pr[\text{"}Yee\text{"} \mid \text{"}Yee\text{"}] = \frac{2}{4} \qquad \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] = \frac{2}{4}$$
$$\Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] = \frac{2}{3} \qquad \Pr[\text{"}Haw\text{"} \mid \text{"}Haw\text{"}] = \frac{1}{3}$$

Then by the bigram model, the probability that we see the sentence "Yee Haw Yee" out of all sentences of length 3 can be calculated as

$$\Pr[\text{"}Yee\text{"}] \times \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] \times \Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] = \frac{5}{8} \times \frac{2}{4} \times \frac{2}{3} \simeq 0.21$$

8.2.2 *Maximum Likelihood Principle*

Recall the Maximum Likelihood Principle introduced in Subsection 4.2.1. It gave a way to measure the "goodness" of a model with probabilistic outputs.

Now we formally prove that the estimation methods given in Definition 8.2.1 satisfy the Maximum Likelihood Principle for the $n = 1$ case. A probabilistic model is "better" than another if it assigns more probability to the actual outcome. Here, the actual outcome is the training corpus, which also consists of words. So let us denote

the training corpus as a string of words $w_1 w_2 \dots w_T$. By definition, a unigram model will assign the probability

$$\Pr[w_1 w_2 \dots w_T] = \prod_{i=1}^{T} \Pr[w_i] \tag{8.9}$$

to this string. Remember that each of the w_i's are a member of a finite set of dictionary words. If we let V be the size of the dictionary, then the model is defined by the choice of V values, the probabilities we assign to each of the dictionary words. Let p_i be the probability that we assign to the i-th dictionary word, and let n_i be the number of times that the i-th dictionary word appears in the training corpus. Then (8.9) can be rewritten as

$$\Pr[w_1 w_2 \dots w_T] = \prod_{i=1}^{V} p_i^{n_i} \tag{8.10}$$

We want to maximize this value under the constraint $\sum_{i=1}^{V} p_i = 1$. A solution to this type of a problem can be found via the Lagrange multiplier method. We will illustrate with an example.

Example 8.2.3. *We revisit the cowperson language from Example 8.2.2. Here $V = 2$ and $T = 8$. Let $p_1 = \Pr[\text{"Yee"}]$ and $p_2 = \Pr[\text{"Haw"}]$. Then the probability assigned to the training corpus by the unigram model is*

$$\Pr[\text{"Yee Haw Haw Yee Yee Yee Haw Yee"}] = p_1^5 p_2^3$$

We want to maximize this value under the constaint $p_1 + p_2 = 1$. Then we want to find the point where the gradient of the following is zero.

$$f(p_1, p_2) = p_1^5 p_2^3 + \lambda(p_1 + p_2 - 1)$$

for some λ. The gradients are given as

$$\frac{\partial f}{\partial p_1} = 5p_1^4 p_2^3 + \lambda \qquad \frac{\partial f}{\partial p_2} = 3p_1^5 p_2^2 + \lambda$$

From $5p_1^4 p_2^3 + \lambda = 3p_1^5 p_2^2 + \lambda = 0$, we get $\frac{p_1}{p_2} = \frac{5}{3}$. Combined with the fact that $p_1 + p_2 = 1$, we get the optimal solution $p_1 = \frac{5}{8}$ and $p_2 = \frac{3}{8}$.

Problem 8.2.4. *Following the same Lagrange multiplier method as in Example 8.2.3, verify that the heuristic solution $p_i = \frac{n_i}{T}$ (the empirical frequency) is the optimal solution that maximizes (8.10) under the constraint $\sum_{i=1}^{V} p_i = 1$.*

8.3 Start and Stop Tokens

In this section, we present a convention that is often useful: start token $\langle s \rangle$ and stop token $\langle /s \rangle$. They signify the start and the end of

each sentence in the training corpus. They are a special type of vocabulary that will be augmented to the dictionary, so you will want to pay close attention to the way they contribute to the vocabulary size, number of words, and the n-gram probabilities. Also, by introducing these tokens, we are able to define a probability distribution over all sentences of finite length, not just a given length of k. For the sake of exposition, we will only consider the bigram model for the most parts of this section.

8.3.1 Re-estimating Bigram Probabilities

Consider the cowperson language again.

Example 8.3.1. *The training corpus "Yee Haw Haw Yee Yee Yee Haw Yee" actually consists of three different sentences: (1) "Yee Haw," (2) "Haw Yee Yee," and (3) "Yee Haw Yee." We can append the start and stop tokens to the corpus and transform it into*

$$\begin{aligned} &\langle s \rangle \ \textit{Yee Haw} \ \langle /s \rangle \\ &\langle s \rangle \ \textit{Haw Yee Yee} \ \langle /s \rangle \\ &\langle s \rangle \ \textit{Yee Haw Yee} \ \langle /s \rangle \end{aligned}$$

With these start and stop tokens in mind, we slightly relax the Assumption 2 of the n-gram model and investigate the probability of a word w being the first or the last word of a sentence, separately from other probabilities. We will denote these probabilities respectively as $\Pr[w \mid \langle s \rangle]$ and $\Pr[\langle /s \rangle \mid w]$. The former probability will be estimated as

$$\Pr[w \mid \langle s \rangle] \approx \frac{\text{Count}(\langle s \rangle \ w)}{\text{total number of sentences}} \tag{8.11}$$

which is the proportion of sentences that start with the word w in the corpus. The latter probability is estimated as

$$\Pr[\langle /s \rangle \mid w] \approx \frac{\text{Count}(w \ \langle /s \rangle)}{\text{Count}(w)} \tag{8.12}$$

which is the proportion of the occurrence of w that is at the end of a sentence in the corpus.

Also, notice that other bigram probabilities are also affected when introducing the stop tokens. In (8.6), the denominator originally did not include the occurrence of the substring at the end of the sentence because there was no word to follow that substring. However, if we consider $\langle /s \rangle$ as a vocabulary in the dictionary, the denominator can now include the case where the substring is at the end of the sentence. Therefore, the denominator is just equivalent to the Count of the substring in the corpus. Therefore, the bigram probabilities after introducing start, stop tokens can be estimated instead as [6]

[6] If we consider $\langle s \rangle$, $\langle /s \rangle$ as vocabularies of the dictionary, (8.13) can also include (8.11), (8.12).

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)}{\text{Count}(w_{j-1})} \tag{8.13}$$

Example 8.3.2. *We revisit Example 8.2.2. The bigram frequency table and the bigram probability table can be recalculated as in Table 8.3 and Table 8.4.* [7]

[7] Note that the values in the *Total* column now correspond to the unigram count of that word.

previous \ next	"Yee"	"Haw"	$\langle /s \rangle$	Total
$\langle s \rangle$	2	1	0	3
"Yee"	1	2	2	5
"Haw"	2	0	1	3

Table 8.3: Bigram frequency table of the cowperson language with start and stop tokens.

previous \ next	"Yee"	"Haw"	$\langle /s \rangle$	Total
$\langle s \rangle$	2/3	1/3	0/3	1
"Yee"	1/5	2/5	2/5	1
"Haw"	2/3	0/3	1/3	1

Table 8.4: Bigram probabilty table of the cowperson language with start and stop tokens.

Therefore, the bigram probabilities of the cowperson language, once we introduce the start and stop tokens, are given as

$$\Pr[\text{"Yee"} \mid \langle s \rangle] = \frac{2}{3} \qquad \Pr[\text{"Haw"} \mid \langle s \rangle] = \frac{1}{3}$$
$$\Pr[\text{"Yee"} \mid \text{"Yee"}] = \frac{1}{5} \quad \Pr[\text{"Haw"} \mid \text{"Yee"}] = \frac{2}{5} \quad \Pr[\langle /s \rangle \mid \text{"Yee"}] = \frac{2}{5}$$
$$\Pr[\text{"Yee"} \mid \text{"Haw"}] = \frac{2}{3} \quad \Pr[\text{"Haw"} \mid \text{"Haw"}] = \frac{0}{3} \quad \Pr[\langle /s \rangle \mid \text{"Haw"}] = \frac{1}{3}$$

8.3.2 *Redefining the Probability of a Sentence*

The biggest advantage of introducing stop tokens is that now we can assign a probability distribution over all sentences of finite length, not just a given length k. Say we want to assign a probability to the sentence $w_1 w_2 \dots w_k$ (without the start and stop tokens). By introducing start and stop tokens, we can interpret this as the probability of $w_0 w_1 \dots w_{k+1}$ where $w_0 = \langle s \rangle$ and $w_{k+1} = \langle /s \rangle$. Following the similar logic from (8.2), we can define this probability by the Chain Rule.

Definition 8.3.3 (Bigram Model with Start, Stop Tokens)**.** *A bigram model, once augmented with start, stop tokens, assigns the following probability to a sentence $w_1 w_2 \dots w_k$* [8]

[8] Notice that we do not have the term $\Pr[w_0]$ in the expansion. A sentence always starts with a start token, so the marginal probability that the first word is $\langle s \rangle$ can be understood to be 1.

$$\Pr[w_1 w_2 \dots w_k] = \prod_{i=1}^{k+1} \Pr[w_i \mid w_{i-1}] \tag{8.14}$$

where the bigram probabilities are estimated as in (8.13).

Example 8.3.4. *The probability that we see the sentence "Yee Haw Yee" in the cowperson language can be calculated as*

$$\begin{aligned}&\Pr[\text{"}Yee\text{"} \mid \langle s\rangle] \times \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] \times \Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] \times \Pr[\langle /s\rangle \mid \text{"}Yee\text{"}] \\ &= \frac{2}{3} \times \frac{2}{5} \times \frac{2}{3} \times \frac{2}{5} \simeq 0.07\end{aligned}$$

Note that this probability is taken over all sentences of finite length.

Problem 8.3.5. *Verify that (8.14) defines a probability distribution over all sentences of finite length.*

8.3.3 Beyond Bigram Models

In general, if we have a n-gram model, then we may need to introduce more than 1 start or stop tokens. For example, in a trigram model, we will need to define the probability that the word is the first word of the sentence as $\Pr[w \mid \langle s\rangle \langle s\rangle]$. Based on the number of start and stop tokens introduced, the n-gram probabilities will need to be adjusted accordingly.

8.4 Testing a Language Model

So far, we discussed how to define a n-gram language model given a corpus. This is analogous to training a model given a training dataset. Naturally, the next step is to test the model on a newly seen held-out data to ensure that the model generalizes well. In this section, we discuss how to test a language model.

8.4.1 Shakespeare Text Production

First consider a bigram text generator — an application of the bigram model. The algorithm initiates with the start token $\langle s\rangle$. It then outputs a random word w_1 from the dictionary, according to the probability $\Pr[w_1 \mid \langle s\rangle]$. It then outputs the second random word w_2 from the dictionary, according to the probability $\Pr[w_2 \mid w_1]$. It repeats this process until the newly generated word is the stop token $\langle /s\rangle$. The final output of the algorithm will be the concatenated string of all outputted words.

It is possible to define a text generator for any n-gram model in general. Figure 8.4 shows the output of the unigram, bigram, trigram, quadrigram text generators when the models were trained on all Shakespeare texts.

Notice the sentence "I will go seek the traitor Gloucester." in the output of the quadrigram text generator. This exact line appears in *King Lear*, Act 3 Scene 7. This is not a coincidence. Figure 8.5 presents

```
<s> I
    I want
      want to
           to eat
              eat tasty
                  tasty food
                       food  </s>
I want to eat tasty food
```

Figure 8.3: An example run of the bigram text generator.

Unigram

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
Every enter now severally so, let
Hill he late speaks; or! a more to leg less first you enter
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram

What means, sir. I confess she? then all sorts, he is trim, captain.
Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Trigram

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
This shall forbid it should be branded, if renown made it empty.
Indeed the duke; and had a very good friend.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

Quadrigram

King Henry.What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
Will you not tell me who I am?
It cannot be but so.
Indeed the short and the long. Marry, 'tis a noble Lepidus.

Figure 8.4: The outputs of unigram, bigram, trigram, quadrigram text generators trained on Shakespeare texts.

the snapshot of the bigram, trigram, and quadrigram text generators once they have outputted the phrase "go seek the." You can see that bigram models and trigram models assign very small probability to the word "traitor" because there are much more instances of phrases "the" or "seek the" in the corpus than "go seek the." On the other hand, the quadrigram model assigns a very large probability to the word "traitor" because there is only a limited number of times that the phrase "go seek the" appears in the corpus.

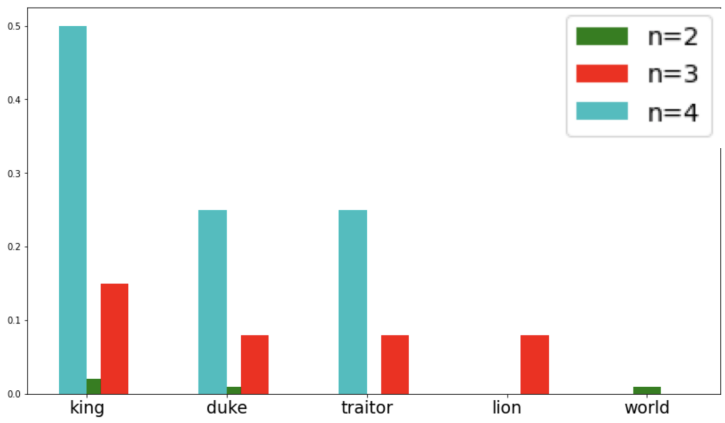


Figure 8.5: The probability of the next word given the previous three words are "go seek the."

Once the quadrigram model outputs the word "traitor" after the

phrase "go seek the," the problem is even worse. As can be see in Figure 8.6, the quadrigram model assigns probability of 1 to the word "Gloucester" meaning that the phrase "seek the traitor" only appears before the word "Gloucester." So the model has *memorized* one completion of the phrase from the training text. From this example, we can see that text production based on n-grams is sampling and remixing text fragments seen in the training corpus.

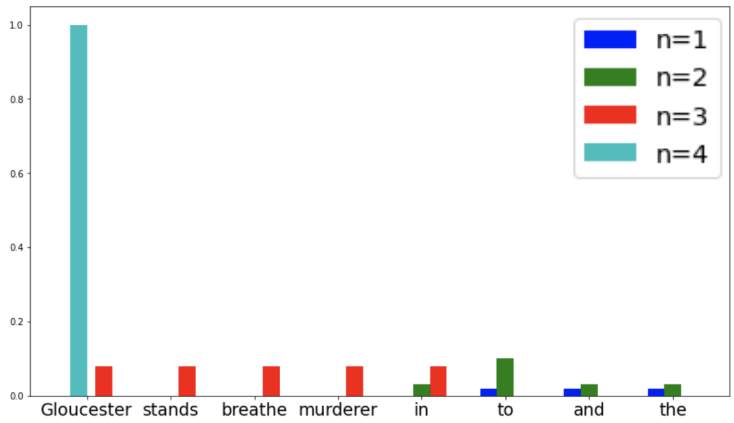


Figure 8.6: The probability of the next word given the previous three words are "seek the traitor."

The Shakespeare corpus consists of $N = 884,647$ words and $V = 29,066$ distinct vocabulary from the dictionary. There are about $V^2 \approx 844$ million possible combinations of bigrams, but Shakespeare only used around $300,000$ of them in his text. So 99.96% of the possible bigrams were never seen. The percentage is much higher for quadrigrams! Furthermore, for the quadrigrams that do appear in the corpus, most of do not even repeat. Thus what comes out of the quadrigram model looks like Shakespeare because it is a *memorized* fragment of Shakespeare.[9]

[9] Do this remind you of overfitting?

8.4.2 *Perplexity*

Having described a way to train a simple language model, we now turn our attention to a formal way of testing [10] a language model.

[10] This method is used even for testing state of the art models.

Just like any other model in ML, a language model will be given a corpus $w_1 w_2 \dots w_T$. Then we can assess the performance of the model by its *perplexity* on the corpus.

Definition 8.4.1 (Perplexity)**.** *The **perplexity** of a language model on the corpus $w_1 w_2 \dots w_T$ is defined as*

$$\Pr[w_1 w_2 \dots w_T]^{-\frac{1}{T}} = \sqrt[T]{\frac{1}{\Pr[w_1 w_2 \dots w_T]}} \tag{8.15}$$

Note that, perplexity is defined for any probabilistic language model: the Chain Rule of joint probability applies to every model,

and does not require the n-gram assumptions. That is, [11]

$$\Pr[w_1 w_2 \dots w_T] = \Pr[w_1] \times \prod_{i=2}^{T} \Pr[w_i \mid w_1 \dots w_{i-1}]$$

Then the perplexity of the model can be rewritten as

$$\sqrt[T]{\frac{1}{\Pr[w_1]} \times \prod_{i=2}^{T} \frac{1}{\Pr[w_i \mid w_1 \dots w_{i-1}]}} \tag{8.16}$$

Example 8.4.2. *Consider the uniform ("clueless") model which assumes that the probability of all words are equal in any given situation. That is, if V is the vocabulary size (i.e., size of the dictionary),*

$$\Pr[w_i] = \Pr[w_i \mid w_1 \dots w_{i-1}] = \frac{1}{V}$$

*for any given $w_1, \dots, w_i \in V$. This model assigns $\left(\frac{1}{V}\right)^T$ to **every** sequence of T words, including the corpus. Therefore, the perplexity of the model is*

$$\left(\left(\frac{1}{V}\right)^T\right)^{-\frac{1}{T}} = V$$

Now we try to understand perplexity at an intuitive level. (8.16) is the geometric mean [12] of the following T values:

$$\frac{1}{\Pr[w_1]}, \frac{1}{\Pr[w_2 \mid w_1]}, \dots, \frac{1}{\Pr[w_T \mid w_1 \dots w_{T-1}]}$$

Now note that a probabilistic model splits the total probability of 1 to fractions and distributes them to the potential options for the next word. So the inverse of an assigned probability for a word can be thought roughly as the *number of choices the model considered for the next word*. With this viewpoint, perplexity as written in (8.16) means: *how much has the model narrowed down the number of choices for the next word* **on average?** The clueless model had not narrowed down the possibilities at all and had the worst-possible perplexity equal to the number of vocabulary words.

Example 8.4.3. *Consider a well-trained language model. At any given place of text, it can identify a set of* 20 *words and assigns probability $\frac{1}{20}$ to each of them to be the next word. It happens that the next word is **always** one of the* 20 *words that the model identifies. The perplexity of the model is*

$$\left(\left(\frac{1}{20}\right)^T\right)^{-\frac{1}{T}} = 20$$

Interestingly enough, the true perplexity of English is believed to be between 15 and 20. That is, if at an "average" place in text, you ask humans to predict the next word, then they are able to narrow down the list of potential next words to around 15 to 20 words. [13]

[11] Assume for now that start and stop tokens do not exist in the corpus.

[12] The *geometric mean* of T numbers $a_1, a_2, \dots, a_T$ is defined as $(\prod_i a_i)^{1/T}$

[13] The perplexity of state of the art language models is under 20 as well.

8.4.3 Perplexity on Test Corpus

The *perplexity* of a language model is analogous to an *loss* of a ML model. [14] Similar to ML models we have been studying so far, it is possible to define a *train perplexity* and a *test perplexity*. The "goodness" of the model will be defined by how low the perplexity was on a previously unseen, held-out data.

[14] It is customary to use the logarithm of the perplexity, as we also did for logistic loss in Chapter 4.

For example, when n-gram models are trained on 38 million words and tested on 1.5 million words from Wall Street Journal articles, they show the following test perplexities in Table 8.5. [15] Note that the state-of-the-art deep learning models achieve a test perplexity of around 20 on the same corpus.

[15] To be more exact, the models were augmented with smoothing, which will be introduced shortly.

Unigram	Bigram	Trigram
962	170	109

Table 8.5: Test perplexities of n-gram models on WSJ corpus.

8.4.4 Perplexity With Start and Stop Tokens

When start and stop tokens are introduced to a corpus, we also need to redefine how to calculate the perplexity of the model. Again, we will only focus on a bigram model for the sake of exposition.

Say the corpus consists of t sentences:

$$\begin{aligned}
&\langle s \rangle\, w_{1,1} w_{1,2}, \ldots, w_{1,T_1}\, \langle /s \rangle \\
&\langle s \rangle\, w_{2,1} w_{2,2}, \ldots, w_{2,T_2}\, \langle /s \rangle \\
&\qquad\qquad \vdots \\
&\langle s \rangle\, w_{t,1} w_{t,2}, \ldots, w_{t,T_t}\, \langle /s \rangle
\end{aligned}$$

The probability of the corpus $w_{1,1} w_{1,2} \ldots w_{t,T_t}$ is redefined as the product of the probability of each of the sentences:

$$\begin{aligned}
\Pr[w_{1,1} w_{1,2} \ldots w_{t,T_t}] &= \prod_{i=1}^{t} \Pr[w_{i,1} w_{i,2} \ldots w_{i,T_i}] \\
&= \prod_{i=1}^{t} \prod_{j=1}^{T_i+1} \Pr[w_{i,j} \mid w_{i,j-1}] \qquad (8.17)
\end{aligned}$$

Now we apply the interpretation of the perplexity that it is the geometric mean of probabilities of each word. Notice that we multiplied $\sum_{i=1}^{t} (T_i + 1)$ probabilities to calculate the probability of the corpus. If we let $T = \sum_{i=1}^{t} T_i$ denote the total number of words (excluding start and stop tokens) of the corpus, the number of probabilities we multiplied can be written as $T^* = T + t$. [16]

[16] This can also be thought as adding the number of stop tokens to the number of words in the corpus.

Definition 8.4.4 (Perplexity with Start, Stop Tokens)**.** *The perplexity of a bigram model with start, stop tokens can be redefined as*

$$\sqrt[T^*]{\frac{1}{\prod_{i=1}^{t}\prod_{j=1}^{T_i+1}\Pr[w_{i,j} \mid w_{i,j-1}]}} \tag{8.18}$$

8.4.5 Smoothing

One big problem with our naive definition of the perplexity of a model is that it does not account for a zero denominator. That is, if the model assigns probability exactly 0 to the corpus, then the perplexity of the model will be ∞! [17]

[17] Mathematically, it is undefined, but here assume that the result is a positive infinity that is larger than any real number.

Example 8.4.5. *Suppose the phrase "green cream" never appeared in the training corpus, but the test corpus contains the sentence "You like green cream." Then a bigram model will have a perplexity of ∞ because it assigns probability 0 to the bigram "green cream."*

To address this issue, we generally apply *smoothing* techniques, which never allow the model to output a zero probability. By *reducing* the naive estimate of *seen* events and *increasing* the naive estimate of *unseen* events, we can always assign nonzero probability to previously unseen events.

The most commonly used smoothing technique is the 1-add smoothing (a.k.a, Laplace smoothing). We describe how the smoothing works for a bigram model. The main idea of the 1-add smoothing can be summarized as "add 1 to all bigram counts in the bigram frequency table." Then the bigram probability as defined in Definition 8.2.1 can be redefined as

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)+1}{\sum_w (\text{Count}(w_{j-1}w)+1)} = \frac{\text{Count}(w_{j-1}w_j)+1}{\sum_w (\text{Count}(w_{j-1}w)) + V} \tag{8.19}$$

where V is the size of the dictionary. If we had augmented the start and the stop tokens to the corpus, the denominator in (8.19) is just equal to $\text{Count}(w_{j-1}) + V^*$ [18] and so the bigram probability can be written as

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)+1}{\text{Count}(w_{j-1}) + V^*} \tag{8.20}$$

[18] $V^* = V + 1$ is the size of the dictionary after adding the start and the stop tokens. It is customary to add only one to the vocabulary. It may help to look at the number of rows and columns in the bigram frequency table 8.3.

Notice that the denominator is just V^*, the new vocabulary size, added to the unigram count of w_{j-1}.

Example 8.4.6. *Recall the cowperson language with the start and stop tokens from Example 8.3.2. Upon further research, it turns out the language actually consists of three words: $\{Yee, Haw, Moo\}$, but the training corpus*

"Yee Haw Haw Yee Yee Yee Haw Yee" left out one of the vocabularies in the dictionary. By applying add-1 smoothing to the bigram model, we can recalculate the bigram frequency and the bigram probability table as in Table 8.6 and Table 8.7

previous \ next	"Yee"	"Haw"	"Moo"	$\langle /s \rangle$	Total
$\langle s \rangle$	3	2	1	1	7
"Yee"	2	3	1	3	9
"Haw"	3	1	1	2	7
"Moo"	1	1	1	1	4

Table 8.6: Bigram frequency table of the cowperson language with start and stop tokens with smoothing.

previous \ next	"Yee"	"Haw"	"Moo"	$\langle /s \rangle$	Total
$\langle s \rangle$	3/7	2/7	1/7	1/7	1
"Yee"	2/9	3/9	1/9	3/9	1
"Haw"	3/7	1/7	1/7	2/7	1
"Moo"	1/4	1/4	1/4	1/4	1

Table 8.7: Bigram probability table of the cowperson language with start and stop tokens with smoothing.

The probability that we see the sentence "Moo Moo" in the cowperson language, which would have been 0 before smoothing, is now assigned a non-zero value:

$$\begin{aligned} &\Pr[\text{"}Moo\text{"} \mid \langle s \rangle] \times \Pr[\text{"}Moo\text{"} \mid \text{"}Moo\text{"}] \times \Pr[\langle /s \rangle \mid \text{"}Moo\text{"}] \\ &= \frac{1}{7} \times \frac{1}{4} \times \frac{1}{4} \simeq 0.01 \end{aligned}$$

Problem 8.4.7. *Verify that (8.19) defines a proper probability distribution over the conditioned event. That is, show that*

$$\sum_{w} \Pr[w \mid w'] = 1$$

for any w in the dictionary.

Another smoothing technique is called *backoff* smoothing. The intuition is that n-gram probabilities are less likely to be zero if n is smaller. So when we run into a n-gram probability that is zero, we replace it with a linear combination of n-gram probabilities of lower values of n.

Example 8.4.8. *Recall Example 8.4.5. The bigram probability of "green cream" can be approximated instead as*

$$\Pr[\text{"}cream\text{"} \mid \text{"}green\text{"}] \approx \Pr[\text{"}cream\text{"}]$$

Also, say we want to calculate the trigram probability of "like green cream," which is also zero in the naive trigram model. We can approximate it instead as

$$\Pr[\text{"}cream\text{"} \mid \text{"}like\ green\text{"}] \approx \alpha \Pr[\text{"}cream\text{"}] + (1-\alpha) \Pr[\text{"}cream\text{"} \mid \text{"}green\text{"}]$$

where α is a hyperparameter for the model.

There are other variants of the backoff smoothing, [19] with some theory for what the "best" choice is, but we will not cover it in these notes.

[19] For instance, Good-Turing and Kneser-Ney smoothing.

9
Matrix Factorization and Recommender Systems

9.1 Recommender Systems

Cataloging and recommender systems have always been an essential asset for consumers who find it difficult to choose from the vast scale of available goods. As early as 1876, Dewey decimal system was invented to organize libraries. In 1892, Sears released their famed catalog to keep subscribers up to date with the latest products and trends, which amounted to 322 pages. Shopping assistants at department stores or radio disc jockeys in the 1940s are also examples of recommndations via human curation. In more contemporary times, bestseller lists at bookstores, or Billboard Hits list aims to capture what is popular among people. The modern recommender system paradigm now focuses on recommending products based on what is liked by people "similar" to you. In this long history of recommender systems, the common theme is that *people like to follow trends*, and recommender systems can help catalyze this process.

9.1.1 Movie Recommendation via Human Curation

Suppose we want to design a recommender system for movies. A human curator identifies r binary attributes that they think is important for a movie (*e.g.*, is a romance movie, is directed by Steven Spielberg, etc.) Then they assign each movie a r-dimensional *attribute vector*, where each element represents whether the movie has the corresponding attribute (*e.g.*, coordinate 2 will have value 1 if a movie is a "thriller" and 0 otherwise).

Now, using a list of movies that a particular user likes, the curator assigns an r-dimensional *taste vector* to a given user in a similar manner (*e.g.*, coordinate w will have value 1 if a user likes "thrillers" and 0 otherwise). With these concepts in mind, we can start with defining the affinity of a user for a particular movie:

Definition 9.1.1 (User Affinity). *Given a taste vector* $\mathbf{A}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,r})$ *for user* i *and the genre vector* $\mathbf{B}_j = (b_{1,j}, b_{2,j}, \ldots, b_{r,j})$ *for movie* j*, we define the* ***affinity of user*** i ***for movie*** j *as*

$$\mathbf{A}_i \cdot \mathbf{B}_j = \sum_{k=1}^{r} a_{i,k} b_{k,j} \tag{9.1}$$

Intuitively, this metric counts the number of attributes which are 1 in both vectors, or equivalently how many of the user's boxes are "checked off" by the movie. Mathematically, the affinity is defined as a dot product, which can be extended to matrix multiplication. Thus if we have a matrix $\mathbf{A} \in \mathbb{R}^{m \times r}$ where each of m rows is a taste vector for a user and a matrix $\mathbf{B} \in \mathbb{R}^{r \times n}$ where each of n columns is a genre vector for a movie, the (i, j) entry of the matrix product $\mathbf{M} = \mathbf{AB}$ represents the affinity score of user i for movie j.

We can also define an additional similarity metric:

Definition 9.1.2 (Similarity Metric). *Given taste vector* A_i *for user* i *and taste vector* $\mathbf{A}_j$ *for user* j*, we define the* ***similarity of user*** i ***and user*** j *as*

$$\sum_{k=1}^{r} a_{i,k} a_{j,k} \tag{9.2}$$

Similarly, the ***similarity of movie*** i ***and movie*** j *is defined as*

$$\sum_{k=1}^{r} b_{k,i} b_{k,j} \tag{9.3}$$

Finally, in practice, each individual is unique and has a different average level of affinity for movies (for example, some users like everything while others are very critical). This means that directly comparing the affinity of one user to another might not be helpful. One way to circumvent this problem is to augment (9.1) in Definition 9.1.1 as

$$\sum_{k=1}^{r} a_{i,k} b_{k,j} + a_{i,0} \tag{9.4}$$

with a bias term $a_{i,0}$.

Based on the affinity scores or similarity scores, the human curator will be able to recommend movies to users. This model design seems like it does the job as a recommender system. In practice, developing such models through human curation comes with a set of pros and cons:

- *Pros:* Using human curation allows domain expertise to be leveraged and this intuition can be critical in the development of a good model (*i.e.*, which attribute is important). In addition, a human curated model will naturally be interpretable.

- *Cons:* The process is tedious and expensive; thus it is difficult to scale. In addition, it can be difficult to account for niche demographics and genres and this becomes a problem for companies with global reach.

We conclude that while human curated models can certainly be useful, the associated effort is often too great.

9.2 *Recommender Systems via Matrix Factorization*

In this section, we provide another technique that can be used for recommender systems — matrix factorization. This method started to become popular since 2005.

9.2.1 *Matrix Factorization*

Matrix factorizations are a common theme throughout linear algebra. Some common techniques include LU and QR decomposition, Rank Factorization, Cholesky Decomposition, and Singular Value Decomposition.

Definition 9.2.1 (Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$*. A* ***matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *such that* $\mathbf{M} = \mathbf{AB}$ *for some* $r < m, n$*.*

Unfortunately, these techniques become less directly applicable once we consider the case where most of the entries of $\mathbf{M}$ are missing (*i.e.*, a missing-data setting). As we saw in Section 9.1.1, this is very common in real-world applications — for example, if the (m, n) entry of M represents the rating of user m for movie n, most entries in M are missing because not everyone has seen every movie. What can we do in such a case?

In turns out, if we assume that M is a low-rank matrix (which is true for many high-dimensional datasets, as noted in Chapter 7), then we can consider an approximate factorization $\mathbf{M} \approx \mathbf{AB}$ on the known entries. We express this as the following optimization problem:

Definition 9.2.2 (Approximate Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$ *where* $\Omega \subset [m] \times [n]$ *is the subset of* (i, j) *where* M_{ij} *is known. An* ***approximate matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *for some* $r < m, n$ *that minimize the loss function:*

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \tag{9.5}$$

We denote the approximation as $\mathbf{M} \approx \mathbf{AB}$*.*

Notice this form is familiar: we are effectively trying to find optimal matrices $\mathbf{A}, \mathbf{B}$ which will minimize the MSE between known entries of M and corresponding entries in the matrix product $\mathbf{AB}$! One thing to note is that by calculating the matrix product $\mathbf{AB}$, we can "predict" entries of $\mathbf{M}$ that are unknown.

You can take the following result from linear algebra as granted.

Theorem 9.2.3. *Given $\mathbf{M} \in \mathbb{R}^{m \times n}$, we can find the matrix factorization $\mathbf{M} = \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if M has rank at most r. Also, we can find the approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}, \mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if $\mathbf{M}$ is "close to" rank r.*

9.2.2 *Matrix Factorization as Semantic Embeddings*

Recall the setup in Section 9.1.1. But instead of calculating the affinity matrix $\mathbf{M}$ as the product of the matrices $\mathbf{A}, \mathbf{B}$, we will approach from the opposite direction. We will start with an affinity matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ (which is only partially known) and find its approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$. We can understand that $\mathbf{A} \in \mathbb{R}^{m \times r}$ represents a set of users and that $\mathbf{B} \in \mathbb{R}^{r \times n}$ represents a set of movies.

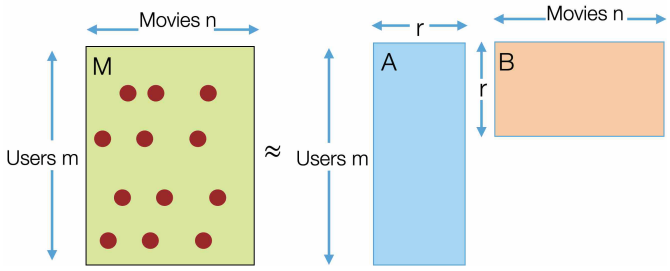


Figure 9.1: Matrix factorization on movie recommendations. Usually the inner dimension r would be much smaller than m, n.

Specifically, if we let A_{i*} denote the i-th row of $\mathbf{A}$ and B_{*j} denote the j-th column of $\mathbf{B}$, then A_{i*} can be understood as the *taste vector* of user i and B_{*j} can be understood as the *attribute vector* of movie j. One difference to note is that the output of a matrix factorization is real-valued, unlike the the $0/1$ valued matrices $\mathbf{A}, \mathbf{B}$ from Section 9.1.1. We can then use the vectors A_{i*} and B_{*j} to find similar users or movies and make recommendations.

Example 9.2.4. *Assume all columns of $\mathbf{B}$ have ℓ_2 norm 1. That is, $\|B_{*j}\|_2 = 1$ for all j. When the inner product $B_{*j} \cdot B_{*j'}$ of two movie vectors is actually 1, the two vectors are exactly the same! They they have the same inner product with every user vector A_{i*} — in other words these movies have the same appeal to all users. Now suppose $B_{*j} \cdot B_{*j'}$ is not quite 1 but close to 1, say 0.9. This means the movie vectors are quite close but not the same. Still, their inner product with typical user vectors will not be too different. We*

conclude that two movies j, j' with inner product $B_{*j} \cdot B_{*j'}$ close to 1 tend to get recommended together to users. One can similarly conclude that high value of inner product between two user vectors is suggestive that the users have similar tastes.

9.2.3 *Netflix Prize Competition: A Case Study*

During 2006-09, DVDs were all the rage. Companies like Netflix were quite interested in recommending movies as accurately as possible in order to retain clients. At the time, Netflix was using an algorithm which had stagnated around $RMSE = 0.95$. [1] Seeking fresh ideas, Netflix curated an anonymized database of $100M$ ratings (each rating was on a $1-5$ scale) of $0.5M$ users for $18K$ movies. Adding a cash incentive of $\$1,000,000$, Netflix challenged the world to come up with a model that could achieve a much lower RMSE! [2] It turned out that matrix factorization would be the key to achieving lower scores. In this example, $m = 0.5M$, $n = 18k$, and Ω corresponds to the $100M$ ratings out of $m \cdot n = 10B$ affinities. [3]

After a lengthy competition, [4] the power of matrix factorization is on full display when we consider the final numbers:

- Netflix's algorithm: $RMSE = 0.95$
- Plain matrix factorization: $RMSE = 0.905$
- Matrix factorization and bias: $RMSE = 0.9$
- Final winner (an ensemble of many methods) : $RMSE = 0.856$

[1] $RMSE$ is shorthand for $\sqrt{MSE}$.

[2] This was an influential competition, and is an inspiration for today's hackathons, Kaggle, etc.

[3] Less than 1% of possible elements are accounted for by Ω.

[4] Amazingly, a group of Princeton undergraduates managed to achieve the second place!

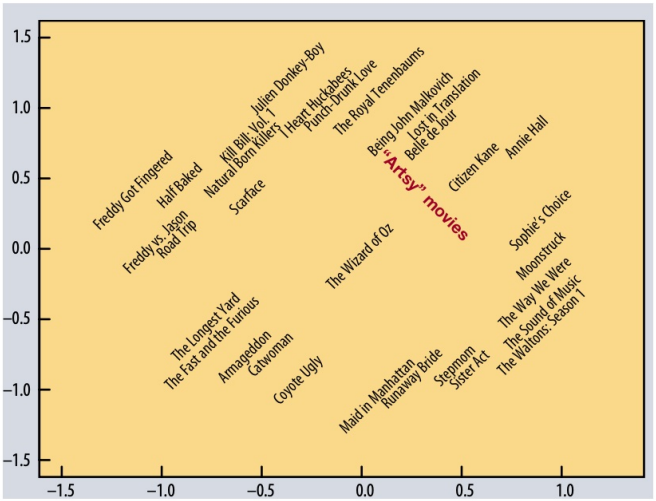


Figure 9.2: 2D visualization of embeddings of film vectors. Note that you see clusters of "artsy" films on top right, and romantic films on the bottom. *Credit: Koren et al.,* Matrix Factorization Techniques for Recommender Systems, *IEEE Computer* 2009.

9.2.4 *Why Does Matrix Factorization Work?*

In general, we need mn entries to completely describe a $m \times n$ matrix $\mathbf{M}$. However, if we find factor $\mathbf{M}$ into the product $\mathbf{M} = \mathbf{AB}$ of $m \times r$ matrix $\mathbf{A}$ and $r \times n$ matrix $\mathbf{B}$, then we can describe $\mathbf{M}$ with essentially only $(m+n)r$ entries. When r is small enough such that $(m+n)r \ll mn$, some entries of $\mathbf{M}$ (including the missing entries) are not truly "required" to understand $\mathbf{M}$.

Example 9.2.5. *Consider the matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & 2 \\ 1 & 1 & * & * \\ 4 & * & 8 & * \\ 4 & * & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? Since* $r = 1$*, it means that all the rows/columns of* $\mathbf{M}$ *are the same up to scaling. By observing the known entries, the second row should be equal to the first row, and the third and the fourth row should be equal to the the first row multiplied by* 4*. Therefore, we can fill in the missing entries as*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 4 & 4 & 8 & 8 \\ 4 & 4 & 8 & 8 \end{bmatrix}$$

It is not hard to infer that $\mathbf{M} = \mathbf{AB}$ *where* $\mathbf{A} = (1,1,4,4)^T$ *and* $\mathbf{B} = (1,1,2,2)$

Example 9.2.6. *Consider another matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & * \\ 1 & 7 & * & * \\ 4 & * & * & 2 \\ * & 4 & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? This time, the answer is no. Following a similar logic from Example 9.2.5, the second row should be equal to the first row multiplied by a constant. This is not feasible since* $M_{2,1}/M_{1,1} = 1$ *and* $M_{2,2}/M_{1,2} = 7$*.*

9.3 *Implementation of Matrix Factorization*

In this section, we look more deeply into implementing matrix factorization in a ML setting. As suggested in Definition 9.2.2, we can consider the process of approximating a matrix factorization to be an optimization problem. Therefore, we can use gradient descent.

9.3.1 Calculating the Gradient of Full Loss

Recall that for an approximate matrix factorization of a matrix $\mathbf{M}$, we want to find matrices $\mathbf{A}, \mathbf{B}$ that minimize the following loss:

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \qquad \text{(9.5 revisited)}$$

Here $(AB)_{ij} = A_{i*} \cdot B_{*j}$. Now we find the gradient of the loss $L(\mathbf{A}, \mathbf{B})$ by first finding the derivatives of L with respect to elements of $\mathbf{A}$ (a total of mr derivatives), then finding the derivatives of L with respect to elements of $\mathbf{B}$ (a total of nr derivatives).

First, consider an arbitrary element $A_{i'k'}$:

$$\begin{aligned}
\frac{\partial}{\partial A_{i'k'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{i'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \cdot (-B_{k'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \qquad (9.6)
\end{aligned}$$

Note that the second step is derived because $(AB)_{ij} = \sum_k A_{ik}B_{kj}$ and if $i \neq i'$, then $\frac{\partial (AB)_{ij}}{\partial A_{i'k'}} = 0$. Enumerating $(i,j) \in \Omega$ can be changed to only enumerating $(i',j) \in \Omega$. Similarly, we can consider an arbitrary element $B_{k'j'}$:

$$\begin{aligned}
\frac{\partial}{\partial B_{k'j'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \cdot (-A_{ik'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \qquad (9.7)
\end{aligned}$$

Whew! That's a lot of derivatives, but we now have $\nabla L(\mathbf{A}, \mathbf{B})$ at our disposal.

9.3.2 Stochastic Gradient Descent for Matrix Factorization

Of course, we could use $\nabla L(\mathbf{A}, \mathbf{B})$ for a plain gradient descent as shown in Chapter 3. However, given that each derivative in the gradient involves a sum over a large number of indices, it would be

worthwhile to use stochastic gradient descent in order to estimate the overall gradient via a small random sample (as shown in Section 3.2).

If we take a sample $S \subset \Omega$ of the known entries at each iteration, the loss becomes

$$\widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i,j \in S} (M_{ij} - (AB)_{ij})^2 \tag{9.8}$$

and the gradient becomes

$$\frac{\partial}{\partial A_{i'k'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{j:\ (i',j) \in S} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \tag{9.9}$$

$$\frac{\partial}{\partial B_{k'j'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i:\ (i,j') \in S} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \tag{9.10}$$

However, if we take a uniform sample S of Ω, the computation will not become much cheaper, since $(i, j) \in S$ can spread into many different rows and columns. One clever (and common) way to do so is to select a set of columns C by sampling k out of the overall n columns. This method is called *column sampling*. We then only need to consider entries $(i, j) \in \Omega$ where $j \in C$ and compute gradients only for the entries $B_{k,j}$ where $j \in C$. We can also perform row sampling in a very similar manner. In practice, whether we should use column sampling or row sampling, or gradient descent of full loss depends on the actual sizes of m and n.

Part III

Deep Learning

10
Introduction to Deep Learning

Deep learning is currently the most successful machine learning approach, with notable successes in object recognition, speech and language understanding, self-driving cars, automated Go playing, etc. It is not easy to give a single definition to such a broad and influential field; nevertheless here is a recent definition by Chris Manning:[1]

> *Deep Learning is the use of large multi-layer (artificial) neural networks that compute with continuous (real number) representations, a little like the hierarchically-organized neurons in human brains. It is currently the most successful ML approach, usable for all types of ML, with better generalization from small data and better scaling to big data and compute budgets.*

[1] Source: https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf.

Deep learning does not represent a specific a model per se, but rather categorizes a group of models called (artificial) neural networks (NNs) (or *deep nets*) which involve several computational layers. Linear models studied in earlier chapters, such as logistic regression in Section 4.2, can be seen as special sub-cases involving only a single layer. The main difference, however, is that general deep nets employ nonlinearity in between each layer, which allows a much broader scale of expressivity. Also, the multiple layers in a neural net can be viewed as computing "intermediate representations" of the data, or "high level features" before arriving at its final answer. By contrast, a linear model works only with the data representation it was given.

Deep nets come in various types, including Feed-Forward NNs (FFNNs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Residual Nets, and Transformers. [2] Training uses a variant of *Gradient Descent*, and the gradient of the loss is computed using an algorithm called *backpropagation*.

[2] Interestingly, a technique called *Neural Architecture Search* uses deep learning to design custom deep learning architectures for a given task.

Due to the immense popularity of deep learning, a variety of software environments such as Tensorflow and PyTorch allow quick implementation of deep learning models. You will encounter them in the homework.

10.1 A Brief History

Neural networks are inspired by the biological processes present within the brain. The concept of an artificial neuron was first outlined by Warren MuCulloch and Walter Pitts in the 1940s. [3] Later in 1986, backpropagation was discovered and provided a way for efficiently applying gradient-based training methods to these models. [4] The basic frameworks for CNNs and modern training soon followed in the late 1980s. [5] However, by the 21st century deep learning had gone out of fashion. This changed in 2012, when Krizhevsky, Sutskever, and Hinton leveraged deep learning techniques through their *AlexNet* model and set new standards for performance on the ImageNet dataset. [6] Deep learning has since begun a resurgence throughout the last decade, boosted by some key factors:

- Hardware, such as GPU and TPU (Tensor Processing Unit, specifically developed for neural network machine learning) technology have made training faster.
- The development of novel neural network architecutres as well as better algorithms for training neural networks.
- A vast amount of data collection, boosted by the spread of the internet, have augmented the performance of NN models.
- Popular frameworks, such Tensorflow and PyTorch, have made it easier to prototype and deploy NN architectures.
- Commercial payoff has caused tech corporations to invest more financial resources.

Each of the reasons listed above have interfaced in a positively reinforcing cycle, causing the acceleration of this technology into the foreseeable future.

[3] Paper: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[4] Paper: https://www.nature.com/articles/323533a0.

[5] Paper: https://link.springer.com/article/10.1007/BF00344251.

[6] Paper: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

10.2 Anatomy of a Neural Network

10.2.1 Artificial Neuron

An *artificial neuron*, or a *node*, is the main component of a neural network. Artificial neurons were inspired by early work on neurons in animal brains, with the analogies in Table 10.1.

Formally, a node is a computational unit which receives m scalar inputs and outputs 1 scalar. This scalar output can be used as an input for a different neuron.

Consider the vector $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_m)$ of m inputs. A neuron internally maintains a trainable weight vector $\vec{\mathbf{w}} = (w_1, w_2, \ldots, w_m)$

Biological neuron	Artificial neuron
Dendrites	Input
Cell Nucleus / Soma	Node
Axon	Output
Synapse	Interconnections

Table 10.1: A comparison between biological neurons in the brain and artificial neurons in neural networks

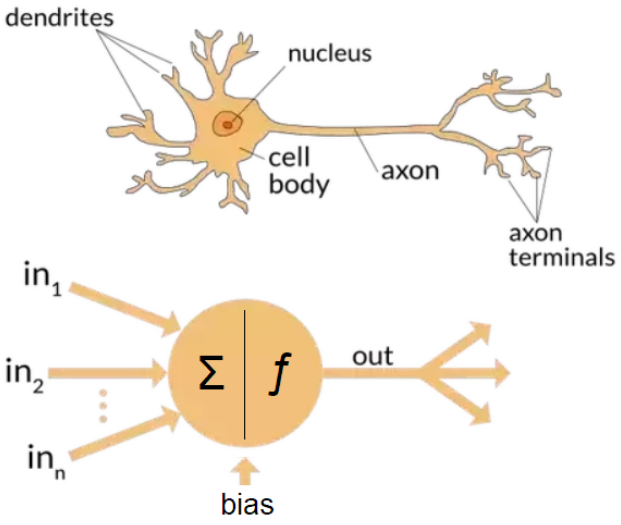


Figure 10.1: A comparison between a brain neuron and an artificial neuron.

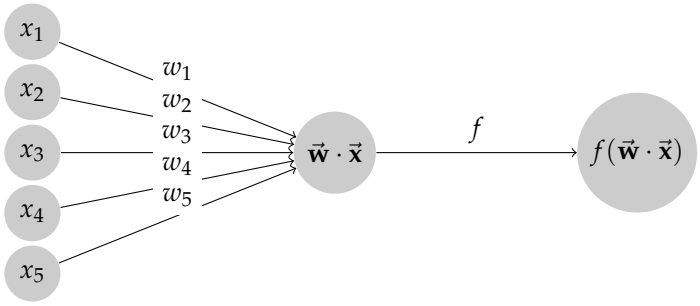


Figure 10.2: A sample artificial neuron.

and optionally a nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$ and outputs the following value: [7]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) \tag{10.1}$$

We can also add a scalar bias b before applying the activation function $f(z)$ in which case the output will look like the following: [8]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b)$$

10.2.2 *Activation Functions*

An artificial neuron can choose its nonlinear activation function $f(z)$ from a variety of options. One such choice is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10.2}$$

[7] If no activation function is chosen, we can assume that f is an identity function $f(z) = z$.

[8] If we introduce a dummy variable for the constant bias term as in Chapter 1 we can absorb the bias term into the equation in (10.1).

Note that in this case, the neuron represents a logistic regression unit. [9] Another popular activation function is the hyperbolic tangent, which is similar to the sigmoid function:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{10.3}$$

In fact, we can rewrite the hyperbolic tangent in terms of sigmoid:

$$\tanh(z) = 2\sigma(2z) - 1 \qquad \text{(10.3 revisited)}$$

According to this expression, tanh function can be viewed as a rescaled sigmoid function. The key difference is: the range of $\sigma(z)$ is $(0, 1)$ and the range of $\tanh(z)$ is $(-1, 1)$.

Arguably the most commonly used activation function is the Rectified Linear Unit, or ReLU:

$$\text{ReLU}(z) = [z]_+ = \max\{z, 0\} \tag{10.4}$$

There are several benefits to the ReLU activation function. It is far cheaper to compute than the previous two alternatives and avoids the "vanishing gradient" problem. [10] With sigmoid and hyperbolic tangent activation functions, the vanishing gradient problem happens when $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ has high absolute values, but ReLU avoids this problem because the derivative is exactly 1 even for high values of z.

Example 10.2.1. *Consider a vector $\vec{\mathbf{x}} = (-2, -1, 0, 1, 2)$ of inputs and a neuron with the weights $\vec{\mathbf{w}} = (1, 1, 1, 1, 1)$. If the activation function of this neuron is the sigmoid, then the output will be:*

$$y = \sigma(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) = \sigma(0) = \frac{1}{2}$$

If the activation is ReLU, it will output:

$$[\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}]_+ = [0]_+ = 0$$

Problem 10.2.2. *Consider a neuron with the weights $\vec{\mathbf{w}} = (1, 1, 5, 1, 1)$ and the ReLU activation function. What will the outputs y_1 and y_2 be for the inputs $\vec{\mathbf{x}}_1 = (-2, -2, 0, 1, 2)$ and $\vec{\mathbf{x}}_2 = (2, -1, 0, 1, 2)$ respectively?*

10.2.3 Neural Network

A neural network consists of nodes connected with directed edges, where each edge has a trainable parameter called its "weight" and each node has an activation function as well as associated parameter(s). There are designated *input* nodes and *output nodes*. The input nodes are given some input values, and the rest of the network then computes as follows: each node produces its output by taking the

[9] However, in this context the output is *not* considered to be a subjective probability as in the case of standard logistic regression.

[10] The vanishing gradient problem refers to a situation where the derivative of a certain step is too close to 0, which can stall the gradient-based learning techniques common in deep learning.

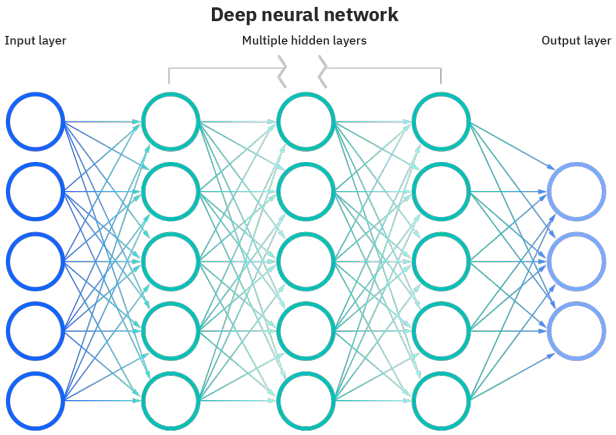


Figure 10.3: A sample neural network design. Each circle represents one artificial neuron. Two nodes being connected by an edge means that the output of the node on the left is being used as one of the inputs for the node on the right.

values produced by all nodes that have a directed edge to it. If the directed graph of connections is *acyclic* — which is the case in most popular architectures — this process of producing the values takes finite time and we end up with a unique value at each of the output nodes. [11] The term *hidden nodes* is used for nodes that are not input or output nodes.

[11] We will not study *Recurrent Neural Nets (RNNs)*, where the graph contains cycles. These used to be popular until a few years ago, and present special difficulties due to the presence of directed loops. For instance, can you come up with instances where the output is not well-defined?

10.3 Why Deep Learning?

Now that we are aware of the basic building blocks of neural networks, let's consider why we prefer these models over techniques explored in previous chapters. The key understanding is that the models previously discussed are fundamentally *linear* in nature. For instance, if we do binary classification, where the data point $\vec{\mathbf{x}}$ is mapped to a label based on $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$, then this corresponds to separating the points with label $+1$ from the points with label -1 via a linear hyperplane $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$. But such models are not a good choice for datasets which are not linearly separable. Deep learning is inherently *nonlinear* and is able to do classification in many settings where linear classification cannot work.

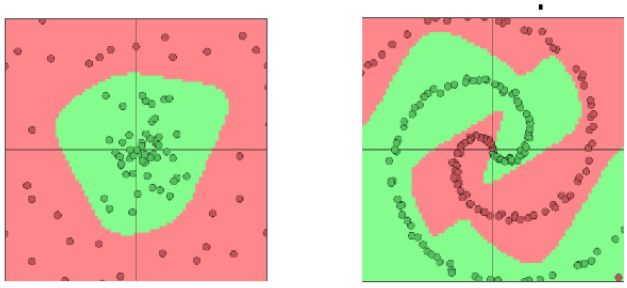


Figure 10.4: Some examples of datasets that are not linearly separable.

10.3.1 The XOR Problem

Consider the boolean function *XOR* with the truth table in Table 10.2.

x_1	x_2	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.2: The truth table for the *XOR* Boolean function.

Let us first attempt to represent the *XOR* function with a single linear neuron. That is, consider a neuron that takes two inputs x_1, x_2 with weights w_1, w_2, a bias term b, and the following Heaveside step activation function: [12]

$$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases} \tag{10.5}$$

[12] This neuron is called a *linear perceptron*. It uses a nonlinear activation function, but the nonlinearity is strictly for the binary classification in the final step. The boundary of the classification is still linear.

Proposition 10.3.1. *There are no values of w_1, w_2, b such that the linear neuron defined by the values represent the XOR function.*

Proof. Assume to the contrary that there are such values. Let $\vec{\mathbf{x}}_1 = (0,0), \vec{\mathbf{x}}_2 = (0,1), \vec{\mathbf{x}}_3 = (1,0), \vec{\mathbf{x}}_4 = (1,1)$. Then we know that

$$g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b) = 0$$
$$g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b) = 1$$

which implies that

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b \leq 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b \leq 0$$
$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b > 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b > 0$$

Now let $\vec{\mathbf{x}} = \left(\frac{1}{2}, \frac{1}{2}\right)$. Since we have $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_1 + \frac{1}{2}\vec{\mathbf{x}}_4$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b)) \leq 0$$

since we are taking the average of two non-positive numbers. But at the same time, since $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_2 + \frac{1}{2}\vec{\mathbf{x}}_3$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b)) > 0$$

since we are taking the average of two positive numbers. This leads to a contradiction. □

Problem 10.3.2. *Verify that the AND, OR Boolean functions can be represented by a single linear node.*

Figure 10.5 visualizes the truth table for *XOR* in the 2*D* plane. The two axes represent the two inputs x_1 and x_2; blue circles denote that $y = 1$; and white circles denote that $y = 0$. A single linear neuron can be understood as drawing a red line that can separate white points from blue points. Notice that it is possible to draw such a line for *AND* and *OR* functions, but the data points that characterize the *XOR* function are not linearly separable.

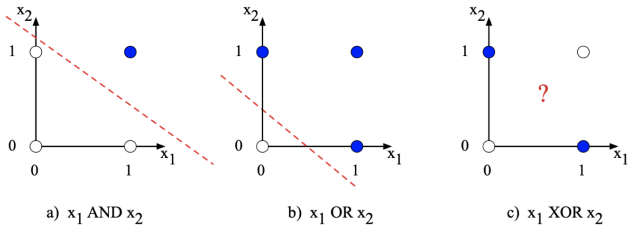


Figure 10.5: The data points that characterize the *XOR* function are not linearly separable.

Instead, we will leverage neural networks to solve this problem. Let us design an architecture with inputs x_1, x_2, a hidden layer with two nodes h_1, h_2, and a final output layer with one node y_1. We assign the *ReLU* activation function to the hidden nodes and define weights and biases as shown in Figure 10.6.

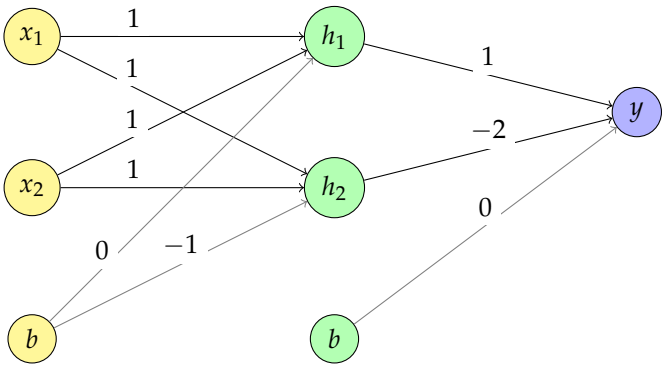


Figure 10.6: A sample neural network which computes the *XOR* of its inputs x_1 and x_2. The weights for inputs are shown by black arrows, while bias terms are shown by grey arrows.

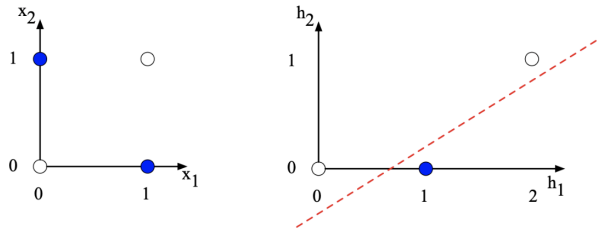
To be more explicit, the neural network is defined by the following three neurons:

$$h_1 = ReLU(x_1 + x_2)$$
$$h_2 = ReLU(x_1 + x_2 - 1)$$
$$y_1 = ReLU(h_1 - 2h_2)$$

Problem 10.3.3. *Verify that the model in Figure 10.6 represents the XOR function by constructing a truth table.*

The main difference between the single linear neuron approach and the neural network for the *XOR* function is that the network now

has two layers of neurons. If we only focus on the final layer of the neural network, we expect the boundary of the binary classification to be linear to the values of h_1, h_2. However, the values of h_1, h_2 are *not* linear to the input values x_1, x_2 because the hidden nodes utilize a *nonlinear* activation function. Hence the boundary of the classification is also *not* linear to the input values x_1, x_2. The nonlinear activation function transforms the input space into a space where the *XOR* operation is *linearly separable*. As shown in Figure 10.7, the h space is quite clearly linearly separable in contrast to the original x space.



a) The original x space b) The new (linearly separable) h space

Figure 10.7: Unlike the x space, after applying the nonlinear *ReLU* activation function, the mapped h space is linearly separable.

10.4 Multi-class Classification

Neural networks, like multi-class regression in Chapter 4, can be used for classification tasks where the number of possible labels is larger than 2. Real-life scenarios include hand-written digit recognition on the MNIST dataset, where the model designer could use ten different classes to correspond to each possible digit. Another possible example is an speech recognition language model, where the model is trained to distinguish between sounds of $|V|$ vocabularies.

It turns out that such functionality can be added by simply including as many output neurons as desired classes in the output layer. Then, the values of the output neurons will be converted into a probability distribution $\Pr[y = k]$ over the number of classes.

10.4.1 Softmax Function

Just as in Chapter 4, we use the softmax function for the purpose of the multi-class classification. See Chapter 19 for the definition of the softmax function.

Example 10.4.1. *Say $\vec{\mathbf{o}} = (3, 0, 1)$ are the values of the output neurons of a neural network before applying the activation function. If we decide to apply the softmax function as the activation function, the final outputs of the*

network will be $softmax(\vec{\mathbf{o}}) \simeq (0.84, 0.04, 0.11)$. *If the network was trying to solve a multi-class classification task, we can understand that the given input is most likely to be of class* 1, *with probability* 0.84 *according to the model.*

One notable property of the softmax function is that the output of the function is the same if all coordinates of the input vector is shifted by the same amount; that is $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ for any $c \in \mathbb{R}$, where $\vec{\mathbf{1}} = (1, 1, \ldots, 1)$ is a vector of all ones.

Example 10.4.2. *Consider two vectors* $\vec{\mathbf{z}}_1 = (5, 2, 3)$ *and* $\vec{\mathbf{z}}_2 = (3, 0, 1)$. *Then* $softmax(\vec{\mathbf{z}}_1) = softmax(\vec{\mathbf{z}}_2)$ *because* $\vec{\mathbf{z}}_2 = \vec{\mathbf{z}}_1 - (2, 2, 2)$.

Problem 10.4.3. *Prove the property that* $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ *for any* $c \in \mathbb{R}$. *(Hint: multiply both the numerator and the denominator of* $softmax(\vec{\mathbf{z}})_k$ *by* $\exp(c)$.*)*

11
Feedforward Neural Network and Backpropagation

Feedforward Neural Networks (FFNNs) are perhaps the simplest kind of deep nets and are characterized by the following properties:

- There are nodes connected with no cycles.
- Nodes are partitioned into layers numbered 1 to k for some k. The nodes in the first layer receive input of the model and output some values. Then the nodes in layer $i+1$ receive output of the nodes in layer i as their input and output some values. The output of the model can be computed with the output of the nodes in layer k.
- No outputs are passed back to lower layers.

Now, we only consider fully-connected layers — a special case of a layer in feedforward neural networks.

Definition 11.0.1 (Fully-Connected Layer)**.** *A **fully-connected layer** is a neural network layer in which all the nodes from one layer are fully connected to every node of the next layer.*

Note that not all layers of feedforward neural networks are necessarily fully-connected (a typical case is a Convolutional Neural Network, which we will explore in Chapter 12). However, feedforward neural networks with fully-connected layers are very common and also easy to implement.

11.1 Forward Propagation: An Example

Forward propagation refers to how the network converts a specific input to the output, specifically the calculation and storage of intermediate variables from the input layer to the output layer. In this section, we use concrete examples to motivate the topic. We will provide a more general formula in the next section. Readers who have a stronger background in math may feel to skip this section altogether.

11.1.1 One Output Node

We start with the network in Figure 11.1 as an example. The network receives three inputs x_1, x_2, x_3 and has a first hidden layer with two nodes $h_1^{(1)}, h_2^{(1)}$, a second hidden layer with two nodes $h_1^{(2)}, h_2^{(2)}$, and a final output layer with one node o. We assign the *ReLU* activation function to the hidden units, and define weights as shown in Figure 11.1.

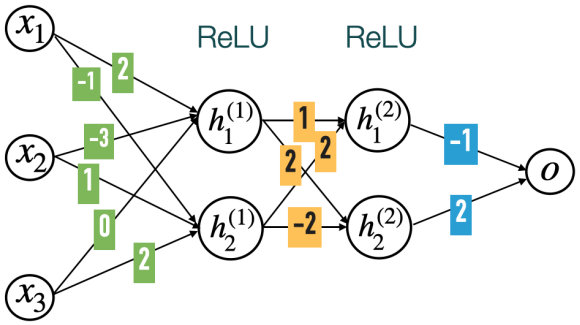


Figure 11.1: A sample feedforward neural network with two hidden layers and one output node.

The two hidden nodes in the first hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(1)} &= ReLU(2x_1 - 3x_2) \\ h_2^{(1)} &= ReLU(-x_1 + x_2 + 2x_3) \end{aligned} \tag{11.1}$$

and the two hidden nodes in the second hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(2)} &= ReLU(h_1^{(1)} + 2h_2^{(1)}) \\ h_2^{(2)} &= ReLU(2h_1^{(1)} - 2h_2^{(1)}) \end{aligned} \tag{11.2}$$

and the output node is characterized by the following equation:

$$o = -h_1^{(2)} + 2h_2^{(2)}$$

Therefore, if we know the input values x_1, x_2, x_3, we can first calculate the values $h_1^{(1)}, h_2^{(1)}$, then using these values, calculate $h_1^{(2)}, h_2^{(2)}$, and finally using these values, we can calculate the output o of the network.

Example 11.1.1. *If the provided input vector to the neural network in Figure 11.1 is $\vec{\mathbf{x}} = (1, 1, 1)$, we can calculate the first hidden layer as*

$$\begin{aligned} h_1^{(1)} &= ReLU(2 - 3) = 0 \\ h_2^{(1)} &= ReLU(-1 + 1 + 2) = 2 \end{aligned}$$

and the second hidden layer as

$$h_1^{(2)} = ReLU(0 + 2 \cdot 2) = 4$$
$$h_2^{(2)} = ReLU(0 - 2 \cdot 2) = 0$$

and the output as

$$o = -4 + 0 = -4$$

11.1.2 *Multiple Output Nodes*

Networks can have more than one output node. An example is the network in Figure 11.2.

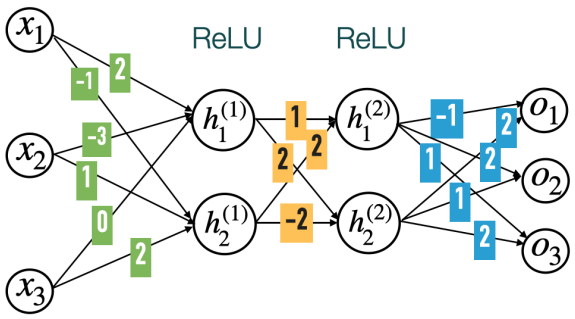


Figure 11.2: A sample feedforward neural network with two hidden layers and three output nodes.

The networks in Figure 11.1 and Figure 11.2 are the same except for the output layer; the former has one output node, while the latter has three output nodes. Now the output values of the network in Figure 11.2 can be calculated as:

$$\begin{aligned} o_1 &= -h_1^{(2)} + 2h_2^{(2)} \\ o_2 &= 2h_1^{(2)} + h_2^{(2)} \\ o_3 &= h_1^{(2)} + 2h_2^{(2)} \end{aligned} \tag{11.3}$$

Recall from the previous Chapter 10 that a FFNN with multiple output nodes is used for multi-class classfication. After the naive output values are calculated, the output nodes will use the softmax activation function to transform the values into the probabilities for each of the three classes. That is, the probability for predicting each class will be calculated as:

$$\begin{aligned} \hat{o}_1 &= softmax(o_1, o_2, o_3)_1 \\ \hat{o}_2 &= softmax(o_1, o_2, o_3)_2 \\ \hat{o}_3 &= softmax(o_1, o_2, o_3)_3 \end{aligned} \tag{11.4}$$

Example 11.1.2. *If the provided input vector to the neural network in Figure 11.1 is* $\vec{\mathbf{x}} = (1, 1, 1)$*, we can calculate the output layer as*

$$\begin{aligned} o_1 &= -4 + 0 = -4 \\ o_2 &= 2 \cdot 4 + 0 = 8 \\ o_3 &= 4 + 0 = 4 \end{aligned}$$

and the probabilities of each class as

$$\begin{aligned} \widehat{o}_1 &= softmax(-4, 8, 4)_1 = \frac{e^{-4}}{e^{-4} + e^{8} + e^{4}} \simeq 0.00 \\ \widehat{o}_2 &= softmax(-4, 8, 4)_2 = \frac{e^{8}}{e^{-4} + e^{8} + e^{4}} \simeq 0.98 \\ \widehat{o}_3 &= softmax(-4, 8, 4)_3 = \frac{e^{4}}{e^{-4} + e^{8} + e^{4}} \simeq 0.02 \end{aligned}$$

11.1.3 Matrix Notation

Let $w_{i,j}^{(1)}$ be the weight between the i-th node $h_i^{(1)}$ in the first hidden layer and the j-th input x_j. Then (11.1) can be rewritten as

$$\begin{aligned} h_1^{(1)} &= ReLU(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,3}^{(1)} x_3) \\ h_2^{(1)} &= ReLU(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{2,3}^{(1)} x_3) \end{aligned}$$

Notice that if we set $\vec{\mathbf{x}} = (x_1, x_2, x_3) \in \mathbb{R}^3$ and $\vec{\mathbf{h}}^{(1)} = (h_1^{(1)}, h_2^{(1)}) \in \mathbb{R}^2$ and define a matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{2\times 3}$ where its (i, j) entry is $w_{i,j}^{(1)}$, then we can further rewrite (11.1) as [1]

$$\vec{\mathbf{h}}^{(1)} = ReLU\left(\mathbf{W}^{(1)} \vec{\mathbf{x}}\right) \tag{11.5}$$

[1] Here we interpret the vectors $\vec{\mathbf{x}}$, $\vec{\mathbf{h}}^{(1)}$ as a column vector, or equivalently a 3×1 matrix and a 2×1 matrix respectively. This will be a convention throughout this chapter.

where the $ReLU$ function is applied element-wise.

Similarly, if we let $w_{i,j}^{(2)}$ be the weight between the i-th node $h_i^{(2)}$ in the second hidden layer and the j-th node $h_j^{(1)}$ in the first hidden layer, (11.2) can be rewritten as

$$\begin{aligned} h_1^{(2)} &= ReLU(w_{1,1}^{(2)} h_1^{(1)} + w_{1,2}^{(2)} h_2^{(1)}) \\ h_2^{(2)} &= ReLU(w_{2,1}^{(2)} h_1^{(1)} + w_{2,2}^{(2)} h_2^{(1)}) \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{h}}^{(2)} = ReLU\left(\mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)}\right) \tag{11.6}$$

where $\vec{\mathbf{h}}^{(2)} = (h_1^{(2)}, h_2^{(2)}) \in \mathbb{R}^2$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{2\times 2}$ is a matrix whose (i, j) entry is $w_{i,j}^{(2)}$.

Next, if we let $w_{i,j}^{(o)}$ be the weight between the i-th output node o_i (before softmax) and the j-th node $h_j^{(2)}$ in the second hidden layer, (11.3) can be rewritten as

$$\begin{aligned} o_1 &= w_{1,1}^{(o)} h_1^{(2)} + w_{1,2}^{(o)} h_2^{(2)} \\ o_2 &= w_{2,1}^{(o)} h_1^{(2)} + w_{2,2}^{(o)} h_2^{(2)} \\ o_3 &= w_{3,1}^{(o)} h_1^{(2)} + w_{3,2}^{(o)} h_2^{(2)} \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} \tag{11.7}$$

where $\vec{\mathbf{o}} = (o_1, o_2, o_3) \in \mathbb{R}^3$ and $\mathbf{W}^{(o)} \in \mathbb{R}^{3\times 2}$ is a matrix whose (i, j) entry is $w_{i,j}^{(o)}$.

Finally, if we let $\hat{\vec{\mathbf{o}}} = (\hat{o}_1, \hat{o}_2, \hat{o}_3) \in \mathbb{R}^3$, then (11.4) can be rewritten as

$$\hat{\vec{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) \tag{11.8}$$

We summarize the results above into the following matrix equations

$$\begin{aligned} \vec{\mathbf{h}}^{(1)} &= ReLU\left(\mathbf{W}^{(1)} \vec{\mathbf{x}}\right) \\ \vec{\mathbf{h}}^{(2)} &= ReLU\left(\mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)}\right) \\ \vec{\mathbf{o}} &= \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} \\ \hat{\vec{\mathbf{o}}} &= softmax(\vec{\mathbf{o}}) \end{aligned} \tag{11.9}$$

Example 11.1.3. *If the provided input vector to the neural network in Figure 11.2 is $\vec{\mathbf{x}} = (1, 1, 1)$, we can calculate the first hidden layer as*

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)} \vec{\mathbf{x}} = ReLU\left(\begin{bmatrix} 2 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

and the second hidden layer as

$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)} = ReLU\left(\begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

and the output layer $\vec{\mathbf{o}}$ (before the softmax) as

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} = \begin{bmatrix} -1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \end{bmatrix} = \begin{bmatrix} -4 \\ 8 \\ 4 \end{bmatrix}$$

The probability distribution $\hat{\vec{\mathbf{o}}}$ of the three classes can then be calculated as

$$\hat{\vec{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) = \left(\frac{e^{-4}}{e^{-4} + e^{8} + e^{4}}, \frac{e^{8}}{e^{-4} + e^{8} + e^{4}}, \frac{e^{4}}{e^{-4} + e^{8} + e^{4}}\right)$$

11.2 Forward Propagation: The General Case

We now consider an arbitrary feedforward neural network with $L \geq 1$ layers. Let $\vec{\mathbf{x}} \in \mathbb{R}^{d_0}$ be the vector of d_0 *inputs* to the network. For $k = 1, 2, \ldots, L$, let $\vec{\mathbf{h}}^{(k)} = (h_1^{(k)}, h_2^{(k)}, \ldots, h_{d_k}^{(k)}) \in \mathbb{R}^{d_k}$ represent the d_k nodes of the *k-th hidden layer*. The L-th hidden layer is also known as the *output layer*, and we alternatively denote $d_0 = d_{in}$ and $d_L = d_{out}$ to emphasize that they are respectively the number of inputs and the number of output nodes.

Additionally, we consider $\mathbf{W}^{(k)} \in \mathbb{R}^{d_k \times d_{k-1}}$ to represent the weights for the k-th hidden layer. Its (i, j) entry is the weight between the i-th node $h_i^{(k)}$ of the k-th hidden layer and the j-th node $h_j^{(k-1)}$ of the $(k-1)$-th hidden layer. We also alternatively denote $\mathbf{W}^{(L)} = \mathbf{W}^{(o)}$ to emphasize that it represents the weights for the output layer.

Finally, let $f^{(k)}$ be the nonlinear activation function for layer k. For instance, consider the output layer. If $d_{out} = 1$ (*i.e.*, there is one output node), we can assume that $f^{(L)}$ is the identity function. On the other hand, if $d_{out} > 1$ (*i.e.*, there are multiple output nodes), we can assume that $f^{(L)}$ is the softmax function. It is also possible to use different activation functions for each layer.

With all these new notations in mind, we can express the nodes of layer k as:

$$\vec{\mathbf{h}}^{(k)} = f^{(k)}(\mathbf{W}^{(k)}\vec{\mathbf{h}}^{(k-1)})$$

for each $k = 1, 2, \ldots, L$.

If $d_{out} = 1$, we let $o = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the final output of the model. If $d_{out} > 1$, we let $\vec{\mathbf{o}} = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the output layer before the softmax and $\vec{\hat{\mathbf{o}}} = f^{(L)}(\vec{\mathbf{o}})$ denote the output layer after the softmax.

11.2.1 Number of Weights

We now briefly consider the number of weights in a feedforward network. There are $d_{in} \cdot d_1$ weights (or variables) for the first hidden layer. Similarly, there are $d_1 \cdot d_2$ weights for the second hidden layer. In total, the number of weights is $\sum_{i=0}^{L-1} d_i \cdot d_{i+1}$.

Example 11.2.1. *The number of weights in the model in Figure 11.2 can be calculated as*

$$3 \times 2 + 2 \times 2 + 2 \times 3 = 16$$

11.2.2 What If We Remove Nonlinearity?

If we removed the nonlinear activation function *ReLU* in our model from (11.9), we would have the following forward propagation equa-

tions:

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)}\vec{\mathbf{x}}$$
$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)}\vec{\mathbf{h}}^{(1)}$$
$$\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(2)}$$
$$\vec{\hat{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$$

Notice that if we set $\mathbf{W}' = \mathbf{W}^{(o)}\mathbf{W}^{(2)}\mathbf{W}^{(1)} \in \mathbb{R}^{3\times 3}$, then

$$\vec{\hat{\mathbf{o}}} = softmax(\mathbf{W}'\vec{\mathbf{x}})$$

We have thus reduced our neural net to a standard multi-classification logistic regression model! As we have discussed the limitation of linear models earlier, this indicates the importance of having nonlinear activation functions between layers.

11.2.3 Training Loss

Just like we have defined a loss function for ML models so far, we also define an appropriate loss function for neural networks, where the objective of the network becomes finding a set of weights that minimize the loss. While there are many different definitions of loss functions, here we present two — one that is more appropriate when there is a single output node, and another that is more appropriate for multi-class classification.

When there is only one scalar node in the output layer (*i.e.*, $d_{out} = 1$), we can use a *squared error loss*, similar to the least squares loss from (1.4):

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} (y - o)^2 \qquad \text{(Squared Error Loss)}$$

where $\vec{\mathbf{x}} \in \mathbb{R}^{d_{in}}$ is the input vector, y is its gold value (*i.e.*, actual value in the training data), and $o = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$ is the final output (*i.e.*, prediction) of the neural network.

Example 11.2.2. *If the provided input vector to the neural network in Figure 11.1 is $\vec{\mathbf{x}} = (1, 1, 1)$ and the training output is $y = 0$, we can calculate the squared error loss as*

$$(y - o)^2 = (0 - (-4))^2 = 16$$

When there are multiple nodes in the output layer (*e.g.*, for multi-class classification), we can use a loss function that is similar to the logistic loss. Recall that in logistic regression, we defined the logistic loss as a sum of log loss over a set of data points:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \Pr[\text{label } y \text{ on input } \vec{\mathbf{x}}] \qquad \text{(4.5 revisited)}$$

where $y \in \{-1, 1\}$ denotes the gold label. For neural networks, we can analogously define the *cross-entropy loss*:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \widehat{o}_y \qquad \text{(Cross-Entropy Loss)}$$

where $y \in \{1, \ldots, d_{out}\}$ denotes the gold label, and $\widehat{o}_y$ denotes the probability that the model assigns to class y — that is, the y-th coordinate of the output vector $\vec{\widehat{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$ after applying the softmax function.

Example 11.2.3. *If the provided input vector to the neural network in Figure 11.2 is $\vec{\mathbf{x}} = (1, 1, 1)$ and the training output is $y = 2$, we can calculate the cross-entropy loss for this data point as*

$$-\log \widehat{o}_y = -\log \frac{e^8}{e^{-4} + e^8 + e^4} \simeq 4.02$$

11.3 Backpropagation: An Example

Just like in previous ML models we have learned, the process of training a neural network model involves three steps:

1. Defining an an appropriate loss function.
2. Calculating the gradient of the loss with respect to the training data and the model parameters.
3. Iteratively updating the parameters via the gradient descent algorithm.

But once a neural network grows in size, the second step of calculating the gradients starts to become a problem. Naively trying to calculate each of the gradients separately becomes inefficient. Instead, *Backpropagation* [2] is an efficient way to calculate the gradients with respect to the network parameters such that the number of steps for the computation is *linear* in the size of the nueral network.

The key idea is to perform the computation in a very specific sequence — from the output layer to the input layer. By the Chain Rule, we can use the already computed gradient value of a node in a higher layer in the computation of the gradient of a node in a lower layer. This way, the gradient values *propagate* back from the top layer to the bottom layer.

[2] Reference: https://www.nature.com/articles/323533a0

11.3.1 The Delta Method: Reasoning from First Principles

The main goal of backpropagation is to compute the partial derivative $\partial f / \partial w$ where f is the loss and w is the weight of an edge. This

will allow us to apply the gradient descent algorithm and appropriately update the weight w. Students often find backpropagation a confusing idea, but it is actually just a simple application of Chain Rule in multivariate calculus.

In this subsection, we motivate the topic with the *Delta Method*, which is an intuitive way to compute $\partial f/\partial w$. We perturb a weight w by a small amount Δ and measure how much the output changes. In doing so, we also measure how the rest of the network changes. As we will see later, the process of computing the partial derivative of the form $\partial f/\partial w$ requires us to also compute the partial derivative of the form $\partial f/\partial h$ where h is the value at a node.

Readers who are familiar with Chain Rule can quickly browse through the rest of this subsection.

Example 11.3.1. *Consider the model from Figure 11.2 but now with the inputs $\vec{\mathbf{x}} = (3, 1, 2)$. We use the same notation for the nodes and the weights that we used throughout Section 11.1. The goal of this simple example is to illustrate what the derivatives mean.*

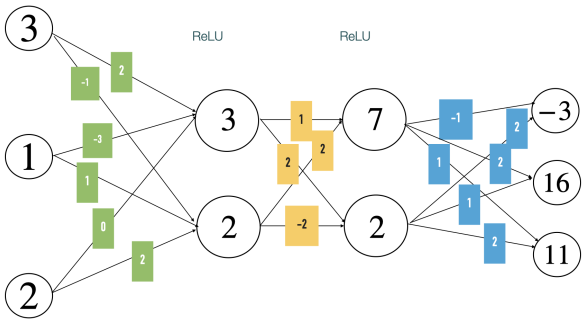


Figure 11.3: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$.

Suppose we want to take partial derivative of first output node o_1 with respect to the weight $w^{(1)}_{2,2}$ (i.e., the weight on the edge between the second input x_2 and the second node $h^{(1)}_2$ of first hidden layer). This is denoted as $\partial o_1/\partial w^{(1)}_{2,2}$. Its definition involves considering how changing $w^{(1)}_{2,2}$ by an infinitesimal amount Δ changes o_1, whose current value is -3.

Adding Δ to $w^{(1)}_{2,2}$ will change the values of the first hidden layer to

$$h^{(1)}_1 = ReLU(2 \cdot 3 + (-3) \cdot 1 + 0 \cdot 2) = 3$$
$$h^{(1)}_2 = ReLU((-1) \cdot 3 + (1 + \Delta) \cdot 1 + 2 \cdot 2) = 2 + \Delta$$

Letting $\Delta \to 0$, we see that the rate of change of $h^{(1)}_1$ and $h^{(1)}_2$ with respect to change of $w^{(1)}_{2,2}$ is 0 and 1 respectively. In more formal terms, $\partial h^{(1)}_1/\partial w^{(1)}_{2,2} = 0$ and $\partial h^{(1)}_2/\partial w^{(1)}_{2,2} = 1$.

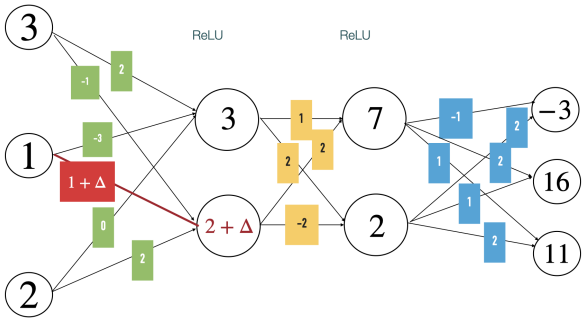


Figure 11.4: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the first hidden layer is also affected.

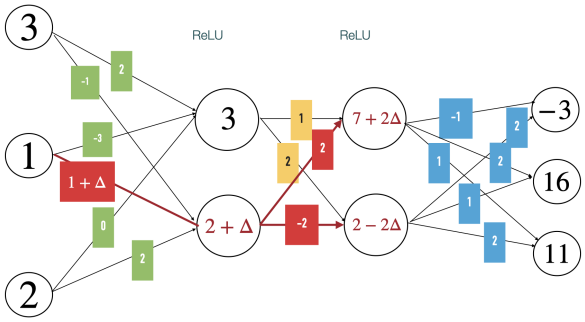


Figure 11.5: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the second hidden layer is also affected.

Using the updated values of the first hidden layer, the second hidden layer will be calculated as

$$h^{(2)}_1 = ReLU(1 \cdot 3 + 2 \cdot (2 + \Delta)) = 7 + 2\Delta$$
$$h^{(2)}_2 = ReLU(2 \cdot 3 + (-2) \cdot (2 + \Delta)) = 2 - 2\Delta$$

This shows that the rate of change of $h^{(2)}_1$ and $h^{(2)}_2$ with respect to change of $w^{(1)}_{2,2}$ is 2 and -2 respectively.

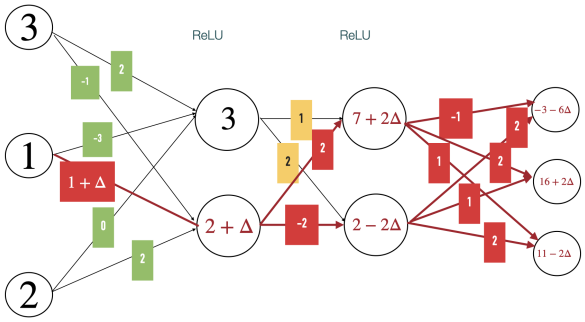


Figure 11.6: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the output layer is also affected.

Finally the output layer can now be calculated as

$$o_1 = (-1)\cdot(7+2\Delta)+2\cdot(2-2\Delta) = -3-6\Delta$$
$$o_2 = 2\cdot(7+2\Delta)+1\cdot(2-2\Delta) = 16+2\Delta$$
$$o_3 = 1\cdot(7+2\Delta)+2\cdot(2-2\Delta) = 11-2\Delta$$

This shows that the rate of change of o_1 with respect to change of $w^{(1)}_{2,2}$ is -6.

Example 11.3.2. *Now we consider the meaning of $\partial o_1/\partial h^{(2)}_1$: how changing the value of $h^{(2)}_1$ by an infinitesimal Δ affects o_1. Note that this is a thought experiment that does not correspond to a change that is possible if the net were a physical object constructed of nodes and wires — the value of $h^{(2)}_1$ is completely decided by the previous layers and cannot be changed in isolation without touching the previous layers. However, treating these values as variables, it is possible to put on a calculus hat and and think about the rate of change of one with respect to the other.*

If only the value of $h^{(2)}_1$ is changed from 7 to $7+\Delta$ in Figure 11.3, then o_1 can be calculated as

$$o_1 = (-1)\cdot(7+\Delta)+2\cdot 2 = -3-\Delta$$

which shows that $\partial o_1/\partial h^{(2)}_1 = -1$.

Problem 11.3.3. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial o_1/\partial h^{(2)}_2$, $\partial h^{(2)}_1/\partial w^{(1)}_{2,2}$, and $\partial h^{(2)}_2/\partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1}\cdot\frac{\partial h^{(2)}_1}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2}\cdot\frac{\partial h^{(2)}_2}{\partial w^{(1)}_{2,2}}$$

Problem 11.3.4. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial h^{(2)}_1/\partial h^{(1)}_2$, $\partial h^{(2)}_2/\partial h^{(1)}_2$, and $\partial h^{(1)}_2/\partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1}\cdot\frac{\partial h^{(2)}_1}{\partial h^{(1)}_2}\cdot\frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2}\cdot\frac{\partial h^{(2)}_2}{\partial h^{(1)}_2}\cdot\frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} \tag{11.10}$$

Some readers may notice that (11.10) is just the result of chain rule in multivariable calculus. It shows that the effect of $w^{(1)}_{2,2}$ on o_1 is the sum of its effect through all possible paths that the values propagate through the network, and the amount of effect for each path can be calculated by multiplying the appropriate partial derivative between each layer.

In this section, we calculated by hand one partial derivative $\partial o_1/\partial w^{(1)}_{2,2}$. But in general, to compute the loss gradient, we see below that we want to calculate the partial derivative of each output node o_i with respect to each weight in the network. Manually applying chain rule for each partial derivative as in (11.10) is too inefficient.[3] Instead,

[3] Putting on your COS 226 hat, you can check that the computational cost of this naive method scales quadratically in the size of the network.

in the next section, we will learn how to utilize matrix operations to combine the computation for multiple partial derivatives into one process.[4]

[4] This efficiency holds even without taking into account the fact that today's GPUs are optimized for fast matrix operations.

11.4 Backpropagation: The General Case

11.4.1 Jacobian Matrix

Suppose some vector $\vec{\mathbf{y}} = (y_1, y_2, \ldots, y_m) \in \mathbb{R}^m$ is a function of $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ — that is, there is a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ such that $\vec{\mathbf{y}} = f(\vec{\mathbf{x}})$, or equivalently, if there are m functions $f_i : \mathbb{R}^n \to \mathbb{R}$ for each $i = 1, 2, \ldots, m$ such that $y_i = f_i(\vec{\mathbf{x}})$.

Then the *Jacobian matrix* of $\vec{\mathbf{y}}$ with respect to $\vec{\mathbf{x}}$, denoted as $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$, is an $m \times n$ matrix whose (i, j) entry is the partial derivative $\partial y_i / \partial x_j$. Note that each entry of this matrix is itself a function of $\vec{\mathbf{x}}$. A bit confusingly, a Jacobian matrix is also often denoted as $\partial \vec{\mathbf{y}} / \partial \vec{\mathbf{x}}$ when it is clear from the context that $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ are vectors and hence this object is not a partial derivative or gradient. [5]

[5] Note that the i-th row of the Jacobian matrix contains the gradient of y_i, *i.e.* the i-th coordinate of $\vec{\mathbf{y}}$.

The mathematical interpretation of the Jacobian matrix is that if we change $\vec{\mathbf{x}}$ such that each coordinate x_i is updated to $x_i + \delta_i$ for an infinitesimal value δ_i, then the output $\vec{\mathbf{y}}$ changes to $\vec{\mathbf{y}} + J(\vec{\mathbf{y}}, \vec{\mathbf{x}})\vec{\boldsymbol{\delta}}$.

Example 11.4.1. *Suppose $\vec{\mathbf{y}}$ is a linear function of $\vec{\mathbf{x}}$ — that is, there exists a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ such that $\vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}}$. Then notice that y_i, the i-th coordinate of $\vec{\mathbf{y}}$, can be expressed as*

$$y_i = \mathbf{A}_{i,*}\vec{\mathbf{x}} = A_{i,1}x_1 + A_{i,2}x_2 + \ldots + A_{i,n}x_n$$

Notice that the partial derivative $\partial y_i / \partial x_j$ is equal to A_{ij}. This means that the (i, j) entry of the Jacobian matrix is the (i, j) entry of the matrix $\mathbf{A}$, and hence $J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = \mathbf{A}$.

Problem 11.4.2. *If $\vec{\mathbf{y}} \in \mathbb{R}^2$ is a function of $\vec{\mathbf{x}} \in \mathbb{R}^3$ such that*

$$y_1 = 2x_1 - x_2 + 3x_3$$
$$y_2 = -x_1 + 2x_3$$

then what is the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$?

Example 11.4.3. *If $\vec{\mathbf{x}} \in \mathbb{R}^n$ and $\vec{\mathbf{y}} = ReLU(\vec{\mathbf{x}}) \in \mathbb{R}^n$, then notice that*

$$\frac{\partial y_i}{\partial x_i} = \begin{cases} 1 & x_i > 0 \\ 0 & otherwise \end{cases}$$

We can also denote this with an indicator function $\mathbb{1}(x_i > 0)$. Also for any $j \neq i$, we see that $\partial y_i / \partial x_j = 0$. Therefore, the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$ is a diagonal matrix whose entry down the diagonal is $\mathbb{1}(x_i > 0)$; that is

$$J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\vec{\mathbf{x}} > 0))$$

where we take the indicator function element-wise to the vector $\vec{\mathbf{x}}$.

Definition 11.4.4 (Jacobian Chain Rule)**.** *Suppose vector* $\vec{\mathbf{z}} \in \mathbb{R}^\ell$ *is a function of* $\vec{\mathbf{y}} \in \mathbb{R}^m$ *and* $\vec{\mathbf{y}}$ *is a function of* $\vec{\mathbf{x}} \in \mathbb{R}^n$*, then by the chain rule, the Jacobian matrix* $J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) \in \mathbb{R}^{\ell \times m}$ *is represented as the matrix product:*

$$J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}}) J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) \tag{11.11}$$

In context of the feedforward neural network, each hidden layer is a function of the previous layer. Specifically, vector of activations of a hidden layer is a function of the vector of activations of the previous layer as well as of the trainable weights within the layer.

Example 11.4.5 (Gradient calculation for a single layer with ReLU's)**.** *If* $\vec{\mathbf{x}} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m \times n}, \vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}} \in \mathbb{R}^m$ *and* $\vec{\mathbf{z}} = ReLU(\vec{\mathbf{y}}) \in \mathbb{R}^m$*, then the Jacobian matrix* $J(\vec{\mathbf{z}}, \vec{\mathbf{x}})$ *can be calculated as*

$$J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}}) J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\mathbf{A}\vec{\mathbf{x}} > 0))\mathbf{A}$$

11.4.2 *Backpropagation — Efficiency Using Jacobian Viewpoint*

Now we return to backpropagation, and show how the Jacobian viewpoint allows computing the gradient of the loss (with respect to network parameters) with a number of mathematical operations (*i.e.*, additions and multiplications) proportional to the size of the fully connected net.

Recall that we want to find the weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(o)}$ that minimize the cross-entropy loss ℓ. To apply the standard/stochastic gradient descent algorithm, we need to find the partial derivative $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}_{i,j}}$ of the loss function with respect to each weight $\mathbf{W}^{(k)}_{i,j}$ of each layer k.

To simplify notations, we introduce a new matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ which has the same dimensions as $\mathbf{W}^{(k)}$ (*e.g.*, $\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} \in \mathbb{R}^{2 \times 3}$ in Figure 11.2) and the (i, j) entry of the matrix is:

$$\left(\frac{\partial \ell}{\partial \mathbf{W}^{(k)}} \right)_{i,j} = \frac{\partial \ell}{\partial \mathbf{W}^{(k)}_{i,j}} \tag{11.12}$$

for any layer k. The matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ will be called the gradient with respect to the weights of the k-th layer. [6] Now the update rule for the gradient descent algorithm can be written as the following:

$$\mathbf{W}^{(k)} \to \mathbf{W}^{(k)} - \eta \cdot \frac{\partial \ell}{\partial \mathbf{W}^{(k)}} \tag{11.13}$$

where η is the learning rate. Now the question remains as how to calculate these gradients. As the name "backpropagation" suggests, we will first compute the gradient of the loss ℓ with respect to the output nodes; we then inductively compute the gradient for the previous layers, until we reach the input layer.

[6] Alternatively, you can think of flattening $\mathbf{W}^{(k)}$ into a single vector, then finding the Jacobian matrix $\partial \ell / \partial \mathbf{W}^{(k)}$, and later converting it back to a matrix form.

1. Output Layer: First recall that the cross-entropy loss is

$$\begin{aligned}\ell &= -\log\left(\frac{e^{o_y}}{\sum_{i=1}^{d_{out}} e^{o_i}}\right)\\ &= -\log(e^{o_y}) + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\\ &= -o_y + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\end{aligned}$$

where $y \in \{1, 2, \ldots, d_{out}\}$ is the ground truth value. Therefore, the gradient with respect to the output layer is

$$\frac{\partial \ell}{\partial o_i}_{1\le i\le d_{out}} = \begin{cases} -1 + \widehat{o}_i & y = i \\ \widehat{o}_i & y \neq i \end{cases}$$

To simplify notations, we introduce a *one-hot encoding vector* $\vec{\mathbf{e}}_y$, which has 1 only at the y-th coordinate and 0 everywhere else. Then, we can rewrite the equation above as: [7]

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\widehat{\mathbf{o}}} - \vec{\mathbf{e}}_y\right)^\intercal \in \mathbb{R}^{1\times d_{out}} \tag{11.14}$$

This is the Jacobian matrix of the loss ℓ with respect to the output layer $\vec{\mathbf{o}}$.

[7] Note that $\partial\ell/\partial\vec{\mathbf{o}}$, the term on the left hand side, is a Jacobian matrix in $\mathbb{R}^{1\times d_{out}}$. But $\vec{\widehat{\mathbf{o}}}$ and $\vec{\mathbf{e}}_y$, the terms on the right hand side, are both column vectors, or equivalently a $d_{out} \times 1$ matrix. We resolve the problem by taking the transpose.

2. Jacobian With Respect To Hidden Layer: We first compute $\partial\ell/\partial\vec{\mathbf{h}}^{(L-1)}$, the Jacobian matrix with respect to the last hidden layer before the output layer. Since $\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$, we can apply the result from Example 11.4.1 and get

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}} J(\vec{\mathbf{o}}, \vec{\mathbf{h}}^{(L-1)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}}\mathbf{W}^{(o)}\end{aligned} \tag{11.15}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian matrix) $\partial\ell/\partial\vec{\mathbf{h}}^{(k+1)}$. We now compute $\partial\ell/\partial\vec{\mathbf{h}}^{(k)}$ using the result from Example 11.4.5.

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} J(\vec{\mathbf{h}}^{(k+1)}, \vec{\mathbf{h}}^{(k)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)}\end{aligned} \tag{11.16}$$

3. Gradient With Respect to Weights: We first compute $\partial\ell/\partial\mathbf{W}^{(o)}$, the gradients with respect to the weights of the output layer. Notice that

a particular weight $w_{i,j}^{(o)}$ is only used in computing o_i out of all output nodes:

$$o_i = w_{i,1}^{(o)} h_1^{(L-1)} + \ldots + w_{i,j}^{(o)} h_j^{(L-1)} + \ldots + w_{i,d_{L-1}}^{(o)} h_{d_{L-1}}^{(L-1)}$$

Therefore, the gradient with respect to $w_{i,j}^{(o)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot h_j^{(L-1)}$$

We can combine these results into the following matrix form

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^\intercal \left(\vec{\mathbf{h}}^{(L-1)}\right)^\intercal \tag{11.17}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian) $\partial \ell / \partial \vec{\mathbf{h}}^{(k)}$. We now compute $\partial \ell / \partial \mathbf{W}^{(k)}$.

To do this, we introduce an intermediate variable $\vec{\mathbf{z}}^{(k)} = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$ such that $\vec{\mathbf{h}}^{(k)} = ReLU(\vec{\mathbf{z}}^{(k)})$. Then the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot \frac{\partial z_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot h_j^{(k-1)}$$

We can combine these results into the following matrix form

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{W}^{(k)}} &= \left(\frac{\partial \ell}{\partial \vec{\mathbf{z}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= \left(J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^\intercal \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= diag(\mathbb{1}(\mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal
\end{aligned} \tag{11.18}$$

4. Full Backpropagation Process We summarize the results above into the following four steps:

1. Compute the Jacobian matrix with respect to the output layer, $\frac{\partial \ell}{\partial \vec{\mathbf{o}}} \in \mathbb{R}^{1 \times d_{out}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\hat{\mathbf{o}}} - \vec{\mathbf{e}}_y\right)^\intercal \qquad \text{((11.14) revisited)}$$

2. Compute the Jacobian matrix with respect to the last hidden layer, $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} \in \mathbb{R}^{1 \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} = \frac{\partial \ell}{\partial \vec{\mathbf{o}}} \mathbf{W}^{(o)} \qquad \text{((11.15) revisited)}$$

Then, compute the gradient with respect to the output weights, $\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} \in \mathbb{R}^{d_{out} \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^\intercal \left(\vec{\mathbf{h}}^{(L-1)}\right)^\intercal \qquad \text{((11.17) revisited)}$$

3. For each successive layer k, calculate the Jacobian matrix with respect to the k-th hidden layer $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} \in \mathbb{R}^{1 \times d_k}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)} \quad \text{((11.16) revisited)}$$

Next, compute the gradient with respect to the $(k+1)$-th hidden layer weights $\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} \in \mathbb{R}^{d_{k+1} \times d_k}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k)}\right)^\intercal$$
((11.18) revisited)

4. Repeat Step 3 until we reach the input layer.

Note that these instructions are based on a model that adopts the cross-entropy loss and the *ReLU* activation function. Using alternative losses and/or activation functions would result in a similar form, although the actual derivatives may be slightly different.

Problem 11.4.6. *(i) Show that if $\boldsymbol{A}$ is an $m \times n$ matrix and $\vec{\mathbf{h}} \in \mathbb{R}^n$ then computing $\boldsymbol{A}\vec{\mathbf{h}}$ requires mn multiplications and m additions. (ii) Using the previous part, argue that the number of arithmetic operations (additions or multiplications) in backpropagation algorithm on an FC net with ReLU activations is proportional to the number of parameters in the net.*

While the above calculation is in line with your basic algorithmic training, it doesn't exactly describe running time in modern ML environments with GPUs, since certain operations are parallelized, and compilers are optimized to run backpropagation as fast as possible.

11.4.3 Using a Different Activation Function

We briefly consider what happens if we choose a different activation function for the hidden layers. Consider the sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$. Its derivative is given by:

$$\begin{aligned} \sigma'(z) &= \frac{1}{1+e^{-z}} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z) \cdot \left(\frac{e^{-z}}{1+e^{-z}}\right) \\ &= \sigma(z) \cdot (1-\sigma(z)) \end{aligned} \qquad (11.19)$$

There is also the hyperbolic tangent function $\tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}$.

Problem 11.4.7. *Compute $f'(z)$ for $f(z) = \tanh(z)$; show how $f'(z)$ can be written in terms of $f(z)$.*

Problem 11.4.8. *Say a neural network uses an activation function $f(z)$ at layer $k+1$ such that $f'(z)$ is a function of $f(z)$. That is, $f'(z) = g(f(z))$ for some function g. Then verify that (11.16, 11.18) can be rewritten as:*

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)}))\mathbf{W}^{(k+1)}$$
$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)})) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k)}\right)^{\mathsf{T}}$$

11.4.4 Final Remark

Directly following the steps of backpropagation is complicated and involves a lot of calculations. But remember that backpropagation is simply *computing gradients by the chain rule*. At a high level, we can think of the loss as a function of inputs and all the weights and note that backpropagation simply entails calculating derivatives with respect to each variable. The good news is that modern deep learning software does all the gradient calculations for users. All the model designer needs to do is to determine the neural network architecture (*e.g.*, choose number of layers, number of hidden units, and the activation functions).

One note of caution is that the loss function for deep neural nets is highly non-convex with respect to the parameters of the network. Just as we discussed in Chapter 3, the gradient descent algorithm is not guaranteed to find the actual minimizer in such situation, and the choice of the initial values of the parameters matter a lot.

11.5 Feedforward Neural Network in Programming

In this section, we discuss how to write Python code to build neural network models and perform forward propagation and backpropagation. As usual, we use the *numpy* package to speed up computation. Additionally, we use the *torch* package to easily design and train the neural network.

```
# import necessary packages
import numpy as np
import torch
import torch.nn as nn

# define the neural network
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                        hidden_dim3=2):
```

```
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                            ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                            , [1., 2.]]))

        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                             0., 0.]]))

        self.activation = nn.ReLU()

    # single step of forward propagation
    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3

net = Net()

# forward propagation with sample input
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)

# backpropagation with sample input
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                    ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

We start the code by importing all necessary packages.

```
import numpy as np
impoort torch
import torch.nn as nn
```

With *PyTorch,* we can design the architecture of any neural network by defining the corresponding *class*.

```
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                        hidden_dim3=2):
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                            ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                            , [1., 2.]]))
```

```
        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                            0., 0.]]))

        self.activation = nn.ReLU()

    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3
```

In the constructor, we define all the layers and activation functions we are going to use in the network. In particular, we specify that we need fully-connected layers by making instances of the *nn.Linear* class and that we need *ReLU* activation function by making an instance of the *nn.Relu* class. Then in the *forward()* function, we specify the order in which to apply the layers and activations. See Figure 11.7 for a visualization of this neural network architecture.

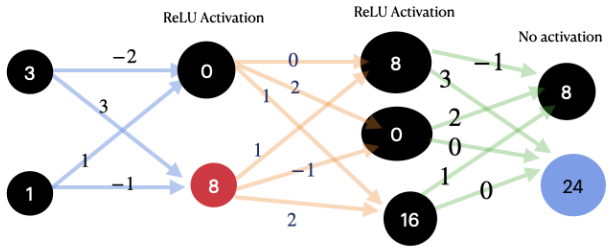


Figure 11.7: A sample feedforward neural network with two hidden layers and two output nodes.

We can simulate one step of forward propagation by calling the *forward()* function of the class *Net* we defined.

```
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)
```

Similarly, we can implement backpropagation by specifying which loss function we want to use, and calling its *backward()* function.

```
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                    ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

Each function call of *backward()* will evaluate the gradients of *loss* with respect to every parameter in the network. Gradients can be manually accessed through the following code.

```
print(net.hidden1.weight.grad)
```

Note that calling *backward()* multiple times will cause gradients to accumulate. While we do not update model weights in this code sample, it is important to periodically clear the gradient buffer when doing so to prevent unintended training. [8] We will discuss how to do this in the next chapter.

[8] For more information, see https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html

12
Convolutional Neural Network

In Chapter 11, we focused on a type of a neural network called feed-forward neural networks. But different data has different structure (*e.g.*, image, text, audio, etc.) and we need better ways of exploiting them. This can help reduce the number of parameters needed in the network, which may allow easier or more data-efficient training. In this chapter, we present a type of a neural network common in image processing called *Convolutional Neural Network* (CNN); these models use a mathematical technique called convolution in order to extract important visual features from input images.

12.1 Introduction to Convolution

Roughly speaking, *convolution* refers to a mathematical operation where two functions are "mixed" to output a new function. In machine learning, the main idea of convolution is to reuse the same set of parameters on different portions of input. This is particularly effective at exploiting the structure of images. It was originally motivated by studies of the structure of cortical cells in the $V1$ visual cortex of the human brain (Hubel and Wiesel won the Nobel Prize in 1981 for this breakthrough discovery). [1] Let's first consider an example of a $1D$ convolution.

[1] Paper: https://www.jstor.org/stable/24965293.

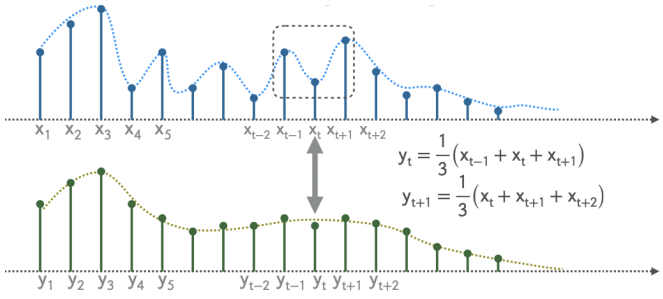


Figure 12.1: The effects of $1D$ convolution on graph of COVID-19 positive cases.

Example 12.1.1. *Consider the 3-day moving average of daily COVID-19 cases as shown in Figure 12.1. Let x_t denote the number of daily cases on day t. We can then take three consecutive values, compute their average and create a new output sequence from averages: $y_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$. Then if we set $w_1 = w_2 = w_3 = \frac{1}{3}$ and denote $\vec{\mathbf{w}} = (w_1, w_2, w_3)$, we can write:* [2]

$$y_t = w_1 x_{t-1} + w_2 x_t + w_3 x_{t+1} = \vec{\mathbf{w}} \cdot (x_{t-1}, x_t, x_{t+1})$$
$$y_{t+1} = w_1 x_t + w_2 x_{t+1} + w_3 x_{t+2} = \vec{\mathbf{w}} \cdot (x_t, x_{t+1}, x_{t+2})$$
$$y_{t+2} = w_1 x_{t+1} + w_2 x_{t+2} + w_3 x_{t+3} = \vec{\mathbf{w}} \cdot (x_{t+1}, x_{t+2}, x_{t+3})$$

Notice that we are reusing the same weights and applying them to multiple different values of x_t to calculate y_t. It is almost like sliding a filter down the array of x_t's and applying it to every set of 3 consecutive inputs. For this reason, we call $\vec{\mathbf{w}}$ the ***convolution filter weight*** *of length 3.*

[2] If we set the weights to a different value, we can find a weighted moving average.

Example 12.1.2. *Consider an input sequence $\vec{\mathbf{x}} = (2, 1, 1, 7, -1, 2, 3, 1)$ and a convolution filter $\vec{\mathbf{w}} = (3, 2, -1)$. The first two output values will be:*

$$y_1 = 2 \times 3 + 1 \times 2 + 1 \times (-1) = 7$$
$$y_2 = 1 \times 3 + 1 \times 2 + 7 \times (-1) = -2$$

Following a similar calculation for the other values, we see that the full output sequence is $\vec{\mathbf{y}} = (7, -2, 18, 17, -2, 11)$. Note that the length of $\vec{\mathbf{y}}$ should be $|\vec{\mathbf{x}}| - |\vec{\mathbf{w}}| + 1 = 8 - 3 + 1 = 6$.

Problem 12.1.3. *If $y_t = 2x_{t-1} - x_{t+1}$, $y_{t+1} = 2x_t - x_{t+2}$, and $y_{t+2} = 2x_{t+1} - x_{t+3}$, what is the convolution filter weight?*

12.2 Convolution in Computer Vision

In this section, we now focus on the application of convolution in computer vision. By the nature of image data, we will be primarily dealing with 2D convolution. Generally, 2D convolution filters are called *kernels*.

Figure 12.2: The effect of local smoothing on a sample image. (The person depicted here is Admiral Grace Murray Hopper, a computing pioneer.)

Example 12.2.1 (Local Smoothing (Blurring))**.** *An image can be blurred by constructing a filter that replaces each pixel by the average of neighboring pixels:*

$$y_{i,j} = \frac{1}{9} \sum_{b_1, b_2 \in \{-1,0,1\}} x_{i+b_1, j+b_2}$$

An example is shown in Figure 12.2.

Figure 12.3: The effect of local sharpening on a sample image

Example 12.2.2 (Local Sharpening (Edge Detection))**.** *The edge of objects in an image can be detected by constructing a filter that replaces each pixel by its difference with the average of neighboring pixels:*

$$y_{i,j} = x_{i,j} - \frac{1}{9} \sum_{b_1,b_2 \in \{-1,0,1\}} x_{i+b_1,j+b_2}$$

An example is shown in Figure 12.3.

12.2.1 *Convolution Filters for Images*

Computationally, we perform 2D convolution on an image by "sliding" the filter around every possible location in the image and taking the inner product:

$$y_{i,j} = \sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s} \tag{12.1}$$

The result is a new image and we can view each filter as a transformation which takes an image and returns an image. In the above equation, the filter size is $(2k+1) \times (2k+1)$. For example, if $k = 1$, we can consider the convolution weight filter to be

$$\begin{bmatrix} w_{-1,-1} & w_{-1,0} & w_{-1,+1} \\ w_{0,-1} & w_{0,0} & w_{0,+1} \\ w_{+1,-1} & w_{+1,0} & w_{+1,+1} \end{bmatrix}$$

An image of size $m \times n$ can only be applied the filter at a location where the filter completely fits inside the image. Therefore, the locations in the input image where the center of the filter can be placed are $k < i \le m - k$, $k < j \le n - k$ and the size of the output image is $(m - 2k) \times (n - 2k)$.

Example 12.2.3. *If the input and convolution filter are given as following:*

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{W} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

then the pixel at $(1,1)$ *of the resulting image can be calculated by applying the filter at the top left corner of the input image. That is, we take the inner product of the following parts (in red) of the two matrices*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

which is

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

Therefore, the $(1,1)$ *entry of the resulting image is* 4*. Similarly, the remaining pixels of the resulting image can be calculated by moving around the filter as in Figure 12.4. The output image is given as:*

$$\mathbf{Y} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

In this example, $\mathbf{X} \in \mathbb{R}^{5\times5}, \mathbf{W} \in \mathbb{R}^{3\times3}, k = 1$ *and* $\mathbf{Y} \in \mathbb{R}^{3\times3}$*.*

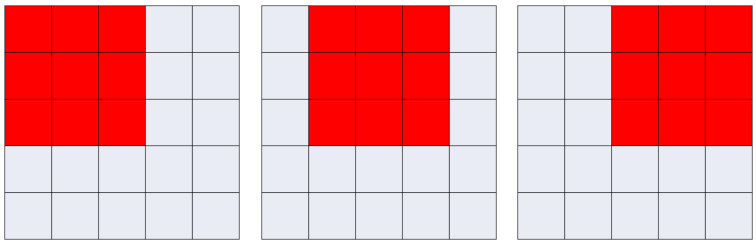


Figure 12.4: Visual representation of applying a 3×3 convolutional filter to a 5×5 image.

Problem 12.2.4. *Suppose we have a* 10×10 *image and a* 5×5 *filter. What is the size of the output image?*

Figure 12.5 shows some common filters used in image processing. Note that all these filters are hand-crafted and require domain-specific knowledge. However, in convolutional neural networks, we don't set these weights by hand and we learn all the filter weights from the data!

12.2.2 *Padding*

In standard 2*D* convolution, the size of the output image is not equal to the size of the input image because we only consider locations where the filter fits completely in the image. However, sometimes we may want their sizes to be the same. In such a case, we apply a

Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&-476&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	

Figure 12.5: Some common filters and corresponding weights used in image processing. Source: https://en.wikipedia.org/wiki/Kernel_(image_processing)

technique called *padding*. The idea is to pad pixels to all four edges of the input image (left, right, up, and down) so that the number of valid locations for the filter is the same as the number of pixels in the original image. In particular, if the filter size is $(2k+1) \times (2k+1)$, we need to pad k pixels on each side.

There are multiple ways to implement padding. *Zero padding* is when the values at all padded pixels are set to 0. *"Same" padding* is when the values at padded pixels are set to the value of the nearest pixel at the edge of the input image. In practice, zero padding is a more common form of padding (it is equivalent to adding a black frame around the image).

12.2.3 Downsampling Input with Stride

Another common operation in convolutional neural networks is called *stride*. Stride controls how the filter convolves around the input image. Instead of moving the filter by 1 pixel every time, we can also move the filter every 2 (or in general, s) pixels. Essentially, we are applying each of the filter weights at fewer locations of the image than before. This can be viewed as a downsampling strategy, which gives a smaller output image while greatly preserving the information from the original input.

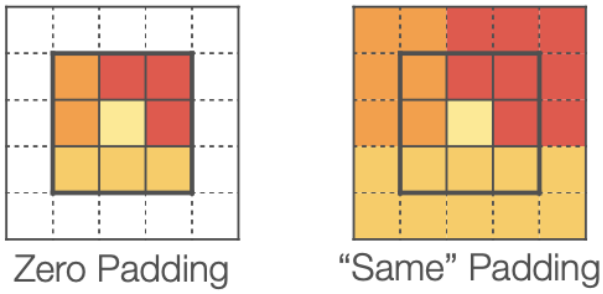


Figure 12.6: A visual comparison between two common types of padding: zero padding and "same" padding

Suppose we have an input image of size $m \times n$ and a filter of size $(2k+1) \times (2k+1)$. If padding is applied to the image, the output image size is $\lfloor (m+s-1)/s \rfloor \times \lfloor (n+s-1)/s \rfloor$. [3] If padding is not applied to the image, the output image size is $\lfloor (m+s-2k-1)/s \rfloor \times \lfloor (n+s-2k-1)/s \rfloor$; this is because convolution with stride is performed directly on the input image itself, making the effective input image size $(m-2k) \times (n-2k)$.

[3] As a sanity check, you can verify that in the special case of $s = 1$ the output image size will be the same as the input image size

Example 12.2.5. *Suppose we have an input image of size* 5×5*. If we apply padding and take stride size* $s = 2$*, then output size is* 3×3*.*

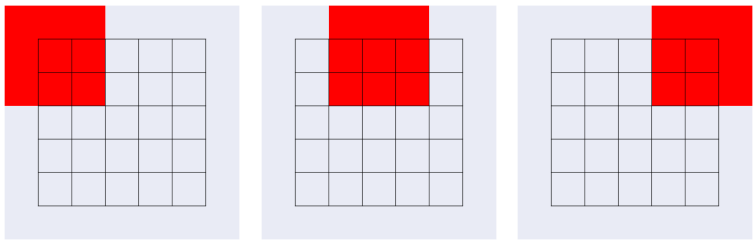


Figure 12.7: Visual representation of applying a 3×3 convolutional filter to a 5×5 image with padding and stride size 2.

12.2.4 *Nonlinear Convolution*

For each location in the image (original or padded) and a single convolution filter, we can apply a nonlinear activation function after the convolution

$$y_{i,j} = g\left(\sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s}\right) \tag{12.2}$$

where g is some function like *ReLU*, sigmoid, tanh. The intuition is similar to what we had earlier in feedforward neural networks — if we don't add non-linear activation functions, a multi-layered convolutional neural network can be easily reduced to a linear model!

12.2.5 *Channels*

In general, we do not only use one convolution filter. We construct a network of multiple layers, and for each of the layers, we apply multiple convolutional filters. Different filters will be able to detect different features of the image (*e.g.*, one filter detects edges and one filter detects dots), and we want to apply different filters independently on the input image. The result of applying a given filter on a single input image is called a *channel*. We can stack the channels to create a $3D$ structure, as shown in Figure 12.8.

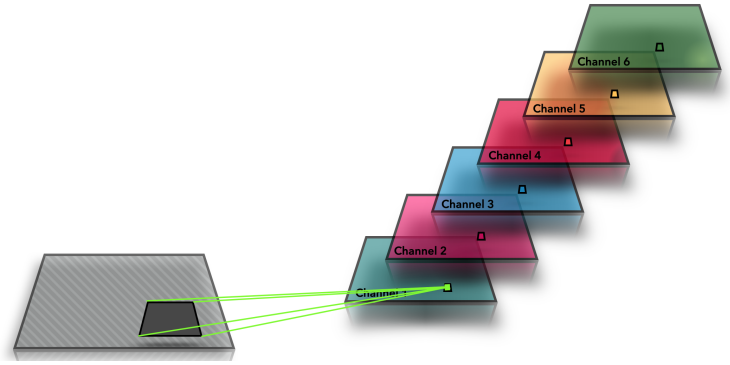


Figure 12.8: Each filter creates one channel. The output of a convolutional layer has multiple output channels.

Next, let's imagine that we want to build a deep neural network with multiple convolutional layers (state-of-the-art CNNs have 100 or even 1000 layers!). A typical convolutional layer in the middle of the network will have several input channels (equivalent to the number of output channels from the previous layer) and multiple output channels. How can we determine the number of filters needed?

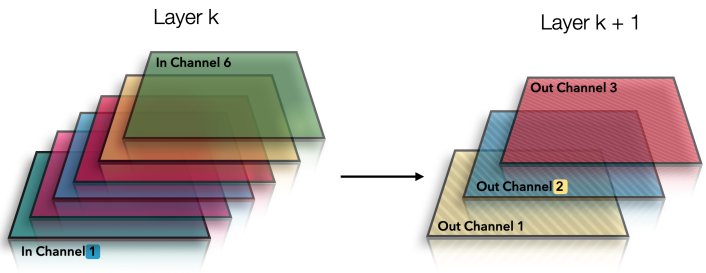


Figure 12.9: A convolutional layer which has multiple input and multiple output channels.

In this case, we want to define a filter for every possible pair of input and output channels. The output image of a particular output channel will be the summation of the output images from each of the input channels, after applying the corresponding filter. We can also add a nonlinear activation function g after taking the summation of the output images. That is, (12.2) can be rewritten for the output

image in the v-th output channel as:

$$y_{i,j}^{(v)} = g\left(\sum_{u=1}^{n_{in}} \sum_{-k\le r,s\le k} w_{r,s}^{(u,v)} x_{i+r,j+s}^{(u)}\right) \tag{12.3}$$

where n_{in} is the number of input channels, $\mathbf{X}^{(u)}$ is the image at the u-th input channel, $\mathbf{Y}^{(v)}$ is the image at the v-th output channel, and $\mathbf{W}^{(u,v)}$ is the filter between the u-th input channel and the v-th output channel.

Example 12.2.6. *Assume there are 6 input channels and 3 output channels, and the filter size is* 5×5*. Then for every* 6×3 *pair of input and output channel, we have a kernel of weights of size* 5×5*, so there are a total of* $6 \times 3 \times 5 \times 5 = 450$ *weights.*

12.2.6 Pooling

Pooling is another popular way to reduce the size of the output of a convolutional layer. In contrast to stride, which applies convolution operation every s pixels, pooling partitions each image (channel) to patches of size $\Delta \times \Delta$ and performs a reduction operation on each patch. You can think of this as similar to what happens when you lower the resolution of an image. The reduction operation can either involve taking the max of all the values in the patch ("max-pooling"):

$$y_{i,j} = \max_{1\le r,s\le \Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

or taking the average of all the values in the patch ("mean-pooling"):

$$y_{i,j} = \frac{1}{\Delta^2}\sum_{r,s=1}^{\Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

The pooling operation can reduce the image size by a factor of Δ^2.

If the input image is of size $m \times n$, the size of the image after pooling will be $\lfloor m/\Delta \rfloor \times \lfloor n/\Delta \rfloor$.

Example 12.2.7. *If the size of an input image to a pooling layer is* 6×6 *and* $\Delta = 2$*, then the output is of size* 3×3*.*

12.2.7 A Full Convolutional Neural Network

Let's put everything together and consider a full convolutional neural network. Figure 12.11 shows a typical example of a convolutional neural network. A convolutional neural network typically begins by stacking multiple convolutional layers and pooling layers. Each convolutional layer has its own kernel size and number of output channels; similarly, each pooling layer has its own kernel size. This is

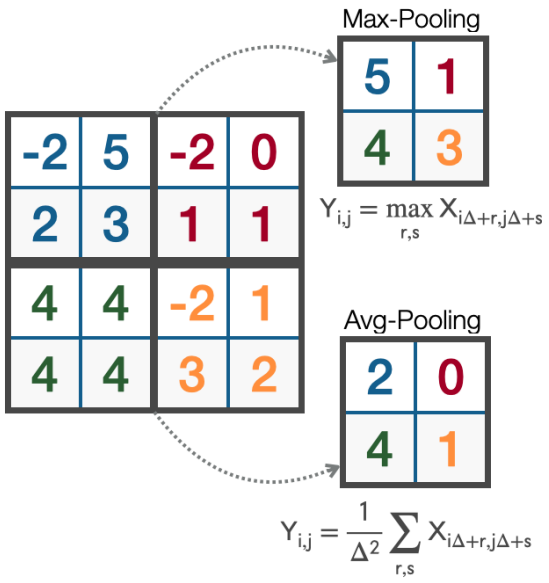


Figure 12.10: Max-pooling vs mean-pooling.

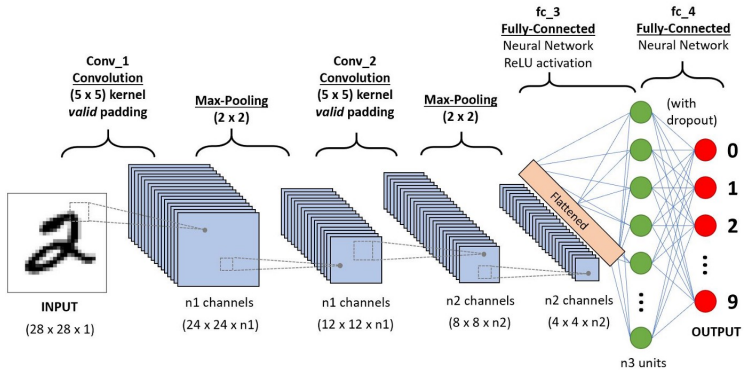


Figure 12.11: A illustration of a full convolutional neural network.

followed by several fully-connected layers at the end. Since the output images of convolutional layers are 2-dimensional, it is customary to "flatten" the images into a $1D$ vector (*i.e.*, append one row after another) before applying the fully connected layers. Intuitively, we can think of the convolutional and pooling layers as learning interesting image features (*e.g.*, stripes or dots) while the fully-connected layers map these features to output classes (*e.g.*, zebras have a lot of stripes).

All the weights in a convolutional neural network (including weights in kernels, fully-connected layers) can be learned via the backpropagation algorithm in Chapter 11. Again, modern deep learning libraries (*e.g.*, PyTorch, Tensorflow) have all the convolutional and pooling layers implemented and can compute gradients

automatically!

Finally, the above convolutional neural network is still a simple network, compared to modern convolutional neural networks. Interested students can look up architectures such as AlexNet, Inception, VGG, ResNet and DenseNet.

12.2.8 Designing a Convolutional Network

While we described convolutional nets above, we did not give a good explanation of why they are well-suited to solve vision tasks. Working through the next few examples will help you understand their power. The idea is that convolution is a parameter-efficient [4] architecture that can "search for patterns" anywhere in the image. For example, suppose the net has to decide if the input image has a triangle anywhere in it. If the net were fully connected, it would have to learn how to detect triangles centered at every possible pixel location (i, j). By contrast, if a simple convolutional filter can detect a triangle, we can just replicate this filter in all patches to detect a triangle anywhere in the image.

[4] Which usually goes with sample-efficiency!

Now consider the CNN architecture in Figure 12.12. The architecture has two convolutional layers, the first with a *ReLU* activation function, and the second with a sigmoid activation function. [5] We will choose an appropriate convolutional weight and bias such that the architecture can detect a particular simple visual pattern.

[5] In both Example 12.2.8 and Example 12.2.9, the second convolutional layer can be considered a fully connected layer if we flatten image Y

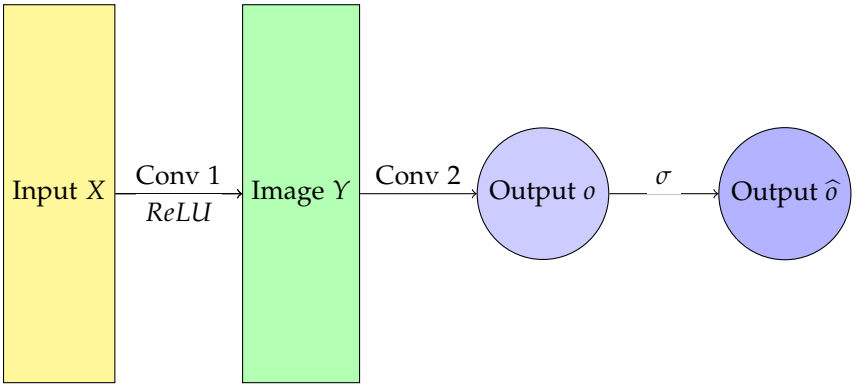


Figure 12.12: A sample CNN architecture that can be used to detect the patterns as aligned in Example 12.2.8 and Example 12.2.9.

Example 12.2.8. *The input to the network is a gray-scale image of size* 8×8 *(*1 *channel), and each pixel takes an integer value between* 0 *and* 255*, inclusive. If at least one pixel of the image has value exactly* 255*, the output of the CNN should have a value close to* 1 *and otherwise the output should have a value close to* 0*.*

We will now solve Example 12.2.8 by individually configuring the parameters for each convolutional layer in Figure 12.12. The first

convolutional layer will have a 1×1 filter of weight 1, a bias of -254, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU(x_{i,j} - 254) \qquad 1 \leq i, j \leq 8$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. Notice that this value is zero everywhere, except if $x_{i,j} = 255$, in which case $y_{i,j}$ takes the value 1. That is,

$$y_{i,j} = \begin{cases} 1 & x_{i,j} = 255 \\ 0 & otherwise \end{cases}$$

See Figure 12.13 to see the effect of this choice of convolutional layer on a sample image. We see that we have now successfully identified the pixels in the input image that take the value 255.

$$\begin{bmatrix} 0 & 100 & 200 \\ 50 & 150 & 250 \\ 55 & 155 & 255 \end{bmatrix} \xrightarrow{\text{Conv } 1} \begin{bmatrix} -254 & -154 & -54 \\ -204 & -104 & -4 \\ -199 & -99 & 1 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 12.13: The effect of the choice of first convoluational layer for Example 12.2.8 on a sample image. Only a portion of the image is shown.

Next, consider the second convolutional layer with a 8×8 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. [6] The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=1}^{8} 10 y_{i,j} \right) - 5$$

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, $x_{i,j} = 255$). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one pixel in the input image whose value is 255. That is

$$o = \begin{cases} \geq 5 & \exists (i, j) : x_{i,j} = 255 \\ -5 & otherwise \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \geq 0.99 & \exists (i, j) : x_{i,j} = 255 \\ 0.01 & otherwise \end{cases}$$

This is exactly what we wanted in Example 12.2.8.

[6] Once the output image of the first convolutional layer is flattened to a vector of length 64, this can also be thought of as a fully-connected layer with input size 64 and output size 1.

Example 12.2.9. *The input to the network is a gray-scale image of size* 8×8 *(*1 *channel), and each pixel takes an integer value between* 0 *and* 255*, inclusive. If any part of the input image contains the following pattern:*

$$\begin{bmatrix} * & 255 & * \\ 255 & 255 & 255 \\ * & 255 & * \end{bmatrix} \tag{12.4}$$

the output of the CNN should have a value close to 1 *and otherwise the output should have a value close to* 0*.*

We use the same architecture as in Figure 12.12, but now with a different choice of parameters for the convolutional layers. The first convolutional layer will have a 3×3 filter with the following weights:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

a bias of -1274, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU\left(x_{i-1,j} + x_{i,j-1} + x_{i,j} + x_{i,j+1} + x_{i+1,j} - 1274\right) \quad 2 \leq i, j \leq 7$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. [7] Notice that this value is zero everywhere, except if $x_{i,j} + x_{i-1,j} + x_{i,j-1} + x_{i,j+1} + x_{i+1,j} = 1275$, in which case $y_{i,j}$ takes the value 1. This can only happen if $x_{i-1,j} = x_{i,j-1} = x_{i,j} = x_{i,j+1} = x_{i+1,j} = 255$. That is, if the input image has the pattern in (12.4) centered around (i, j).

[7] Since there is no padding, the values $y_{1,1}, y_{1,8}, y_{8,1}, y_{8,8}$ are not defined.

$$y_{i,j} = \begin{cases} 1 & \textit{Pattern in (12.4) exists at } (i, j) \\ 0 & \textit{otherwise} \end{cases}$$

See Figure 12.14 to see the effect of this choice of convolutional layer on two sample images.

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 250 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & -4 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 0 & * \\ * & * & * \end{bmatrix}$$

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 255 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & +1 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 1 & * \\ * & * & * \end{bmatrix}$$

Figure 12.14: The effect of the choice of first convoluational layer for Example 12.2.9 on two sample images. Only a portion of the images is shown.

Next, consider the second convolutional layer with a 6×6 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=2}^{7} 10 y_{i,j} \right) - 5$$

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, the pattern exists at (i, j)). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one copy of the given pattern. That is

$$o = \begin{cases} \geq 5 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ -5 & \textit{otherwise} \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \geq 0.99 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ 0.01 & \textit{otherwise} \end{cases}$$

This is exactly what we wanted in Example 12.2.9.

12.3 *Backpropagation for Convolutional Nets*

A convolutional neural network is a special case of a feedforward neural network where we use convolutional layers, instead of fully-connected layers as in Chapter 11. Therefore, we can apply the same basic idea of backpropagation so that we can run the gradient descent algorithm, although the details of the calculation are slightly different.

The biggest difference is that in a fully-connected layer, each weight is used exactly once, while in a convolutional layer, each weight is applied multiple times throughout the input image. [8] This makes the computation for the gradient slightly more convoluted. But the basic idea is the same — identify all paths through which the corresponding weight affects the output of the model and add up the amount of effect for each path.

[8] This phenomenon is also known as *weight sharing*.

Figure 12.15 shows a portion of a sample neural network where weight sharing occurs. That is, the same weight w is used between the following four pairs of nodes: $(x_1, y_1), (x_2, y_2), (x_2, y_3), (x_3, y_4)$. If we wanted to find the gradient $\partial o / \partial w$, we need to consider the four paths that the weight w affects the output: $w \to y_i \to o$ where $1 \leq i \leq 4$.

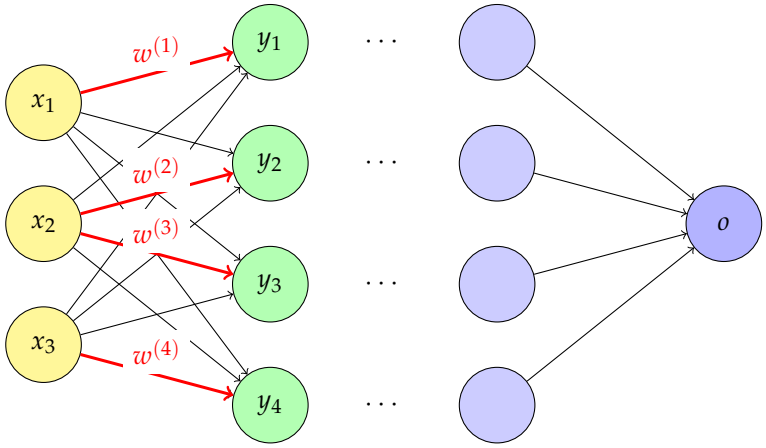


Figure 12.15: A sample neural network where weight sharing occurs. $w^{(1)}, w^{(2)}, w^{(3)}, w^{(4)}$ are the copies of the same weight w.

What we will do is consider the four copies of the weight w as separate weights that will be denoted as $w^{(i)}$ where $1 \leq i \leq 4$. Since these weights are only used in one place in the layer, we are already familiar with computing the gradients $\partial o / \partial w^{(i)}$. Then we will add (or *pool*) these values to get the gradient $\partial o / \partial w$. This works because we can think of each $w^{(i)}$ as a function of w where $w^{(i)} = w$. Then by Chain Rule, we have

$$\frac{\partial o}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}} \cdot \frac{\partial w^{(i)}}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}}$$

12.3.1 *Deriving Backpropagation Formula for Convolutional Layers*

In this subsection, we derive the backpropagation formula for a convolutional layer. (As in many other places, if your instructor did not teach it in COS 324, consider this to be advanced reading.)

Recall that in a fully-connected layer, which computes $\vec{\mathbf{h}}^k = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$, the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be simply computed as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot \frac{\partial h_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot h_j^{(k-1)}$$

This is because the weight $w_{i,j}^{(k)}$ is only used to compute $h_i^{(k)}$ out of all nodes in the next hidden layer. In comparison, consider a convolutional layer, which computes an output image $\mathbf{Y} \in \mathbb{R}^{n \times n}$ from an input image $\mathbf{X} \in \mathbb{R}^{m \times m}$ and filter $\mathbf{W} \in \mathbb{R}^{(2k+1) \times (2k+1)}$. Notice that the weight $w_{i,j}$ is used to compute all of the pixels in the output image. Therefore, we just need to add (or *pool*) the gradient flow from each of these paths. The gradient with respect to a particular weight

$w_{i,j}$ will be

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{1,1}} \cdot \frac{\partial y_{1,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{1,2}} \cdot \frac{\partial y_{1,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot \frac{\partial y_{1,n}}{\partial w_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot \frac{\partial y_{2,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{2,2}} \cdot \frac{\partial y_{2,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot \frac{\partial y_{2,n}}{\partial w_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot \frac{\partial y_{n,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{n,2}} \cdot \frac{\partial y_{n,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot \frac{\partial y_{n,n}}{\partial w_{i,j}}\right)\end{aligned}$$

Assuming there is zero padding, this can be calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{1,1}} \cdot x_{i+1,j+1} + \frac{\partial \ell}{\partial y_{1,2}} \cdot x_{i+1,j+2} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot x_{i+1,j+n}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot x_{i+2,j+1} + \frac{\partial \ell}{\partial y_{2,2}} \cdot x_{i+2,j+2} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot x_{i+2,j+n}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot x_{i+n,j+1} + \frac{\partial \ell}{\partial y_{n,2}} \cdot x_{i+n,j+2} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot x_{i+n,j+n}\right)\end{aligned}$$

Notice that the equation above can be rewritten as

$$\frac{\partial \ell}{\partial w_{i,j}} = \sum_{1 \leq r,s \leq n} \frac{\partial \ell}{\partial y_{r,s}} \cdot x_{i+r,j+s} \tag{12.5}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{W}$ is the output when applying a convolution filter $\partial \ell / \partial \mathbf{Y}$ to the input matrix $\mathbf{X}$.

Similarly, we can try to calculate the Jacobian matrix with respect to the input matrix $\mathbf{X}$. Each input pixel $x_{i,j}$ is used to calculate the output pixels $y_{i+r,j+s}$ where $-k \leq r, s \leq k$. The gradient with respect to a particular input pixel will be

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot \frac{\partial y_{i-k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot \frac{\partial y_{i-k,j+k}}{\partial x_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot \frac{\partial y_{i-k+1,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot \frac{\partial y_{i-k+1,j+k}}{\partial x_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot \frac{\partial y_{i+k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot \frac{\partial y_{i+k,j+k}}{\partial x_{i,j}}\right)\end{aligned}$$

Assuming zero padding, this is calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} = &\left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot w_{k,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot w_{k,-k}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot w_{k-1,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot w_{k-1,-k}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot w_{-k,k} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot w_{-k,-k}\right)\end{aligned}$$

which can be rewritten as

$$\frac{\partial \ell}{\partial x_{i,j}} = \sum_{-k \le r,s \le k} \frac{\partial \ell}{\partial y_{i+r,j+s}} \cdot w_{-r,-s} \tag{12.6}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{X}$ is the output when applying the horizontally and vertically inverted image of $\mathbf{W}$ as the convolutional filter to the input matrix $\partial \ell / \partial \mathbf{Y}$.

12.4 *CNN in Programming*

In this section, we discuss how to write Python code to implement Convolutional Neural Networks (CNN). As usual, we use the *numpy* package to speed up computation and the *torch* package to easily design and train the neural network. We also introduce the *torchvision* package:

- *torchvision*: This package focuses on computer vision applications and is integrated with the broader PyTorch framework. It provides access to pre-built models, popular datasets, and a variety of image transform capabilities. [9]

[9] Documentation is available at https://pytorch.org/vision/stable/index.html

The following code sample implements a CNN and trains it on a single image.

```
# import necessary packages
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# set random seeds to ensure reproducibility
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

```
# load CIFAR10 data
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                          transform=transform)

# helps iterate through the train/test data in batches
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                          num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                          num_workers=0)

# define the CNN architecture
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

# choose the optimization technique to implore
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

# extract one image from the dataset
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)

# forward propagation
net = ConvNet()
output = net(image)

# backpropagation
loss = torch.norm(output - torch.ones(output.shape[1]))**2
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

As usual, we start by importing packages.

```
import random
import numpy as np
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

The *DataLoader* class helps iterate through a dataset in batches.

Next, we fix all random seeds to ensure reproducibility.

```
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

Recall that programming languages on a classical computer can only implement pseudorandom methods, which always produce the same result for a given seed.

Then we load the CIFAR-10 dataset.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                         transform=transform)
```

The CIFAR-10 dataset contains simple images of a single object, and the images are labeled with the category of the objects they contain. Note that we normalize the dataset with a mean of 0.5 and standard deviation of 0.5 per color channel. Figure 12.16 shows a sampling of images from the dataset after the normalization.

Figure 12.16: Sample images from the CIFAR10 dataset.

Next we create *DataLoader* objects to help iterate through the dataset in batches. Each batch will consist of 8 images and 8 labels.

```
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                          num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                          num_workers=0)
```

Then we define our CNN architecture in the *ConvNet* class.

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
```

```
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Just like the FFNN code from the previous chapter, we define all the layers and activations we are going to use in the constructor. Note that in addition to instances of the *nn.Linear* class and the *nn.Relu* class, we also make use of classes like *nn.Conv2d* and *nn.MaxPool2d* which are specifically designed for CNNs.

We extract one training image with the following code.

```
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)
```

The *unsqueeze()* function adds one dimension to the training data. This is called a *batch dimension*. Normally, we would run the training in batches, and the size of the data along the batch dimension will be equal to the number of images in each batch. Here, we only use one image for the sake of exposition.

We can now run forward propagation on a sample image with the code below.

```
net = ConvNet()
output = net(image)
```

We then implement the *squared error* loss. Alternatively, we could have chosen the cross-entropy loss or any other valid loss function.

```
loss = torch.norm(output - torch.ones(output.shape[1]))**2
```

Next, we calculate the gradients of the *loss* with the following line of code

```
loss.backward()
```

and update each of the parameters according to the Gradient Descent algorithm with the following line.

```
optimizer.step()
```

Finally, we reset the values of the gradients to zero with the following code.

```
optimizer.zero_grad()
```

Recall as discussed in the previous chapter that failing to do so will cause unintended training during subsequent iterations of backpropagation. Here, we called the *zero_grad()* function at the end of one iteration of backpropagation, but it may be a good idea to call this function right before calling *backward()*, just in case there are already gradients in the buffer before program execution (*e.g.*, if someone was working with the model beforehand in the interpreter).

In this section, we only showed how to run forward propagation and backpropagation on a single data point. In general, we train the model on the entire dataset multiple times. A single pass over the entire dataset is called an *epoch*.

Part IV

Reinforcement Learning

13
Introduction to Reinforcement Learning

This part of the course concerns *Reinforcement Learning (RL)*, the conceptual underpinning of several modern technologies such as self-driving technologies in new cars. It is the third major category of machine learning, in addition to the two previously seen categories of supervised and unsupervised learning. In class we saw a video of robots (made by Boston Dynamics) doing parkour, dancing, and overall doing a pretty good job of imitating the peak human physique. That is also achieved via RL.

The basic idea of RL involves the concept of an *agent* learning to make a *sequence of actions* in a *dynamic* environment. At each discrete time step, the agent is able to take one of a menu of actions. Each choice of action leads to changes in the state of the world (*i.e.*, the agent and its surroundings). The agent has an internal representation of the current and potential states of the world (*e.g.*, using vision or other sensing modules). Under this setting, the agent takes a sequence of actions towards a certain goal.

The world contains uncertainty due to a variety of factors. For instance, there may be other agents in the environment that also take actions to their own benefit, or the sensing modules may be imperfect. Thus taking the same action from the same state of the world may lead to different evolution of state in the future — that is, RL is *non-deterministic*.

In this chapter, we introduce the basic elements of RL using real-world examples, and what it means for the agent to act optimally. Chapter 14 focuses on the setting where the underlying environment (*e.g.*, the number of states, the current state, the probability distribution) is completely known to the agent. [1] In Chapter 15, we will present the case where the environment is not fully available to the agent, and the agent learns about the environment while also learning to act in it. [2]

[1] Think of playing a game where you know the complete set of rules.

[2] Think of playing a Role-playing Game (RPG) where you need to unlock parts of the map by advancing the story.

13.1 Basic Elements of Reinforcement Learning

Now we formalize several of the basic elements of reinforcement learning that were sketched above.

13.1.1 States and Actions

There is a finite set S of states, and the entire system AGENT + ENVIRONMENT exists in one of these states at any time. At each state $s \in S$, the agent makes an action $a \in A_s$, where A_s is the set of allowed actions at state s. We denote $A = \bigcup_{s \in S} A_s$ to be the set of all possible actions in the whole RL environment.

Example 13.1.1. *Consider a game of chess. Each state s can be represented as a pair (C, p) where C denotes the current configuration of pieces and p denotes the player to play next. For example, "white king at $e1$, black king at $e8$, and it is white turn to move" would be a possible state s of the game. An action a is a valid movement of a piece, given a state of the game. For example, "white king to $e2$" (i.e., $Ke2$) would be a possible action of the agent playing white in state s.*

Example 13.1.2. *Self-driving cars, like those built by Tesla, are becoming increasingly popular. Let's imagine how we could construct a state diagram for the task of driving autonomously. Each state can be represented by the current configuration of a number of factors (e.g., the car speed, distance from lane boundaries, distance to nearest vehicle, etc.) Possible actions include increasing/decreasing speed, changing gear, changing direction, changing lane, etc.*

13.1.2 Modeling Uncertainty via Transition Probabilities

As mentioned, the agent has many sources of uncertainty in its knowledge about the environment, and we can use concepts from probability to model uncertainty.

Suppose $S = \{s_1, s_2, \ldots, s_n\}$ contains n states. When the agent takes action a while in state s, it will *transition* into another (potentially the same) state s'. The catch is: the agent does not know exactly which state it will end up in. Instead, there is a probability p_i of ending up in state s_i for each $s_i \in S$. Here $\sum_i p_i = 1$, meaning each (state, action) pair is associated with a *probability distribution* over the next state that the agent will enter. Formally, we define it as follows:

Definition 13.1.3 (Transition Probabilities)**.** *Given a state $s \in S$ and an action $a \in A_s$, there is an associated **transition probability** $p(* \mid s, a)$ distributed over S such that state $s' \in S$ happens with probability $p(s' \mid s, a)$*

when action a is taken at state s and $\sum_{s' \in S} p(s' \mid s, a) = 1$. *If* $p(s' \mid s, a) > 0$, *we say that the state s' is reachable from s when action a is taken.*

In general, not all states are reachable, given a state s and an action a. That is, some transition probability $p(s' \mid s, a)$ is zero. For these states, it is conventional to leave out the corresponding transitions when representing the RL environment as a state diagram as in Figure 13.1 or Figure 13.2.

Example 13.1.4. *Consider the state diagram shown in Figure 13.1. This is a special case where there is only one action a in the set A. In other words, the agent is not making any choices; instead, it is just following probabilistic transition over time steps. To calculate the probability of reaching state s_3 from s_0, we note there are two different paths. The first path is $s_0 - s_1 - s_3$ and the second path is $s_0 - s_2 - s_3$. We thus calculate the probabilities of each of these paths and note that the overall probability of reaching s_3 will be the sum of both:* $0.2 \cdot 0.7 + 0.8 \cdot 0.4 = 0.46$.

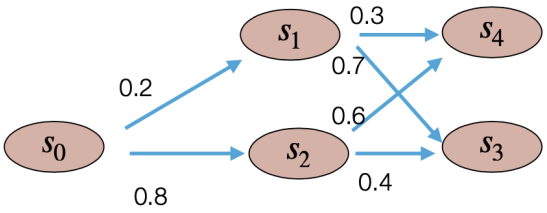


Figure 13.1: An example diagram where $|A| = 1$. The agent simply follows probabilistic transitions.

Now we consider an example where there is more than one action to make. In this case, each action induces a different probability distribution on the set of states, so we need to draw a diagram for each option.

Example 13.1.5. *We can model a baby learning to walk through RL. As shown in Figure 13.2, we can define the state $s_0 =$* ***standing but feeling unsteady****, $s_1 =$* ***standing and feeling secure****, and $s_2 =$* ***on ground****. The baby has two actions to take: $a =$* ***not grab onto nearest support*** *and $a' =$* ***grab onto nearest support****. The state diagram on the top represents the transition probabilities when the baby takes the action a. See that the baby has a high chance of entering state s_2 — falling to the ground. On the other hand, the state diagram on the bottom represents the transition probabilities when the baby takes the action a'. The baby now has a high chance of entering state s_1 — standing securely on the ground. The two actions have* ***different*** *probability distributions associated with the relevant transitions.*

Example 13.1.6. *Mechanical ventilators are used to stabilize breathing for patients. Suppose we wished to construct a state diagram. We could define*

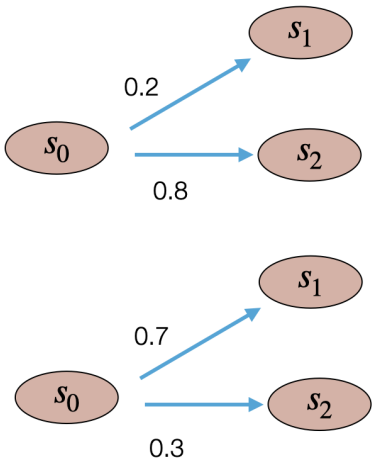


Figure 13.2: An example diagram showing how an action determines the probability of outgoing transitions.

states that consider the pressure and CO_2 level in the patient's level for the past k seconds. Actions might include adjusting the flow rate of oxygen via valve settings as needed. Possible transitions might include the typical mechanical response of the lungs, or unexpected spasms. Finally, we can define the goal to be maintaining steady pressure in the patient without "overshooting" and causing damage.

13.1.3 Agent's Motivation/Goals

In general, an agent is a participant in RL models driven by the need to maximize "rewards." In a probabilistic setting, the agent wishes to maximize their *expected* rewards. In a natural setting, the "rewards" could be innate satisfaction, such as getting to eat food, being entertained, etc. But in the usual artificial settings such as robots and self-driving cars, rewards are sprinkled by the system designer into the framework. Some examples appear later.[3]

At each step, the agent takes an action, and is given a reward (which could be negative, *i.e.*, is a punishment) based on the action, current state, and next state.

[3] While reward/punishment as a way to shape human or animal behavior is a very old idea, mathematical modeling of agents as reward-maximisers appears in several disciplines that flowered around the middle of the 20th century (*e.g., behaviorism* in psychology, profit-maximisation in economics, and of course RL).

Definition 13.1.7 (Reward)**.** *For each valid 3-tuple (a, s, s') where $s' \in S$ is a state reachable from state $s \in S$ by taking action $a \in A_s$, we define a corresponding **reward** $r(a \mid s_1, s_2) \in \mathbb{R}$.*

Example 13.1.8 (Example 13.1.5 revisited)**.** *When the baby stands and feels secure after grabbing onto something, the parents applaud the baby, and the baby receives a positive reward: $r(a' \mid s_0, s_1) = 5$. When the baby feels secure without grabbing onto the nearest support, the parents feel even prouder and the baby gets a more positive reward: $r(a \mid s_0, s_1) = 10$. When the baby falls to the ground, the baby feels pain and receives negative reward:*

$r(a \mid s_0, s_2) = r(a' \mid s_0, s_2) = -5$.

Typically, the designer of a RL model gets to define the rewards throughout the framework based on the designer's judgment. For instance, in Example 13.1.2, we might design an RL model such that if the car drifts into an adjacent lane, we assign a negative reward. If another vehicle is in the lane, we might assign an even larger negative reward. This will induce a RL model to "learn" the proper way to driving — staying in lane.

13.1.4 Comparison with NFA

Recall the Non-deterministic Finite Automata (NFA) you learned in COS 126. In an NFA, there is a finite number of states, and for each state, we know the set of next possible states, based on the next input character.

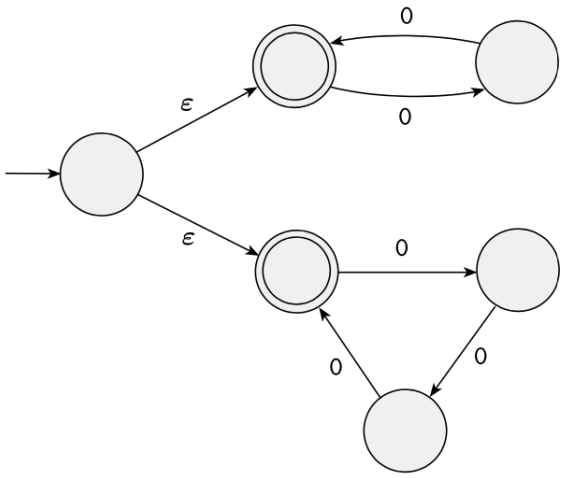


Figure 13.3: A sample Non-deterministic Finite Automata. Source: *Introduction to the Theory of Computation* by Michael Sipser.

We can consider the following analogy between RL and NFA — there is someone behind an NFA, who can observe its current state and type in the next input character. This person will be called an *agent*, and the choice of input character that is typed in will be called an *action*. Each action can lead to a finite set of next possible states, but because of some uncertainty in the world, the agent cannot specify which particular state will be the next one. This is similar to an NFA in the sense that the actions are non-deterministic. Also, just like in an NFA, the change in the current state is also referred to as a *transition*. One major difference between RL and NFA is that while an NFA only cares about the final state of the automata (*i.e.*, whether it is an accept state or a reject state), in RL, the agent is given a *reward* after each transition. The goal of the agent will be to take a sequence

of actions so as to maximize the sum of the reward throughout the sequence of actions.

13.2 *Useful Resource: MuJoCo-based RL Environments*

Real-life robots with precise and reliable hardware can get very expensive to buy, let alone train. An easier playground for students (especially those trying to work with a single GPU on CoLab) is doing RL in a virtual environment.

MuJoCo is a famous physics engine that allows creating virtual objects with somewhat realistic "joints" that can be commanded to move similar to real-life robots. OpenAI and DeepMind have open-source environments that allow experimentation in the MuJoCo environment. The official website gives a pretty good overview of the software: [4]

[4] Source: https://mujoco.org.

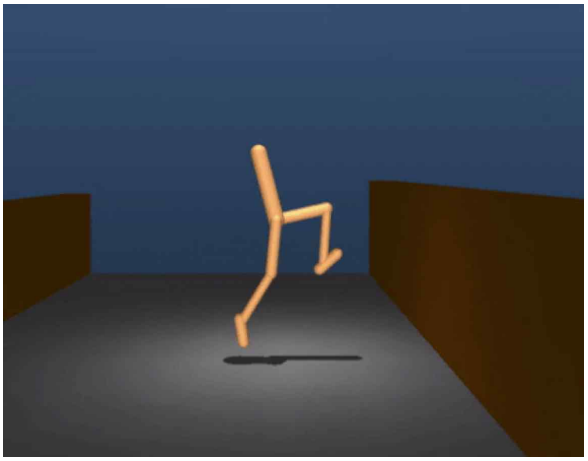


Figure 13.4: An example of a MuJoCo Walker.

> *MuJoCo is a physics engine that aims to facilitate research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed. MuJoCo offers a unique combination of speed, accuracy and modeling power, yet it is not merely a better simulator. Instead it is the first full-featured simulator designed from the ground up for the purpose of model-based optimization, and in particular optimization through contacts.*

One aspect of MuJoCo simulation involves a representation of a humanoid figure (*i.e.*, the agent) learning how to navigate an obstacle course (*i.e.*, the environment). Training videos are readily available online and show how the agent learns over time (sometimes, to comedic effect).

Example 13.2.1. *Let's analyze the example of an agent navigating an obstacle course through a RL framework. The states can be considered to be the set of coordinates, velocity, and acceleration for each limb, the velocity*

and acceleration for the motors in each joint, and the environment itself straight ahead. The actions can include the agent increasing or decreasing motor speed in their joints. Finally, the final goal is to stay upright, run forward at a reasonable pace, and avoid obstacles.

13.3 Illustrative Example: Optimum Cake Eating

Let's consider an extended example which ties together the elements of RL discussed previously. Suppose you buy a small cake with three slices. The reward of eating one slice at one sitting is 1, but eating two or three slices at one sitting is 1.5 and 1.8 respectively. [5]

[5] This sense of diminishing rewards is known as the satiation effect.

Problem 13.3.1. *Suppose you plan to eat the cake over a period of three days. What eating schedule will maximize the internal reward?*

Now let's introduce your roommate, who is oblivious to basic understandings of ownership and adheres to the "finders keepers" faith. We define the probability $\Pr[\text{sneakily eats a slice overnight}] = \frac{1}{2}$. To account for this uncertainty, we can create a *look-ahead tree* for different initial actions. We first consider the action where you decide to eat two out of the three slices on the first night. Successive states and associated probabilities are shown in the Figure 13.5.

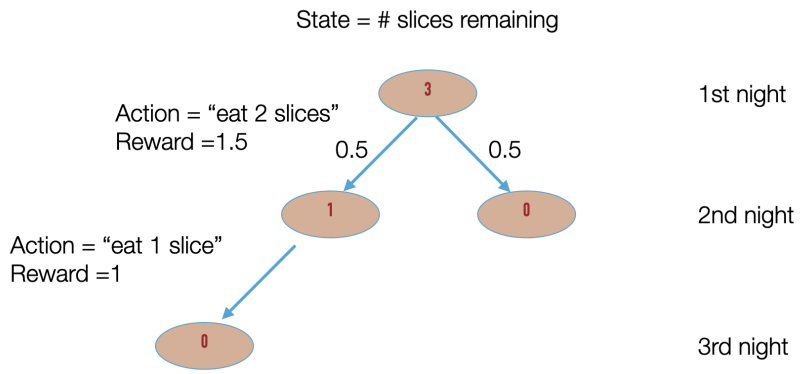


Figure 13.5: The diagram (look-ahead tree) of the cake problem where you decide to eat two slices on the first night. Each state represents the number of slices remaining, and each action represents the number of slices eaten on one night.

Even though you only eat only two out of the three slices during the first night, there is a $\frac{1}{2}$ chance that your roommate eats the remaining slice overnight. Therefore, the action of "eating 2 slices" can lead to two possible states — "1 slice left" or "0 slice left" — each with probability $\frac{1}{2}$.

Example 13.3.2. *We can calculate the expected reward associated with eating two slices on the first night by analyzing the Figure 13.5. You first gain reward of* 1.5 *by eating the two slices on the first night. Then with probability* $\frac{1}{2}$ *(where the roommate does not eat the remaining slice overnight),*

you get to eat the last slice on the second night and gain additional reward of 1. *With probability of* $\frac{1}{2}$ *(where the roommate eats the remaining slice), you cannot gain anymore reward. That is, the expected reward is* $1.5 + 0.5 \cdot 1 + 0.5 \cdot 0 = 2$.

Problem 13.3.3. *Consider the result of Example 13.3.2. Would you prefer to take two slices on the first night or three slices?*

We next consider the action where you decide to eat one out of the three slices on the first night. Successive states and associated probabilities are shown in the Figure 13.6.

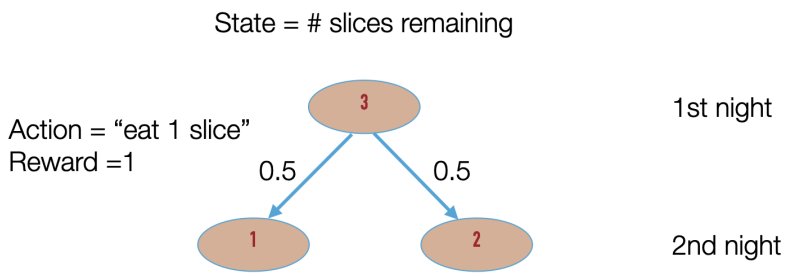


Figure 13.6: The diagram (look-ahead tree) of the cake problem where you decide to eat one slice on the first night.

The difficulty in this example in contrast to the Figure 13.5 is that if the roommate does not eat a slice after the first night, you have two slices at your disposal on the second night. You have two actions you can take in this "2 slices left" state — "eat 1 slice" (and hope the third slice is still there on the third night) or "eat 2 slices" — and it is not immediately obvious which one is more optimal. It turns out that the expected reward you can get from the remaining 2 slices is 1.5 for both options.

Problem 13.3.4. *Verify the previous claim that both options on the second night have the same expected reward.*

Example 13.3.5. *Given the previous analysis and the look-ahead tree in the Figure 13.6, we note that the total expected reward is* $1 + 0.5 \cdot 1 + 0.5 \cdot 1.5 = 2.25$. *You first receive a reward of* 1 *by eating* 1 *slice on the first night. Then with probability* $\frac{1}{2}$, *the roommate eats one slice over night, and you gain reward of* 1 *by eating the last slice on the second night. With the remaining probability* $\frac{1}{2}$, *the roommate does not eat a slice, and you are expected to gain reward of* 1.5 *from the remaining* 2 *slices, regardless of the action you choose to take on the second night.*

Problem 13.3.6. *Consider the result of Example 13.3.5. Would you prefer to take two slices on the first night or one slice?*

14
Markov Decision Process

In this chapter, we formally introduce the *Markov Decision Process (MDP)*, a way to formulate an RL environment. We then present ways to find the optimal strategy of an agent, provided that the agent knows the full details of the MDP — that is, knows everything about the environment.

14.1 Markov Decision Process (MDP)

Let's review the key ingredients of RL. We have the *agent*, who senses the environment and captures it as the current *state*. There is a finite number of actions available at any given state, and taking an action a in state s will cause a transition to s' with probability $p(s' \mid s, a)$. Each transition is accompanied by a reward $r(a \mid s, s_i) \in \mathbb{R}$. Finally, the goal of the agent is to maximize the expected reward via a sequence of actions.

A *Markov Decision Process (MDP)* is a formalization of these concepts. It is a *directed graph* which consists of four key features:

- A set S which contains all possible states
- A set A which contains all possible actions
- For each valid tuple of action a and states s_1, s_2, there is an assigned probability $p(s_2 \mid s_1, a)$ probability of transition to s_2 if action a is taken in s_1
- For each valid tuple of action a and states s_1, s_2, there is an assigned reward $r(a \mid s_1, s_2)$, which is obtained if action a is taken to transition from s_1 to s_2

If a designed MDP has M actions and N states, we can specify the MDP by a table of transition probabilities (with MN^2 numbers) and a table for rewards (with MN^2 numbers).

14.1.1 *Revisiting the Cake Eating Example*

Let's return to the case study on eating cake from Subsection 13.3, and formally express it through a MDP. The set of states is given as $S = \{0, 1, 2, 3\}$, where each state represents the number of slices left. The set of actions is given as $A = \{1, 2, 3\}$, where each action represents the number of slices you choose to eat on a given night. Notice that reward only depends on how many slices you take, not how many slices are left after your roommate goes through the fridge. That is, we can define the reward $r(a \mid s, *)$ for each $a \in A$ to be the same for every $s \in S$ where a is feasible. [1]

[1] We still need to include the previous state s because not all actions are feasible at each state. For example, you can't eat 2 slices when there is only 1 slice left.

Example 14.1.1. *Let's revisit Example 13.3.2 as a motivating example. If we let $a = 2$, $s_1 = 3$, and $s_2 = 0$, then the probability of the specified transition is $p(s_2 \mid s_1, a) = 0.5$. The associated reward is $r(a \mid s_1, s_2) = 1.5$ as discussed earlier.*

We are now ready to generalize to the full MDP, which is shown in Figure 14.1. Note that every transition is labeled with its probability, associated action, and associated reward.

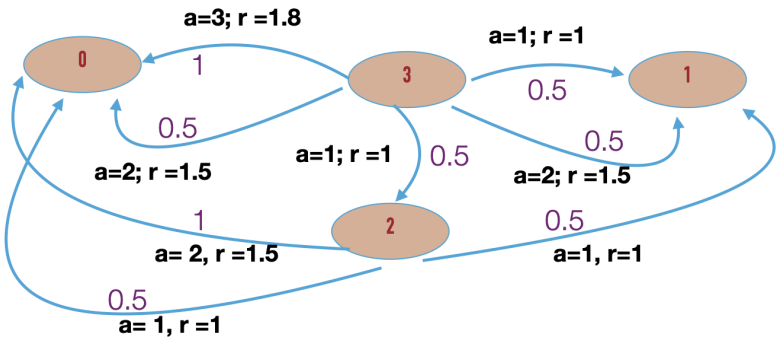


Figure 14.1: The full diagram of the cake problem when described as a MDP.

14.1.2 *Discounting the Future*

The MDP describing cake eating in the previous subsection was acyclic. [2] However, in general, MDPs can have directed cycles, and the agent's actions can allow it to continuously collect rewards along that cycle. For instance, continuing our cake theme, we may have a scenario in which you receive a fresh cake every 3 days. But now we run into a problem: how can we calculate the expected reward when there is an unbounded number of steps?

[2] This is also known as an *Episodic MDP*.

The solution lies in the concept of *future discounting*. The basic idea is to reduce, or *discount*, the amount of reward we get from

future steps. In an MDP, we represent this through a discount factor $0 < \gamma \leq 1$ and an associated infinite sum. [3]

[3] This is related to notions of discounting commonly considered in economics.

Definition 14.1.2 (Future Discounting). *If a reward r_t is received at time $t = 0, 1, 2, \ldots$, then the perceived value of these rewards r_d, or the **discounted reward**, at $t = 0$ is:*

$$r_d = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

Example 14.1.3. *Consider the cake eating problem again and let r_t denote the reward we get on night t. If the reward is discounted by a factor of γ every night, the total expected discounted reward $E[total]$ can be rewritten as*

$$\mathbb{E}[total] = \mathbb{E}[r_1] + \gamma \cdot \mathbb{E}[r_2] + \gamma^2 \cdot \mathbb{E}[r_3]$$

Consider taking the action $a = 2$ on the first night. If $\gamma = 0.9$, then the expected discounted reward is

$$1.5 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 0) = 1.95$$

This is the same as in Example 13.3.2 except the reward taken from the second night is discounted by a factor of 0.9. Now consider taking the action $a = 1$ on the first night and on the second night. If $\gamma = 0.9$, the expected discounted reward is

$$1 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 1) + 0.9^2 \cdot (0.5^2 \cdot 1) = 2.1025$$

Here, we first take the reward of 1 on the first night without any discount factor. Then, we calculate the expected reward from the second night — 1 whether or not the roommate eats a slice — and discount it by a factor of 0.9. Finally, we calculate the expected reward from the third night — 1 only if the roommate did not eat any slice on the first two nights — and discount it by a factor of 0.9^2.

Note that in Definition 14.1.2, if each $r_t \in [-R, R]$ and if $\gamma < 1$, then the discounted reward of the infinite sequence has the following upper bound:

$$|r_d| \leq R(1 + \gamma + \gamma^2 + \cdots) = \frac{R}{1 - \gamma} \tag{14.1}$$

(14.1) is derived by considering the formula for the sum of an infinite geometric series, which we can invoke if $\gamma < 1$. In general, γ is up to the system designer. A lower γ would imply that the agent places little importance on future rewards, whereas $\gamma = 1$ would imply that there is effectively no discounting.

14.2 *Policy and Markov Reward Process*

Now that we have discussed what an action is and what it does in an MDP, we want to specify what action an agent has to take in each state. This is known as a *policy*.

Example 14.2.1. *Consider again the cake eating MDP example without a discount factor. We already established through Example 13.3.2 and Example 13.3.5 that to maximize the expected reward, you need to eat one slice per day until all slices are gone. That is, in any state j where $j = 1, 2, 3$, you need to take action 1.*

In general, if S is the set of states, and A is the set of actions, then a *policy* (not necessarily the optimum) π can be defined as a function $\pi : S \to A$

Definition 14.2.2 (Policy)**.** *If S is the set of states, and A is the set of actions, any function $\pi : S \to A$ is called a **policy** that describes which action to take at each state. In particular, each state s should only be mapped to a valid action $a \in A_s$ at that state.*

Recall that if there are M actions and N states, there are at most MN^2 transitions in the graph of the MDP. Because a policy specifies one action per state, there are at most N^2 transitions that remain when we choose a specific policy. Therefore, it can be understood that a policy trims out the MDP.

14.2.1 Markov Reward Process (MRP)

When we have an MDP and a fixed policy, we have what is called a *Markov Reward Process (MRP)*. There are no more decisions to make; instead, all we need to do is take the action specified by the policy; probabilistically follow a transition into a new state; and collect the associated reward.

Example 14.2.3. *Let's revisit Figure 14.1. If we fix the policy to be $\pi(s) = 1$ for any $s \in S$, we can focus our attention to the action $a = 1$. Then there are three trajectories that will lead from state 3 to state 0, based on what the roommate does overnight. The first trajectory is $3 \to 1 \to 0$ with probability 0.5×1 and reward $1 + 1$. The second trajectory is $3 \to 2 \to 0$ with probability 0.5×0.5 and reward $1 + 1$. The last trajectory is $3 \to 2 \to 1 \to 0$ with probability $0.5 \times 0.5 \times 1$ and reward $1 + 1 + 1$.*

In general, when we fix a policy π and an initial state s, we can redraw the transition diagram of an MDP into a tree diagram for the MRP, where each node corresponds to a state, and each edge corresponds to a probabilistic transition. The top node represents the initial state, and each subsequent row of the tree represents the set of possible states after taking an action from their parent node.

Example 14.2.4. *We revisit Example 14.2.3. We now transform Figure 14.1 into a tree diagram for the MRP as shown in Figure 14.2. The top node is the initial state 3. The second row of the tree is all states that can be achieved by taking the action 1 at state 3, and so on.*

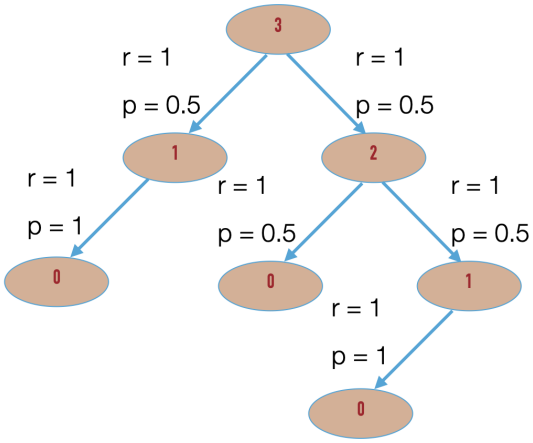


Figure 14.2: A tree representing the MRP in Example 14.2.3.

Note that in an MRP tree, the same state can appear multiple times, but each copy of the same state is identical — that is, the subtree rooted at each copy must be identical. In Figure 14.2, the state 1 appears three time in the tree. Every time it appears, it can only lead to state 0 with probability 1. This is simply the result of fixing a policy π — once we know the state we are in, we only have one choice for the action to take.

The policy also induces a *value function* on this tree. The value function assigns a value to each node of the tree, and each value intuitively measures how much reward the agent should expect to collect once the agent knows they have arrived at that node. By the observation from the previous paragraph, this expected reward should be the same for two nodes, if they are the copy of the same state. Therefore, we can equivalently define the value function for each state s instead. Formally, we define the value function as the following.

Definition 14.2.5 (Value Function)**.** *$v_\pi(s)$, the **value of state** s **under the policy** π, is the expected discounted reward of a random trajectory starting from s. We can define this value by using the following recursive formula:*

$$v_\pi(s) = \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_\pi(s')\big) \tag{14.2}$$

*Computing the value function as in (14.2) is also known as the **Bellman equation**.*

Let us unpack the intuition behind (14.2). Once we take action $\pi(s)$ at state s, it will bring us to state s' with probability $p(s' \mid s, a)$, immediately giving us a reward $r(a \mid s, s')$. Then, the expected reward from that point on is already captured by the value $v_\pi(s')$. We just need to apply the discount factor γ because we already took one time step to reach s' from s.

On the other hand, if we pick any random trajectory starting from s, its next node will be some state s' that is reachable from s. Therefore, the contribution of this particular trajectory to $v_\pi(s)$ is accounted for when we sum over that particular s'.

14.2.2 *Connection with Dynamic Programming*

In COS 226, you may have seen an implementation of a bottom-up dynamic programming.

```
int[] opt = new int[n+1];
for (int v = 1; v <= V; v++)
{
   opt[v] = Integer.MAX_VALUE;
   for (int i = 1; i <= n; i++)
   {
      if (d[i] > v) continue;
      if (opt[v] > 1 + opt[v - d[i]])
          opt[v] = 1 + opt[v - d[i]];
   }
}
```

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$

Figure 14.3: A Dynamic Programming implementation of a coin changing problem that uses the bottom-up approach.

In such implementations, the algorithm divides the problem into subproblems arranged as directed acyclic graphs and compute "bottom-up." The MDP from the cake eating problem is acylic and our method using a look-ahead tree is similar to the dynamic programming algorithms. Therefore, it seems like we can apply a similar algorithm to the cake eating problem.

Example 14.2.6. *Consider Example 14.2.3 again, but now with a discount factor of* 0.9*. We will find the value $v_\pi(s)$ of each state s by going bottom-up from the tree in Figure 14.2. We start by noticing that $v_\pi(0) = 0$ as can be seen from the bottom row. Then from the third node of the third row, we can calculate*

$$v_\pi(1) = 1 \cdot (1 + 0.9 \cdot 0) = 1$$

From the second node of the second row, we can calculate

$$v_\pi(2) = 0.5 \cdot (1 + 0.9 \cdot 0) + 0.5 \cdot (1 + 0.9 \cdot 1) = 1.45$$

Finally, from the top node, we can calculate

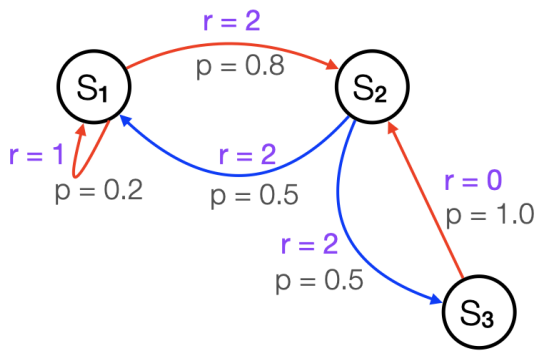
$$v_\pi(3) = 0.5 \cdot (1 + 0.9 \cdot 1) + 0.5 \cdot (1 + 0.9 \cdot 1.45) = 2.1025$$

But in general, the dynamic programming approach does not completely apply to MDP. The biggest assumption for dynamic programming algorithms is that the graph is *acyclic*, but MDPs are generally allowed to have directed cycles if we can return to the same state after a sequence of actions. Therefore, computing the expected reward for even a single policy π involves solving a system of linear equations.

Example 14.2.7. *Assume that we have three states s_1, s_2, s_3 and transitions as in Figure 14.2.2 with a discount factor of $\gamma = 0.7$. Then the value at each state is given as*

$$v_\pi(s_1) = 0.2 \times (1 + 0.7v_\pi(s_1)) + 0.8 \times (2 + 0.7v_\pi(s_2))$$
$$v_\pi(s_2) = 0.5 \times (2 + 0.7v_\pi(s_1)) + 0.5 \times (2 + 0.7v_\pi(s_3))$$
$$v_\pi(s_3) = 1 \times (0 + 0.7v_\pi(s_2))$$

Unlike in Example 14.2.6, we cannot compute any of these values one by one because the values are interdependent in a cyclic manner. Instead, we need to solve the linear equation as a whole, which gives us the solution: $v_\pi(s_1) \approx 5.47, v_\pi(s_2) \approx 5.18, v_\pi(s_3) \approx 3.63$.



14.3 *Optimal Policy*

Out of all choices for a policy, we are interested in the *optimal policy*, the one that maximizes the expected (discounted) reward. Surprisingly, it is known that there always exists a policy π^* that obtains the maximum expected reward from all initial states *simultaneously*; that is $\pi^* = \arg\max_\pi v_\pi(s)$ for every state s. [4]. The value function of the optimal policy is called the *optimal value function* and is often denoted as $v^*(s)$. Then we can express the optimal value function using (14.2) as:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_\pi(s'))$$

This is just restating the fact that the optimal value of state s is the maximum of all possible values $v_\pi(s)$ of s under a policy π — *i. e.*, the Bellman equation evaluated with the values $v_\pi(s')$ of each children node s' under that specific policy π.

But we can even go further than this result. It is known that the optimal value also satisfies the following:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v^*(s')) \tag{14.3}$$

[4] If there are multiple such policies, we denote any one of them by π^*.

Notice that $v_\pi(s')$ in the summation has now been replaced with $v^*(s')$. This property, known as the *Bellman Optimality condition*, states that the optimal value is even the maximum when the Bellman equation is evaluated with the values $v^*(s')$, regardless of the choice of the policy π.

Notice that the right-hand side of (14.3) only depends on the choice of the action a of the given state s, not any other states. Therefore, we can rewrite (14.3) as:

$$v^*(s) = \max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.4}$$

which also suggests that the optimal action at state s can be expressed as:

$$\pi^*(s) = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.5}$$

But the problem is: it is unclear how to turn this into an efficient algorithm. Computing the value $v^*(s)$ depends on the value $v^*(s')$, which can also depend on $v^*(s)$, which becomes recursive.

In this section, we present an iterative algorithm called the *value iteration* method which will be used to compute the optimal policy. Before we describe the algorithm, we unpack the underlying ideas.

14.3.1 Developing Intuition about Optimality: Gridworld

To develop intuition about how to find an optimum policy, let's consider a classic example called Gridworld. [5]

[5] Source: Sutton and Barton 2020, https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

Example 14.3.1 (Gridworld)**.** *Consider a* 5×5 *grid. The set of states is given as the cells of this grid. At each state except for at* $A = (1,2)$ *and* $B = (1,4)$*, there are four available actions: move left/right/up/down, each with reward* 0*, except in the following setting: if the action will make you move off the grid. then the reward is* -1*, and you are made to stay at the same state instead.*

At A*, there is only one action: move to* $A' = (5,2)$ *with reward* 10 *and similarly at* B*, there is one action: move to* $B' = (3,4)$ *with reward* 5*.* [6] *The discount factor is given as* 0.9*.*

[6] The outgoing transition from A and B can be thought of as "wormholes."

How can we compute the reward for a policy in the example above? When beginners try to calculate the exact value using the above definitions, they quickly get bogged down in keeping track of too many variables, equations, and recurrences.

Instead, let's try to think intuitively about what an optimal policy **should** be trying to do. Since the wormholes are the only source of rewards, an optimal policy should be trying to utilize the wormholes as much as possible. Using this kind of intuition, we can design a

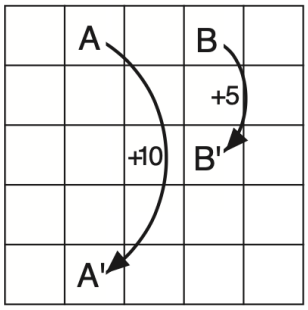


Figure 14.4: Visual representation of the Gridworld.

policy that looks at least near-optimal, and use its value as a **lower bound** for the optimal policy.

First, let $v^*(s)$ denote the value $v_{\pi^*}(s)$ of state s for an optimal policy π^*. Since there is only one action to choose from at state A, we know that

$$v^*(A) = 10 + \gamma v^*(A') \tag{14.6}$$

Now, at the state A', one possible trajectory you can follow is "go up four steps" (each with reward 0) back to A. We know that the optimal value has to be at least as great as this value. That is

$$v^*(A') \geq \gamma^4 v^*(A) \tag{14.7}$$

Combining (14.6) and (14.7), we get

$$v^*(A) \geq 10 + \gamma^5 v^*(A)$$

If we solve for $v^*(A)$, we get

$$v^*(A) \geq \frac{10}{1-\gamma^5} \approx 24.4$$

The value iteration method discussed below is based on this intuition — we can provide a lower bound for the optimal policy by suggesting some potential policy. If we repeat this process, the lower bound for the optimal policy can only go up. At the end of the section, we will prove that this process converges to the actual optimal value.

14.3.2 Value Iteration Method

Value Iteration is a method guaranteed to find the optimal policy. At each step of the iteration, we are given a lower bound on the optimal values of each state s. Using the values of the immediate children nodes in the tree, we can compute an improved lower bound on $v^*(s)$.

Example 14.3.2. *See Figure 14.5. Suppose there are two actions to take at state s. The first action, labeled as blue, will lead to state s_1 with reward -1 with probability 0.5 and s_3 with reward -1 with probability 0.5. The second action, labeled as red, will lead to state s_2 with reward 2 with probability 1. The discount factor is given as 0.6. Now assume that someone tells us that they know a way to get expected reward of 12 starting from s_1, 1 from s_2, and 4 from s_3, regardless of the choice of initial action at s. In other words, the optimal values for these three states are lower bounded by: $v^*(s_1) \geq 12, v^*(s_2) \geq 1$ and $v^*(s_3) \geq 4$. Using this fact, we consider two strategies [7] — (1) first take action blue at state s and play optimally thereon based on the other person's knowledge; (2) first take action red at state s and play optimally thereon. The lower bound for the expected reward for each of the two strategies can be computed as:*

$$v_{blue}(s) \geq 0.5 \times (-1 + 0.6 \times 12) + 0.5 \times (-1 + 0.6 \times 4) = 3.8$$
$$v_{red}(s) \geq 1.0 \times (2 + 0.6 \times 1) = 2.6$$

The Bellman Optimality condition in (14.4) guarantees that the optimal policy is at least as good as either of these strategies. Therefore $v^*(s)$ has to be larger than both v_{blue}, v_{red}; that is, $v^*(s) \geq 3.8$.

[7] This is not necessarily a *policy* because the second part of playing optimally may require you to return to state s and take an action that is inconsistent with your initial choice of action.

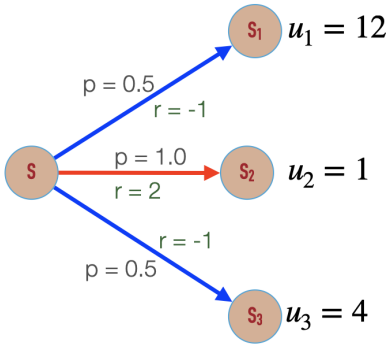


Figure 14.5: There are two actions you can take at state s, and you will end up in one of the three states: s_1, s_2, s_3.

In general, the value iteration algorithm looks like:

1. Initialize some values $v_0(s)$ for each state s such that we are guaranteed $v_0(s) \leq v^*(s)$
2. For each time step $k = 1, 2, \ldots$, and for each state s, use the values $v_k(s')$ of the immediate children s' to compute an updated value $v_{k+1}(s)$ such that $v_{k+1}(s) \leq v^*(s)$. [8]
3. When $k \to \infty$, each $v_k(s)$ will converge to the optimal value $v^*(s)$.

[8] These values $v_k(s)$ maintained by the algorithm is *not* necessarily associated with a specific policy. They are just a lower bound for the optimal value $v^*(s)$ that will be improved over time.

Recall from (14.1) that if all transition rewards are within $[-R, R]$, then the expected rewards at any state for any policy lies in $\left[-\frac{R}{1-\gamma}, \frac{R}{1-\gamma}\right]$.

Therefore, we can set the initial value $v_0(s) = -\frac{R}{1-\gamma}$ to be the lower bound for each state s. [9]

After the k-th iteration of the algorithm, we will maintain a value $v_k(s)$ for state s, where the condition $v_k(s) \leq v^*(s)$ is maintained as an invariant. Now at the $(k+1)$-th iteration, the algorithm will update the values at each state s as the following:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \tag{14.8}$$

This is just the Bellman equation evaluated with the values $v_k(s')$ of each children node.

Example 14.3.3 (Example 14.3.1 revisited)**.** *Say we start the value iteration on the gridworld with all values equal to zero. Now let us compute $v_1(A)$, the value of A after the first iteration. Recall that A has only one action to choose from: moving to A'. Denote this action by a. Therefore,*

$$\begin{aligned} v_1(A) &= p(A' \mid A, a) \cdot \big(r(a \mid A, A') + \gamma v_0(A')\big) \\ &= 1.0 \cdot (10 + 0.9 \cdot 0) = 10 \end{aligned}$$

Problem 14.3.4 (Example 14.3.1 revisited)**.** *Start value iteration with all values equal to zero. What is $v_2((1,3))$, the value of $(1,3)$ after second iteration?*

14.3.3 *Why Does Value Iteration Find an Optimum Policy?*

We prove that the values $v_k(s)$ maintained by the value iteration method converge to the optimal values $v_\pi(s)$ in a finite number of steps. We break this proof down in two parts. We first prove that the invariant $v_k(s) \leq v^*(s)$ holds throughout the algorithm. Then we prove that in general, $v_{k+1}(s)$ is a tighter lower bound for $v^*(s)$ than $v_k(s)$.

Proposition 14.3.5. *For each time step $k = 1, 2, \ldots$, and for each state s, the invariant $v_k(s) \leq v^*(s)$ holds.*

Proof. Proof by mathematical induction. As discussed earlier, our choice of initial values $v_0(s) = -\frac{R}{1-\gamma}$ satisfies the invariant. Now assume that the invariant holds for some k. Now consider the update rule of the value iteration algorithm:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big)$$

Notice that for any specific policy π and for any next state s', we have

$$\begin{aligned} &p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\ \leq &p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \end{aligned}$$

[9] Our proof assumes this special initialization where all $v_0(s) = -\frac{R}{1-\gamma}$ for all states s. It turns out the value iteration method converges to the optimal value for arbitrary initialization, but the proof is more complicated.

because of the inductive hypothesis that $v_k(s') \leq v^*(s')$. Therefore, if we sum over all state s', we have

$$\begin{aligned}
&\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\
\leq &\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big)
\end{aligned}$$

Since this inequality holds for every policy π, we have the following:

$$\begin{aligned}
v_{k+1}(s) &= \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\
&\leq \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \quad = v^*(s)
\end{aligned}$$

where we apply the Bellman Optimality condition (14.4) in the last equality. This concludes the inductive step, and it suffices for the proof. □

Now to prove that these values $v_k(s)$ eventually converge to $v^*(s)$, we introduce the following definition:

Definition 14.3.6. *The **residual** at s at the k-th iteration is defined as $\delta_{s,k} = v^*(s) - v_k(s) > 0$.*

Notice that as long as the residuals at the k-th iteration converge to 0, the values $v_k(s)$ also converge to $v^*(s)$. Since the residuals take finite values when the algorithm is initiated, it suffices to prove that the residuals decrease non-trivially in every iteration. [10]

[10] Our exposition of Value Iteration with our particular initialization is new. The usual textbook description requires a slightly more complicated argument.

Proposition 14.3.7. *If the largest residual at iteration k is denoted as $\delta_k = \max_s \delta_{s,k}$, then the largest residual δ_{k+1} at iteration $k+1$ satisfies $\delta_{k+1} \leq \gamma \delta_k$*

Proof. Let a^* be the action at s under the optimum policy π^*. Then by (14.2),

$$v^*(s) = \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v^*(s')) \tag{14.9}$$

Note that taking the action a^* is always an option at the $(k+1)$-th iteration, so the $v_{k+1}(s)$, the maximum value across all actions has to be greater than or equal to the value computed with the action a^*; that is,

$$\begin{aligned}
v_{k+1}(s) &= \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_k(s')) \\
&\geq \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v_k(s')) \qquad (14.10)
\end{aligned}$$

Subtracting (14.10) from (14.9), we get

$$v^*(s) - v_{k+1}(s) \leq \gamma \left(\sum_{s'} p(s' \mid s, a^*)(v^*(s') - v_k(s')) \right)$$

By the definition of δ_k, each of $v^*(s') - v_k(s') = \delta_{s',k} \leq \delta_k$. Therefore,

$$v^*(s) - v_{k+1}(s) \leq \gamma \delta_k \sum_{s'} p(s' \mid s, a^*)$$

Finally note that $\sum_{s'} p(s' \mid s, a^*) = 1$ because p is a probability distribution. □

Theorem 14.3.8. *For each $s \in S$, $v_k(s)$ converges to $v^*(s)$ when $k \to \infty$.*

Proof. By Proposition 14.3.5 and Proposition 14.3.7,

$$|v^*(s) - v_k(s)| = v^*(s) - v_k(s) \leq \delta_k \leq \gamma^k \delta_0$$

which converges to 0 when k goes to infinity. □

14.3.4 *Retrieving Optimal Policy from the v^*'s*

One important thing to note is that the value iteration method finds the optimal *value* of each state, not the optimal *policy*. So we need an extra step to retrieve the optimal policy from the output of the value iteration algorithm. This can be done by considering the Bellman Optimality condition. For each state s, define $\pi^*(s) = a^*$ such that

$$a^* = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \qquad \text{(14.5 revisited)}$$

where $v^*(s)$ is the value that the value iteration algorithm converges to. If there are multiple actions a that satisfy the equation above, arbitrarily choose an action.

Example 14.3.9 (Example 14.3.1 revisited)**.** *Say we ran the value iteration algorithm on the Gridworld. The output of the algorithm (the optimal values of each state) is given in Table 14.1.*

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Table 14.1: Optimal values $v^*(s)$ of the Gridworld.

Consider the state $A' = (5, 2)$. There are four actions to take: left-/right/up/down. Each action would yield the following values when evaluat-

ing the Bellman equation:

$$v_{left}(A') = 0 + 0.9 \times 14.4 = 13.0$$
$$v_{right}(A') = 0 + 0.9 \times 14.4 = 13.0$$
$$v_{up}(A') = 0 + 0.9 \times 17.8 = 16.0$$
$$v_{down}(A') = -1 + 0.9 \times 16.0 = 13.4$$

The only action that maximizes the value is the action "go up." Therefore, we can conclude that the optimal policy π^* will adopt the action "go up" for the state A'.

Problem 14.3.10 (Example 14.3.1 revisited)**.** *Verify that an optimal policy can assign either the action "go up" or the action "go left" for the state* $(5, 3)$.

15
Reinforcement Learning in Unknown Environment

In the previous Chapter 14, we established the principles of reinforcement learning using a Markov Decision Process (MDP) with set of states S, set of actions A, transition probabilities $p(s' \mid a, s)$, and the rewards $r(a \mid s, s')$. We saw a method (value iteration) to find the optimal policy that will maximize the expected reward for every state. The main assumption of the chapter was that the agent has access to the full description of the MDP — the set of states, the set of actions, the transition probabilities, rewards, etc.

But what can we do when some of the parameters of the MDP are not available to the agent in advance — specifically, the transition probabilities and the rewards. Instead, the agent makes actions and observes the new state and the reward it just received. Using such experiences it begins to learns the reward and transition structure, and then to translate this incremental knowledge into improved actions.

The above scenario describes most real-life agents: the system designer does not know a full description of the probabilities and transitions. For instance, think of the sets of possible states and transitions in the MuJoCo animals and walkers that we saw. Even with a small number of joints, the total set of scenarios is too vast. Thus the designer can set up an intuitive reward structure and let the learner figure out from experience (which is painless since it involves a simulation).

Settings where agent must determine (or "figure out") the MDP through *experience*, specifically by taking actions and observing the effects, is called the "model-free" setting of RL. This chapter will introduce basic concepts, including the famous Q-learning algorithm.

In many settings today, the underlying MDP is too large for the agent to reconstruct completely, and the agent uses deep neural networks to represent its knowledge of the environment and its own policy.

15.1 Model-Free Reinforcement Learning

In model-free RL, we know the set of states S and the set of actions A, but the transition probabilities and rewards are unknown. The agent now needs to explore the environment to estimate the transition probabilities and rewards. Suppose the agent is originally in state s_1, chooses to take an action a, and ends up in state s_2. The agent immediately observes some reward $r(a \mid s_1, s_2)$, but we need more information to figure out $p(s_2, |s_1, a)$.

One way we can estimate the transition probabilities is through Maximum Likelihood Principle. This concept has been used before when considering estimating unigram probabilities in Chapter 8. In model-free RL, an agent can keep track of the number of times they took action a at state s_1 and ended up in state s_2 — denote this as $\#(s_1, a, s_2)$. Then the estimate of the transition probability $p(s'|s, a)$ is:

$$p(s_2|s_1, a) = \frac{\#(s_1, a, s_2)}{\sum_{s'} \#(s_1, a, s')} \tag{15.1}$$

The Central Limit Theorem (see Chapter 18) guarantees that estimates will improve with more observations and quickly converge to underlying state-action transition probabilities and rewards.

15.1.1 Groundhog Day

Groundhog Day is an early movie about a "time loop" and the title has even become an everyday term. The film tracks cynical TV weatherman Phil Connors (Bill Murray) who is tasked with going to the small town of Punxsutawney and filming its annual Groundhog Day celebration. He ends up reliving the same day over and over again, and becomes temporarily trapped. Along the way, he attempts to court his producer Rita Hanson (Andie MacDowell), and is only released from the time loop after a concerted effort to improve his character.

Sounds philosophically deep! On the internet you can find various interpretations of the movie: *Buddhist* interpretation ("many reincarnations ending in Nirvana") and *psychoanalysis* ("revisiting of the same events over and over again to reach closure"). The RL interpretation is that Phil is in an model-free RL environment, [1] revisiting the same events of the day over and over again and figuring out his optimal actions.

[1] Specifically a model-free RL environment with an ability to *reset* to an initial state. This happens for example with a robot vacuum that periodically returns to its charging station. After charging, it starts exploring the MDP from the initial state again.

15.2 *Atari Pong (1972): A Case Study*

In 1972, the classic game of Pong was released by Atari. This was the first commercially successful video game, and had a major cultural impact on the perception of video games by the general public. The rules of the game are simple: each player controls a virtual paddle which can move vertically in order to rally a ball back and forth (one participant may be a computer AI). If a player misses the ball, the other player wins a point. We can consider the total number of points accumulated by a player to be their *reward* so far. While technology and video games have become far more advanced in the present, it is still useful to analyze Pong today. This is because it is a simple example of a *physics-based* system, similar to (but far less advanced than) the MuJoCo stick figure simulations discussed in Chapter 13. It thus provides a useful case study to demonstrate how an agent can learn basic principles of physics through random exploration and estimation of transition probabilities.

Let's apply some simplifications in the interest of brevity. We define the pong table to be 5×5 pixels in size, the ball to have a size of 1 pixel, and the paddles to be 2 pixels in height. We define the state at a time t as the locations of the two paddles at time t, and the locations of the ball at time t and time $t-1$. [2]

[2] Storing the location of the ball at time $t-1$ and time t allows us to calculate the difference between the two locations and thus gives an estimate for the velocity.

We additionally restrict attention to the problem of tracking and returning the ball, also known as "Pico Pong." Thus, we define the game to *begin* when the opponent hits the ball. The agent gets a reward of $+1$ if they manage to hit the ball, -1 if they miss, and 0 if the ball is still in play. As soon as the agent either hits the ball or misses, we define that the game *ends*. Of course, these additional rules of the game are not available to the agent playing the game. The agent needs to "learn" these rules by observing the possible states, transitions, and corresponding rewards.

In general, these simplifications remove complications of modeling the opponent and makes the MDP acyclic; an explanatory diagram is shown in Figure 15.1. Throughout this section, we will build intuition about different aspects of our Pico-Pong model through some examples.'

15.2.1 *Pico-Pong Modeling: States*

Suppose the agent is playing the random paddle movements. Consider the possible states of the game shown in Figure 15.2. We note that out of the three, the third option is never seen. By the definition of the game, the ball can never move away from the agent. Of course, the agent is oblivious to this fact at first, but once the game proceeds,

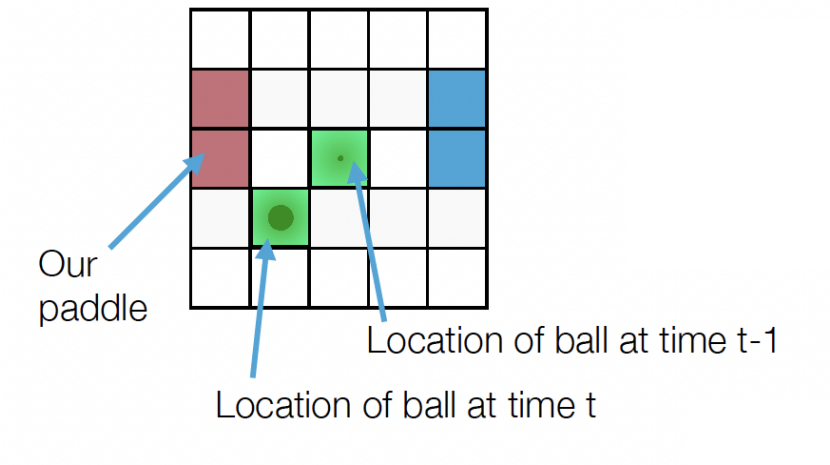


Figure 15.1: The simplified Pico-Pong setup which will be considered in this case study.

the agent will be able to implicitly "learn" that the ball can never move away from them.

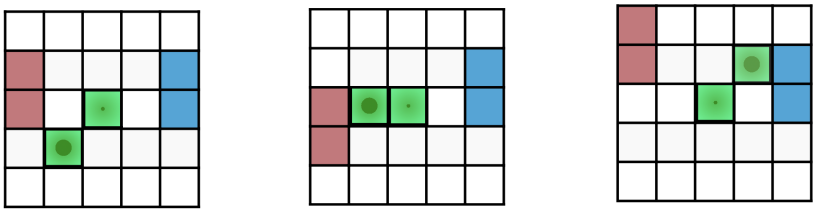


Figure 15.2: Out of these possible states, the third option is never seen.

15.2.2 *Pico-Pong Modeling: Transitions*

Let us now add another restriction to the game that the ball always moves at a speed of 1 pixel every time step (*i.e.*, moves to one of the 8 adjacent pixels) and in a straight linear path unless being bounced against the top/bottom wall. Consider the possible transitions shown in Figure 15.3. We note that out of the three, the third option is never seen. By the restriction of the game, the ball cannot move 2 pixels in one time step. The agent thus implicitly "learns" that the ball moves at a constant speed of 1 pixel per time step.

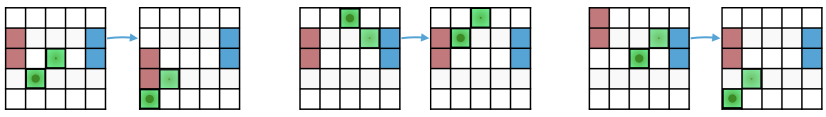


Figure 15.3: Out of these possible transitions, the third option is never seen.

Problem 15.2.1. *Suppose the agent is playing randomly and the ball is traveling at a speed of* 1 *pixel per step. Which of the transitions in Figure 15.4 is never seen, and why?*

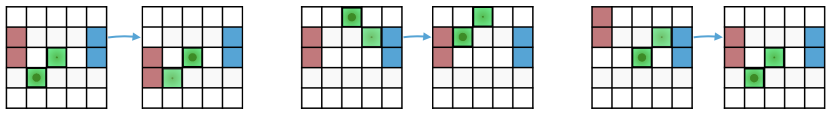


Figure 15.4: Out of these possible transitions, one option is never seen.

15.2.3 *Pico-Pong Modeling: Rewards*

Suppose the agent is playing randomly and the ball is traveling at a speed of 1 pixel per step. Consider the action in Figure 15.5. We note that the associated reward will be $+1$ because in the resulting state the agent has "hit" the ball. The agent thus implicitly learns that if the ball is 1 pixel away horizontally, it should move towards it to obtain a positive reward.

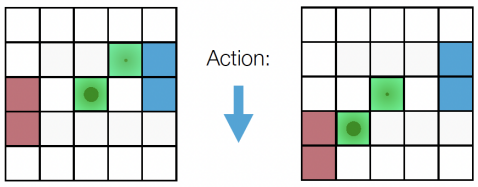


Figure 15.5: Taking action $\downarrow$ results in a reward of $+1$.

Problem 15.2.2. *Suppose the agent is playing randomly and the ball is traveling at a speed of* 1 *pixel per step. What reward is achieved given the current state and chosen action in Figure 15.6, and why?*

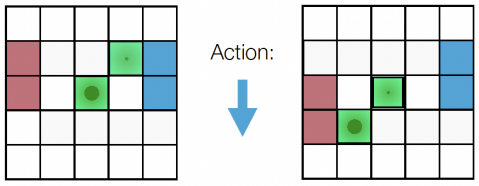


Figure 15.6: What reward will result when taking action $\downarrow$?

15.2.4 *Playing Optimally in the Learned MDP*

After allowing the agent to *explore* enough, the agent has "learnt" some information about the underlying MDP of the Pico-Pong model. First thing the agent can learn is that, out of all possible states, there is a subset of states that never appear in the game (*e.g.*, ball moving

away from the agent or ball moving too fast). The agent will be able to ignore these states, while learning how to play optimally in states that *did* occur while exploring.

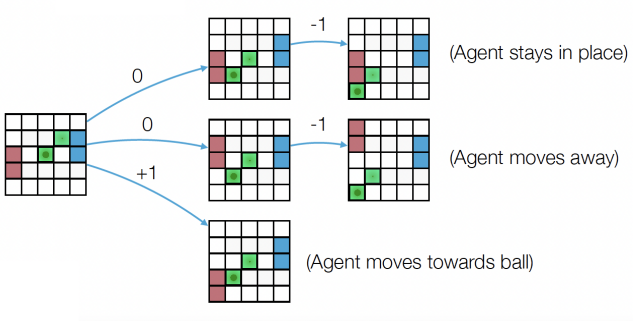


Figure 15.7: An example look-ahead tree for the Pico-Pong model.

Also, the agent has now "learnt" the transition probabilities and rewards of the MDP. Using these estimates, the agent is able to build up a representation of the MDP. Since the underlying MDP for the simplified Pico-Pong model is acyclic, the optimal policy can be determined using a simple look-ahead tree. An example diagram is shown in Figure 15.7.

We provide a specific example to aid the exposition. Suppose an agent finds themselves in the state shown in Figure 15.8. Since the path of the ball is already determined, the next possible state is uniquely determined by the choice of the action — "go down" or "stay in place" or "go up." If the agent chooses to "go down," the game will end with a reward of $+1$. If the agent chooses to "stay in place" or "go up," the game continues for another time step, but no matter the choice of action on that step, the game will end with a reward of -1. Therefore, the agent will learn that the optimal policy will assign the action of "go down" in the state shown in Figure 15.8.

Problem 15.2.3. *Draw out the look-ahead tree from the state shown in Figure 15.8.*

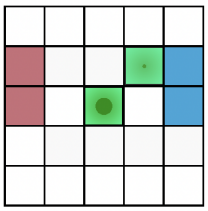


Figure 15.8: A sample state in the game play of Pico Pong.

Problem 15.2.4. *Suppose we start from the state shown in Figure 15.9. Assuming optimal play, what is the expected reward for the agent? (Hint: consider if the agent be able to reach the ball in time)*

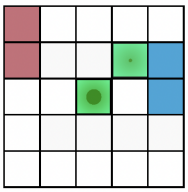


Figure 15.9: A sample state in the game play of Pico Pong.

Impressive! The agent has learnt how to return the ball in Pico-Pong by first building up the MDP and its transitions/rewards through repeated observations, and then computing the optimum policy for the constructed MDP through a look-ahead tree. [3]

[3] How would you extend these ideas to design a rudimentary ping pong robot which can track and return the ball?

15.3 Q-learning

15.3.1 Exploration vs. Exploitation

Let us analyze the case study with Pico-Pong more deeply. We can separate the process of learning into two different stages — *exploration* and *exploitation*:

- *Exploration:* This pertains to what the agent did in the first phase. Random paddle movements were used to help build up previously unknown knowledge of the MDP — transition probabilities and rewards.
- *Exploitation:* This pertains to what the agent did in the second phase. Specifically, the agent used the learnt MDP to play optimally.

In general, an RL environment is more complicated than Pico Pong, and there is no clear-cut boundary of when an agent has explored "sufficiently." It is best to combine the two stages (*i.e.*, exploration and exploitation) into one and "learn as you go." Also, it is difficult to balance between these two processes, and how to find the correct trade-off between exploration and exploitation is a recurring topic in RL.

15.3.2 Q-function

We now introduce the *Q-function*, an important concept that helps tie together concepts of exploration and exploitation when considering general MDPs with discounted rewards.

Definition 15.3.1 (Q-function)**.** *We define the* ***Q-function*** $Q : S \times A \to \mathbb{R}$ *as a table which assigns a real value* $Q(s, a)$ *to each pair* (s, a) *where* $s \in S$ *and* $a \in A$*.*

Intuitively, the value $Q(s, a)$ is the current *estimate* of the *expected discounted reward when we take action a from state s*. In other words, it is the estimate of the value $v_\pi(s)$ if π is any policy that will assign the action a to state s. Using the currently stored values of the Q-function, we can define a canonical policy π_Q. For each state s, the policy will assign the action a that maximizes the $Q(s, a)$ value; that is,

$$\pi_Q(s) = \arg\max_a Q(s, a)$$

Since the agent only has access to the estimate values $Q(s, a)$, but not the actual value function v, this is the most optimal policy to the agent's knowledge. Therefore, if the agent choose to take an exploitation step, they will take an action prescribed by the policy π_Q with respect to the currently maintained Q-function.

Instead of relying on the currently stored Q-function, we can also choose to take an exploration step. Every time we take an exploration step and receive additional information about the RL environment, we update the values of the Q-function accordingly. The goal of the Q-learning is to learn the *optimal Q-function*, which approximates the optimal policy π^* and the optimal value function v^* as closely as possible. We formalize the notion as follows:

Definition 15.3.2 (Optimal Q-function)**.** *The **optimal Q-function** is a Q-function that satisfies the following two conditions:*

- *The corresponding canonical policy π_Q is an optimal policy for the MDP.*
- *The Q-function satisfies the following condition:*

$$Q(s, a) = \sum_{s'; a} p(s' \mid s, a)(r(a \mid s, s') + \gamma \max_b Q(s', b)) \tag{15.2}$$

The first condition of Definition 15.3.2 states that for a fixed state s, the action a that maximizes $Q(s, a)$ is $a = \pi^*(s)$. This condition only cares about the relative ordering of the values of $Q(s, a)$ — as long as $Q(s, \pi^*(s))$ is the maximum value among all $Q(s, a)$, then it is fine. This condition guarantees that the action we take in the exploitation step is an optimal action.

The second condition is formally stating that the values of the Q-function are estimates of the expected reward when we take aciton a from state s. It also suggests that Q-function needs to "behave like" a value function v_π for some policy π. However, whereas a similar condition for a value function v_π only needs to hold for one particular action (*i.e.*, $a = \pi(s)$) given a state s, this condition for a Q-function should hold for any arbitrary action a. Note that for an optimal Q-function, the term $\max_b Q(s', b)$ in (15.2) is equivalent to $v_{\pi_Q}(s')$.

15.3.3 *Q-learning*

Now that we have defined the Q-function and the optimal Q-function, it is time for us to study how to learn the optimal Q-function. This process is called *Q-learning*. The basic idea is to probabilistically choose between exploration or exploitation: we define some probability $\epsilon \in [0, 1]$ such that we choose a random action a with probability ϵ (exploration) or choose the action a according to the current canonical policy π_Q with probability $1 - \epsilon$ (exploitation). If we choose the exploration option, we use its outcome to update the $Q(s, a)$ table. But how should we define the update rule?

Let's take a step back and consider a (plausibly?) real life scenario. You are a reporter for the Daily Princetonian at Princeton, and want to estimate the average wealth of alumni at a Princeton Reunions event. The alumni, understandably vexed by such a request, strike a compromise that you are only allowed to ask *one* alum about their net worth. Can you get an estimate of the average? Well, you could pick an alum at random and ask them their net worth! [4]

[4] The expectation gives the right average. But typically the answer would be far from the true average; especially if Jeff Bezos happens to be attending the reunion.

With this intuition, we return to the world of Q-learning. Suppose you start at some state s_t, take an action a_t, receive a reward of r_t, and arrive at state s_{t+1}. We call this process an *experience*. Now, when we update the current estimate of $Q(s_t, a_t)$, we ideally want to mimic the behavior of the optimal Q-function in (15.2) and update it to:

$$Q(s_t, a_t) = \sum_{s';\, a_t} p(s' \mid s_t, a_t) \cdot (r(a_t \mid s_t, s') + \gamma \max_b Q(s', b)) \qquad (15.3)$$

Notice that this is the weight average of the expected reward $r(a_t \mid s_t, s') + \gamma \max_b Q(s', b)$ over all possible next state s' given the action a_t. But in practice, the agent only has the ability to take a *single* experience; they lack the ability to "rest" and try all states s' according to the transition probability $p(s' \mid s_t, a_t)$. We thus must consider an alternative idea — we define the *estimate* for $Q(s_t, a_t)$ according to the experience at time step t as

$$Q'_t = r_t + \max_b Q(s_{t+1}, b)$$

This estimate can be calculated using the observed reward r_t and looking up the Q values of the state s_{t+1} on the Q-function table. Note that the expectation of Q'_t is exactly the right hand side of (15.3). That is,

$$\mathbb{E}[Q'_t] = \sum_{s';\, a_t} p(s' \mid s_t, a_t) \cdot (r(a_t \mid s_t, s') + \gamma \max_b Q(s', b))$$

This is because the agent took a transition to state s_{t+1} with probability $p(s_{t+1} \mid s_t, a_t)$ (of course, the agent does not know this value). This

is thus analogous to the single-sample estimate of average alumni wealth at the Princeton Reunions event. We can now define the following update rule of the Q-learning process:

$$\begin{aligned} Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \eta(Q'_t - Q(s_t, a_t)) \\ &= (1 - \eta)Q(s_t, a_t) + \eta Q'_t \end{aligned} \tag{15.4}$$

for some learning rate $\eta > 0$. You can understand this update rule in two different ways. First, we are gently nudging the value of $Q(s_t, a_t)$ towards the estimate Q'_t from the most recent experience. We can alternatively think of the updated value of $Q(s_t, a_t)$ as the weighted average of the previous value of $Q(s_t, a_t)$ and the estimate Q'_t. In either approach, the most important thing to note is that we combine both the previous Q value and the new estimate to compute the updated Q value. This is because the new estimate is just a single sample that can be far off from the actual expectation, and also because after enough iterations, we can assume the previous Q value to contain information from past experience.

Example 15.3.3. *Let's return to our adventures in Pico-Pong and consider the situation in Figure 15.10. Denote the state in the left diagram as s_t and the state in the right as s_{t+1}. Suppose the current value of $Q(s_t, a) = 0.4$ with $a = \uparrow$. Assuming that $Q(s_{t+1}, a) = 0$ for all a, we can compute the estimate Q'_t from this experience as*

$$Q'_t = r_t + \max_b Q(s_{t+1}, b) = 1$$

Then the Q value will be increased to $0.4 + 0.6\eta$.

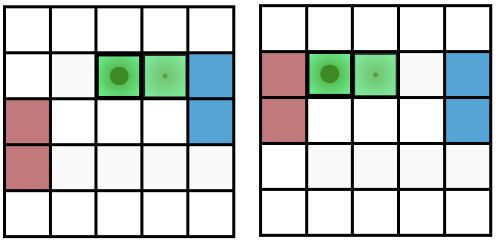


Figure 15.10: The diagram representing two states in a game of Pico-Pong.

15.3.4 Deep Q-learning

Note that the update rule in (15.4) looks similar to the Gradient Descent algorithm. They are both iterative processes which incorporate a learning rate η. In fact, you can consider the Q-learning update rule to be trying to minimize the squared difference between $Q(s_t, a_t)$ and Q'_t. The similarity between the Q-learning update rule and the Gradient Descent algorithm allows us to utilize deep neural network

to learn the optimal Q-function. Such a network is called the *Deep Q Network (DQN)*.

In a DQN, the Q-function can be represented by the parameters $\mathbf{W}$ of the network. We emphasize this by denoting the Q-function as $Q_{\mathbf{W}}(s, a)$. Now instead of directly updating the Q-function as in the update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta(Q'_t - Q(s_t, a_t)) \qquad \text{(15.4 revisited)}$$

we instead update the parameters $\mathbf{W}$ such that the Q-function is updated accordingly.

First consider the case that $Q'_t > Q(s_t, a_t)$. That is, the estimated Q-value is larger than the currently stored value. Then the update rule (15.4) will increase the value of $Q(s_t, a_t)$. To mimic this behavior, we want to find an update rule for $\mathbf{W}$ that will increase the Q-value. This is given as:

$$\mathbf{W} \leftarrow \mathbf{W} + \beta \cdot \nabla_{\mathbf{W}} Q_{\mathbf{W}}(s_t, s_a)$$

for some learning rate $\beta > 0$.

Problem 15.3.4. *Suppose $Q'_t < Q(s_t, a_t)$. How should we design the weight updates?*

One final thing to note is a technique called *experience replay*. Experiencing the environment can be expensive (*i.e.*, computation time, machine wear, etc.). Therefore, it is customary to keep a history of old experiences and their rewards, and periodically take a random sample out of the old experiences to update the Q values. In particular, experience replay ensures that DQNs are efficient and avoid "catastrophic forgetting." [5]

[5] Catastrophic forgetting is a phenomenon where a neural network, after being exposed to new information, "forgets" information it had learned earlier

15.4 *Applications of Reinforcement Learning*

15.4.1 *Q-learning for* Breakout *(1978)*

We previously considered using reinforcement learning for *Pong*. We can also use it for another famous Atari game called *Breakout*. One particular design uses a CNN to process the screen and uses the "score" as a reward. As shown in Figure 15.11, the model becomes quite successful after several epochs.

15.4.2 *Self-help Apps*

Self-help apps are designed to aid in recovery of the user from addiction, trauma heart disease, etc. A typical design involves an RL algorithm which determines the next advice/suggestion based upon reversals, achieved milestones, etc. so far. These can be a helpful supplement to expensive therapy/consultation.

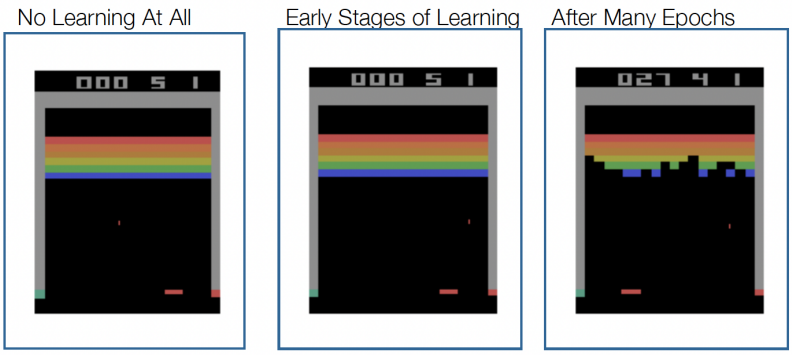


Figure 15.11: An application of Q-learning to the famous Atari game *Breakout*.

15.4.3 *Content Recommendation*

At reputable websites, we might imagine that there exists a page creation system designed to capture the "reward" of user engagement. We can use MDP techniques to model this situation. Specifically, we can define s_0 as the outside link which brought the user to the landing page and/or the past history of the user on the site. If the user clicks on a link, a new page is created and we can define s_1 as a concatenation of s_0 and the new link. If the user again clicks on a link, another new page is created and we can define s_2 as the concatenation of s_1 and the new link.

15.5 *Deep Reinforcement Learning*

Deep Reinforcement Learning is a subfield of machine learning that combines the methods of Deep Learning and Reinforcement Learning that we have discussed earlier. [6] The goal of it is to create an artificial agent with human-level intelligence (AGI). In general, Reinforcement Learning defines the objective and Deep Learning gives mechanism for optimizing that objective. Deep RL method combines the problem given by the RL with the solution given by the DL. In the cited source video, RL expert David Silver made three broad conjectures related to this topic.

[6] Source: https://www.youtube.com/watch?v=x5Q79XCxMVc

1. RL is enough to formalize the problem of intelligence
2. Deep neural networks can represent and learn any computable function
3. Deep RL can solve the problem of intelligence

Many Deep RL models are trained to play games (*e.g.*, chess, Go) because it is easy to evaluate progress. By letting them compete against humans, we can easily compare them to human-level intelligence. As

an example, Google Deepmind trained a Deep RL model called DQN to play 49 arcade games. [7] The computer is not given the explicit set of rules; instead, given only the pixels and game score as input, it learns by using deep reinforcement learning to maximize its score. Amazingly, on about half of the games, the model played at least at human level of intelligence!

[7] For the full paper, visit https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

15.5.1 *Chess: A Case Study*

Founders of AI considered chess to be the epitome of human intelligence. In principle, the best next move can be calculated via a look-ahead tree (similar to Figure 13.5 from the cake-eating example). Since chess is a two-player game, we can use an algorithm called the *min-max search* on the look-ahead game tree. [8]

[8] Source: https://www.youtube.com/watch?v=l-hh51ncgDI

Usually, RL agents are playing against the nature that causes them to take random transitions according to the MDP's transition probabilities. But in chess, the agent plays against an opponent that is trying to make you take the largest possible loss (the largest possible gain for the opponent). That is why we need a min-max evaluation of the look-ahead tree.

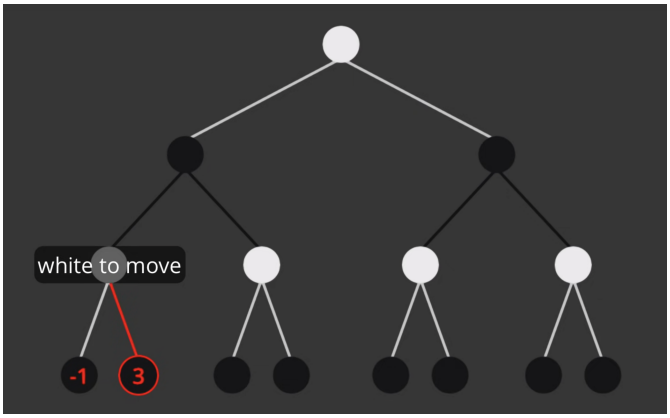


Figure 15.12: An example look-ahead game tree for chess with depth 3. White will choose the right option.

In Figure 15.12, the numbers at the leaf nodes represent a static evaluation of how good the game configuration is for white. This is an approximation for the actual value of the node. An example metric in chess would be the difference in the number of pieces (# white − # black). These numbers are evaluated either when the game terminates or when the algorithm has reached the specified number of steps to look ahead. If the game ever reaches the specified node, the white has two options to choose from: if white chooses the left child node, it will end up with reward of -1; whereas if it chooses the right child node, the reward will be 3. Then to maximize reward, the best move of white will be to choose 3. [9]

[9] For those who are familiar with chess or game theory in general, this is known as the *best response*.

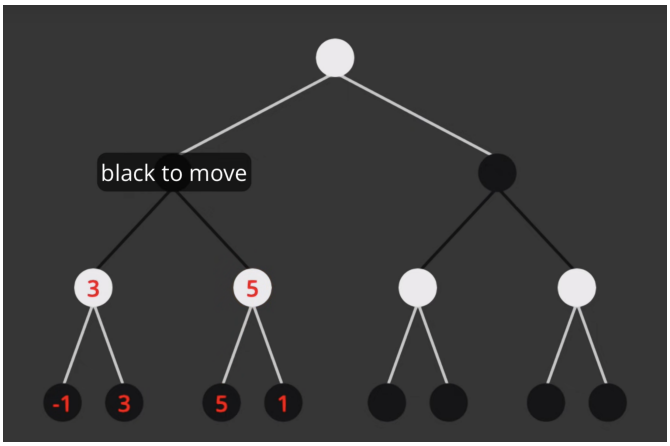


Figure 15.13: Black will choose the left option.

In Figure 15.13, it is now black's turn to choose. Note that the reward for black is the opposite of the reward for white, so black wants to *minimize* the value on the tree. Therefore, black will want to choose the left child node.

So whenever we are at a configuration, we can create a look-ahead tree for a reasonable number of steps and try to calculate the best move. But the size of a game tree is astronomical, so it is computationally infeasible to search all levels of the tree. [10]

[10] There is an optimization method called the alpha-beta pruning. Consult the video referenced above for an implementation on the game of chess.

15.5.2 *AlphaGo: A Case Study*

Go is a game invented in China around 500 BC. It is played by 2 players on a 19×19 grid. Players take turns placing stones on the grid, and if any set of stones is entirely surrounded by opponent stones, the enclosed stones are taken away from the board and awarded to the opponent as points. Even though the rules are very simple, no computer could beat a good human amateur at Go until 2015. [11]

[11] In comparison, IBM's Deep Blue model beat the world chess champion Kasparov in 1997.

How can we utilize RL concepts to play this game? In general, we can create a Deep Policy Net (DPN) to learn W, which is a function that takes state s as an input and outputs a probability distribution $p_W(a \mid s)$ over the next possible actions from s. AlphaGo is an example of a DPN engineered by the Google Deepmind lab. It takes the current board position as the input and uses ConvNet to learn the internal weights, and outputs the value given by a softmax function. In its initial setup, the DPN was trained using a big dataset of past games. [12]

[12] Source: `https://www.youtube.com/watch?v=Wujy7OzvdJk`

To be more specific, AlphaGo used supervised learning from human data to learn the optimal policy (action to take at each game setting). In other words, it used convolutional layers to replicate the moves of professional players as closely as possible. Since the CNN is just mimicking human players, it cannot beat human champions.

Figure 15.14: The diagram representing the process of training AlphaGo.

However, it can be used to search the full game tree more efficiently than the alpha-beta search. Formally, this method is called the Monte Carlo Tree Search, where the CNN is used to decide the order in which to explore the tree. After the policy network was sufficiently trained, reinforcement learning was used to train the value network for position evaluation. Given a board setting, the network was trained to estimate the value (*i.e.*, likelihood of winning) of that setting.

AlphaGo Zero is a newer version of the model that does not depend on any human data or features. In this model, policy and value networks are combined into one neural network, and the model does not use any randomized Monte-Carlo simulations. It learns solely by self-play reinforcement learning and uses neural network (resnet) to evaluate its performance. Within 3 days of training, AlphaGo Zero surpassed an earlier version of AlphaGo that beat Lee Se Dol, the holder of 8 world titles; within 21 days, it surpassed the version that beat Ke Jie, the world champion. Interestingly enough, AlphaGo Zero adopted some opening patterns commonly played by human players, but it also discarded some common human patterns and it also discovered patterns unknown to humans.

The newest version of AlphaGo is called the AlphaZero. It is a model that can be trained to play not just Go but simultaneously Chess and Shogi (Japanese chess). After just a few hours of training, AlphaZero surpassed the previous computer world champions (*Stockfish* in Chess, *Elmo* in Shogi, and *AlphaGo Zero* in Go). Just as AlphaGo Zero did, AlphaZero was able to dynamically adopt or discard known openings in chess.

Part V

Advanced Topics

16
Machine Learning and Ethics

Throughout this course, we have discussed the technical aspects of model design, training, and testing in depth. However, we have not yet discussed some of the social implications of this technology. What are some ethical and legal issues in deployment of ML techniques in society? What are the caveats and limitations to temper our exuberance about the possibilities of ML? This brief chapter addresses these issues, and we hope as technologists you will continue to investigate and consider such issues throughout your career.

16.1 Facebook's Suicide Prevention

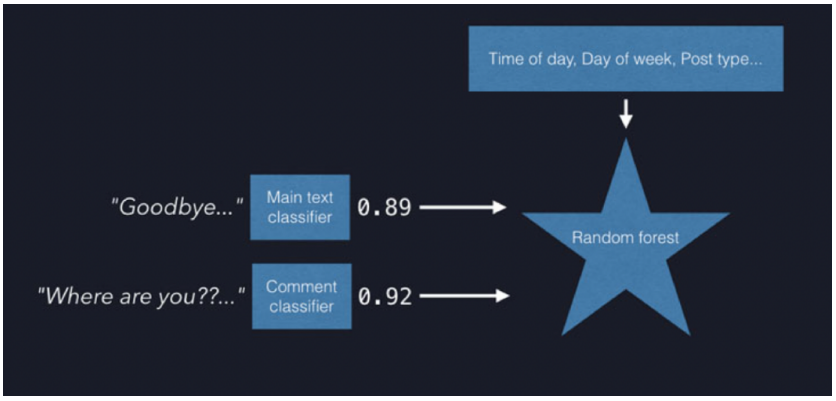


Figure 16.1: A visualization of the Facebook model to predict suicides.

In 2017, Facebook launched a program to use a machine learning algorithm to predict suicide risk amongst its user population. It has continued with various iterations over the years. Figure 16.1 gives a visualization of the four-step process:

1. ML algorithm automatically analyzes a post by processing its text content and comments

2. Algorithm additionally uses spatial-temporal context of the post to perform a risk prediction
3. A human behind the algorithm performs a personal review to finally verify if a threshold is reached
4. If the post is poses a serious risk, Facebook performs a wellness check through the person's contacts, community organizations, etc.

At first sight, this may appear to be very good idea: even if it saves just one life, surely the project is worth it? But the announcement of the project cause a lot of controversy among people. The following are some of the potential problems that people identified:

1. False positives may result in stigmatization.
2. Many people who contemplate suicide do not end up going through with it. Facebook's reporting could lead to criminal penalties (in regions where suicide is a crime), involuntary hospitalization, stigmatization etc.
3. Involvement of authorities (*e.g.*, law enforcement) raises risk of illegal seizures
4. Should Facebook be liable for any problem caused by misdetection?

Beyond these points, there are deep philosophical questions associated with the concept of suicide as well. For instance, is suicide actually immoral? Even if it is immoral, is it the responsibility of Facebook to get involved? Is it moral for Facebook to use personal information to assess suicide risk? Opinions differ.

16.2 Racial Bias in Machine Learning

Suppose we are designing a machine learning approach for loan approval. The general approach will be to take a dataset of $(\vec{\mathbf{x}}, y)$, where $\vec{\mathbf{x}}$ is a vector of the individual's attributes (*e.g.*, age, education, alma mater, address, etc.) who got a loan and $y \in \{-1, 1\}$ indicates whether they actually paid off the loan or not. Using the approaches we learned in Section 4.2, we could train a binary classifier through logistic regression. Civil rights legislation forbids using the individual's race in many of these decisions, so while training we could simply mask out any coordinates which identify race. However, this does not guarantee that the classifier will be entirely "race-neutral." [1]

In 2016, a study [2] found that COMPAS, a leading software for assessing the probability that a prison inmate would commit another

[1] The reason is that race happens to be correlated with many other attributes. Thus if a classifier uses any of the correlated attributes, it may be implicitly using racial information in the decision making process.

[2] *Machine Bias*, by Anwin et al., in *Pro Publica* 2016.

serious crime, disproportionately tags African-American as being likely to commit crimes — in the sense that African-Americans who were tagged as likely to commit another crime were only half as likely to actually commit a crime than a similarly-tagged person of another race.

	White	**African-American**
Labeled Higher Risk & Did *Not* Re-offend	23.5%	44.9%
Labeled Lower Risk & *Did* Re-offend	47.7%	28.0%

Table 16.1: COMPAS correctly predicts recidivism 61 percent on average. But African-Americans are almost twice as likely as whites to be labeled a higher risk but not actually re-offend. Conversely, whites are twice as likely as African-Americans to be labeled lower risk but go on to commit other crimes.

16.3 *Conceptions of Fairness in Machine Learning*

We will briefly consider possible ways to formulate fairness in machine learning. Keep in mind that this task is intrinsically difficult, as we are attempting to assign a technological perspective to a fundamentally normative problem. The first property we might want a ML classifier to have is called *demographic parity*, which effectively enforces that the output of classifier does not depend on a protected attribute (*e.g.*, race, ethnicity, gender).

Definition 16.3.1 (Demographic Parity)**.** *We say that a binary classifier that outputs $y \in \{-1, 1\}$ satisfies* ***demographic parity*** *if $\Pr[y \mid x_i = a] = \Pr[y \mid x_i = b]$ where a, b are any two values that a protected attribute x_i can take.*

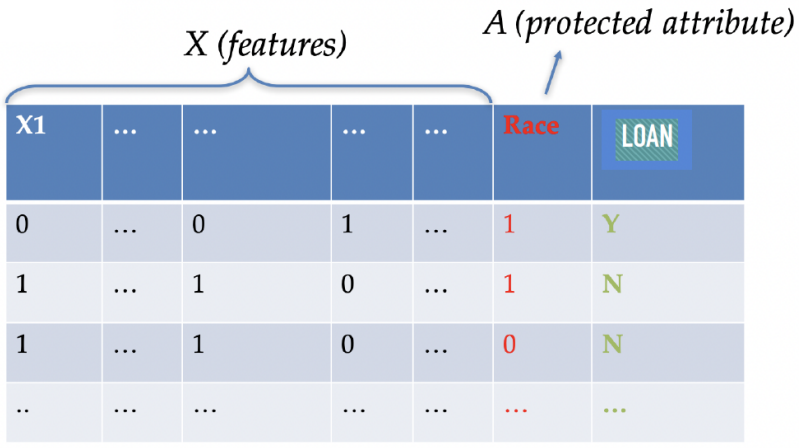


Figure 16.2: A hypothetical application of ML to a loan approval application. *Race* has been made a protected attribute in an attempt to prevent bias during training.

A visualization of how a protected attribute could be specified in a dataset is shown in Figure 16.2. Consider the loan approval example

from the previous section. If the binary classification model for the loan approval satisfies the demographic parity property, then the model approve loans for different races at similar rates. One way to achieve this condition is to use a regularizer term $\lambda(\Pr[y \mid x_i = a] - \Pr[y \mid x_i = b])^2)$ during training. [3]

[3] Does this seem like a good formulation of fairness?

Another property we want a "fair" model to satisfy is called the *predictive parity*. This is the property that the model in Table 16.1 failed to satisfy.

Definition 16.3.2 (Predictive Parity)**.** *We say that a binary classifier that outputs $y \in \{-1, 1\}$ satisfies **predictive parity** if the true negative/false negative/false positive/true positive rates are the same for any values of a sensitive attribute.*

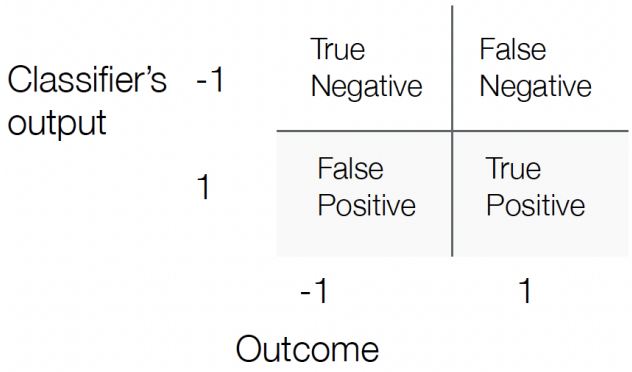


Figure 16.3: A table of all possible outcomes based on the model output and the ground truth outcome. This is also known as a *confusion matrix*.

Ideally, we want a ML model to satisfy both the demographic parity and predictive parity. However, it turns out that these two notions are incompatible!

Theorem 16.3.3 (Fairness Impossibility Theorem)**.** [4] *Under fairly general conditions,* demographic parity *and* predictive parity *are incompatible.*

[4] See *Inherent Trade-Offs in the Fair Determination of Risk Scores*, Kleinberg, Mullainathan, and Raghavan, *ITCS* 2017. The paper actually considered three possible definitions of "fairness" and showed every pair of them are mutually incompatible.

There are other formulations of fairness, but it is difficult to find a combination of these notions that are compatible with each other. So one way or another, we need to sacrifice some notions of "fairness."

16.4 Limitations of the ML Paradigm

The predictive power of ML seems immense, but is it true that if we have enough data and the right algorithm, then everything become predictable? If yes, then one could imagine societal programs leveraging this to precisely target help to where it would be more effective. We first consider a famous — and somewhat amusing — example of a study [5] that turned out to be false.

[5] *Extraneous factors in judicial decisions*, Danziger et al., *PNAS* 2011.

16.4.1 *Hungry Judge Effect*

The study analyzed the parole decisions made by 8 Israeli judges in over 1,100 cases. The data in Figure 16.4 shows that prisoners were much more likely to be granted parole after the judge took a lunch break or a coffee break. The study therefore suggested that judges tend to be stricter before a break (maybe because they are "hangry") but more lenient when they return from the break.

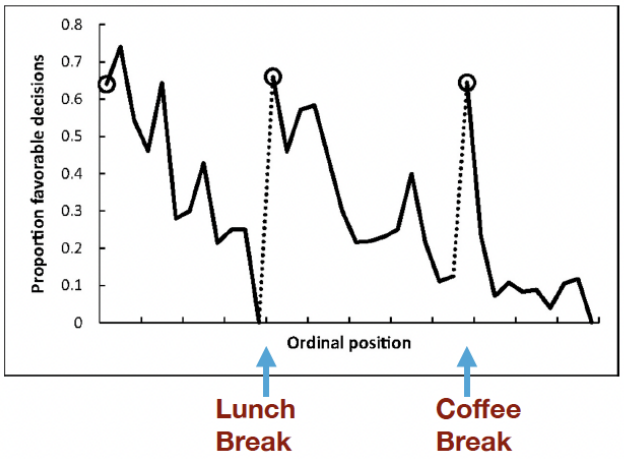


Figure 16.4: Data from the study shows an uptick in favorable decisions following a lunch break or a coffee break.

Nevertheless, it turns out that this "hungry judge effect" can be explained by a completely different reason. A followup study [6] found that the ordering of cases presented to the judge was not random: prisoners with attorneys were scheduled at the beginning of each session, while prisoners without an attorney were scheduled at the end of a session. The former group were let on parole with a rate of 67%, while the rate was just 39% for those without attorneys. Another important observation was that attorneys tended to present their cases in decreasing order of strength of case, with the average attorney having 4.1 clients. Computer simulations of hunger-immune judges faced with cases presented according to these percentages showed the same see-saw effect of Figure 16.4.

[6] *Overlooked factors in the analysis of parole decisions*, Weinshall-Margel and Shepard, *PNAS*, 2012.

16.4.2 *Fragile Families Challenge*

The Fragile Families Challenge is a collaborative project initiated by the Center for Research on Child Wellbeing at Princeton University. A brief description of the initiative's motivation is provided on the website: [7]

> *The Fragile Families Challenge is a mass collaboration that combines predictive modeling, causal inference, and in-depth interviews to yield insights that can improve the lives of disadvantaged children in the United States.*

[7] Source: `https://www.fragilefamieschallenge.org`.

By working together, we can discover things that none of us can discover individually.

The Fragile Families Challenge is based on the Fragile Families and Child Wellbeing Study, which has followed thousands of American families for more than 15 years. During this time, the Fragile Families study collected information about the children, their parents, their schools, and their larger environments.

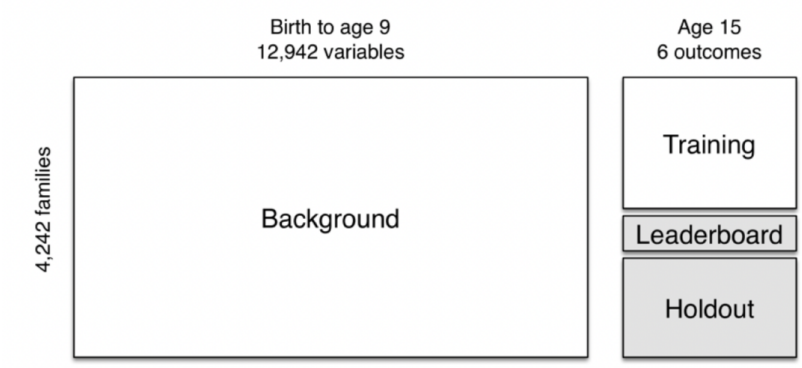


Figure 16.5: Diagram illustrating the dataset of the Fragile Families Challenge.

The initiative has collected immense data on multiple families, including interviews with mothers, fathers, and/or primary caregivers at several ages. Interviewees were inquired as to attitudes, relationships, parenting behavior, economic and employment status, etc. Additionally, in-home assessments of children and their home environments were performed to assess cognitive and emotional development, health, and home environment. The goal was to predict six key outcomes at age 15 (*e.g.*, whether or not the child is attending school) given background data from birth to age 9 as shown in 16.5. However, up to this point no method has done better than random guessing.

This is food for thought: what is going on?

16.4.3 General Limits to Prediction

Matt Salganick and Arvind Narayanan, professors at Princeton University, recently started a course [8] which aims to explore the extent to which interdisciplinary problems in social science and computer science can be predictable. In general, following are some major themes that can make prediction difficult:

1. The distribution associated with data can shift over time
2. The relationship between input data and desired outputs can change over time

[8] The course, COS 597E/SOC 555 is a seminar first offered in Fall 2020.

3. There is a possibility for unknown coordinates to be unintentionally ignored (*i.e.*, as in the hungry judge effect)

4. The "8 billion problem," which outlines how data available in the real world is fundamentally finite and limited

16.5 Final Thoughts

As described in the preceding sections, users and designers of machine learning will often face ethical dilemmas. Designers may have to operate without moral clarity or easy technical fixes. In fact, technical solutions may even be impossible. To appropriately acknowledge these limitations, it is important to embrace a culture of measuring and openly discussing the impact of the system being built. Indeed, a general principle to follow is to avoid harm when trying to do good.

17
Deep Learning for Natural Language Processing

17.1 Word Embeddings

In traditional NLP, each word is regarded as discrete symbols each with a single value of weight. For example, in Chapter 1, we learned how to use linear regression on sentiment prediction. But with this approach, it is hard for the computer to learn the *meaning* of the word; instead, each of the words remain as some abstract symbols with numeric weights.

But how do computers know the meaning of words? We can easily think of one solution: we can look up words in a dictionary. For example, WordNet is a project that codes the meaning of the words and the relationship between the words, so that the data can be used for computers to parse. [1] But resources like WordNet require human labor to create and adapt, and it is impossible to keep up-to-date (because new words are coined up and new meanings appear out of existing words).

[1] For more information, check http://wordnetweb.princeton.edu.

An alternative approach is to represent words as short (50 - 300 dimensions [2]), real-valued vectors. These vectors encode the meaning and other properties of words. In this representation, the distance between vectors represent the *similarity* between words. This vectorized form of the words are much easier to be used as inputs in modern ML systems (especially neural networks). This vector form of words is known as *word embedding*. In this section, we explore the process of how to learn a good word embedding.

[2] The dimension of word vectors is a hyperparameter that needs to be decided first.

17.1.1 Distributional Hypothesis

Word embedding is based on a concept called the *distributional hypothesis*, a theory developed by John Rupert Firth. The hypothesis, one of the most successful ideas of modern statistical NLP, says that words that occur in similar contexts tend to have similar meaning.

Definition 17.1.1 (Context). *When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).*

Example 17.1.2. *Assume that you first heard the word **tejuino** and have no idea what the word means. But you learn that the word may appear in the following four contexts.*

- *C1: A bottle of ____ is on the table.*
- *C2: Everybody likes ____.*
- *C3: Don't have ____ before you drive.*
- *C4: We make ____ out of corn.*

Based on these contexts, it is reasonable to conclude that the word "tejuino" refers to some form of alcoholic drink made from corn.

Problem 17.1.3. *To find words with similar meanings as "tejuino," we tried filling out the contexts from Example 17.1.2 with 5 other words. The results are given in Table 17.1, where 1 means that the word was appropriate to be used in that context, and 0 means that it was inappropriate.*

	C1	C2	C3	C4
tejuino	1	1	1	1
loud	0	0	0	0
motor-oil	1	0	0	0
tortillas	0	1	0	1
choices	0	1	0	0
wine	1	1	1	0

Table 17.1: Data showing if 6 words are appropriate for the four contexts in Example 17.1.2.

Which word is closest to "tejuino"?

17.1.2 Word-word Co-occurrence Matrix

Given a very large collection of documents with words from a dictionary V, we construct a $|V| \times |V|$ matrix X, where the entry at the i-th row, j-th column denotes the number of times (*i.e.*, frequency) that w_j appears in the context window of w_i. This matrix is called the *word-word co-occurrence matrix*.

Example 17.1.4. *Table 17.2 shows a portion of a word-word co-occurrence matrix. Each row corresponds to the* center word w_i*, and each column corresponds to the* context word w_j*. The value X_{ij} at the (i, j) entry means that the context word w_j appeared X_{ij} times in the context (of length 4) of w_i in total.*

Although the portion shown in Table 17.2 mostly has non-zero entries, in general, the entries of the matrix are mostly zero.

	⋯	computer	data	result	pie	sugar	⋯
cherry	⋯	2	8	9	442	25	⋯
strawberry	⋯	0	0	1	60	19	⋯
digital	⋯	1670	1683	85	5	4	⋯
information	⋯	3325	3982	378	5	13	⋯

Table 17.2: A portion of a word-word co-occurrence matrix for a corpus of Wikipedia articles. Source: https://www.english-corpora.org/wiki/.

17.1.3 *Factorization of Word-word Co-occurrence Matrix*

Recall the example of movie recommendation through matrix factorization in Chapter 9. In that example $m \times n$ matrix M was factorized into $M \approx AB$ where the i-th row of A was a d-dimensional vector that represented user i and the j-th column of B was a d-dimensional vector that represented movie j.

We can imagine a similar factorization on the word-word co-occurrence matrix. That is, we can represent each *center word* and each *context word* as a d-dimensional vector such that $X_{ij} \approx A_{i*} \cdot B_{*j}$. But this particular idea does not work on the word-word co-occurrence matrix. The key difference is that X is a complete matrix with no missing entries (although most entries are zero). Therefore we instead use other standard matrix factorization techniques (*e.g.*, Singular Value Decomposition).

One popular choice of factorization is running the Singular Value Decomposition (SVD) on a weighted co-occurrence matrix. [3] This idea originates from a concept called *Latent Semantic Anlysis*. [4] If the SVD returns the following decomposition,

[3] The particular weighting scheme is called *PPMI*. We will not get into the detail here.
[4] From *Indexing by Latent Semantic Analysis* by Deerwester et al., 1990.

$$\begin{bmatrix} & X & \end{bmatrix} = \begin{bmatrix} & W & \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d \end{bmatrix} \begin{bmatrix} & W^\intercal & \end{bmatrix}$$

where X is a $|V| \times |V|$ matrix and W is a $|V| \times d$ matrix, then the i-th row of matrix W can be regarded as the embedding for word w_i.

Other modern approaches tend to treat word vectors as parameters to be optimized for some objective function and apply the gradient descent algorithm. But the principle is the same: "words that occur in similar contexts tend to have similar meanings." Some of the popular algorithms with this approach include: *word2vec* (Mikolov et al., 2013), *GloVe* (Pennington et al., 2014), and *fastText* (Bojanowski et al., 2017).

Here we briefly explain the GloVe algorithm. Given the co-occurrence table X, we will construct a *center word vector* $\vec{\mathbf{u}}_i \in \mathbb{R}^d$ and a *context word vector* $\vec{\mathbf{v}}_j \in \mathbb{R}^d$ such that they optimize the follow-

ing objective:

$$J(\theta) = \sum_{i,j \in V} f(X_{ij}) \left(\mathbf{u}_i \cdot \mathbf{v}_j + b_i + \widetilde{b}_j - \log X_{ij} \right)^2 \tag{17.1}$$

where f is some non-linear function and $b_i, \widetilde{b}_j$ are bias terms. This is within the same line of logic as optimizing

$$L(A, B) = \frac{1}{|\Omega|} \sum_{i,j \in \Omega} (M_{ij} - (AB)_{ij})^2 \qquad ((9.5) \text{ revisited})$$

17.1.4 Properties of Word Embeddings

A good word embedding should represent the meaning of the words and their relationship with other words as accurately as possible. Therefore there are some properties that we would like a word embedding to preserve. We will discuss three such properties and see how the current algorithms for word embedding perform on preserving those properties.

1. Similar words should have similar word vectors: This is the most important property we can think of.

Example 17.1.5. *In a certain word embedding, the following is the list of 9 most nearest words to the word "sweden."*

Word	*Cosine distance*
norway	*0.760124*
denmark	*0.715460*
finland	*0.620022*
switzerland	*0.588132*
belgium	*0.585835*
netherlands	*0.574631*
iceland	*0.562368*
estonia	*0.547621*
slovenia	*0.531408*

Notice Scandanavian countries are the top 3 entries on the list, and the rest are also European country names.

2. Vector difference should encode the relationship between words: If there are two or more pairs of words where each pair of words are distinguishable by the same attribute, you can imagine that the vector difference within each pair is nearly the same.

Example 17.1.6. *In Figure 17.1, notice that $v_{man} - v_{woman} \approx v_{king} - v_{queen}$. The vector difference in common can be understood as representing the male-female relationship. Similarly, there seems to be a common vector difference for representing the difference in verb tense.*

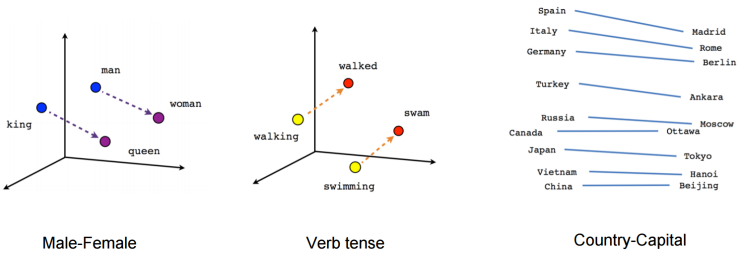


Figure 17.1: Two pairs of words that differ in the same attribute show a similar difference in their word embeddings.

3. The embeddings should be translated between different languages: When we independently find the word embedding in different languages, we can expect to have a bijective mapping that preserves the structure of the words in each language. [5]

[5] From *Exploiting Similarities among Languages for Machine Translation* by Mikolov et at., 2013.

Example 17.1.7. *In Figure 17.2, notice that if we let W to be the mapping from English to Spanish word embeddings, $v_{cuatro} \approx W \circ v_{four}$*

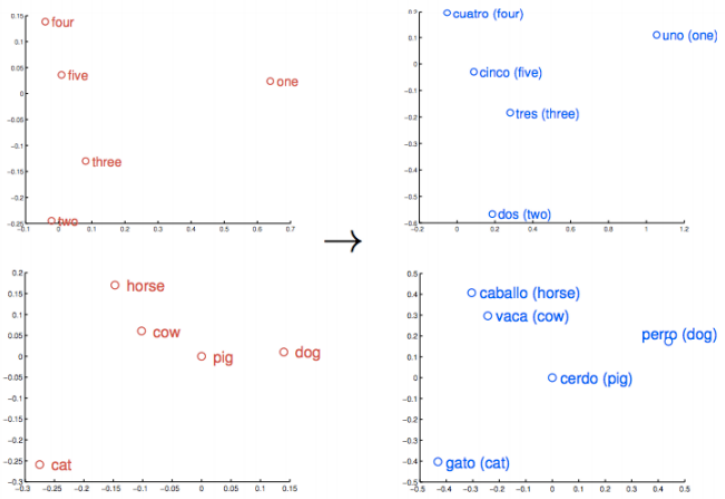


Figure 17.2: Word embeddings are translated into the embeddings of other languages.

17.2 *N-gram Model Revisited*

Recall the n-gram model from Chapter 8. It assigned a probability $\Pr[w_1 w_2 \dots w_n]$ to every word sequence $w_1 w_2 \dots w_n$. We discussed the concept of perplexity of the model to compare the performance of unigram, bigram, and trigram models. While the n-gram model is impressive, it has obvious limitations.

Problem 17.2.1. *"The students opened their _____." Can you guess the next word?*

Problem 17.2.2. *"As the proctor started the clock, the students opened their _____." Can you guess the next word?*

In a lot of cases, words in a sentence are closely related to other words and phrases that are far away. But the n-gram model cannot look beyond the specified frame.

Example 17.2.3. *The following is a text generated by a 4-gram model*

> *Today the price of gold per tan, while production of shoe lasts and shoe industry, the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks, sept 30 and primary 76 cts a share.*

The generated text is surprisingly grammatical, but incoherent.

Example 17.2.3 shows that we need to consider more than three words at a time if we want to model language well. But if we use a larger value of n for the n-gram model, the data will become too sparse to estimate the probabilities. But even when we restrict ourselves to words that appear in the dictionary, there are 10^{21} distinct sequences of 4 words.

17.2.1 Feedforward Neural Language Model

The idea of *feedforward neural language model* was proposed by Bengio et al. in 2003 in a paper called *A Neural Probabilistic Language Model*. The intuition is to use a neural network to learn the probabilistic distribution of language, instead of estimating raw probabilities. The key ingredient in this model is the word embeddings we discussed earlier.

Example 17.2.4. *Assume we are given two contexts "You like green _____" and "You like yellow _____" to fill the blanks in. A n-gram model will try to calculate the raw probabilities* $\Pr[w \mid \textit{You like green}]$ *and* $\Pr[w \mid \textit{You like yellow}]$*. However, if the word embeddings showed that* $v_{green} \approx v_{yellow}$*, then we can imagine that the two contexts are similar enough. Then we may be able to estimate the probabilities better.*

Now we show how to use feedforward neural language model on a n-gram model. Assume we want to estimate the probability $\Pr[w_{n+1} \mid w_1 \dots w_n]$. Then the first step is to find a find a word embedding

$$v_1, v_2, \dots, v_n \in \mathbb{R}^d$$

of each word $w_1, w_2, \dots, w_n$. Then we concatenate the word embeddings into [6]

$$\vec{\mathbf{x}} = (v_1, \dots, v_n) \in \mathbb{R}^{nd}$$

[6] the order of the input vectors cannot change

This will be the input layer. Then we define the fully connected hidden layer as

$$\vec{\mathbf{h}} = \tanh(\mathbf{W}\vec{\mathbf{x}} + \vec{\mathbf{b}}) \in \mathbb{R}^h$$

where $\mathbf{W} \in \mathbb{R}^{h \times nd}$ and $\vec{\mathbf{b}} \in \mathbb{R}^h$. Then we define the output layer as

$$\vec{\mathbf{z}} = \mathbf{U}\vec{\mathbf{h}} \in \mathbb{R}^{|V|}$$

where $\mathbf{U} \in \mathbb{R}^{|V| \times h}$. Then finally, the probability will be calculated with the softmax function:

$$\Pr[w = i \mid w_1 \dots w_n] = \text{softmax}_i(\vec{\mathbf{z}}) = \frac{e^{z_i}}{\sum_{k \in V} e^{z_k}}$$

So the total number of parameters to train in this network is

$$d\,|V| + ndh + h + h\,|V|$$

where the terms are respectively for the input embeddings, $\mathbf{W}, \vec{\mathbf{h}}, \mathbf{U}$. When $d = h$, sometimes we tie the input and output embeddings. That is, we can consider $\mathbf{U}$ to be the parameters required for the output embeddings. At this point, the language model reduces to a $|V|$-way classification, and we can create lots of training example by sliding the input-output indices. That is, when given a huge text, we can create lots of input-output tuple as follows: $((w_1, \dots, w_n), w_{n+1}), ((w_2, \dots, w_{n+1}), w_{n+2}), \dots$.

17.2.2 *Beyond Feedforward Neural Language Model*

But the feedforward language model still has its limitations. The main reason is that $\mathbf{W} \in \mathbb{R}^{h \times nd}$ scales linearly with the window size. Of course, this is better than the traditional n-gram model which scales exponentially with n. Another limitation of the neural LM is that the model learns separate patterns for the same item. That is, a substring $w_k w_{k+1}$, for example, will correspond to different parameters in $\mathbf{W}$ when trained on $(w_k w_{k+1} \dots w_{k+n-1})$ or on $(w_{k-1} w_k \dots w_{k+n-2})$.

To mitigate these limitations, we can choose to use similar modeling ideas but use better and bigger neural network architectures like *recurrent neural networks (RNN)* or *transformers*.

Here we briefly explain the core ideas of a RNN. RNNs are a family of neural networks that handle variable length inputs. Whereas feedforward NNs map a fixed-length input to a fixed-length output, recurrent NNs map *a sequence* of inputs to *a sequence* of outputs. The sequence length can vary and the key is to *reuse the weight matrices* at different time steps. When the inputs are given as $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \dots \vec{\mathbf{x}}_T \in \mathbb{R}^d$ and we want to find outputs $\vec{\mathbf{h}}_1, \vec{\mathbf{h}}_2, \dots \vec{\mathbf{h}}_T \in \mathbb{R}^h$, we train the parameters

$$\mathbf{W} \in \mathbb{R}^{h \times h}, \mathbf{U} \in \mathbb{R}^{h \times d}, \vec{\mathbf{b}} \in \mathbb{R}^h$$

such that

$$\vec{\mathbf{h}}_t = g(\mathbf{W}\vec{\mathbf{h}}_{t-1} + \mathbf{U}\vec{\mathbf{x}}_t + \vec{\mathbf{b}}) \in \mathbb{R}$$

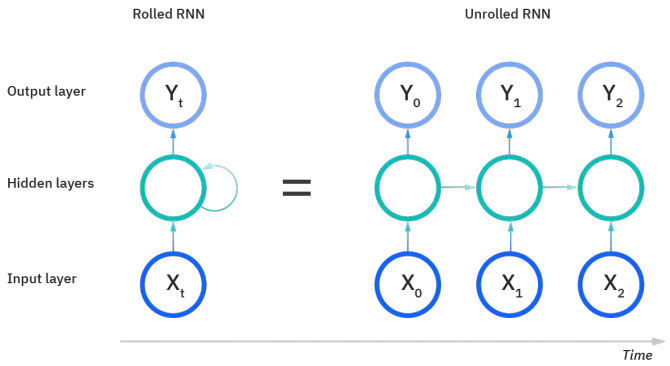


Figure 17.3: A visual representation of an RNN architecture.

where g is some non-linear function (*e.g.*, ReLU, tanh, sigmoid). We can also set $\vec{\mathbf{h}}_0 = \vec{\mathbf{0}}$ for simplicity.

Part VI

Mathematics for Machine Learning

18
Probability and Statistics

18.1 Probability and Event

18.1.1 Sample Space and Event

Probability is related to the uncertainty and randomness of the world. It measures the likelihood of some outcome or event happening. To formalize this concept, we introduce the following definition:

Definition 18.1.1 (Sample Space and Event)**.** *A set S of all possible outcomes of a random phenomenon in the world is called a **sample space**. A subset $A \subset S$ is called an **event**.*

Example 18.1.2. *The sample space of "the outcome of tossing two dice" is the set $S = \{(1,1),(1,2),\ldots,(6,6)\}$ of 36 elements. The event "the sum of the numbers on the two dice is 5" is the subset $A = \{(1,4),(2,3),(3,2),(4,1)\}$ of 4 elements.*

18.1.2 Probability

Given a sample space S, we define a probability for each event A of that space. This probability measures the likelihood that the outcome of the random phenomenon belongs A.

Definition 18.1.3 (Probability)**.** *A **probability** $\Pr : \mathcal{S} \to \mathbb{R}_{\geq 0}$ is a mapping from each **event** $A \subset S$ to a non-negative real number $\Pr[A] \geq 0$ such that the following properties are satisfied:*

1. $0 \leq \Pr[A] \leq 1$ *for any* $A \subset S$
2. $\Pr[S] = 1$
3. *For any countable collection $\{A_1, A_2, \ldots\}$ of events that are pairwise disjoint (i.e., $A_i \cap A_j = \emptyset$ for any $i \neq j$),*

$$\Pr\left[\bigcup_{i=1}^{\infty} A_i\right] = \sum_{i=1}^{\infty} \Pr[A_i]$$

When the sample space is finite or countably infinite, [1] *the properties above can be simplified into the following condition:*

$$\sum_{x \in S} \Pr[\{x\}] = 1$$

[1] A countably infinite set refers to a set whose elements can be numbered with indices. The set $\mathbb{N}$ of natural numbers or the set $\mathbb{Q}$ of rational numbers are examples of countably infinite sets.

Example 18.1.4. *Consider the sample space of "the outcome of tossing two dice" again. Assuming the two dice are fair, the probability of each outcome can be defined as* $1/36$*. Then the probability of the event "the sum of the numbers on the two dice is* 5*" is* $4/36$*.*

Example 18.1.5. *We are picking a point uniformly at random from the sample space* $[0, 2] \times [0, 2]$ *in the Cartesian coordinate system. The probability of the event that the point is drawn from the bottom left quarter* $[0, 1] \times [0, 1]$ *is* $1/4$*.*

18.1.3 Joint and Conditional Probability

In many cases, we are interested in not just one event, but multiple events, possibly happening in a sequence.

Definition 18.1.6 (Joint Probability)**.** *For any set of events* $\mathcal{A} = \{A_1, \ldots, A_n\}$ *of a sample space* S*, the* ***joint probability*** *of* $\mathcal{A}$ *is the probability* $\Pr[A_1 \cap \ldots \cap A_n]$ *of the intersection of all of the events. The probability* $\Pr[A_i]$ *of each of the events is also known as the* ***marginal probability****.*

Example 18.1.7. *Consider the sample space of "the outcome of tossing two dice" again. Let* A_1 *be the event "the number on the first die is* 1*" and let* A_2 *be the event "the number on the second die is* 4*." The joint probability of* A_1, A_2 *is* $1/36$*. The marginal probability of each of the events is* $1/6$*.*

It is also useful to define the probability of an event A, based on the knowledge that other events $A_1, \ldots, A_n$ have occurred.

Definition 18.1.8 (Conditional Probability)**.** *For any event* A *and any set of events* $\mathcal{A} = \{A_1, \ldots, A_n\}$ *of a sample space* S*, where* $\Pr[A_1 \cap \ldots \cap A_n] > 0$*, the* ***conditional probability of*** A ***given*** $\mathcal{A}$ *is*

$$\Pr[A \mid A_1, \ldots, A_n] = \frac{\Pr[A \cap A_1 \cap \ldots \cap A_n]}{\Pr[A_1 \cap \ldots \cap A_n]}$$

Example 18.1.9. *Consider the sample space of "the outcome of tossing two dice" again. Let* A_1 *be the event "the number on the first die is* 1*" and let* A_2 *be the event "the sum of the numbers on the two dice is* 5*." The conditional probability of* A_1 *given* A_2 *is* $1/4$*. The conditional probability of* A_2 *given* A_1 *is* $1/6$*.*

Using the definition of a conditional probability, we can define a formula to find the joint probability of a set $\mathcal{A}$ of events of a sample space.

Proposition 18.1.10 (Chain Rule for Conditional Probability)**.** *Given a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of events of a sample space S, where all appropriate conditional probabilities are defined, we have the following*

$$\begin{aligned}\Pr[A_1 \cap \ldots \cap A_n] &= \Pr[A_1 \mid A_2 \cap \ldots \cap A_n] \cdot \Pr[A_2 \cap \ldots \cap A_n] \\ &= \Pr[A_1 \mid A_2 \cap \ldots \cap A_n] \cdot \Pr[A_2 \mid A_3 \cap \ldots \cap A_n] \cdot \Pr[A_3 \cap \ldots \cap A_n] \\ &\vdots \\ &= \Pr[A_1 \mid A_2 \cap \ldots \cap A_n] \cdot \Pr[A_2 \mid A_3 \cap \ldots \cap A_n] \cdots \Pr[A_n]\end{aligned}$$

Finally, from the definition of a conditional probability, we see that

$$\Pr[B \mid A] \Pr[A] = \Pr[A \cap B] = \Pr[A \mid B] \Pr[B]$$

This shows that

$$\Pr[B \mid A] = \frac{\Pr[A \mid B] \Pr[B]}{\Pr[A]}$$

This is known as the *Bayes's Rule*.

18.1.4 Independent Events

Definition 18.1.11 (Independent Events)**.** *Two events A, B are **independent** if $\Pr[A], \Pr[B] > 0$ and*

$$\Pr[A] = \Pr[A \mid B]$$

or equivalently

$$\Pr[B] = \Pr[B \mid A]$$

or equivalently

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

Example 18.1.12. *Consider the sample space of "the outcome of tossing two dice" again. Let A_1 be the event "the number on the first die is 1" and let A_2 be the event "the number on the second die is 4." A_1 and A_2 are independent.*

Example 18.1.13. *Consider the sample space of "the outcome of tossing two dice" again. Let A_1 be the event "the number on the first die is 1" and let A_2 be the event "the sum of the numbers on the two dice is 5." A_1 and A_2 are not independent.*

18.2 Random Variable

In the previous section, we only learned how to assign a probability to an event, a subset of the sample space. But in general, we can assign a probability to a broader concept called a *random variable*, associated to the sample space.

Definition 18.2.1 (Random Variable)**.** *Given a sample space S, a mapping $X : S \to \mathbb{R}$ that maps each outcome $x \in S$ to a value $i \in \mathbb{R}$ is called a* ***random variable****.*

Example 18.2.2. *Consider the sample space of "the outcome of tossing two dice" again. Then the random variable $X =$ "sum of the numbers on the two dice" maps the outcome $(1, 4)$ to the value 5.*

Definition 18.2.3 (Sum and Product of Random Variables)**.** *If $X, X_1, \ldots, X_n$ are random variables defined on the same sample space S such that $X(x) = X_1(x) + \ldots + X_n(x)$ for every outcome $x \in S$, then we say that X is the* ***sum*** *of the random variables $X_1, \ldots, X_n$ and denote*

$$X = X_1 + \ldots + X_n$$

If $X(x) = X_1(x) \times \ldots \times X_n(x)$ for every outcome $x \in S$, then we say that X is the ***product*** *of the random variables $X_1, \ldots, X_n$ and denote*

$$X = X_1 \cdots X_n$$

Example 18.2.4. *Consider the sample space of "the outcome of tossing two dice" again. Then the random variable $X =$ "sum of the numbers on the two dice" is the sum of the two random variables $X_1 =$ "the number on the first die" and $X_2 =$ "the number on the second die."*

18.2.1 Probability of Random Variable

There is a natural relationship between the definition of an event and a random variable. Given a sample space S and random variable $X : S \to \mathbb{R}$, the "event that X takes a value in B" is denoted $\Pr[X \in B]$. It is the total probability of all outcomes $x \in S$ such that $X(x) \in B$. In particular, the event that X takes a particular value $i \in \mathbb{R}$ is denoted as $X = i$ and the event that X takes a value in the interval $[a, b]$ is denoted as $a \leq X \leq b$ and so on.

Example 18.2.5. *Consider the sample space of "the outcome of tossing two dice" and the random variable $X =$ "sum of the numbers on the two dice" again. Then*

$$\Pr[X = 5] = \Pr[\{(1, 4), (2, 3), (3, 2), (4, 1)\}] = 4/36$$

Often we are interested in the probability of the events of the form $X \leq x$. Plotting the values of $\Pr[X \leq x]$ with respect to x completely identifies the *distribution* of the values of X.

Definition 18.2.6 (Cumulative Distribution Function)**.** *Given a random variable X, there is an associated* ***cumulative distribution function (cdf)*** *$F_X : \mathbb{R} \to [0, 1]$ defined as*

$$F_X(x) = \Pr[X \leq x]$$

Proposition 18.2.7. *The following properties hold for a cumulative distribution function F_X:*

1. *F_X is increasing*
2. *$\lim_{x\to-\infty} F_X(x) = 0$ and $\lim_{x\to\infty} F_X(x) = 1$*

18.2.2 Discrete Random Variable

If the set of possible values of a random variable X is finite or countably infinite, we call it a *discrete random variable*. For a discrete random variable, the probability $\Pr[X = i]$ for each value i that the random variable can take completely identifies the *distribution* of X. In view of this fact, we denote the *probability mass function (pmf)* by

$$p_X(i) = \Pr[X = i]$$

Proposition 18.2.8. *The following properties hold for a probability mass function p_X:*

1. *$\sum_i p_X(i) = 1$*
2. *$F_X(x) = \sum_{i\leq x} p_X(i)$*

18.2.3 Continuous Random Variable

We now consider the case where the set of all possible values of a random variable X is an interval or a disjoint union of intervals in $\mathbb{R}$. We call such X a *continuous random variable*. In this case, the probability of the event $X = i$ is zero for any $i \in \mathbb{R}$. Instead, we care about the probability of the events of the form $a \leq X \leq b$.

Definition 18.2.9 (Probability Density Function)**.** *Given a continuous random variable X, there is an associated* ***probability density function (pdf)*** *$f_X : \mathbb{R} \to \mathbb{R}_{\geq 0}$ such that*

$$\Pr[a \leq X \leq b] = \int_a^b f(x)dx$$

for any $a, b \in \mathbb{R}$.

Proposition 18.2.10. *The following properties hold for a probability density function f_X:*

1. *$\int_{-\infty}^{\infty} f_X(x)dx = 1$*
2. *$F_X(x) = \int_{-\infty}^{x} f_X(y)dy$*

18.2.4 Expectation and Variance

Definition 18.2.11 (Expectation)**.** *The **expectation** or the **expected value** of a discrete random variable X is defined as*

$$\mathbb{E}[X] = \sum_i i \cdot p_X(i) = \sum_i i \cdot \Pr[X = i]$$

where p_X is its associated probability mass function. Similarly, the expectation for a continuous random variable X is defined as

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f_X(x) dx$$

where f_X is the associated probability density function. In either case, it is customary to denote the expected value of X as μ_X or just μ if there is no source of confusion.

Example 18.2.12. *Consider the sample space of "the outcome of tossing one die." Then the expected value of the random variable $X =$ "the number on the first die" can be computed as*

$$\mathbb{E}[X] = 1 \cdot \frac{6}{36} + 2 \cdot \frac{6}{36} + 3 \cdot \frac{6}{36} + 4 \cdot \frac{6}{36} + 5 \cdot \frac{6}{36} + 6 \cdot \frac{6}{36} = 3.5$$

Proposition 18.2.13 (Linearity of Expectation)**.** *If X is the sum of the random variables $X_1, \ldots, X_n$, then the following holds:*

$$\mathbb{E}[X] = \mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n]$$

Also, if $a, b \in \mathbb{R}$ and X is a random variable, then

$$\mathbb{E}[aX + b] = a\,\mathbb{E}[X] + b$$

Example 18.2.14. *Consider the sample space of "the outcome of tossing two dice." Then the expected value of the random variable $X =$ "the sum of the numbers of the two dice" can be computed as*

$$\mathbb{E}[X] = 3.5 + 3.5 = 7$$

since the expected value of the number on each die is 3.5*.*

Definition 18.2.15 (Variance)**.** *The **variance** of a random variable X, whose expected value is μ, is defined as*

$$Var[X] = \mathbb{E}[(X - \mu)^2]$$

*Its **standard deviation** is defined as*

$$\sigma_X = \sqrt{Var[X]}$$

It is customary to denote the variance of X as σ_X^2.

Proposition 18.2.16. *If $a \in \mathbb{R}$ and X is a random variable, then*

$$Var[aX] = a^2 Var[X] \quad \sigma_{aX} = |a|\,\sigma_X$$

Problem 18.2.17. *Prove **Chebyshev's inequality**:*

$$\Pr[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2}$$

for any $k > 0$. (Hint: Suppose the probability was greater than $1/k^2$. What could you conclude about $\mathbb{E}[(X-\mu)^2]$?)

18.2.5 Joint and Conditional Distribution of Random Variables

Just as in events, we are interested in multiple random variables defined on the sample space.

Definition 18.2.18 (Joint Distribution)**.** *If X, Y are discrete random variables defined on the same sample space S, the **joint probability mass function** $p_{X,Y}$ is defined as*

$$p_{X,Y}(i,j) = \Pr[X = i, Y = j]$$

where the event $X = i, Y = j$ refers to the intersection $(X = i) \cap (Y = j)$.

*If X, Y are continuous random variables defined on S, there is an associated **joint probability density function** $f_{X,Y} : \mathbb{R} \to \mathbb{R}_{\geq 0}$ such that*

$$\Pr[a \leq X \leq b, c \leq Y \leq d] = \int_c^d \int_a^b f_{X,Y}(x,y)dxdy$$

*The joint probability mass/density function defines the **joint distribution** of the two random variables.*

Definition 18.2.19 (Marginal Distribution)**.** *Given a joint distribution $p_{X,Y}$ or $f_{X,Y}$ of two random variables X, Y, the **marginal distribution** of X can be found as*

$$p_X(i) = \sum_j p_{X,Y}(i,j)$$

if X, Y are discrete and

$$f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x,y)dy$$

if continuous. We can equivalently define the marginal distribution of Y.

Definition 18.2.20 (Conditional Distribution)**.** *Given a joint distribution $p_{X,Y}$ or $f_{X,Y}$ of two random variables X, Y, we define the **conditional distribution of X given Y** as*

$$p_{X \mid Y}(i \mid j) = \frac{p_{X,Y}(i,j)}{p_Y(j)}$$

if X, Y are discrete and

$$f_{X \mid Y}(x \mid y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}$$

if continuous. We can equivalently define the marginal distribution of Y given X.

18.2.6 Bayes' Rule for Random Variables

Sometimes it is easy to calculate the conditional distribution of X given Y, but not the other way around. In this case, we can apply the *Bayes' Rule* to compute the conditional distribution of Y given X. Here, we assume that X, Y are discrete random variables. By a simple application of Bayes' Rule, we have

$$\Pr[Y = j \mid X = i] = \frac{\Pr[X = i \mid Y = j] \Pr[Y = j]}{\Pr[X = i]}$$

Now by the definition of a marginal distribution, we have

$$\Pr[X = i] = \sum_{j'} \Pr[X = i, Y = j] = \sum_{j'} \Pr[X = i \mid Y = j'] \Pr[Y = j']$$

for all possible values j' that Y can take. If we plug this into the denominator above,

$$\Pr[Y = j \mid X = i] = \frac{\Pr[X = i \mid Y = j] \Pr[Y = j]}{\sum_{j'} \Pr[X = i \mid Y = j'] \Pr[Y = j']}$$

Example 18.2.21. *There is a coin, where the probability of **Heads** is unknown and is denoted as θ. You are told that there is a 50% chance that it is a fair coin (i.e., $\theta = 0.5$) and 50% chance that it is biased to be $\theta = 0.7$. To find out if the coin is biased, you decide to flip the coin. Let D be the result of a coin flip. Then it is easy to calculate the conditional distribution of D given θ. For example,*

$$\Pr[D = H \mid \theta = 0.5] = 0.5$$

But we are more interested in the probability that the coin is fair/biased based on the observation of the coin flip. Therefore, we can apply the Bayes' Rule.

$$\Pr[\theta = 0.7 \mid D = H] = \frac{\Pr[D = H \mid \theta = 0.7] \Pr[\theta = 0.7]}{\Pr[D = H]}$$

which can be calculated as

$$\frac{\Pr[D = H \mid \theta = 0.7] \Pr[\theta = 0.7]}{\Pr[D = H \mid \theta = 0.7] \Pr[\theta = 0.7] + \Pr[D = H \mid \theta = 0.5] \Pr[\theta = 0.5]}$$
$$= \frac{0.7 \cdot 0.5}{0.7 \cdot 0.5 + 0.5 \cdot 0.5} \simeq 0.58$$

*So if we observe one **Heads**, there is a* 58% *chance that the coin was biased and a* 42% *chance that it was fair.*

Problem 18.2.22. *Consider Example 18.2.21 again. This time, we decide to throw the coin* 10 *times in a row. Let N be the number of observed **Heads**. What is the probability that the coin is biased if $N = 7$?*

18.2.7 Independent Random Variables

Analogous to events, we can define the independence of two random variables.

Definition 18.2.23 (Independent Random Variables)**.** *Two discrete random variables X, Y are **independent** if for every i, j, we have*

$$p_X(i) = p_{X \mid Y}(i \mid j)$$

or equivalently,

$$p_Y(j) = p_{Y \mid X}(j \mid i)$$

or equivalently

$$p_{X,Y}(x, y) = p_X(x) \cdot p_Y(y)$$

*Two continuous random variables X, Y are **independent** if the analogous conditions hold for the probability density functions.*

Definition 18.2.24 (Mutually Independent Random Variables)**.** *If any pair of n random variables $X_1, X_2, \ldots, X_n$ are independent of each other, then the random variables are **mutually independent**.*

Proposition 18.2.25. *If $X_1, \ldots, X_n$ are mutually independent random variables, the following properties are satisfied:*

1. $\mathbb{E}[X_1 \cdots X_n] = \mathbb{E}[X_1] \cdots \mathbb{E}[X_n]$
2. $Var[X_1 + \ldots + X_n] = Var(X_1) + \ldots + Var(X_n)$

We are particularly interested in independent random variables that have the same probability distribution. This is because if we repeat the same random process multiple times and define a random variable for each iteration, the random variables will be *independent and identically distributed*.

Definition 18.2.26. *If $X_1, \ldots, X_n$ are mutually independent random variables that have the same probability distribution, we call them **independent, identically distributed** random variables, which is more commonly denoted as **iid** or **i.i.d.** random variables.*

18.3 Central Limit Theorem and Confidence Intervals

Now we turn our attention to two very important topics in statistics: *Central Limit Theorem* and *confidence intervals*.

You may have seen *confidence intervals* or *margin of error* in the context of election polls. The pollster usually attaches a caveat to the prediction, saying that there is some probability that the true opinion of the public is $\pm\epsilon$ of the pollster's estimate, where ϵ is typically a few percent. This section is about the most basic form of confidence intervals, calculated using the famous Gaussian distribution. It also explains why the Gaussian pops up unexpectedly in so many settings.

A running example in this chapter is estimating the bias of a coin we have been given. Specifically, $\Pr[\textbf{Heads}] = p$ where p is unknown and may not be $1/2$. We wish to estimate p by repeatedly tossing the coin. If we toss the coin n times, we expect to see around np Heads. Confidence intervals ask the converse question: after having seen the number of heads in n tosses, how "confidently" can we estimate p?

18.3.1 Coin Tossing

Suppose we toss the same coin n times. For each $i = 1, 2, \ldots, n$, define the random variable X_i as an *indicator random variable* such that

$$X_i = \begin{cases} 1 & i\text{-th toss was } \textbf{Heads} \\ 0 & \textit{otherwise} \end{cases}$$

It is easily checked that $X_1, \ldots, X_n$ are iid random variables, each with $\mathbb{E}[X_i] = p$ and $Var[X_i] = p(1-p)$. Also if we have another random variable $X =$ "number of heads," notice that X is the sum of $X_1, \ldots, X_n$. Therefore, $\mathbb{E}[X] = np$ and $Var[X] = np(1-p)$.

Problem 18.3.1. *Show that if* $\Pr[\textbf{Heads}] = p$ *then* $\mathbb{E}[X] = np$ *and* $Var[X] = np(1-p)$. *(Hint: use linearity of expectation and the fact that* X_i*'s are mutually independent.)*

Suppose $p = 0.8$. What is the distribution of X? Figure 18.1 gives the distribution of X for different n's.

Let's make some observations about Figure 18.1.

Expected value may not happen too often. For $n = 10$, the expected number of Heads is 8, but that is seen only with probability 0.3. In other words, with probability 0.7, the number of Heads is different from the expectation. [2]

The highly likely values fall in a smaller and smaller band around the expected value, as n increases.

For $n = 10$, there is a good chance that the number of Heads is

[2] In such cases, *expectation* can be a misleading term. It may in fact be *never* seen. For instance, the expected number of eyes in an individual drawn from the human population is somewhere between 1 and 2 but no individual has a non-integral number of eyes. Thus *mean value* is a more intuitive term.

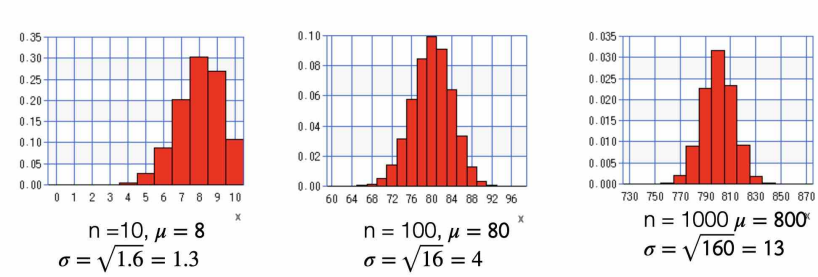


Figure 18.1: Distribution of X when we toss a coin n times, and $p = 0.8$. The plots were generated using a calculator.

quite far from the the expectation. For $n = 100$, the number of Heads lies in $[68, 90]$ with quite high probability. For $n = 1000$ it lies in $[770, 830]$ with high probability.

The probability curve becomes more symmetrical around the mean. Contrast between the case where $n = 10$ and the case where $n = 100$.

Probability curve starts resembling the famous Gaussian distribution . Also called *Normal Distribution* and in popular math, the *Bell curve*, due to its bell-like shape.

18.3.2 Gaussian Distribution

We say that a real-valued random variable X is distributed according to $\mathcal{N}(\mu, \sigma^2)$, the Gaussian distribution with mean μ and variance σ^2, if

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{18.1}$$

It is hard to make an intuitive sense of this expression. The following figure gives us a better handle.

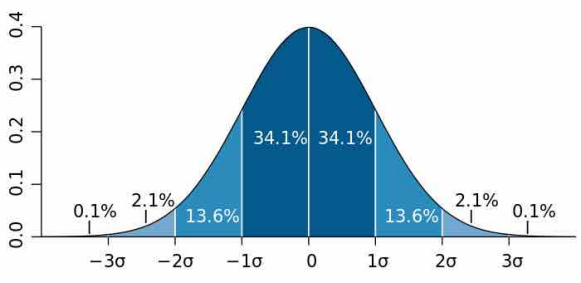


Figure 18.2: Cheatsheet for the Gaussian distribution with mean μ and variance σ^2. It is tightly concentrated in the interval $[\mu - k\sigma, \mu + k\sigma]$ for even $k = 1$ and certainly for $k = 2, 3$. *Source:* https://en.wikipedia.org/wiki/Normal_distribution

Figure 18.2 shows that X concentrates very strongly around the mean μ. The probability that X lies in various intervals around μ of the type $[\mu - k\sigma, \mu + k\sigma]$ are as follows: (i) For $k = 1$ it is 68.2%; (ii) For $k = 2$ it is 95.4%; (iii) For $k = 3$ it is 99.6%.

18.3.3 Central Limit Theorem (CLT)

This fundamental result explains our observations in Subsection 18.3.1.

Theorem 18.3.2 (Central Limit Theorem, informal statement)**.** *Suppose $X_1, X_2, \ldots,$ is a sequence of random variables that are mutually independent and each of whose variance is upper bounded by some constant C. Then as $n \to \infty$, the sum $X_1 + X_2 + \ldots + X_n$ tends to $\mathcal{N}(\mu, \sigma^2)$ where $\mu = \sum_i \mathbb{E}[X_i]$ and $\sigma^2 = \sum_i Var(X_i)$.*

We won't prove this theorem. We will use it primarily via the "cheatsheet" of Figure 18.2.

18.3.4 Confidence Intervals

We return to the problem of estimating the bias of a coin, namely $p = \Pr[\textbf{Heads}]$. Suppose we toss it n times and observe X heads. Then $X = \sum_i X_i$ where X_i is the indicator random variable that signifies if i-th toss is Heads.

Since the X_i's are mutually independent, we can apply the CLT and conclude that X will approximately follow a Gaussian distribution as n grows. This is clear from Figure 18.1, where the probability histogram (which is a discrete approximation to the probability density) looks quite Gaussian-like for $n = 1000$. In this course we will assume for simplicity that CLT applies exactly. Using the mean and variance calculations from Problem 18.3.1, X is distributed like $\mathcal{N}(\mu, \sigma^2)$ where $\mu = np$, $\sigma^2 = np(1-p)$. Using the cheatsheet of Figure 18.2, we can conclude that

$$\Pr[X \notin [np - 2\sigma, np + 2\sigma]] \leq 4.6\%$$

Since $X \in [np - 2\sigma, np + 2\sigma]$ if and only if $np \in [X - 2\sigma, X + 2\sigma]$, some students have the following misconception:

> *Given the observation of X heads in n coin tosses, the probability that $np \notin [X - 2\sigma, X + 2\sigma]$ is at most 4.6%.*

But there is no *a priori distribution* on p. It is simply some (unknown) *constant* of nature that we're trying to estimate. So the correct inference should be:

> *If $np \notin [X - 2\sigma, X + 2\sigma]$, then the probability (over the n coin tosses) that we would have seen X heads is at most 4.6%.*

The above is an example of confidence bounds. Of course, you may note that σ also depends on p, so the above conclusion doesn't give us a clean confidence interval. In this course we use a simplifying assumption: to do the calculation we estimate σ^2 as $np'(1-p')$ where $p' = X/n$. (The intuitive justification is that we expect p to be close to X/n.)

Example 18.3.3. *Suppose $X = 0.8n$. Using our simplified calculation, $\sigma^2 \approx n(0.8)(0.2)$, implying $\sigma = 0.4\sqrt{n}$. Thus we conclude that if $p \notin [0.8 - 0.4/\sqrt{n}, 0.8 + 0.4/\sqrt{n}]$, then the probability of observing this many Heads in n tosses would have been less than $100 - 68.2\%$, that is, less than 31.8%.*

The concept of confidence intervals is also relevant to ML models.

Example 18.3.4. *A deep neural network model was trained to predict cancer patients' chances of staying in remission a year after chemotherapy, and we are interested in finding out its accuracy p. When the model is tested on $n = 1000$ held-out data points, this problem is equivalent to the coin flipping problem. For each of the held-out data point, the probability that the model makes the correct prediction is p. By observing the number of correct predictions on the held-out data, we can construct a confidence interval for p. Say the test accuracy was $p' = 70\%$. Then the 68% confidence interval can be written as*

$$np \in [np' - \sigma, np' + \sigma]$$

Substituting $p' = 0.7, \sigma \approx \sqrt{np'(1-p')}, n = 1000$, we get

$$1000p \in [685.5, 714.5]$$

or equivalently,

$$p \in [0.6855, 0.7145]$$

18.3.5 Confidence Intervals for Vectors

In the above settings, sampling was being used to estimate a real number, namely, $\Pr[\textbf{Heads}]$ for a coin. How about estimating a vector? For instance, in an opinion poll, respondents are being asked for opinions on multiple questions. Similarly, in *stochastic gradient descent* (Chapter 3), the gradient vector is being estimated by sampling a small number of data points. How can we develop confidence bounds for estimating a vector in $\mathbb{R}^k$ from n samples?

The confidence intervals for the coin toss setting can be easily extended to this case using the so called *Union Bound*:

$$\Pr[A_1 \cup A_2 \cup \cdots \cup A_k] \leq Pr[A_1] + \Pr[A_2] + \cdots + \Pr[A_k] \qquad (18.2)$$

This leads to the simplest confidence bound for estimating a vector in $\mathbb{R}^k$. Suppose the probability of the estimate being off by δ_i in the i-th coordinate is at most q_i. Then

$$\Pr[\text{estimate is off by } \vec{\delta}] \leq q_1 + q_2 + \cdots + q_k$$

where $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_k)$

18.4 Final Remarks

The CLT applies to many settings, but it doesn't apply everywhere. It is useful to clear up a couple of frequent misconceptions that students have:

1. Not every distribution involving a large number of samples is Gaussian. For example, scores on the final exam are usually not distributed like a Gaussian. Similarly, human heights are not really distributed like Gaussians.

2. Not everything that looks Gaussian-like is a result of the Central Limit Theorem. For instance, we saw that the distribution of weights in the sentiment model in Chapter 1 looked vaguely Gaussian-like, but they are not the sum of independent random variables as far as we can tell.

19
Calculus

19.1 Calculus in One Variable

In this section, we briefly review calculus in one variable.

19.1.1 Exponential and Logarithmic Functions

When we multiply the same number a by n times, we denote it as a^n. The *exponential function* is a natural extension of this concept.

Definition 19.1.1 (Exponential Function)**.** *There is a unique function $f : \mathbb{R} \to \mathbb{R}$ such that $f(n) = e^n$ for any $n \in \mathbb{N}$ and $f(x+y) = f(x)f(y)$ for any $x, y \in \mathbb{R}$. This function is called the **exponential function** and is denoted as e^x or $\exp(x)$.*

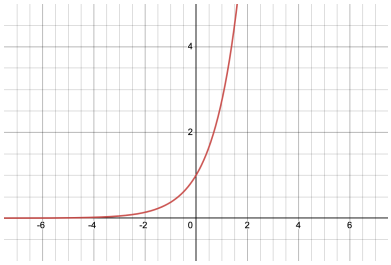


Figure 19.1: The graph of the exponential function.

Proposition 19.1.2. *The following properties hold for the exponential function:*

1. $\exp(x) > 0$ *for any* $x \in \mathbb{R}$
2. $\exp(x)$ *is increasing*
3. $\lim_{x \to -\infty} \exp(x) = 0$
4. $\lim_{x \to \infty} \exp(x) = \infty$
5. $\exp(-x) = \frac{1}{\exp(x)}$

We are also interested in the inverse function of the exponential function.

Definition 19.1.3 (Logarithmic Function)**.** *The **logarithmic function** $\log : (0, \infty) \to \mathbb{R}$ is defined as the inverse function of the exponential function. That is, $\log(x) = y$ where $x = e^y$.*

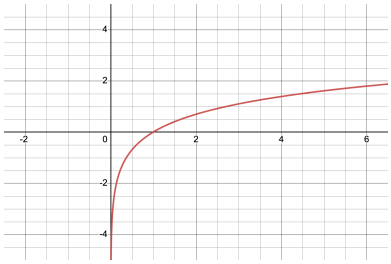


Figure 19.2: The graph of the logarithmic function.

Proposition 19.1.4. *The following properties hold for the logarithmic function:*

1. $\log(x)$ *is increasing*
2. $\lim_{x \to 0+} \log(x) = -\infty$
3. $\lim_{x \to \infty} \log(x) = \infty$
4. $\log(xy) = \log(x) + \log(y)$

19.1.2 Sigmoid Function

In Machine Learning, a slight variant of the exponential function, known as the *sigmoid function* is widely used.

Definition 19.1.5 (Sigmoid Function)**.** *The* ***sigmoid function*** *denoted as* $\sigma : \mathbb{R} \to \mathbb{R}$ *is defined as*

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

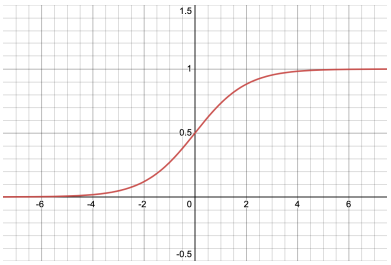


Figure 19.3: The graph of the sigmoid function.

Proposition 19.1.6. *The following properties hold for the sigmoid function:*

1. $0 < \sigma(x) < 1$ *for any* $x \in \mathbb{R}$
2. $\sigma(x)$ *is increasing*
3. $\lim_{x \to -\infty} \sigma(x) = 0$
4. $\lim_{x \to \infty} \sigma(x) = 1$
5. *The graph of* σ *is symmetrical to the point* $\left(0, \frac{1}{2}\right)$*. In particular,*

$$\sigma(x) + \sigma(-x) = 1$$

Because of the last property in Proposition 19.1.6, the sigmoid function is well suited for binary classification (*e.g.*, in logistic regression in Chapter 1). Given some output value x of a classification model, we interpret it as the measure of confidence that the input is of label 1, where we implicitly assume that the measure of confidence that the input is of label 2 is $-x$. Then we apply the sigmoid function to translate this into a probability distribution over the two labels.

19.1.3 Differentiation

Definition 19.1.7 (Derivative)**.** *Given a function* $f : \mathbb{R} \to \mathbb{R}$*, its* ***derivative*** f' *is defined as*

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

We alternatively denote $f'(x)$ *as* $\frac{d}{dx} f(x)$.

Example 19.1.8. *The derivative of the exponential function is itself:*

$$\exp'(x) = \exp(x)$$

and the derivative of the logarithmic function is:

$$\log'(x) = \frac{1}{x}$$

In general, there are more than two variables, that are related to each other through a composite function. The *chain rule* helps us find the derivative of the composite function.

Definition 19.1.9 (Chain Rule)**.** *If there are functions $f, g : \mathbb{R} \to \mathbb{R}$ such that $y = f(x)$ and $z = g(y)$, then*

$$(g \circ f)'(x) = g'(f(x))f'(x) = \frac{d}{dy}g(f(x)) \cdot \frac{d}{dx}f(x)$$

or equivalently

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

19.2 Multivariable Calculus

In this section, we introduce the basics of multivariable calculus, which is widely used in Machine Learning. Since this is a generalization of the calculus in one variable, it will be useful to pay close attention to the similarity with the results from the previous section.

19.2.1 Mappings of Several Variables

So far, we only considered functions of the form $f : \mathbb{R} \to \mathbb{R}$ that map a real value x to a real value y. But now we are interested in mappings $f : \mathbb{R}^n \to \mathbb{R}^m$ that map a vector $\vec{\mathbf{x}} = (x_1, \ldots, x_n)$ with n coordinates to a vector $\vec{\mathbf{y}} = (y_1, \ldots, y_m)$ with m coordinates. In general, a *function* is a special case of a *mapping* where the range is $\mathbb{R}$. If the mappings are of the form $f : \mathbb{R}^n \to \mathbb{R}$ (*i.e.*, $m = 1$), it can still be called a *function* of several variables.

First consider an example where $m = 1$.

Example 19.2.1. *Let $f(x_1, x_2) = x_1^2 + x_2^2$ be a function in two variables. This can be understood as mapping a point $\vec{\mathbf{x}} = (x_1, x_2)$ in the Cartesian coordinate system to its squared distance from the origin. For example, $f(3, 4) = 25$ shows that the point the squared distance between $(3, 4)$ and the origin $(0, 0)$ is 25.*

When $m > 1$, we notice that each coordinate $y_1, \ldots, y_m$ is a function of $x_1, \ldots, x_n$. Therefore, we can decompose f into m functions $f_1, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$ such that

$$f(\vec{\mathbf{x}}) = (f_1(\vec{\mathbf{x}}), \ldots, f_m(\vec{\mathbf{x}}))$$

Example 19.2.2. *Let $f(x_1, x_2) = (x_1^2 x_2, x_1 x_2^2)$ be a mapping from $\mathbb{R}^2$ to $\mathbb{R}^2$. Then we can decompose f into two functions f_1, f_2 in two variables where*

$$f_1(x_1, x_2) = x_1^2 x_2$$
$$f_2(x_1, x_2) = x_1 x_2^2$$

19.2.2 Softmax Function

The *softmax function* is a multivariable function widely used in Machine Learning, especially for multi-class classification (see Chapter 4, Chapter 10). It takes in a vector of k values, each corresponding to a particular class, and outputs a probability distribution over the k classes — that is, a vector of k non-negative values that sum up to 1. The resulting probability is *exponentially proportional* to the input value of that class. We formally write this as:

Definition 19.2.3 (Softmax Function)**.** *Given a vector $\vec{\mathbf{z}} = (z_1, z_2, \ldots, z_k) \in \mathbb{R}^k$, we define the **softmax function** as a probability function $softmax : \mathbb{R}^k \to [0,1]^k$ where the "probability of predicting class i" is:*

$$softmax(\vec{\mathbf{z}})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{19.1}$$

Problem 19.2.4. *Show that for $k = 2$, the definition of the softmax function is equivalent to the sigmoid function (after slight rearrangement/renaming of terms).*

The sigmoid function is used for binary classification, where it takes in a single real value and converts it to a probability of one class (and the probability of the other class can be inferred as its complement). The softmax function is used for multi-class classification, where it takes in k real values and converts them to k probabilities, one for each class.

19.2.3 Differentiation

Just like with functions in one variable, we can define differentiation for mappings in several variables. The key point is that now we will define a *partial derivative* for each pair (x_i, y_j) of coordinate x_i of the domain and coordinate y_j of the range.

Definition 19.2.5 (Partial Derivative)**.** *Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the **partial derivative of y_j with respect to x_i at the point $\vec{\mathbf{x}}$** is defined as*

$$\left.\frac{\partial y_j}{\partial x_i}\right|_{\vec{\mathbf{x}}} = \lim_{h \to 0} \frac{f_j(x_1, \ldots, x_j + h, \ldots, x_n) - f_j(x_1, \ldots, x_j, \ldots, x_n)}{h}$$

Definition 19.2.6 (Gradient). *If $f : \mathbb{R}^n \to \mathbb{R}$ is a function of several variables, the* gradient of f *is defined as a mapping $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ that maps each input vector to the vector of partial derivatives at that point:*

$$\nabla f(\vec{\mathbf{x}}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right) \Bigg|_{\vec{\mathbf{x}}}$$

Similarly to the chain rule in one variable, we can define a chain rule for multivariable settings. The key point is that there are multiple ways that a coordinate x_j can affect the value of z_i. Definition 19.2.7 can be thought as applying the chain rule for one variable in each of the paths, and adding up the results.

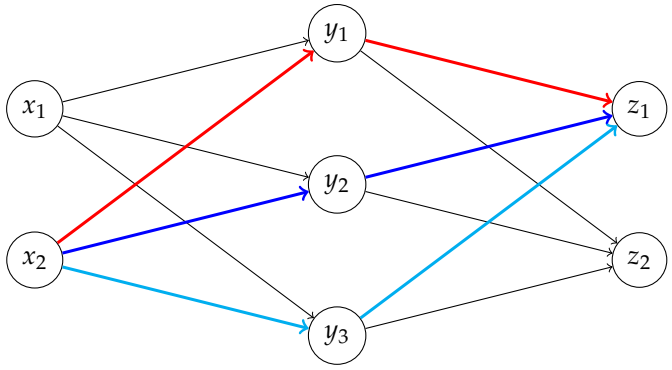


Figure 19.4: A visualization of the chain rule in multivariable settings. Notice that x_2 can affect the value of z_1 in three different paths. The amount of effect from each path will respectively be calculated as $(\partial z_1/\partial y_1)(\partial y_1/\partial x_2)$ (red), $(\partial z_1/\partial y_2)(\partial y_2/\partial x_2)$ (blue), and $(\partial z_1/\partial y_3)(\partial y_3/\partial x_2)$ (cyan).

Definition 19.2.7 (Chain Rule). *If $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^m \to \mathbb{R}^\ell$ are mappings of several variables, where $\vec{\mathbf{y}} = f(\vec{\mathbf{x}})$ and $\vec{\mathbf{z}} = g(\vec{\mathbf{y}})$, the following* ***chain rule*** *holds for each $1 \le i \le \ell$ and $1 \le j \le n$:*

$$\frac{\partial z_i}{\partial x_j} = \sum_{k=1}^{m} \frac{\partial z_i}{\partial y_k} \cdot \frac{\partial y_k}{\partial x_j}$$

Example 19.2.8. *Suppose we define the functions $h = s + t^2$, $s = 3x$, and $t = x - 2$. Then, we can find the partial derivative $\frac{\partial h}{\partial x}$ using the chain rule:*

$$\begin{aligned} \frac{\partial h}{\partial x} &= \frac{\partial s}{\partial x} + \frac{\partial (t^2)}{\partial x} \\ &= \frac{\partial s}{\partial x} + \frac{\partial (t^2)}{\partial t} \cdot \frac{\partial t}{\partial x} \\ &= 3 + 2t \cdot 1 \\ &= 2x - 1 \end{aligned}$$

Problem 19.2.9. *Suppose we define the functions $h = s + t^2$, $s = xy$, and $t = x - 2y$. Compute the partial derivative $\partial h / \partial x$.*

20
Linear Algebra

20.1 Vectors

Vectors are a collection of entries (here, we focus only on real numbers). For example, the pair $(1,2)$ is a real vector of size 2, and the 3-tuple $(1,0,2)$ is a real vector of size 3. We primarily categorize vectors by their size. For example, the set of all real vectors of size n is denoted as $\mathbb{R}^n$. Any element of $\mathbb{R}^n$ can be thought of as representing a point (or equivalently, the direction from the origin to the point) in the n-dimensional Cartesian space. A real number in $\mathbb{R}$ is also known as a *scalar*, as opposed to *vectors* in $\mathbb{R}^n$ where $n > 1$.

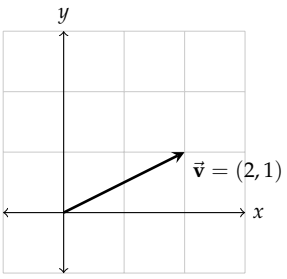


Figure 20.1: A visualization of a vector $\vec{\mathbf{v}} = (2,1)$ in $\mathbb{R}^2$.

20.1.1 Vector Space

We are interested in two operations defined on vectors — vector addition and scalar multiplication. Given vectors $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_n)$ and $\vec{\mathbf{y}} = (y_1, y_2, \ldots, y_n)$ and a scalar $c \in \mathbb{R}$, the *vector addition* is defined as

$$\vec{\mathbf{x}} + \vec{\mathbf{y}} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \in \mathbb{R}^n$$

where we add each of the coordinates element-wise. As shown in Figure 20.2, vector addition is the process of finding the diagonal of the parallelogram made by the two vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$. The *scalar multiplication* is similarly defined as

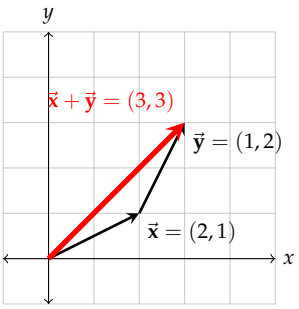


Figure 20.2: A visualization of $\vec{\mathbf{x}} + \vec{\mathbf{y}}$ where $\vec{\mathbf{x}} = (2,1)$ and $\vec{\mathbf{y}} = (1,2)$.

$$c\vec{\mathbf{x}} = (cx_1, cx_2, \ldots, cx_n) \in \mathbb{R}^n$$

As shown in Figure 20.3, scalar multiplication is the process of scaling one vector up or down.

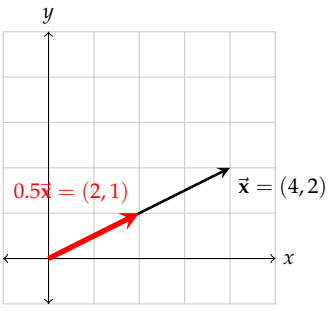


Figure 20.3: A visualization of $0.5\vec{\mathbf{x}}$ where $\vec{\mathbf{x}} = (4,2)$.

$\mathbb{R}^n$ is closed under these two operations — *i.e.*, the resulting vector of either operation is still in $\mathbb{R}^n$. Any subset S of $\mathbb{R}^n$ that is closed under vector addition and scalar multiplication is known as a *subspace* of R^n.

20.1.2 Inner Product

The *inner product* is defined as

$$\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n = \sum_{i=1}^{n} x_i y_i \in \mathbb{R}$$

Closely related to the inner product is the *norm* of a vector, which measures the *length* of it. It is defined as $\|\vec{\mathbf{x}}\| = \sqrt{\vec{\mathbf{x}} \cdot \vec{\mathbf{x}}}$. [1]

[1] There are many other definitions of a norm. This particular one is called an ℓ_2 norm.

Proposition 20.1.1. *The inner product satisfies the following properties:*

- *Symmetry:* $\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \vec{\mathbf{y}} \cdot \vec{\mathbf{x}}$
- *Linearity:* $(a_1 \vec{\mathbf{x}}_1 + a_2 \vec{\mathbf{x}}_2) \cdot \vec{\mathbf{y}} = a_1 (\vec{\mathbf{x}}_1 \cdot \vec{\mathbf{y}}) + a_2 (\vec{\mathbf{x}}_2 \cdot \vec{\mathbf{y}})$

and the norm satisfies the following property:

- *Absolute Homogeneity:* $\|a\vec{\mathbf{x}}\| = |a| \, \|\vec{\mathbf{x}}\|$

20.1.3 Linear Independence

Any vector of the form

$$a_1 \vec{\mathbf{x}}_1 + a_2 \vec{\mathbf{x}}_2 + \ldots + a_k \vec{\mathbf{x}}_k$$

where a_i's are scalars and $\vec{\mathbf{x}}_i$'s are vectors is called a *linear combination* of the vectors $\vec{\mathbf{x}}_i$'s. Notice that the zero vector $\vec{\mathbf{0}}$ (*i.e.*, the vector with all zero entries) can always be represented as a linear combination of an arbitrary collection of vectors, if all a_i's are chosen as zero. This is known as a *trivial linear combination*, and any other choice of a_i's is known as a *non-trivial linear combination*.

Definition 20.1.2. *k vectors $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \ldots, \vec{\mathbf{x}}_k \in \mathbb{R}^n$ are called **linearly dependent** if $\vec{\mathbf{0}}$ can be represented as a non-trivial linear combination of the vectors $\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_k$; or equivalently, if one of the vectors can be represented as a linear combination of the remaining $k-1$ vectors. The vectors that are not linearly dependent with each other are called **linearly independent**.*

Consider the following analogy. Imagine trying to have a family style dinner at a fast food restaurant, where the first person orders a burger, the second person orders a chilli cheese fries, and the third person orders a set menu with a burger and a chili cheese fries. The third person's order did not contribute to the diversity of the food on the dinner table. Similarly, if some set of vectors are linearly dependent, it means that at least one of the vectors is redundant.

Example 20.1.3. *The set $\{(-1,2), (3,0), (1,4)\}$ of three vectors is linearly dependent because*

$$(1,4) = 2 \cdot (-1,2) + (3,0)$$

can be represented as the linear combination of the remaining two vectors.

Example 20.1.4. *The set* $\{(-1,2,1),(3,0,0),(1,4,1)\}$ *of three vectors is linearly independent because there is no way to write one vector as a linear combination of the remaining two vectors.*

20.1.4 Span

Definition 20.1.5. *The* ***span*** *of a set of vectors* $\vec{\mathbf{x}}_1,\ldots,\vec{\mathbf{x}}_k$ *is the set of all vectors that can be represented as a linear combination of* $\vec{\mathbf{x}}_i$*'s.*

Example 20.1.6. $(1,4)$ *is in the span of* $\{(-1,2),(3,0)\}$ *because*

$$(1,4) = 2\cdot(-1,2)+(3,0)$$

Example 20.1.7. $(1,4,1)$ *is not in the span of* $\{(-1,2,1),(3,0,0)\}$ *because there is no way to choose* $a_1,a_2\in\mathbb{R}$ *such that*

$$(1,4,1) = a_1(-1,2,1)+a_2(3,0,0)$$

The span is also known as the *subspace generated by the vectors* $\vec{\mathbf{x}}_1,\ldots,\vec{\mathbf{x}}_k$. This is because if you add any two vectors in the span, or multiply one by a scalar, it is still in the span (*i.e.*, the span is closed under vector addition and scalar multiplication).

Example 20.1.8. *In the* $\mathbb{R}^3$*, the two vectors* $(1,0,0)$ *and* $(0,1,0)$ *span the 2-dimensional XY-plane. Similarly, the vectors* $(1,0,1)$ *and* $(0,2,1)$ *span the 2-dimensional plane* $2x+y-2z=0$*.* [2]

[2] The term *dimension* will be formally defined soon. Here, we rely on your intuition.

In Example 20.1.8, we see examples where 2 vectors span a 2-dimensional subspace. In general, the dimension of the subspace spanned by k vectors can go up to k, but it can also be strictly smaller than k. This is related to the *linear independence* of the vectors.

Proposition 20.1.9. *Given* k *vectors,* $\vec{\mathbf{x}}_1,\ldots,\vec{\mathbf{x}}_k\in\mathbb{R}^n$*, there is a maximum number* $d\geq 1$ *such that there is some subcollection* $\vec{\mathbf{x}}_{i_1},\ldots,\vec{\mathbf{x}}_{i_d}$ *of these vectors that are linearly independent. Then*

$$span(\vec{\mathbf{x}}_1,\ldots,\vec{\mathbf{x}}_k) = span(\vec{\mathbf{x}}_{i_1},\ldots,\vec{\mathbf{x}}_{i_d}) \tag{20.1}$$

is a d*-dimensional subspace of* $\mathbb{R}^n$*.*

Conversely, if we know that the span of the k *vectors is a* d*-dimensional subspace, then the maximum number of vectors that are linearly independent with each other is* d*, and any subcollection of linearly independent* d *vectors satisfies* (20.1)*.*

Proposition 20.1.9 states that the span of some set of k vectors is equivalent to the maximum number d of linearly independent vectors. It also states that the span of the k vectors is equal to the span of the linearly independent d vectors, meaning all of the information

is captured by the d vectors; the remaining $k - d$ vectors are just redundancies. But trying to directly compute the maximum number of linearly independent vectors is inefficient — it may require checking the linear independence of an exponential number of subsets of the vectors. In the next section, we discuss a concept called *matrix rank* that is very closely related to this topic.

20.1.5 Orthogonal Vectors

Definition 20.1.10. *If vectors $\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_k \in \mathbb{R}^n$ satisfy $\vec{\mathbf{x}}_i \cdot \vec{\mathbf{x}}_j = 0$ for any $i \neq j$, then they are called **orthogonal** vectors. In particular, if they also satisfy the condition that $\|\vec{\mathbf{x}}_i\| = 1$ for each i, then they are also **orthonormal**.*

In $\mathbb{R}^n$, orthogonal vectors form a 90 degrees angle with each other.

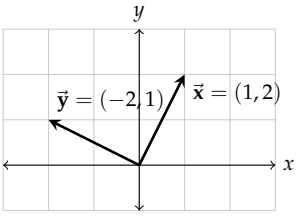


Figure 20.4: A visualization of orthogonal vectors $\vec{\mathbf{x}} = (1, 2)$ and $\vec{\mathbf{y}} = (-2, 1)$.

Example 20.1.11. *The two vectors $(1, 0), (0, 2)$ are orthogonal. So are the vectors $(1, 2), (-2, 1)$.*

Given any set of orthogonal vectors, it is possible to transform it into a set of orthonormal vectors, by normalizing each vector (*i.e.*, scale it such that the norm is 1).

20.1.6 Basis

Definition 20.1.12. *A collection $\{\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_k\}$ of linearly independent vectors in $\mathbb{R}^n$ that span a set S is known as a **basis** of S. In particular, if the vectors of the basis are orthogonal/orthonormal, the basis is called an **orthogonal/orthonormal basis** of S.*

The set S in Definition 20.1.12 can be the entire vector space $\mathbb{R}^n$, but it can also be some subspace of $\mathbb{R}^n$ with a lower dimension.

Example 20.1.13. *The set $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ of three vectors is a basis for $\mathbb{R}^3$. When we exclude the last vector $(0, 0, 1)$, the set $\{(1, 0, 0), (0, 1, 0)\}$ is a basis of the 2-dimensional XY-plane in $\mathbb{R}^3$.*

Given some subspace S, the basis of S is not unique. However, every basis of S must have the same size — this size is called the *dimension* of S. For a finite dimensional space S, it is known that there exists an *orthogonal* basis of S. There is a well-known algorithm — Gram-Schmidt process — that can transform an arbitrary basis into an orthogonal basis (and eventually an orthonormal basis via normalization).

20.1.7 Projection

Vector projection is the key concept used in the Gram-Schmidt process that computes an orthogonal basis. Given a fixed vector $\vec{\mathbf{a}}$, it decom-

poses any given vector $\vec{\mathbf{x}}$ into a sum of two components — one that is orthogonal to $\vec{\mathbf{a}}$ ("distinct information") and the other that is parallel to $\vec{\mathbf{a}}$ ("redundant information").

Definition 20.1.14 (Vector Projection). *Fix a vector $\vec{\mathbf{a}} \in \mathbb{R}^n$. Given another vector $\vec{\mathbf{x}}$, the **projection of $\vec{\mathbf{x}}$ on $\vec{\mathbf{a}}$** is defined as*

$$proj_{\vec{\mathbf{a}}}(\vec{\mathbf{x}}) = \frac{\vec{\mathbf{x}} \cdot \vec{\mathbf{a}}}{\vec{\mathbf{a}} \cdot \vec{\mathbf{a}}} \vec{\mathbf{a}}$$

and is parallel to the fixed vector $\vec{\mathbf{a}}$. The remaining component

$$\vec{\mathbf{x}} - proj_{\vec{\mathbf{a}}}(\vec{\mathbf{x}})$$

*is called the **rejection of $\vec{\mathbf{x}}$ from $\vec{\mathbf{a}}$** and is orthogonal to $\vec{\mathbf{a}}$.*

Proposition 20.1.15 (Pythagorean Theorem). *If $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ are orthogonal, then*

$$\|\vec{\mathbf{x}} + \vec{\mathbf{y}}\|^2 = \|\vec{\mathbf{x}}\|^2 + \|\vec{\mathbf{y}}\|^2$$

In particular, given two vectors $\vec{\mathbf{a}}, \vec{\mathbf{x}}$, we have

$$\|\vec{\mathbf{x}} - proj_{\vec{\mathbf{a}}}(\vec{\mathbf{x}})\|^2 = \|\vec{\mathbf{x}}\|^2 - \|proj_{\vec{\mathbf{a}}}(\vec{\mathbf{x}})\|^2$$

Now assume we are given a space S and a subspace $T \subset S$. Then a vector $\vec{\mathbf{x}} \in S$ in the larger space does not necessarily belong in T. Instead, we can find a vector $\vec{\mathbf{x}}' \in T$ that is "closest" to $\vec{\mathbf{x}}$ using vector projection. [3]

[3] We ask you to prove this in Problem 7.1.3.

Definition 20.1.16 (Vector Projection on Subspace). *Given a space S, its subspace T with an orthogonal basis $\{\vec{\mathbf{t}}_1, \ldots, \vec{\mathbf{t}}_k\}$, and a vector $\vec{\mathbf{x}} \in S$, the **projection of $\vec{\mathbf{x}}$ on** T is defined as*

$$proj_T(\vec{\mathbf{x}}) = \sum_{i=1}^{k} proj_{\vec{\mathbf{t}}_i}(\vec{\mathbf{x}}) = \sum_{i=1}^{k} \frac{\vec{\mathbf{x}} \cdot \vec{\mathbf{t}}_i}{\vec{\mathbf{t}}_i \cdot \vec{\mathbf{t}}_i} \vec{\mathbf{t}}_i$$

the sum of projection of $\vec{\mathbf{x}}$ on each of the basis vectors of T.

20.2 Matrices

Matrices are a generalization of *vectors* in 2-dimension — a $m \times n$ matrix is a collection of numbers assembled in a rectangular shape of m rows and n columns. The set of all real matrices of size $m \times n$ is denoted as $\mathbb{R}^{m \times n}$. A vector of size n is customarily understood as a column vector — that is, a $n \times 1$ matrix. Also, if $m = n$, then the matrix is known as a *square matrix*.

20.2.1 Matrix Operation

Similarly to vector operations, we are interested in four matrix operations — matrix addition, scalar multiplication, matrix multiplication, and transpose. Given a scalar $c \in \mathbb{R}$ and matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times n}$ such that

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix} \quad \textit{and} \quad \mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,n} \end{bmatrix}$$

the matrix addition is defined as

$$\mathbf{X} + \mathbf{Y} = \begin{bmatrix} x_{1,1} + y_{1,1} & x_{1,2} + y_{1,2} & \cdots & x_{1,n} + y_{1,n} \\ x_{2,1} + y_{2,1} & x_{2,2} + y_{2,2} & \cdots & x_{2,n} + y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} + y_{m,1} & x_{m,2} + y_{m,2} & \cdots & x_{m,n} + y_{m,n} \end{bmatrix}$$

where we add each of the coordinates element-wise. The *scalar multiplication* is similarly defined as

$$c\mathbf{X} = \begin{bmatrix} cx_{1,1} & cx_{1,2} & \cdots & cx_{1,n} \\ cx_{2,1} & cx_{2,2} & \cdots & cx_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ cx_{m,1} & cx_{m,2} & \cdots & cx_{m,n} \end{bmatrix}$$

The *matrix multiplication* $\mathbf{XY}$ is defined for a matrix $\mathbf{X} \in \mathbb{R}^{\ell \times m}$ and a matrix $\mathbf{Y} \in \mathbb{R}^{m \times n}$; that is, when the number of columns of the first matrix is equal to the number of rows of the second matrix. The output $\mathbf{XY}$ of the matrix multiplication will be a $\ell \times n$ matrix. The (i, j) entry of the matrix $\mathbf{XY}$ is defined as

$$(\mathbf{XY})_{i,j} = \sum_{k=1}^{m} x_{i,k} y_{k,j}$$

That is, it is defined as the inner product of the i-th row of $\mathbf{X}$ and the j-th column of $\mathbf{Y}$.

Proposition 20.2.1. *The above matrix operations satisfy the following properties:*

- $c(\mathbf{XY}) = (c\mathbf{X})\mathbf{Y} = \mathbf{X}(c\mathbf{Y})$
- $(\mathbf{X}_1 + \mathbf{X}_2)\mathbf{Y} = \mathbf{X}_1\mathbf{Y} + \mathbf{X}_2\mathbf{Y}$
- $\mathbf{X}(\mathbf{Y}_1 + \mathbf{Y}_2) = \mathbf{X}\mathbf{Y}_1 + \mathbf{X}\mathbf{Y}_2$

Finally, the *transpose* $\mathbf{X}^\intercal \in \mathbb{R}^{n\times m}$ of a matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$ is the resulting matrix when the entries of $\mathbf{X}$ are reflected down the diagonal. That is,

$$(\mathbf{X}^\intercal)_{i,j} = \mathbf{X}_{j,i}$$

Proposition 20.2.2. *The transpose of a matrix satisfies the following properties:*

- $(\mathbf{X} + \mathbf{Y})^\intercal = \mathbf{X}^\intercal + \mathbf{Y}^\intercal$
- $(c\mathbf{X})^\intercal = c(\mathbf{X}^\intercal)$
- $(\mathbf{XY})^\intercal = \mathbf{Y}^\intercal\mathbf{X}^\intercal$

20.2.2 Matrix and Linear Transformation

Recall that a vector of size n is often considered a $n \times 1$ matrix. Therefore, given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and a vector $\vec{\mathbf{x}} \in \mathbb{R}^n$, we can define the following operation

$$\vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}} \in \mathbb{R}^m$$

through matrix multiplication. This shows that $\mathbf{A}$ can be understood as a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. We see that $a_{i,j}$ (the (i, j) entry of the matrix $\mathbf{A}$) is the coefficient of x_j (the j-th coordinate of the input vector) when computing y_i (the i-th coordinate of the output vector). Since each y_i is linear in terms of each x_j, we say that $\mathbf{A}$ is a *linear transformation*.

20.2.3 Matrix Rank

Matrix rank is one of the most important concepts in basic linear algebra.

Definition 20.2.3. *Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ of m rows and n columns, the number of linearly independent rows is known to be always equal to the number of linearly independent columns. This common number is known as the **rank** of $\mathbf{A}$ and is denoted as $rank(\mathbf{A})$.*

The following property of rank is implied in the definition, but we state it explicitly as follows.

Proposition 20.2.4. *The rank of a matrix is invariant to reordering rows/-columns.*

Example 20.2.5. *Consider the matrix* $M = \begin{bmatrix} 1 & 1 & -2 & 0 \\ -1 & -1 & 2 & 0 \end{bmatrix}$*, we notice that the second row is simply the first row negated, and thus the* rank *of M is* 1.

Example 20.2.6. *Consider the matrix* $M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$*, the rank of* M *is* 3 *because all the row (or column) vectors are linearly independent (they form basis vectors of* $\mathbb{R}^3$*).*

Example 20.2.7. *Consider the matrix* $M = \begin{bmatrix} 1 & 0 & 1 \\ -2 & -3 & 1 \\ 3 & 3 & 0 \end{bmatrix}$*, the rank of* M *is* 2 *because the third row can be expressed as the first row subtracted by the second row.*

When we interpret a matrix as a linear transformation, the rank measures the dimension of the output space.

Proposition 20.2.8. $\mathbf{A} \in \mathbb{R}^{m \times n}$ *has rank* k *if and only if the image of the linear transformation; i.e., the subspace*

$$\{\mathbf{A}\vec{\mathbf{x}} \mid \vec{\mathbf{x}} \in \mathbb{R}^n\}$$

of $\mathbb{R}^m$ *has dimension* k*.*

There are many known algorithms to compute the rank of a matrix. Examples include Gaussian elimination or certain decompositions (expressing a matrix as the product of other matrices with certain properties). Given m vectors in $\mathbb{R}^n$, we can find the maximum number of linearly independent vectors by constructing a matrix with each row equal to each vector [4] and finding the rank of that matrix.

[4] By Proposition 20.2.4, the order of the rows can be arbitrary.

20.2.4 *Eigenvalues and Eigenvectors*

Say we have a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$. This means that the linear transformation expressed by $\mathbf{A}$ is a mapping from $\mathbb{R}^n$ to itself. Most vectors $\vec{\mathbf{x}} \in \mathbb{R}^n$ is mapped to a very "different" vector $\mathbf{A}\vec{\mathbf{x}}$ under this mapping. However, some vectors are "special" and they are mapped to another vector with the same direction.

Definition 20.2.9 (Eigenvalue/Eigenvector)**.** *Given a square matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$*, if a vector* $\vec{\mathbf{v}} \in \mathbb{R}^n$ *satisfies*

$$\mathbf{A}\vec{\mathbf{v}} = \lambda\vec{\mathbf{v}}$$

for some scalar $\lambda \in \mathbb{R}$*, then* $\vec{\mathbf{v}}$ *is known as an* ***eigenvector*** *of* $\mathbf{A}$*, and* λ *is its corresponding* ***eigenvalue****.*

Each eigenvector can only be associated with one eigenvalue, but each eigenvalue may be associated with multiple eigenvectors.

Proposition 20.2.10. *If* $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ *are both eigenvectors of* $\mathbf{A}$ *for the same eigenvalue* λ*, then any linear combination of them is also an eigenvector for* $\mathbf{A}$ *with the same eigenvalue* λ*.*

Proposition 20.2.10 shows that the set of eigenvectors for a particular eigenvalue forms a subspace, known as the *eigenspace* of that eigenvalue. The dimension of this subspace is known as the *geometric multiplicity* of the eigenvalue. The following result ties together some of the concepts we discussed so far.

Proposition 20.2.11 (Rank-Nullity Theorem)**.** *Given a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, the eigenspace of 0 is the set of all vectors that get mapped to zero vector $\vec{\mathbf{0}}$ under the linear transformation $\mathbf{A}$. This subspace is known as the **null space** of $\mathbf{A}$ and its dimension (i.e., the geometric multiplicity of 0) is known as the **nullity** of $\mathbf{A}$ and is denoted as $nullity(\mathbf{A})$. Then*

$$rank(\mathbf{A}) + nullity(\mathbf{A}) = n$$

20.3 Advanced: SVD/PCA Procedures

Now we briefly introduce a procedure called *Principal Component Analysis (PCA)*, which is commonly used in low-dimensional representation as in Chapter 7.

We are given vectors $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_N \in \mathbb{R}^d$ and a positive integer k and wish to obtain the low-dimensional representation in the sense of Definition 7.1.1 that minimizes ϵ. This is what we mean by "best" representation.

Theorem 20.3.1. *The best low-dimensional representation consists of k eigenvectors corresponding to the top k eigenvalues (largest numerical values) of the matrix $\mathbf{A}\mathbf{A}^\intercal$ where the columns of $\mathbf{A}$ are $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_N$.*

Theorem 20.3.1 shows what the best low-dimensional representation is, but it does not show *how* to compute it. It turns out something called the *Singular Value Decomposition (SVD)* of the matrix $\mathbf{A}$ is useful. It is known that any matrix $\mathbf{A}$ can be decomposed into the following product

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal$$

where $\mathbf{\Sigma}$ is a diagonal matrix with entries equal to the square root of the nonzero eigenvalues of $\mathbf{A}\mathbf{A}^\intercal$ and the columns of $\mathbf{U}$ are the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}^\intercal$, where the i-th column is the eigenvector that corresponds to the eigenvalue at the i-th diagonal entry of $\mathbf{\Sigma}$. There are known computationally efficient algorithms that will perform the SVD of a matrix.

In this section, we will prove Theorem 20.3.1 for the case where $k = 1$. To do this, we need to introduce some preliminary results.

Theorem 20.3.2. *If a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric (i.e., $\mathbf{A} = \mathbf{A}^\intercal$), then there is an orthonormal basis of $\mathbb{R}^n$ consisting of n eigenvectors of $\mathbf{A}$.* [5]

[5] This is known as the Spectral Theorem.

Proof. A real symmetric matrix is known to be *diagonalizable*, and diagonalizable matrices are known to have n eigenvectors that form a basis for $\mathbb{R}^n$. In particular, the eigenvectors are linearly independent, meaning the eigenvectors corresponding to a particular eigenvalue λ will form a basis for the corresponding eigenspace. Through the Gram-Schmidt process, we can replace some of these eigenvectors such that the eigenvectors for λ are orthogonal to each other. That is, if $\vec{\mathbf{u}}, \vec{\mathbf{v}}$ are eigenvectors for the same eigenvalue λ, then $\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = 0$. Now assume $\vec{\mathbf{u}}, \vec{\mathbf{v}}$ are two eigenvectors with distinct eigenvalues λ, μ respectively. Then

$$\begin{aligned}\lambda \vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = (\lambda \vec{\mathbf{u}}) \cdot \vec{\mathbf{v}} = (\mathbf{A}\vec{\mathbf{u}}) \cdot \vec{\mathbf{v}} = \sum_{i,j=1}^{n} a_{i,j} u_j v_i \\ = \vec{\mathbf{u}} \cdot (\mathbf{A}^\intercal \vec{\mathbf{v}}) = \vec{\mathbf{u}} \cdot (\mathbf{A}\vec{\mathbf{v}}) = \vec{\mathbf{u}} \cdot (\mu \vec{\mathbf{v}}) = \mu \vec{\mathbf{u}} \cdot \vec{\mathbf{v}}\end{aligned}$$

where the third and the fourth equality can be verified by direct computation. Since $\lambda \neq \mu$, we conclude $\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = 0$. We have now showed that $\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = 0$ for any pair of eigenvectors $\vec{\mathbf{u}}, \vec{\mathbf{v}}$ — this means that the basis of eigenvectors is also orthogonal. After normalization, the basis can be made orthonormal. □

The following result is not necessarily needed for the proof of Theorem 20.3.1, but the proofs are similar.

Theorem 20.3.3. *If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric, then the unit vector $\vec{\mathbf{x}}$ that maximizes $\|\mathbf{A}\vec{\mathbf{x}}\|$ is an eigenvector of $\mathbf{A}$ with an eigenvalue, whose absolute values is the largest out of all eigenvalues.*

Proof. By Theorem 20.3.2, there is an orthonormal basis $\{\vec{\mathbf{u}}_1, \ldots, \vec{\mathbf{u}}_n\}$ of $\mathbb{R}^n$ consisting of eigenvectors of $\mathbf{A}$. Then any vector $\vec{\mathbf{x}}$ is in the span of the eigenvectors and can be represented as the linear combination

$$\vec{\mathbf{x}} = \alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n$$

for some scalars α_i's. Then

$$\begin{aligned}\|\vec{\mathbf{x}}\|^2 &= \vec{\mathbf{x}} \cdot \vec{\mathbf{x}} \\ &= (\alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n) \cdot (\alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n) \\ &= \sum_{i,j=1}^{n} \alpha_i \alpha_j (\vec{\mathbf{u}}_i \cdot \vec{\mathbf{u}}_j) \\ &= \sum_{i=1}^{n} \alpha_i^2\end{aligned}$$

where for the last equality, we use the fact that $\vec{\mathbf{u}}_i$'s are orthonormal — that is, $\vec{\mathbf{u}}_i \cdot \vec{\mathbf{u}}_j = 0$ if $i \neq j$ and $\vec{\mathbf{u}}_i \cdot \vec{\mathbf{u}}_i = 1$. Since $\vec{\mathbf{x}}$ has norm 1, we see

that $\sum_{i=1}^{n} \alpha_i^2 = 1$. Now notice that

$$\begin{aligned}\mathbf{A}\vec{\mathbf{x}} &= \mathbf{A}(\alpha_1\vec{\mathbf{u}}_1 + \alpha_2\vec{\mathbf{u}}_2 + \ldots + \alpha_n\vec{\mathbf{u}}_n) \\ &= \alpha_1\mathbf{A}\vec{\mathbf{u}}_1 + \alpha_2\mathbf{A}\vec{\mathbf{u}}_2 + \ldots + \alpha_n\mathbf{A}\vec{\mathbf{u}}_n \\ &= \alpha_1\lambda_1\vec{\mathbf{u}}_1 + \alpha_2\lambda_2\vec{\mathbf{u}}_2 + \ldots + \alpha_n\lambda_n\vec{\mathbf{u}}_n\end{aligned}$$

where λ_i is the eigenvalue for the eigenvector $\vec{\mathbf{u}}_i$. Following a similar computation as above,

$$\|\mathbf{A}\vec{\mathbf{x}}\|^2 = \sum_{i=1}^{n} \alpha_i^2\lambda_i^2$$

The allocation of weights α_i that will maximize $\sum_{i=1}^{n} \alpha_i^2\lambda_i^2$ while maintaining $\sum_{i=1}^{n} \alpha_i^2 = 1$ is assigning $\alpha_i = \pm 1$ to the eigenvalue λ_i that has the highest value of λ_i^2. This shows that the unit vector $\vec{\mathbf{x}} = \pm\vec{\mathbf{u}}_i$ is an eigenvector with the eigenvalue λ_i. □

We now prove one last preliminary result.

Theorem 20.3.4. *For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the matrix $\mathbf{A}\mathbf{A}^\intercal$ is symmetric and its eigenvalues are non-negative.*

Proof. The first part can be verified easily by observing that

$$(\mathbf{A}\mathbf{A}^\intercal)^\intercal = (\mathbf{A}^\intercal)^\intercal\mathbf{A}^\intercal = \mathbf{A}\mathbf{A}^\intercal$$

Now assume $\vec{\mathbf{x}}$ is an eigenvector of $\mathbf{A}$ with eigenvalue λ. Then

$$\mathbf{A}\mathbf{A}^\intercal\vec{\mathbf{x}} = \lambda\vec{\mathbf{x}}$$

We multiply $\vec{\mathbf{x}}^\intercal$ on the left on both sides of the equation.

$$\vec{\mathbf{x}}^\intercal\mathbf{A}\mathbf{A}^\intercal\vec{\mathbf{x}} = \vec{\mathbf{x}}^\intercal(\lambda\vec{\mathbf{x}}) = \lambda\|\vec{\mathbf{x}}\|^2$$

At the same time, notice that

$$\vec{\mathbf{x}}^\intercal\mathbf{A}\mathbf{A}^\intercal\vec{\mathbf{x}} = (\mathbf{A}^\intercal\vec{\mathbf{x}})^\intercal(\mathbf{A}^\intercal\vec{\mathbf{x}}) = \|\mathbf{A}^\intercal\vec{\mathbf{x}}\|^2$$

which shows that

$$\lambda\|\vec{\mathbf{x}}\|^2 = \|\mathbf{A}^\intercal\vec{\mathbf{x}}\|^2$$

Since $\|\vec{\mathbf{x}}\|^2, \|\mathbf{A}^\intercal\vec{\mathbf{x}}\|^2$ are both non-negative, λ is also non-negative. □

We are now ready to (partially) prove the main result of this section.

Proof of Theorem 20.3.1. We prove the case where $k = 1$. Recall that we want to find a vector $\vec{\mathbf{u}}$ that minimizes the error of the low-dimensional representation:

$$\sum_{i=1}^{N} \left\| \vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i \right\|^2$$

where $\widehat{\vec{\mathbf{v}}}_i$ is the low-dimensional representation of $\vec{\mathbf{v}}_i$ that can be computed as

$$\widehat{\vec{\mathbf{v}}}_i = (\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})\vec{\mathbf{u}}$$

by the result of Problem 7.1.3. Now by Proposition 20.1.15, we see that

$$\begin{aligned}\sum_{i=1}^{N} \|\vec{\mathbf{v}}_i - (\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})\vec{\mathbf{u}}\|^2 &= \sum_{i=1}^{N} \left(\|\vec{\mathbf{v}}_i\|^2 - \|(\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})\vec{\mathbf{u}}\|^2 \right) \\ &= \sum_{i=1}^{N} \left(\|\vec{\mathbf{v}}_i\|^2 - (\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})^2 \right)\end{aligned}$$

Since we are already given a fixed set of vectors $\vec{\mathbf{v}}_i$, we cannot change the values of $\|\vec{\mathbf{v}}_i\|^2$. Therefore, minimizing the last term of the equation above amounts to maximizing $\sum_{i=1}^{N} (\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})^2$. Notice that

$$\sum_{i=1}^{N} (\vec{\mathbf{v}}_i \cdot \vec{\mathbf{u}})^2 = \|\mathbf{A}^\intercal \vec{\mathbf{u}}\|^2 = \vec{\mathbf{u}}^\intercal \mathbf{A}\mathbf{A}^\intercal \vec{\mathbf{u}}$$

By Theorem 20.3.2 and by Theorem 20.3.4, there is an orthonormal basis $\{\vec{\mathbf{u}}_1, \ldots, \vec{\mathbf{u}}_n\}$ of $\mathbb{R}^n$ that consist of the eigenvectors of the matrix $\mathbf{A}\mathbf{A}^\intercal$. Let λ_i be the eigenvalue corresponding to the eigenvector $\vec{\mathbf{u}}_i$. Then similarly to the proof of Theorem 20.3.3, we can represent any vector $\vec{\mathbf{u}}$ as a linear combination of the eigenvectors as

$$\vec{\mathbf{u}} = \alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n$$

Then we have $\sum_{i=1}^{n} \alpha_i^2 = 1$ and

$$\begin{aligned}\vec{\mathbf{u}}^\intercal \mathbf{A}\mathbf{A}^\intercal \vec{\mathbf{u}} &= (\alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n)^\intercal \mathbf{A}\mathbf{A}^\intercal (\alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n) \\ &= (\alpha_1 \vec{\mathbf{u}}_1 + \alpha_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \vec{\mathbf{u}}_n)^\intercal (\alpha_1 \lambda_1 \vec{\mathbf{u}}_1 + \alpha_2 \lambda_2 \vec{\mathbf{u}}_2 + \ldots + \alpha_n \lambda_n \vec{\mathbf{u}}_n) \\ &= \sum_{i,j=1}^{n} \alpha_i \alpha_j \lambda_j (\vec{\mathbf{u}}_i \cdot \vec{\mathbf{u}}_j) \\ &= \sum_{i=1}^{n} \alpha_i^2 \lambda_i\end{aligned}$$

Again, the allocation of α_i's that maximize $\sum_{i=1}^{n} \alpha_i^2 \lambda_i$ while maintaining $\sum_{i=1}^{n} \alpha_i^2 = 1$ is assigning $\alpha_i = \pm 1$ to the eigenvector corresponding to the highest value of λ_i. □