COMPSCI 389

Introduction to Machine Learning

Prof. Philip S. Thomas

Spring 2022

Manning College of Information and Computer Sciences
University of Massachusetts

Acknowledgements. I would like to thank the many people who contributed to this document with feedback and corrections. This includes the teaching assistants, Scott Jordan (Spring 2021) and Cooper Sigrist (Spring 2021), as well as the many students who took this course. I would particularly like to thank Wes Cowley [link], who provided many edits that improved the clarity and veracity of the initial Spring 2021 version this document. Finally, I would like to thank the Berkeley Existential Risk Initiative for their support.

Contents

Introduction 1

1.1 What is COMPSCI 389?

COMPSCI 389 is an undergraduate (junior level) introduction to machine learning. This course is intended for students who have learned the basics of programming but who may have no previous exposure to machine learning or artificial intelligence. Familiarity with linear algebra is *not* required for this course, but familiarity with calculus and basic probability is assumed.

1.2 What is This Document?

This document is a study aid for the Spring 2022 offering of COMPSCI 389 in the *Manning College of Information and Computer Sciences* (CICS) at the University of Massachusetts. Each chapter corresponds to one lecture. This is not a full-fledged book, nor is it merely a collection of notes that a student might have taken during lecture. Rather, this document lies between these two extremes, providing more context than notes that you might take during lecture without being a stand-alone resource like a textbook.

1.3 What is Artificial Intelligence?

Artificial intelligence (AI) is a field concerned with intelligent behavior in artifacts [1]. To properly parse this definition, we must define the three colored terms: field, intelligent behavior, and artifacts.

[1]: Nilsson (1998), *Artificial Intelligence: A New Synthesis*

A field is an area of study like math, physics, and theology. Within the AI field, the phrase "AI" is typically *not* used to refer to a *thing*. For example, one would not say "I want to create an AI that . . . ", even though this phrasing is common outside of the AI field (e.g.,[1] in video game design). Within the AI field, the *thing* (software, robot, object, etc.) that uses AI methods is called an agent. More precisely, an **agent**[2] is an autonomous entity which acts, directing its activity towards achieving goals, upon an environment using observation through sensors and consequent actuators [2]. So, someone in the AI field is more likely to say "I want to create an agent that . . . ".

1: This text uses the abbreviations "e.g." and "i.e." for the common Latin terms *exempli gratia* (for example) and *id est* (that is).

2: This usage of the word *agent* stems from the Latin word *agere*, which means "to do." [2]: Wikipedia contributors (2021), *Intelligent Agent*

With this modern usage of the word *agent*, we can update Nilsson's definition of AI by replacing artifact with agent: **Artificial intelligence** is a field concerned with intelligent behavior in agents.

Last, but certainly not least, intelligent behavior is behavior that a reasonable person might recognize as indicative of intelligence. You may rightfully be thinking "What a cop-out! What then is *intelligence*?". Rather

than attempt to answer this nuanced and debated question, we point out that it is not particularly important in the context of this discussion. AI is a *field*, and so determining what is, and is not, intelligent behavior is useful in this context to determine what is, and what is not, part of the AI field. However, in reality, the various definitions of AI are of little importance when deciding what is and is not AI; the broader computer science community generally decides by implicit consensus which fields are and are not part of AI, independent of any definitions.

For example, consider two programs, `Optimize` and `Path`. `Optimize` takes as input easy-to-read source code and outputs source code that is fast to run. An example input might be:

Algorithm 1.1: Example input to `Optimize`.

```
1 while value < 100 do
2 |   item = 10;
3 |   value = value + item;
4 end
```

With the above algorithm as input, `Optimize` might output:

Algorithm 1.2: Example output from `Optimize`.

```
1 while value < 100 do
2 |   value = value + 10;
3 end
```

Or, perhaps `Optimize` might be even more intelligent, producing the following as output.

Algorithm 1.3: Example output from `Optimize`.

```
1 value = 100 + (value modulo 10);
```

Looking at the second example output, Algorithm 1.3, I suspect that you had to think a bit to verify that the input and output programs are equivalent. A program that can do this reduction is surely in some way exhibiting intelligent behavior! However, this program is one that would be studied within the field of *compiler optimization*, which is generally viewed as *not* part of AI.

Next, consider a program, `Path`, which takes as input a graph[3] (V, E) and two vertices, `start` $\in V$ and `goal` $\in V$,[4] and outputs `yes` if there is a path from `start` to `goal`, and outputs `no` otherwise. *I* do not find this behavior to be particularly "intelligent" (do you?). Sure, it may be computationally intensive, but I think intelligence is more than just the amount of computation used by a program—if that was the way to measure intelligence, then the most intelligent agent on Earth would be a Chrome browser with more than five tabs open!

However, regardless of whether you believe that `Path` exhibits intelligent behavior or not, it is a type of algorithm called a *search* algorithm, which the AI community agrees is part of AI. How can we reconcile our definition of the AI field with fact that `Optimize` seems like it should be part of AI but is not, while `Path` seems like it should not be part of AI but is? The answer is to view the definition of AI as a guiding principle for crudely summarizing what is and is not part of AI, rather

3: A **graph** (V, E) is a set of *vertices*, V, and a set of edges, E. [Wiki]

4: Throughout this document, I make minor abuses of grammar and notation for brevity. For example, here the two vertices are `start` and `goal`, not the two Boolean values `start` $\in V$ and `goal` $\in V$.

than a precise rule. Furthermore, this guiding principle should be applied inclusively—do not be the zealot arguing that some subfield that has been established as part of AI should not be considered part of the AI field. You will only annoy the people working in that subfield. Instead, try to be inclusive—recognize that inclusion within AI is a communal decision that does not necessarily have to agree with any written definition of AI or intelligence.

A consequence of this usage of the phrase AI is that students are often disappointed by their first classes in AI or machine learning. Students enter expecting an enlightening discussion of intelligence and how to create minds, only to find an introductory course filled with discussion of algorithms like `Path`. When taking my first course on AI, I recall thinking "none of this is AI" for the vast majority of the semester. One goal of this course is to prevent this disillusionment by including discussion of more advanced topics, like *reinforcement learning* algorithms, which closely resemble the algorithms implemented by parts of animal brains. Furthermore, I will explicitly discuss topics related to psychology, neuroscience, and even philosophy of mind in order to situate the topics covered in this course relative to the broader discussion of (natural and synthetic) intelligence and minds.

1.4 What is Machine Learning?

Machine learning (ML) is a subfield of AI concerned with the question of how to construct computer programs that automatically improve with experience [3]. The inputs to a program that characterize an "experience" are typically called **data**.[5] Similarly, the modification of behavioral tendencies by experience is called *learning* [4]. Finally, incorporating the word *agent*, defined above, we can rewrite Mitchell's definition of ML as: **Machine learning** is a subfield of AI concerned with the question of how to construct agents that learn from data.

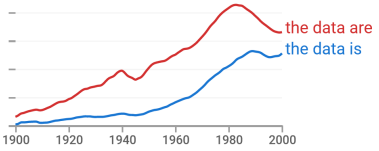
[3]: Mitchell (1997), *Machine Learning*

[4]: Merriam-Webster (2021), *Learning*

As an example, consider a third program, `Predict`, that takes as input data consisting of labeled images of handwritten letters along with a handwritten letter without a label (called the *query*), and which outputs a prediction of the label for the unlabeled letter. Example inputs and outputs for `Predict` are depicted in Figure 1.1 and Table 1.1. Problems of the sort solved by `Predict`, where the labels are discrete (not real numbers), are called *classification* problems and will be covered in a later chapter.

To summarize, *artificial intelligence* (AI) is a subfield of computer science concerned with agents that are in any way *intelligent*, and *machine learning* (ML) is a subfield of AI concerned with agents that become intelligent via *learning*. For example, a program that takes as input the rules to chess and then computes an optimal strategy for the game would fall within AI, but not ML, while a program that learns an optimal chess strategy over time by playing many games of chess would fall within ML. However, the precise boundaries of AI and ML are vague and sometimes debated. As a result, you may find it surprising that certain topics are, or are not, part of AI or ML, and you should try not to be dismissive of other peoples'

5: *Data* is the plural of the word *datum*, much like how *agenda* is the plural of the Latin word *agendum*. Just as people have started using *agenda* as a singular noun in English, the word *data* is increasingly treated as a singular noun in English, as shown in the Google Ngram below.



In this document I use *data* as both the singular and plural forms.

Figure 1.1: Example data that could be provided to the `Predict` algorithm. Each row corresponds to a different label, from *zero* to *nine*. These images are examples of some of the entries in a popular data set called the Modified National Institute of Standards and Technology data set, which is commonly called **MNIST** (pronounced "em-nist"). If you were standing in for the `Predict` algorithm and saw the handwritten number below as an input for which you should predict the label, what would you output?

Input	Label
	Wolf
	Seax
	Pacul
	Wolf
	Pacul
	Seax

Input	Your Predicted Label
	?
	?
	?

Table 1.1: Left. Example input data for the `Predict` algorithm. **Right.** Examples of additional inputs (without labels) for the `Predict` algorithm to produce predictions of what the labels should be. If you were the `Predict` algorithm, what would you predict the correct labels are?

work as "not really AI" just because it does not fit your definitions of artificial or intelligence.

1.5 Artificial General Intelligence

Artificial general intelligence (AGI) is the hypothetical intelligence of a computer program that has the capacity to understand or learn any intellectual task that a human being can [5]. Creating AGI is the long-term goal of many AI researchers, and the consensus among AI researchers is that AGI is feasible in theory. However, the general consensus is that we are far from creating AGI. In fact, we know so little about how AGI might eventually be created that it is challenging to guess how long it will be before it is achieved. If every remaining challenge ends up being easier to overcome than expected, and if breakthroughs come rapidly, then perhaps AGI might be created in 20 years. Similarly, if some particularly nefarious yet unforeseen challenge arises, it could be millennia.

[5]: Wikipedia contributors (2021), *Artificial General Intelligence*

As a result, there are two (sometimes conflicting) motivations for studying AI and ML. Some people do not think about AGI because it is too distant, and instead focus on creating AI and ML systems that can be beneficial today and in the near future—systems that could improve medical treatments, automate dangerous tasks, and be profitable. Others are entirely captivated by the long-term goal of creating AGI. These two motivations are sometimes conflicting, with peer reviewers of research papers sometimes arguing against work that is not practicable today but rather is a small step down the path towards AGI, and vice versa.

However, most AI researchers are motivated by both the short-term practical benefits that advances in AI and ML bring and the long-term goal of creating AGI. I hypothesize that most people enter the field leaning more towards the long-term goal of creating AGI, and over time shift towards focusing more heavily on shorter-term (feasible within a decade) research. This shift is caused by the necessity to show progress—it is hard to make a living by saying: "Pay me to think for 20–200,000 years, and I might have an awesome development then." Rather, real-world pressures and celebrations of smaller successes train researchers to increasingly lean towards the shorter-term goals. However, the passion for creating AGI often persists under the surface. So, if you are interested in creating AGI, do not be discouraged when you find that people more senior in the AI and ML fields often talk very little about it. The interest in AGI is generally shared—there is just very little to say, and we do not know how to make progress beyond focusing on shorter-term subgoals on the path to AGI.

While we will talk about some topics related to AGI later in the course, particularly when discussing the relationship between ML and other fields like psychology, neuroscience, and philosophy, the first two-thirds of this course will *not* focus on AGI. Instead, it will focus on creating agents capable of learning, albeit for extremely simple problems when compared to the problems humans solve. Still, this study of the foundations of intelligence and learning provides useful background for your future deliberations concerning AGI.

1.6 Topics and Overview

This course can be roughly broken into three parts:

1. **Supervised learning.** This part focuses on creating agents that can learn from labeled data, like the `Predict` algorithm.
2. **Reinforcement learning.** This part focuses on creating agents that can learn from their own experiences—by trial and error.
3. **Safety, fairness, ethics, and related fields.** This part focuses on a variety of topics, beginning with those related to the responsible use of ML for real-world problems—topics like safety, fairness, and ethics. After discussing these practical issues, this part turns to describing how ML research relates to other research areas like psychology, neuroscience, and philosophy.

Finally, the last chapter may contain a survey of advanced topics in ML, to show what will come next if you decide to pursue further ML education.

Supervised Learning

Regression I 2

2.1 Regression Problem Formulation

Regression problems are a type of supervised learning problem wherein an agent uses labeled data to learn to predict the labels for new inputs that were not necessarily seen during training, and where the labels are real numbers. This differs from the **classification** setting, which we will cover later, where the labels are discrete (like the 26 discrete letters in the alphabet when classifying handwritten letters).

GPA Prediction Example

As an illustrative example, consider the problem of using information from college applications to predict what an applicant's *grade point average* (GPA) would be if the student were to be admitted.[1] Such a system could be useful for filtering large numbers of applications to determine who to admit, or if a human is in the loop, for filtering applications to select which ones should be looked at by the human. This example is similar to a common use of ML: determining which job applicant résumés [6] should be inspected by a person.

Specifically, we will consider a **data set**[2] containing information about 43,303 students from the *Federal University of Rio Grande do Sul* (UFRGS), a top university in Brazil. This data was collected as part of an evaluation of Brazil's *quota system* [7], and includes each student's scores on nine entrance exams taken specifically when applying to UFRGS, along with students' GPAs after their first three semesters at UFRGS.

You can download the data set in *comma-separated value* (.csv) form here. This data set, which we call the *base GPA data set* or simply the *GPA data set* (later, we will consider an expanded version of this data set) has 10 columns and 43,303 rows. Each row corresponds to one applicant; the first nine columns correspond to the applicant's scores on different entrance exams and the tenth column is the student's GPA after completing the first three semesters at UFRGS. The first few rows are provided in Table 2.1.

1: We assume that GPAs are on a four-point scale in the closed interval $[0, 4]$; that is, an A maps to 4, B to 3, C to 2, D to 1, and F to 0.

[6]: Lauren Weber (2012), *Your Résumé vs. Oblivion*

[7]: Julia Carneiro (2013), *Brazil's universities take affirmative action*

2: **Data sets** are simply collections of data often used to train ML algorithms. In this course, we use "data set" rather than "dataset." For formal writing, use the version that matches your target publication's style.

Exam Number									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**GPA**
538	491	407	529	532	447	528	379	489	2.98
455	440	571	418	454	426	476	476	407	1.97
757	680	531	584	534	521	592	784	588	2.53333

Table 2.1: The first three rows from the base GPA data set, with exam scores rounded to the nearest integer.

2.2 Regression Notation

While all of the available data is called the data set, one row is called a **data point**. Each data point has two components: the example input and the desired output for that input, which we call the **label**. In the GPA data set, the input is the applicant's scores on the nine entrance exams, and the label is the student's subsequent GPA.

The input data is a vector (one-dimensional array) of real numbers, called a **feature vector**. We write x_i to denote the i^{th} feature vector. For example, in the GPA data set,[3]

$$x_1 \approx (538, 491, 407, 529, 532, 447, 528, 379, 489). \tag{2.1}$$

We write $x_{i,j}$ to denote the j^{th} feature within x_i. For example, $x_{1,3} \approx 407$. Also, let m be the number of features in each data point, so we can write $x_i \in \mathbb{R}^m$ to concisely indicate that each x_i is a vector of m real numbers.

Notice that we are using the symbol $\mathbb{R}$ to denote the set of all real numbers.[4] At the end of this document you will find a section titled *Notation* that serves as a reference for common symbols like $\mathbb{R}$.

We write y_i to denote the i^{th} label. For regression problems, y_i is a real number. That is,

$$y_i \in \mathbb{R}. \tag{2.2}$$

Additionally, let $\hat{y}_i$ be the agent's prediction of y_i (this prediction is created based on x_i).[5]

Let $n \in \mathbb{N}_{>0}$ be the number of data points in the data set ($n = 43{,}303$ in the GPA data set). With these definitions, a data set is a set of n data points,

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n), \tag{2.3}$$

where $\forall i \in \{1:n\},\ x_i \in \mathbb{R}^m,\ y_i \in \mathbb{R}$.[6] Even this was a bit cumbersome to write, and so we use a shorthand for sequences, writing $(x_i, y_i)_{i=1}^n$ to denote (2.3).[7]

Finally, let x_{n+1} be a feature vector for which the agent is not provided with the corresponding label, y_{n+1}. In this and later chapters, we will discuss how an agent can use the labeled data to construct an estimate, $\hat{y}_{n+1}$, of the unknown label y_{n+1}.

The previous sentence presents a common point of confusion: What is the difference between something being "unknown" and something "not existing"? When we say that something is "not known," we generally mean that it is not known by the agent. In the previous paragraph, we described how y_{n+1} is not known by the agent. However, that does *not* mean that y_{n+1} does not exist—we are assuming that it does, and we (you and I) may or may not know its value. Another example of this point of confusion is when we make statements like: Assume that s is the state of the universe—a complete description of *everything* in the universe at some given time.[8] A common response is: "How can we do that—we don't know the complete state of the universe!" The key here is that this is *not* a statement about what we or the agent knows, but rather a statement about the existence of such a state. We can then even reason about this state, s, even though we will never know it. Similarly, here, the

3: Notice that this equation uses the symbol $\approx$, which means *approximately equal to*. In this case, the approximation is due to rounding to the nearest integer.

4: $\mathbb{R}$ is the letter R written in *blackboard bold*. Blackboard bold has a fascinating history, which you can read about here.

5: A common convention, which we use here, is to write $\hat{a}$ to denote an estimate or approximation of a, for any symbol a.

6: In this text we will use a nonstandard notation $\{a:b\}$ to be the set of integers in the closed interval $[a,b]$. For example, $\{1:3\} = \{1,2,3\}$. You may have noticed that we are using $[\cdot]$, $(\cdot)$, and $\{\cdot\}$ to mean different things. Here, we use standard notation: we write $[a,b]$, (a,b), $[a,b)$, and $(a,b]$ to denote intervals of the real number line, spanning from a to b, with brackets indicating inclusion of the endpoint and parentheses indicating that the endpoint is excluded. We write $\{a_1, a_2, \ldots\}$ to denote a set with elements a_1, a_2, etc., and we write $(a_1, a_2, \ldots)$ to denote a sequence (ordered set) with elements a_1, a_2, etc. Finally, we use a form of mathematical logic notation similar to first order logic. For example,

$$\forall i \in \{1:n\},\ x_i \in \mathbb{R}^m,\ y_i \in \mathbb{R},$$

could be written in sloppy first order logic notation as:

$$\forall i \in \{1:n\}\big((x_i \in \mathbb{R}^m) \wedge (y_i \in \mathbb{R})\big).$$

More precisely, first order logic does not allow for range specification, and so the proper equivalent expression would be:

$$\forall i\Big(\big(i \in \{1,2,\ldots,n\}\big) \implies \big((x_i \in \mathbb{R}^m) \wedge (y_i \in \mathbb{R})\big)\Big).$$

7: So much notation! Keep with it—we're almost there! Notice how we just referenced (2.3) without writing something like "Equation 2.3" or "Eq. 2.3". Though you may see these latter examples fairly often, they raise an issue: (2.3) is *not* an equation—there is no equality specified! This is most noticeable when authors reference inequalities by calling them "equations."

8: For this discussion, we ignore the strange realities of time that make it nonsense to talk about "some given time" across large distances (even at the global scale, not just the interstellar scale). For a mind-bending adventure into the nature of time, I highly recommend the book *The Order of Time* by Carlo Rovelli [8].

agent will reason about what it thinks y_{n+1} is, despite the fact that it may never actually know what y_{n+1} really is.

2.3 Nonparametric Methods

How might an agent go about using the available data to predict the labels for new, unlabeled, feature vectors? One first idea might be to search through the data set for the feature vector x_i that is "closest" to x_{n+1} and then guess that the label for x_{n+1} will be y_i—that is, the label will be the same as the label for the closest point.

Pseudocode for this algorithm, called **nearest neighbor**, is provided in Algorithm 2.1. This algorithm has two nuances worth discussing here. First, to find the nearest feature vector, we sort the points by their distance to x_{n+1}. This pseudocode is very computationally inefficient, so we place "naïve" in the algorithm name. In practice, the search for nearest neighbors should be performed using a data structure designed to speed up the search, such as a k-d tree [9]. However, as long as n is relatively small, our basic version will be sufficient. Second, notice that this algorithm requires us to specify how we want to measure the distance between feature vectors[9] . Although any distance function could work, we will use Euclidean distance (for vectors) because it is familiar:[10]

$$\forall x \in \mathbb{R}^m, \forall x' \in \mathbb{R}^m, \operatorname{dist}(x, x') = \sqrt{\sum_{j=1}^{m} \left(x'_j - x_j\right)^2}. \tag{2.4}$$

[9]: Wikipedia contributors (2021), *k-d tree*

9: We call these specifications *hyperparameters*, which we will discuss further in Section 2.4.

10: Hereafter, we will write $\forall x, y \in \mathbb{R}^m$ as shorthand for $\forall x \in \mathbb{R}^m, \forall y \in \mathbb{R}^m$. Notice that this is different from $\forall (x, y) \in \mathbb{R}^m$, which would indicate that (x, y) is one element of $\mathbb{R}^m$.

Algorithm 2.1: Nearest Neighbor (Naïve)

Input : Data set $(x_i, y_i)_{i=1}^n$, feature vector x_{n+1}
Output : $\hat{y}_{n+1}$, a prediction of y_{n+1}.
Hyperparameters : Distance function dist

```
/* Initialize variables and compute needed distances   */
Allocate array dists of length n;
for i ← 1 to n do
    dists[i] ← dist(x_i, x_{n+1});
end
/* Sort the points by their distance to x_{n+1} (smallest
   first).                                              */
Sort (x_i, y_i, dists[i])_{i=1}^n by dists[i], breaking ties randomly;
/* Return the label of the closest point               */
return y_1;
```

To visualize the behavior of this algorithm, we applied it to the GPA data set. However, to make the behavior of the algorithm easier to visualize, we made predictions from only one exam score (the first column of the data set) and used only the first 30 data points. See Figure 2.1 for the results of this experiment.

In Figure 2.1, two undesirable behaviors of Algorithm 2.1 become apparent. First, consider the predicted GPAs for students with exam scores around 550. Slight changes in exam scores result in wild shifts in the predictions as the nearest neighbor changes from one data point to another. Even as the amount of available data increases, this trend will not

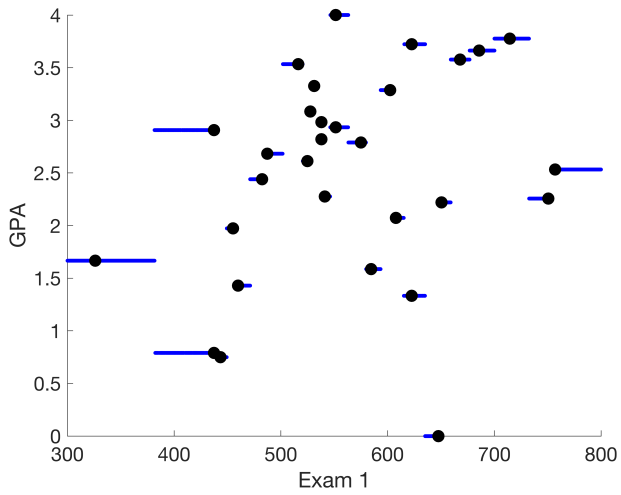


Figure 2.1: Black dots correspond to scores on the first exam (horizontal position) and GPAs (vertical position) for the first 30 rows (students) in the base GPA data set. The blue lines show the predictions of Algorithm 2.1 (nearest neighbor).

go away—if anything, it will get worse because small changes in exam scores will be more likely to result in a different nearest neighbor.

Second, consider the algorithm's behavior for students with exam scores around 400. Two data points (the 12^{th} and 15^{th}) have the same score on Exam 1, but different GPAs. As a result, Algorithm 2.1 selects randomly between the GPAs of these two points. This is usually not desirable behavior of a regression algorithm—imagine if the algorithm selected the lower prediction for you but the higher prediction for someone else who, from the algorithm's perspective, is identical to you!

We might mitigate the latter issue by returning the *average* of the labels for all of the points that are closest to x_{n+1}. For exam scores around 400, this would result in predictions closer to ≈ 2 rather than predictions that randomly flip between ≈ 0.75 and ≈ 3. However, that does not fix the former problem of predictions jumping around when there is only one unique nearest neighbor.

To fix both issues simultaneously, we might consider more than just the nearest neighbor—we could consider the $k \in \mathbb{N}_{>0}$ nearest neighbors. However, once the agent has identified the k nearest points, which label should it output? One popular strategy is to return the average label for the k nearest neighbors. Notice that when $k = 1$ this more sophisticated algorithm degenerates exactly to Algorithm 2.1.

The resulting algorithm, called k **nearest neighbor**, or k**-NN**, is an *extremely* popular algorithm due to its simplicity and efficacy. Pseudocode for the k nearest neighbor algorithm is provided in Algorithm 2.2. Notice that this algorithm differs from Algorithm 2.1 only in its inclusion of the hyperparameter k (and the corresponding assumption that $k \leq n$) and in the final line that determines the value to return.

Figure 2.2 shows the predictions of both nearest neighbor (Algorithm 2.1) and k nearest neighbor (Algorithm 2.2) with $k = 10$. Notice that by considering many of the nearby points the resulting predictions deviate less with individual outliers and better follow the general trend of the data. However, some issues remain. Consider how k is set. When points are densely packed along the horizontal axis (e.g., the middle of the horizontal axis in Figure 2.2), large values of k are appropriate, providing an average of the many points with similar horizontal positions.

Algorithm 2.2: k Nearest Neighbor (Naïve)

Input : Data set $(x_i, y_i)_{i=1}^n$, feature vector x_{n+1}
Output : $\hat{y}_{n+1}$, a prediction of y_{n+1}.
Hyperparameters : Distance function dist, number of nearest neighbors $k \in \mathbb{N}_{>0}$
Assumptions : $k \leq n$

```
/* Initialize variables and compute needed distances   */
Allocate array dists of length n;
for i ← 1 to n do
    dists[i] ← dist(x_i, x_{n+1});
end
/* Sort the points by their distance to x_{n+1} (smallest
   first).                                              */
Sort (x_i, y_i, dists[i])_{i=1}^n by dists[i], breaking ties randomly;
/* Return the average of the labels for the k closest
   points                                               */
return (1/k) Σ_{i=1}^k y_i;
```

However, when points are sparse along the horizontal axis (e.g., the left part of the horizontal axis in Figure 2.2), large values of k are *not* appropriate, because it results in the algorithm considering distant points just as much as nearby points. As a result, on the far left side of the plot in Figure 2.2, the predictions of k-NN with $k = 10$ tend to be higher than the data points. This happens because the 10 nearest neighbors include points corresponding to students with significantly higher scores on Exam 1, and these points are given the same weight in the prediction as the nearby points with much closer exam scores.

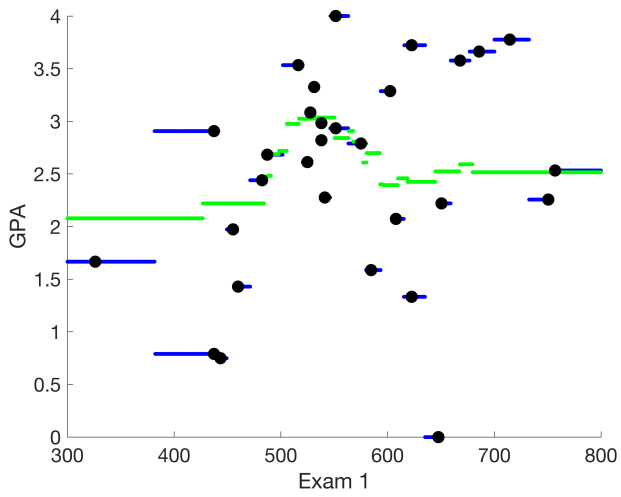


Figure 2.2: Like Figure 2.1, but with the predictions of k nearest neighbor (k-NN) with $k = 10$ plotted in green. Recall that the blue predictions also correspond to k-NN, but with $k = 1$.

So, the choice of k changes the behavior of the algorithm, and the best choice for k varies depending on the position on the horizontal axis (intuitively, we may want larger k when there are many points nearby). While one fix might be to allow k to vary based on the local density of points, we can make a more principled fix by using a *weighted* average that places a larger weight on nearby points and a smaller weight on distant points. Consider the left side of Figure 2.2—with an appropriate weighting scheme, the weighted average can effectively ignore the distant points that happen to be one of the k closest not because they are close in value, but because points are sparse on the far left of the plot. However,

for predictions near the middle of the horizontal axis, all of the k closest points will be considered almost equally because none are particularly distant.

Pseudocode for this algorithm, called *weighted* k nearest neighbors, is provided in Algorithm 2.3. Notice that Algorithm 2.3 takes as input an additional hyperparameter, $w : \mathbb{R}_{\geq 0} \to \mathbb{R}$.[11] This function is used to determine how much weight should be given to each point based on its distance to x_{n+1}. While there are many reasonable choices, a common choice is a bell curve (the *probability density function* (PDF) of the normal distribution with mean zero and standard deviation $\sigma \in \mathbb{R}$). That is, a point distance dist from x_{n+1} is given the weight:

$$w(\text{dist}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\text{dist}^2}{2\sigma^2}}. \tag{2.5}$$

11: The notation $f : \mathcal{A} \to \mathcal{B}$ indicates that f is a function that takes any element of $\mathcal{A}$ as input and always outputs an element of $\mathcal{B}$. So, $w : \mathbb{R} \to \mathbb{R}$ means that w takes any real number as input and produces a real number as output.

Notice that using this weighting function introduces another hyperparameter σ. The impact of this hyperparameter on the weighting scheme is depicted in Figure 2.3.

Also, notice that Algorithm 2.3 divides each weight by the sum of the (k) weights. To see why, consider measuring the temperature around you with three thermometers of different quality—a scientific thermometer, a cheap thermometer from a typical store, and your own guess of the temperature. Rather than directly average the temperature readings, you might take a weighted average that places a larger weight on the thermometer that you expect to be most accurate. Say that the scientific thermometer reads $t_1 = 71.3°\text{F}$, the cheap thermometer reads $t_2 = 70°\text{F}$, and your best guess is $t_3 = 72°\text{F}$. Rather than simply average these, you might take a weighted average with a weight of 10 on the scientific thermometer and weights of 1 on the other two thermometers.

However, if you take the weighted sum using the formula:

$$\text{combined estimate} = 10t_1 + 1t_2 + 1t_3, \tag{2.6}$$

you would obtain a combined estimate of 855°F! Clearly something went wrong. The issue is that our weights did not sum to one, and so the scale of the estimate is off by a factor of $10 + 1 + 1 = 12$. To fix this, we simply divide each weight by the sum of the weights, ensuring that the sum of the weights is one. This gives:

$$\text{combined estimate} = \frac{10}{12}t_1 + \frac{1}{12}t_2 + \frac{1}{12}t_3 = 71.25°\text{F}, \tag{2.7}$$

which is a much more reasonable estimate. Notice in Algorithm 2.3 the variable sum is used to normalize the weights in this same way.

Figure 2.4 shows the predictions of all three variants of the nearest neighbor algorithm. Notice on the left side of the plot that the weighting brings the predictions (magenta) down relative to k-NN without weighting (green) because the distant points with higher GPAs have less of an impact on the predictions for students with low scores on Exam 1. Also, note that the predictions tend to be a smoother function of exam scores.

Algorithm 2.3: Weighted k Nearest Neighbor (Naïve)

Input : Data set $(x_i, y_i)_{i=1}^n$, feature vector x_{n+1}
Output : $\hat{y}_{n+1}$, a prediction of y_{n+1}.
Hyperparameters : Distance function dist, number of nearest neighbors $k \in \mathbb{N}_{>0}$, weight function $w : \mathbb{R}_{\geq 0} \to \mathbb{R}$

```
/* Initialize variables and compute needed distances   */
Allocate arrays dists of length n;
for i ← 1 to n do
    dists[i] ← dist(x_i, x_{n+1});
end
/* Sort the points by their distance to x_{n+1} (smallest
   first).                                              */
Sort (x_i, y_i, dists[i])_{i=1}^n by dists[i], breaking ties randomly;
/* Compute weights                                      */
Allocate array weights of length k;
sum ← 0;
for i ← 1 to k do
    weights[i] ← w(dists[i]);
    sum ← sum + weights[i];
end
/* Return the weighted average of the labels for the k
   closest points                                       */
return Σ_{i=1}^k (weights[i] / sum) y_i;
```

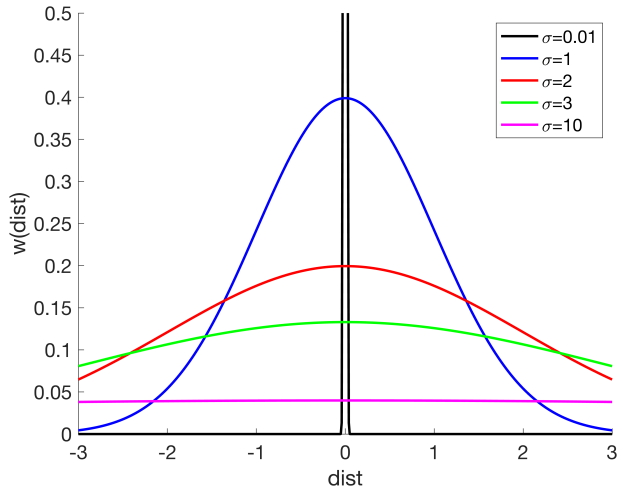


Figure 2.3: The weights assigned by (2.5) to points of distance dist (horizontal axis). Notice that σ sets the width of the weighting function—large values of σ correspond to wider bell curves (more consideration of distant points) and smaller values of σ to thinner bell curves (less consideration of distant points).

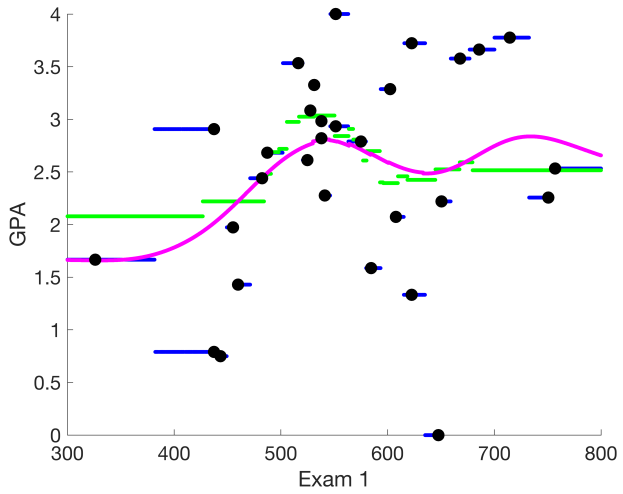


Figure 2.4: Like Figure 2.2, but with the predictions of weighted k nearest neighbor with $k = 20$ and $\sigma = 50$ plotted in magenta. Recall that the green predictions correspond to k-NN with $k = 10$ and the blue predictions correspond to k-NN with $k = 1$.

2.4 Hyperparameters

We have used the term **hyperparameters** to refer to several parameters of the various nearest neighbor algorithms. Hyperparameters are parameters whose values are used to tune the learning process. Intuitively, you can think of hyperparameters as values that are set (typically by a person) before the agent begins learning (reasoning from data) and that shape the learning process. For example, we discussed how the choice of k impacts the behavior of the k nearest neighbor algorithm.

How should one select values for all of an algorithm's hyperparameters? Setting hyperparameters is an art learned with practice—as you gain experience with ML algorithms, you will get better at guessing which values of hyperparameters will be effective, and you will learn how to study the behavior of an agent and make predictions about how the agent's hyperparameters should be changed to improve its performance.

Consider trying to apply ML algorithm `alg` to some application. In some cases, you might find a different ML algorithm, `meta`, that is designed to automatically adjust the hyperparameters of `alg` to make it more effective. In some cases, `meta` will have hyperparameters of its own. However, if `meta` is effective, it will tend to have fewer hyperparameters than `alg` or at least ones that are easier to set. We will discuss algorithms like `meta` later when automatically adjusting a hyperparameter called the *step size* that is used by some ML algorithms.

Also, notice that the quality of predictions generally improved as we made the nearest neighbor algorithm more sophisticated. However, the increased sophistication also came with an increase in the number of hyperparameters—nearest neighbor had one (the distance function), k nearest neighbor had two (the distance function and k), and weighted k nearest neighbor had three (the distance function, k, and the weighting function [or σ if you only consider the bell-curve weighting function]). This presents a trade-off: we can obtain better performance from the more sophisticated algorithms once the hyperparameters have been properly tuned, but it becomes increasingly difficult to properly tune the hyperparameters. We are essentially shifting some of the difficulty of the task from the ML algorithm to the mechanism performing the hyperparameter optimization (often a person).

Are we therefore deceiving ourselves by calling these algorithms better or more sophisticated, when really they are offloading the work to the hyperparameter optimization process? How should we compare these algorithms to determine which one to use for a specific application? Should we focus on the performance of the algorithm once the hyperparameters have been tuned? Should we focus on the performance of the algorithm with random hyperparameters? Is there some other way to account for the difficulty of tuning hyperparameters? These are surprisingly difficult questions to answer. Some efforts have been made to quantify how difficult it is to apply an ML algorithm to a new problem, taking into account the difficulty of setting hyperparameters. However, these efforts often make an assumption about how hyperparameters are tuned that does not reflect the reality that a human will often be a part of the process. Additionally, this human in the loop presents a serious confounding variable—are the predictions of weighted k nearest neighbor in Figure

2.4 the predictions of the ML algorithm or *my* predictions as the one who manually tuned the hyperparameters until the predictions looked good to me? Surely, we (myself and the algorithm) both deserve partial credit for the predictions, but who deserves more credit?

An unfortunate reality of modern ML, and particularly the reinforcement learning subfield, is that the amount of effort, computation, and time spent optimizing hyperparameters typically *dwarfs* all other aspects of applying ML. This should be obvious—tuning hyperparameters generally involves setting the hyperparameters, allowing the agent to learn, then resetting the agent with new (and hopefully improved) hyperparameters, and repeating the process. As this process involves multiple agent "lifetimes" (learning from scratch, then being reset), it is necessarily more time-consuming than a single agent lifetime.

The difficulty of tuning the hyperparameters of algorithms suggests that in many cases the mechanism tuning the hyperparameters of an agent, whether it is a person or a massively computationally intensive brute force search, is *more* responsible for the seemingly intelligent behavior of the agent than any learning process implemented by the agent. Furthermore, because of how much work it takes to tune hyperparameters, I encourage you to be skeptical of any "improvements" to ML algorithms that come with an increase in the difficulty of tuning hyperparameters (either by introducing more hyperparameters or by making the algorithm more sensitive to the hyperparameter settings). Often, purported improvements are consequences of increased work during hyperparameter optimization that is not accounted for in performance reports.

Lastly, you may have noticed a similarity to natural intelligences—recall from the first homework assignment that the computational power spent by evolution to tune your hyperparameters dwarfs all of the computation and learning performed by your brain during your lifetime, and so one might argue that your intelligence is primarily shaped by evolution and events from before your birth, not the learning that occurred since your birth. This provides an accurate analogy to keep in mind when working with ML algorithms—think of hyperparameter tuning as performing the job of evolution, and do not ignore the difficulty of this task nor the fact that seemingly intelligent behavior of an agent may actually be more reflective of an effective hyperparameter optimization mechanism than an effective ML algorithm.

2.5 Parametric Methods

Look back at Figure 2.4, and notice that all of the reported variants of the nearest neighbor algorithm present a troubling behavior: in some cases, they predict that your GPA will be higher if your exam score was lower! To overcome this, we might want to place a constraint on the agent, requiring it to make higher GPA predictions as exam scores increase. Including constraints within ML algorithms is an important topic and one that we will discuss later in the course, particularly when discussing how to place constraints on agents to prevent them from exhibiting racist, sexist, or otherwise unfair behavior.

For now, we will consider a particular type of constraint: what if we constrain the agent by assuming that the predictions should have a particular form? For example, what if we require the agent to make predictions that are linear in the first exam score, that is, predictions that form a line in Figure 2.4? We might *parameterize* this line with two parameters, m and b, using the formula:[12]

$$\hat{y}_{n+1} = mx_{n+1} + b, \tag{2.8}$$

so that m is the slope of the line and b is the y-intercept (the height at which the line crosses the y-axis). Now, the problem of learning to make predictions has been distilled to the seemingly simpler problem of finding the values for m and b that make accurate predictions for the available data.

12: Here we use m as the standard symbol for the slope parameter of a line, which is unrelated to our usage of m elsewhere as the number of input features.

Let f be the function within the agent that takes as input a feature vector and produces as output a prediction of the corresponding label:

$$\hat{y}_i = f(x_i). \tag{2.9}$$

In supervised learning (but not reinforcement learning)[13] this function is called a **model**.

13: In reinforcement learning this function is called a *policy*, and the word *model* is used to refer to a completely different mechanism within the agent that can be used to generate new training data (including generating new feature vectors!).

In order to restrict the agent to only consider linear predictors, we assume that the model is parameterized by a vector, w. In the example of fitting a line using (2.8), this parameter vector is $w = (m, b)$. The symbol w is often used because these parameters are sometimes called **weights**. More formally, these parameters are called **model parameters**. For brevity, we will call them "parameters."

Incorporating parameters into our definition of f, we obtain a **parameterized model**:

$$\hat{y}_i = f_w(x_i). \tag{2.10}$$

Here, we write w as a subscript of f to indicate that different parameters result in different models, each of which may provide a different way of mapping inputs to label predictions. However, when thinking about how to manipulate f_w using algebra and calculus, note that writing $f_w(x_i)$ is nearly[14] equivalent to writing $f(x_i, w)$. That is, $f_w(x_i)$ corresponds to a function with two arguments, w and x_i.

14: The two forms are not perfectly equivalent. For example, $df_w(x_i)/dx_i$ is a valid way of writing the derivative of the function f_w. However, $df(x_i, w)/dx_i$ is not correct notation because this is a partial derivative of f, not a total derivative, and so it should be written as $\partial f(x_i, w)/\partial x_i$ (notice the difference between d and ∂). For the purposes of this introductory course, you may ignore these differences.

Notice that model parameters differ from hyperparameters. Whereas hyperparameters are set by a mechanism external to the agent (typically a person), model parameters are learned by the agent based on its observations (the available data). That is, hyperparameters change *how* the agent adjusts its model parameters. In fact, we refer to the process of an agent adjusting its parameters based on the available data as **learning**. Hence, when an agent *learns*, it adjusts the parameters of its model based on the data that it observes, while hyperparameters are set before the agent learns and shape how the agent learns.

Notice that the nearest neighbor algorithms had hyperparameters, but not model parameters. Algorithms like these, which do not use a fixed parametric form for the model, are called **nonparametric** methods. By contrast, algorithms that do assume a particular parametric form for the model (a formula for how labels are generated from model parameters w) are called **parametric** methods.

Also, notice that the choice of model parameterization—how changes to the model parameters w change the model's predictions—is itself a hyperparameter. For some problems we might guess that a simple parameterized model like a line will be effective, while for other problems we might guess that a more complicated parameterized model will be needed. Either way, the decision of how to parameterize the model is a decision generally made by the person applying a parametric ML algorithm, and is therefore a hyperparameter.

Regression II 3

3.1 Linear Parametric Models

We will begin by focusing on **linear models**, which are models of the form:

$$f_w(x_i) := w_1 x_{i,1} + w_2 x_{i,2} + \cdots + w_m x_{i,m} \tag{3.1}$$
$$= \sum_{j=1}^{n} w_j x_{i,j} \tag{3.2}$$
$$= w \cdot x_i \tag{3.3}$$
$$= w^\intercal x_i, \tag{3.4}$$

where (3.3) and (3.4) are alternative ways of writing the same concept (taking the sum of the products of the weights and features). These alternate forms come from linear algebra, and this course does *not* assume any familiarity with linear algebra. Still, we may in some cases use these forms because they are shorter to write. So, although we will not make use of techniques from linear algebra, we may make use of this shorthand notation—particularly (3.4).[1]

1: The $^\intercal$ symbol in $a^\intercal b$ is called *transpose*—this is not an exponent. Rather, this symbol indicates that a vector or matrix should be rotated by 90 degrees, turning the rows into columns and the columns into rows. If we assume that a and b are column vectors, which we can think of as $k \times 1$ matrices for some k, then $a^\intercal$ converts a into a $1 \times k$ matrix. So, $a^\intercal b$ corresponds to multiplying a $1 \times k$ matrix by a $k \times 1$ matrix, which is equal to $\sum_{i=1}^{k} a_i b_i$.

Including an Offset Feature

Notice that the linear model, $f_w(x_i) = w^\intercal x_i$ from (3.1), is not yet equivalent to the $\hat{y}_{n+1} = m x_{n+1} + b$ form of (2.8) because the former does not include a y-intercept parameter (b). That is, consider the case where $m = 1$, so that $f_w(x_i) = w_1 x_{i,1}$. When $x_{i,1} = 0$, this linear model always predicts $\hat{y}_{n+1} = 0$ regardless of how the model parameter is set; i.e., the line always passes through the y-axis at a height of zero.

To include a y-intercept parameter, which specifies how high the line should be when passing through the y-axis, we can change our data set, appending a one to every feature vector. That is, we increment the number of features, $m \leftarrow m + 1$, and define the new feature to be 1 for every data point: $x_{i,m} = 1$ for all $i \in \{1, \ldots, n\}$.

With this change, the previous $m = 1$ example now has $m = 2$, and we obtain:

$$f_w(x_i) = w_1 x_{i,1} + w_2 x_{i,2} \tag{3.5}$$
$$\tag{3.6}$$
$$= w_1 x_{i,w} + w_2 1 \tag{3.7}$$
$$= w_1 x_{i,1} + w_2. \tag{3.8}$$

So, w_1 is the slope of the line and w_2 is the y-intercept. We use this form for $f_w(x_i)$ to avoid having to constantly handle the y-intercept parameter as a special case: it is a model parameter like any other.

3.2 Defining "Best Fit"

Once a parametric model has been chosen, like the linear model that we are considering so far, the remaining challenge is to search for settings of the model parameters that result in good predictions. That is, which model parameters result in the "best fit" of the parameterized model to the training data? In order to answer this question, we need to precisely define what "best fit" means.

Intuitively, we might define the quality of a model based on how far its predictions are from the labels. Let us define a function, $l : \mathbb{R}^{|w|} \to \mathbb{R}$, so that $l(w)$ is a measure of how *bad* the parametric model is when using model parameters w. Because larger values of $l(w)$ correspond to worse fits, we call l a **loss function**. Once we have chosen a specific definition for the loss function l, we can then search for the model parameters, w, that minimize the loss function (i.e., the parameters that result in the best fit).

First, we focus on precisely defining the loss function l, thereby defining "best fit." How good or bad a parametric model fits a data set will likely depend on how far the predicted labels are from the labels in the data set, so this quantity is worth naming. Let a **residual**, r_i, be the difference between the model's prediction and the label in the data set:

$$r_i = y_i - \hat{y}_i \tag{3.9}$$
$$= y_i - f_w(x_i). \tag{3.10}$$

Sometimes people call these residuals *errors*, but technically a prediction's error is a slightly different quantity that we will not discuss here.[2]

2: For the curious reader, errors and residuals differ when the labels in the training data are themselves noisy. If y_i^* is the actual label for x_i but y_i is the label in the training data, then the *residual* is $y_i - \hat{y}_i$ while the *error* is $y_i^* - \hat{y}_i$.

Notice that the choice of loss function is yet another hyperparameter. For some applications, the choice of loss function may not be critical, like when roughly fitting a line to data points for a noncritical application. However, in other cases, the loss function must be carefully constructed. For example, if the ML algorithm is predicting how far a landslide will travel based on features of a hill in order to inform how far houses should be built from a slope (a real use of regression [10]), then underpredictions are far more severe than overpredictions, and this difference in severity should be incorporated into the loss function.

[10]: Jibson (2007), 'Regression models for estimating coseismic landslide displacement'

For now, we will construct a general purpose loss function that is not customized to a specific application. At first, we might consider using the sum of the residuals as the loss function:

$$l(w) = \sum_{i=1}^{n} r_i \tag{3.11}$$
$$= \sum_{i=1}^{n} (y_i - \hat{y}_i). \tag{3.12}$$

However, this loss function is flawed: it encourages models with large negative residuals. In general, negative residuals are just as bad as positive residuals (over- and underpredicting are equally bad). One fix would be to take the absolute value of the residuals to explicitly ignore the sign:[3]

3: Recall that $|z|$ is the cardinality (size) of z if z is a set, but the absolute value of z if z is a real number.

$$l(w) = \sum_{i=1}^{n} |r_i|. \tag{3.13}$$

While this is a reasonable choice, a more common loss function penalizes larger residuals more than smaller residuals. That is, what if a residual of 2 is twice as bad as a residual of 1, but a residual of 4 is more than twice as bad as a residual of 2? To account for this, we can use squared residuals (which are also never negative) rather than the absolute value of the residuals:

$$l(w) = \sum_{i=1}^{n} r_i^2 \tag{3.14}$$

$$= \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3.15}$$

$$= \sum_{i=1}^{n} (y_i - f_w(x_i))^2 . \tag{3.16}$$

This loss function, called the **least squares** loss, is extremely popular. Hereafter, unless otherwise specified, for regression algorithms we use the symbol l to refer to the least squares loss.[4]

4: The following comment is beyond the scope of this introductory course, but provided as a connection for the curious reader. The least squares loss is more than just a reasonable heuristic constructed as we have shown. If one assumes that residuals are normally distributed, then the model parameters that maximize the probability of the observed labels are the model parameters that minimize the least squares loss. Using terminology that we have not defined, the model parameters that minimize the least squares loss result in the *maximum likelihood* model [11, Page 29].

3.3 Optimization Perspective of Regression

Having defined a parameterized model f_w and loss function l, we can now formally define the problem of searching for model parameters, $w*$, which result in the best fit:

$$w^* \in \arg\min_w l(w). \tag{3.17}$$

This usage of the * superscript is common to indicate an optimized value of a symbol and is pronounced "star," as in "double-u star."

To avoid confusion, we will review some common points of confusion regarding (3.17). First, recall that $\min_w l(w)$ would return the minimum value that $l(w)$ takes for any value of w. Similarly, $\arg\min_w l(w)$ returns a w that causes $l(w)$ to achieve this minimum value. That is, $\arg\min_w l(w)$ "is the model parameter vector that minimizes the loss function."

I put the last sentence in quotes because, although it provides the correct intuition, it is not actually correct. What if there are many values of w that cause $l(w)$ to be minimized? To handle this situation, the arg min operator returns a set containing all w that cause $l(w)$ to be minimized. In some cases there may be only one element in this set.[5] Nevertheless, arg min returns a *set*, and so (3.17) uses the $\in$ symbol, rather than $=$, to indicate that w^* is an element of this set, not the entire set of model parameters. Returning to the last sentence of the previous sentence, we can now state the proper claim: That is, $\arg\min_w l(w)$ is the set of all model parameters that minimize the loss function.

5: A set with a single element is called a **singleton**.

The expression in (3.17) is a common type of expression called an *optimization problem*. In general, optimization problems have the following

form (here we temporarily redefine the symbols x, $\mathcal{X}$, and f):

$$\arg\min_{x \in \mathcal{X}} f(x), \tag{3.18}$$

where f is called the **objective function** and $\mathcal{X}$ is the set of possible values for x, called the **feasible set**.

So, we have converted regression problems using parametric models into optimization problems. As you will see throughout this course, nearly all ML problems (including regression, classification, and even reinforcement learning problems) can be formulated as optimization problems. In general, this optimization problem is of the form: find model parameters that optimize (minimize or maximize) an objective function that quantifies how good or bad model parameters are. The many different ML problem settings, from regression to reinforcement learning and even classification algorithms designed to avoid producing racist and sexist behaviors, correspond to different types of objective functions and assumptions about what is known about the objective function and feasible set.

Soon we will discuss optimization algorithms for finding or approximating solutions to the optimization problem in (3.17). However, first note that *any* algorithm that returns a w^* satisfying (3.17) using the least squares loss is called a **least mean squares** (LMS) algorithm. In summary, an LMS algorithm is a regression algorithm that returns model parameters that minimize the least squares loss:

$$w^* \in \arg\min_{w} \sum_{i=1}^{n} (y_i - f_w(x_i))^2, \tag{3.19}$$

and *linear* LMS additionally assumes a linear parametric model:

$$w^* \in \arg\min_{w} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{m} w_j x_{i,j} \right)^2, \tag{3.20}$$

where a y-offset parameter can be incorporated by appending a feature to each x_i that always takes the value one. The linear LMS fit to the base GPA data set is depicted in Figure 3.1.

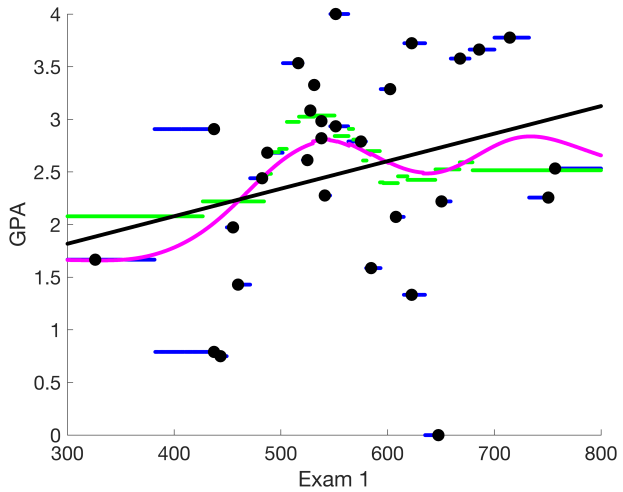


Figure 3.1: Like Figure 2.4, but with the linear LMS fit plotted as a solid black line. Recall that the magenta predictions correspond to weighted k nearest neighbor with $k = 20$ and $\sigma = 50$, the green predictions correspond to k-NN with $k = 10$, and the blue predictions correspond to k-NN with $k = 1$.

3.4 Evaluating a Model

Consider using an ML method like k-nearest neighbors to make predictions for a real application, like predicting whether a tumor is benign or malignant [12] or how far a landslide will travel. In these and most other real applications, using a model that makes inaccurate predictions could be costly or dangerous. So, we need a way to evaluate how good the predictions of a model would be if the model were to be used to make predictions for new and unseen data points.

[12]: Abdel-Zaher et al. (2016), 'Breast cancer classification using deep belief networks'

We will start with a dangerous method for evaluating a model. Let $D = (x_i, y_i)_{i=1}^n$ be the available data, which is used to train a model, for example by finding $w^\star \in \arg\min_w \sum_{i=1}^n (y_i - f_w(x_i))^2$ or by using D as the data for weighted k-nearest neighbors. To evaluate the resulting model, we might report the least squares loss of the model on D:

$$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{3.21}$$

This approach has two problems. The first, and more minor, is that the least squares loss is not an easily interpreted value. For the GPA prediction problem, would a loss of 7,000 be good or bad? To answer this question one would need to know n. The GPA data set had $n = 43{,}303$ points, and so a least squares loss of 7,000 equates to an averaged squared residual of approximately 0.16, which is actually very good. To make the reported loss easier for a person to interpret without knowing n and doing quick mental calculations, we might instead report the *average* squared loss, which is also called the **sample mean squared error** (sample MSE):

$$\text{sample MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{3.22}$$

Notice that the sample MSE reports the average *squared* residual. For many people it is more natural to think about the magnitude of the residual rather than the squared residual. We can obtain a quantity scaled similarly to the magnitude of the average residual by taking the square root of the sample MSE. This results in the **sample root mean squared error** (sample RMSE):

$$\text{sample RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}. \tag{3.23}$$

Though there are many different statistics that can be used to measure how accurate the predictions of a regression algorithm are, the sample RMSE is by far the most common.

The second problem with the approach described above is far more serious. Consider what would happen if the data set contains no duplicate points and if we were to evaluate the weighted k-nearest neighbor algorithm with $k = 1$. For the i^{th} point in the data set, the algorithm would search for the single nearest neighbor, which would be the i^{th} point. It would then output the label for this point, which would be precisely the correct label. So, the resulting residuals would all be zero! Hence, the sample RMSE would be zero, suggesting that the model

always makes perfect predictions. Obviously this is not correct—when applied to new data points not in D, the algorithm's predictions will not always be exactly correct.

The problem is that we are training and testing the ML algorithm using the same data. This usually results in over-predictions of how good the model's predictions will be. This property is related to the topic of **overfitting**, which we will cover later in the course. This property applies to all ML algorithms, not just nearest neighbor algorithms. As another example, consider an LMS algorithm using a linear parametric model. If $m = 1$ and D contains two data points, this corresponds to fitting a line to two points. The best fit line will pass precisely through the two points resulting in both residuals being zero. However, for new data points not in D the average residual would likely be far larger than zero. For linear LMS this over-prediction of the accuracy of the model's predictions becomes less severe as the amount of data grows, but it never entirely goes away.

To address this issue, practitioners should partition the available data D into two sets, a training set D_{train} and a testing set D_{test}. The model should be trained using D_{train} only. For example, the best weights $w^\star$ for a parametric model should be computed from D_{train} and the predictions of a weighted k-nearest neighbors model should be computed from D_{train} only. To evaluate how good the resulting model is, one can compute the sample RMSE using only the data in D_{test}. Intuitively, using all of D for both training and testing is like giving the agent the exact test that will be used to evaluate its performance, and so it can "cheat" by memorizing the answers to the test without really understanding the concepts in a way that allows it to apply its knowledge to future problems. As a result, the agent's score on the test is not representative of its actual ability to solve future problems. By splitting D into D_{train} and D_{test} and only allowing the agent to observe D_{train} prior to the test, which uses D_{test}, we prevent the agent from "cheating" in this way.

Lastly, one must be careful how the data points in D are assigned to D_{train} and D_{test}. An unscrupulous practitioner might cherry-pick which data points are placed in each set, for example, placing points that are easier to make accurate predictions for into D_{test}. This would again result in an over-estimation of the model's performance. To prevent the partitioning of D into D_{train} and D_{test} from inserting bias in this way, it is common practice to randomly select which points are placed into each set.[6]

In summary, to evaluate a model, practitioners should partition the available data D into two sets, D_{train} and D_{test}, randomly determining which points are each each set. Next, the model should be trained using D_{train} only, and the sample RMSE from (3.23) should then be computed and reported using D_{test} only.

This raises the question: How much data should be placed in D_{train} and how much in D_{test}? This is a hyperparameter (in this case a hyperparameter that changes how the algorithm is evaluated, not what model the algorithm produces). Common splits include placing $60\% - 80\%$ of the data in D_{train} and the remainder in D_{test}, though the appropriate split will depend on the exact setting. For example, if there is little data and knowing precisely how accurate the model will be is more important

6: Though one should typically avoid looking at the values of x_i and y_i when determining whether to place the point in D_{test} or D_{train}, there is one exception: **stratified sampling**. Notice that randomly deciding which points should be placed in the training set and which should be placed in the testing set could result in precisely the biased split that the aforementioned unscrupulous practitioner might create. We therefore might want to ensure that the random split preserves some properties of the data. For example, if each data point corresponds to a person and $\alpha\%$ own a car, we might want to ensure that roughly $\alpha\%$ of the points in both D_{train} and $\alpha\%$ of the points in D_{test} correspond to people who own cars. If the random split happens to result in D_{train} containing all of the points that correspond to people who own cars and D_{test} contains all of the points that correspond to people who do not own cars, that would likely not result in an accurate evaluation. Stratified sampling is a technique for ensuring that such properties of D are preserved in D_{train} and D_{test}.

than obtaining a slightly more accurate model (e.g., for safety critical applications), one might place significantly more data in D_{test}.

Regression III

4

4.1 Black Box Optimization

Black box optimization (BBO) algorithms are designed to solve problems of the form:

$$w^* \in \arg\min_w l(w), \tag{4.1}$$

when little information about l is available. They are called "black box" because the objective function, l, is thought of as an opaque box that we cannot open to see its inner workings; we can only provide inputs to it and observe its outputs. That is, we can query the value of $l(w)$ for any input w, and our goal is to find an input w^* that results in the minimum possible output, or at least a value close to the minimum.

There are many different BBO algorithms including genetic algorithms, hill climbing, simulated annealing, the cross entropy method, the covariance matrix adaptation evolutionary strategy (CMA-ES), finite difference methods like simultaneous perturbation stochastic approximation, and Bayesian optimization methods. Here we focus on a simple example: a variant of **hill climbing**.

The idea behind hill climbing algorithms is to create a sequence of inputs to the loss function, i.e., a sequence of model parameters, that gets "better" with respect to the loss function as the sequence progresses. To discuss sequences of model parameter vectors, we have to change our model parameter notation: Let $w_0, w_1, \ldots$ be a sequence of model parameter vectors, such that w_0 is an arbitrary initial guess of model parameters that might minimize the loss function. Given some model parameter vector, w_k, we will specify a procedure for computing the next vector in the sequence, w_{k+1}, such that $l(w_{k+1}) < l(w_k)$.

In our earlier discussion, the subscripts on w indicated a particular weight within one vector of model parameters, while now w_k is itself an entire vector of model parameters. Therefore, we now write $w_{k,j}$ to reference the j^{th} weight within the k^{th} vector of model parameters.

Intuitively, the hill climbing algorithm selects w_{k+1} by first randomly sampling potential new model parameters, w', from around the current model parameters w_k. Next, it checks whether $l(w') < l(w_k)$. If so, $w_{k+1} = w'$, and if not, it samples a new w' and repeats the process. Pseudocode for this variant of hill climbing is provided in Algorithm 4.1.

Notice that hill climbing has multiple hyperparameters: the initial vector, the sampling distribution, and the stopping criterion. How would you set these? One choice for w_0 is the zero vector: $w_{0,j} = 0$ for all $j \in \{1, \ldots, m\}$. There are many options for the sampling distribution, d, which changes how the algorithm picks a point near w_k to test next. For example, one

might use the continuous uniform distribution on some interval $[-c, c]$:

$$d = U(-c, c). \tag{4.2}$$

Another option is a mean-zero normal distribution with variance σ^2:

$$d = N(0, \sigma^2). \tag{4.3}$$

In either of these cases, the chosen distribution d incurs an extra hyperparameter, either c or σ^2.

Finally, the last hyperparameter of hill climbing is a stopping criterion, `stop`, which is used to determine when the algorithm should stop trying to find even better solutions. Common stopping criteria include:

1. Stopping after some allowed maximum time has passed.
2. Stopping when recent decreases in $l(w_k)$ are small.
3. Stopping when enough w' have been tested without resulting in a new w_{k+1}.

Notice that all of these criteria introduce even more hyperparameters, highlighted in red.

Algorithm 4.1: Hill Climbing

Input : Black box loss function $l : \mathbb{R}^m \to \mathbb{R}$, dimension m of domain of l.
Output : An approximation of an element of $\arg\min_w l(w)$.
Hyperparameters : Initial model parameters w_0, sampling distribution d, stopping criterion `stop`.

$k \leftarrow 0$;
while *stopping criterion* `stop` *not satisfied* **do**
  /* Sample a nearby point, one feature at a time. */
  for $j \leftarrow 1$ **to** m **do**
    $\eta_j \sim d$; // Sample noise.
    $w'_j \leftarrow w_{k,j} + \eta_j$; // Add noise to current point to obtain candidate for next point.
  end
  /* Check if w' is better than w_k. */
  if $l(w') < l(w_k)$ **then**
    /* w' is better, so make it w_{k+1} and increment k. */
    $w_{k+1} = w'$;
    $k = k + 1$;
  end
end
return w_k;

While BBO algorithms can be simple, like hill climbing, they can also be quite sophisticated, like the popular CMA-ES algorithm [13] and its many newer variants. However, by design, BBO algorithms do not use information about the inner workings of the loss function. While this makes BBO algorithms effective for optimization problems wherein one does not have any additional information about l, in our case we know the exact form of l (think back to (3.16)). In the next section, we investigate

[13]: Hansen (2006), 'The CMA evolution strategy: a comparing review'

how we can use our knowledge about l to improve our hill climbing algorithm so that it selects w' more intelligently.

4.2 Gradient Descent

We know more than just how to query the value of l for any input, w—we have an exact expression for $l(w)$. The idea behind gradient descent is that we can use this additional knowledge about l to do better than randomly sampling a w' close to w_k and checking to see if it is better. Instead, we can derive the exact direction of change to w_k that will result in the largest decrease in the loss function.

First, let us consider the direction of steepest descent of l at w_k. To visualize this direction, consider the case where $m = 2$. In this case $w_i = (w_{k,1}, w_{k,2})$, so we can think of the input as specifying the coordinates of a point on the floor of a room, where a corner at the floor is $(0, 0)$, and $w_{k,1}$ specifies how far to move down one wall and $w_{k,2}$ specifies how far to move down the other wall. We can then visualize l by taking every possible input coordinate, $w = (w_{k,1}, w_{k,2})$, computing $l(w)$, and plotting a point at a height of $l(w)$ above the floor at the position w. An example is provided in Figures 4.1 and 4.2. Some might equivalently visualize this as a landscape with mountains and perhaps valleys.

Now, when the algorithm is at w_k, it is at a position within the room, which results in the surface having some height. The **direction of steepest descent** of l at w_k is the direction that a ball would roll if placed at w_k on this surface—the steepest downhill direction. This direction is a **local** quantity.[1] That is, it only depends on the shape of $l(w)$ immediately around w_k, not its value elsewhere. Before continuing with our discussion of the direction of steepest descent, we briefly review *gradients*.

As you may recall from calculus classes, the **gradient** of l at w_k is itself a vector of length m (the same length as w_k), containing the partial derivatives of l with respect to each of the model parameters:

$$\nabla l(w_k) = \left(\frac{\partial l(w_k)}{\partial w_{k,1}}, \frac{\partial l(w_k)}{\partial w_{k,2}}, \ldots, \frac{\partial l(w_k)}{\partial w_{k,m}} \right). \tag{4.5}$$

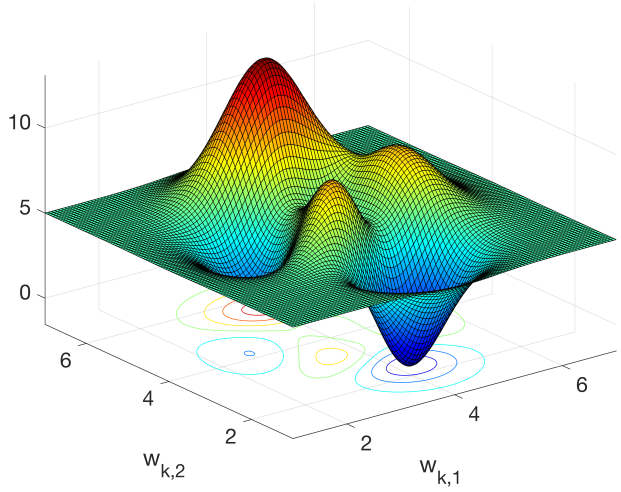


Figure 4.1: Example of what a loss function might look like, where the height of the surface corresponds to the value of the loss function at the specified location $w_k = (w_{k,1}, w_{k,2})$. The actual shape of the surface will depend on the data and chosen loss function. Underneath the surface is a **contour plot**, which is shown on its own and described in Figure 4.2.

1: Though the precise definition of the direction of steepest descent is beyond the scope of this course, we describe it here for the curious reader. Consider a circle of radius ϵ around w_k. Let w' be the point on this circle that causes $l(w')$ to be as small as possible. The direction from w_k to w' is the vector $w' - w_k$. If we consider the limit of this direction as $\epsilon \to 0$ (in the limit as the circle becomes infinitesimally small), we obtain the *direction of steepest descent*. This can be written mathematically as:

$$\lim_{\epsilon \to 0} \operatorname*{arg\,min}_{\Delta \in \mathbb{R}^m : \|\Delta\| = 1} f(w_i + \epsilon\Delta), \tag{4.4}$$

where $\|\cdot\|$ is called the l^2 norm. We can solve analytically for this direction by using a Taylor expansion to remove curvature terms and Lagrange multipliers to handle the constraint. Also, notice that this direction is *local* in that it only considers the value of l at points an infinitesimal distance from w_k.

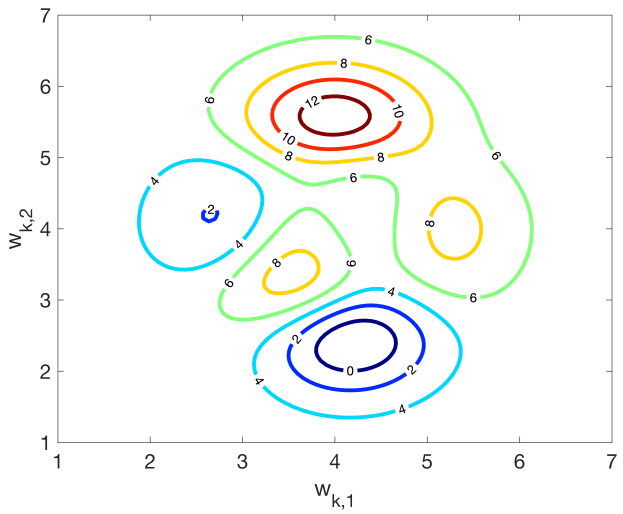


Figure 4.2: Contour plot for the loss function depicted in Figure 4.1. This plot is constructed by selecting several *levels* (heights of the function) and plotting all of the locations where the value of the loss function (height) is one of these levels. That is, all of the points of the same color form a **level set**—the set of all the points where the surface has some specific height. In this contour plot the level sets are labeled with their heights, though often these labels are omitted.

Recall that $\frac{\partial l(w_k)}{\partial w_{k,j}}$ is the *partial derivative of* $l(w_k)$ *with respect to* $w_{k,j}$. In other words, it is how quickly $l(w_k)$ increases if $w_{k,j}$ is increased. Intuitively, if you imagine $l(w_k)$ as having all of the inputs fixed at w_k except for one element, $w_{k,j}$, which is still allowed to vary as an input, you would obtain a new function that we will call g, which takes a real number as input and produces a real number as output. For example, $g(2)$ is the value of $l(w_k)$ if you replaced $w_{k,j}$ with the value 2. The partial derivative $\frac{\partial l(w_k)}{\partial w_{k,j}}$ is simply the slope of this function, g, at the location $w_{k,j}$. So, just as the slope of a function in one dimension tells us how quickly the function is rising, the gradient tells us how quickly the function increases when you change each of the inputs, from $w_{k,1}$ through $w_{k,m}$.

Given the gradient, which characterizes how the function changes locally around w_k as each of the inputs to l is increased, it is straightforward to compute the direction of steepest **ascent** (the steepest uphill direction): it is the gradient! This may actually be counter intuitive at first. Imagine the $m = 2$ setting again and the case where $\frac{\partial l(w_k)}{\partial w_{k,1}} = 10$ and $\frac{\partial l(w_k)}{\partial w_{k,2}} = 1$. Since increasing $w_{k,1}$ increases the loss function more quickly than increasing $w_{k,2}$, you might think that we should focus all of our efforts on changing $w_{k,1}$, and so the direction of steepest ascent would be to increase $w_{k,1}$ and not change $w_{k,2}$. However, this is not the case. Take a piece of paper and label the edges $w_{k,1}$ and $w_{k,2}$. Draw a dot in the middle, and label it w_k. How, hold up the paper so that the slope along the $w_{k,1}$ edge is ten times the slope along the $w_{k,2}$ edge. At the point w_k, which direction is the steepest uphill direction? You should notice that it is *not* just moving in the direction $w_{k,1}$, but that it also involves moving slightly in the $w_{k,2}$ direction. Specifically, for every ten units moved in the $w_{k,1}$ direction, the steepest uphill direction moves one unit in the $w_{k,2}$ direction.

Though it should be clear that the direction of steepest ascent does not solely focus on the largest element of the gradient, it may not be obvious that the gradient *is* the direction of steepest ascent. The proof of this result is beyond the scope of this course (but it was likely covered in your calculus courses, and we can discuss it during office hours if you are curious).

While the gradient is the direction of steepest *ascent*, we aim to minimize the loss function, and so we desire the direction of steepest *descent*.

However, these directions are opposites, and so the direction of steepest descent is simply the negative of the gradient.

Bringing these pieces together, we can now improve our hill climbing algorithm by making it automatically select a point w' by moving from w_k in the direction of steepest descent. That is, we perform the following update for all $j \in \{1, \ldots, m\}$:

$$w_{k+1,j} = w_{k,j} - \frac{\partial l(w_k)}{\partial w_{k,j}}. \tag{4.6}$$

However, recall that the direction of steepest descent is a *local* direction—it only characterizes the shape of l for inputs extremely close to w_k. If this update moves too far away from $w_{k,j}$, then it could result in the algorithm stepping over a basin in the surface and landing farther up on the other side! That is, the loss function could *increase* after one step if the step is sufficiently large. To avoid this, we scale down the size of the step using a *step size* $\alpha \in \mathbb{R}$. This new hyperparameter is generally a small positive number. The resulting update including the step size is, for all $j \in \{1, \ldots, m\}$:

$$w_{k+1,j} = w_{k,j} - \alpha \frac{\partial l(w_k)}{\partial w_{k,j}}. \tag{4.7}$$

In practice, α is not actually set to a small enough value to ensure that increases in the loss function never occur. This is because very small values of α result in very slow downhill movement, while large steps can result in overshooting. Picking a good value for α balances these trade-offs between the speed of improvement and the likelihood of frequently stepping too far, thus increasing the loss function. Pseudocode for the gradient descent algorithm is provided in Algorithm 4.2. The stopping criteria options for gradient descent are similar to those for hill climbing, though other possible stopping criteria include stopping when the magnitude of the gradient is small, indicating that further improvement will be slow and suggesting that the process might be approaching a minimum (where the gradient is zero).

Algorithm 4.2: Gradient Descent

Input : Loss function $l : \mathbb{R}^m \to \mathbb{R}$, dimension m of domain of l.
Output : An approximation of an element of $\arg\min_w l(w)$.
Hyperparameters : Initial model parameters w_0, step size $\alpha \in \mathbb{R}_{>0}$, stopping criterion `stop`.

$k \leftarrow 0$;
while *stopping criterion* `stop` *not satisfied* **do**
  `/* Compute next point in the sequence. */`
  for $j \leftarrow 1$ **to** m **do**
    $w_{k+1,j} = w_{k,j} - \alpha \frac{\partial l(w_k)}{\partial w_{k,j}}$;
  end
  $k = k + 1$;
end
return w_k;

Regression 4 | 5

5.1 Gradient Descent Review

[In this review section we ignore the notation used so far and begin by discussing general properties of functions f of inputs x.]

Let x be a vector in $\mathbb{R}^m$ and let $f : \mathbb{R}^m \to \mathbb{R}$. That is, f is a function that takes a vector of length m as input and produces a real number as output. Recall that the gradient is the concatenation of the partial derivatives:

$$\nabla f(x) = \frac{df(x)}{dx} = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_m} \right). \tag{5.1}$$

Recall that here x, $\nabla f(x)$, and $\frac{df(x)}{dx}$ are all vectors with m elements. Also, each x_i and each $\frac{\partial f(x)}{\partial x_i}$ is a single real number.

There are at least two ways of interpreting each of the partial derivatives $\frac{df(x)}{dx_i}$, where $i \in \{1, \ldots, m\}$.

1. If all of the inputs except for x_i are held constant and we view $f(x)$ only as only a function of x_i, what is the slope?
2. How should x_i be changed to increase $f(x)$ as much as possible?

There are also at least two ways of interpreting $\nabla f(x)$ or $df(x)/dx$.

1. It is the direction of steepest ascent of the function f at the location x.
2. It is the direction to change x to increase $f(x)$ as much as possible.

This second interpretation is worth emphasizing, as we will return to this interpretation again later in the course. In general, when you see an expression $\frac{d\square}{d\triangle}$, you can think of this as an expression for how $\triangle$ should be changed in order to make $\square$ larger. For example, $\frac{df(x)}{dx_i}$ is how x_i should be changed to make $f(x)$ larger, $\frac{df_w(x)}{dw_j}$ is how w_j should be changed to make $f_w(x)$ larger, and $\frac{df_w(x)}{dx_j}$ is how x_j should be changed to make $f_w(x)$ larger.

With this intuition, the gradient descent algorithm produces the next point, x_{k+1} from the current point, x_k, by taking a small step in the opposite of the direction of steepest ascent—a small step in the direction of steepest *descent*:

$$\forall j \in \{1, \ldots, m\},\ x_{k+1,j} = x_{k,j} - \alpha \frac{\partial f(x_k)}{\partial x_{k,j}}. \tag{5.2}$$

Or, in vector notation:

$$x_{k+1} = x_k - \alpha \frac{\partial f(x_k)}{\partial x_k}. \tag{5.3}$$

In this course, we apply this technique to the loss function, l rather than f, which takes as input weights w_k rather than x_k.

To practice working with gradient descent, consider an example of trying to find the minimum of the function

$$f(z) = 17z_1^2 - 7z_1z_2 + 14z_2^2, \tag{5.4}$$

where $z = (z_1, z_2)$. Notice that here we have used z as the input to f rather than x. This is to point out that the symbol used as an input to f is merely a symbol. For example, when programming you might define a function `sum(int a, int b)` which you later might call with `x=5; y=7; z=sum(x,y)`. The choice of a and b as the symbols used as the inputs to `sum` is arbitrary, and the function `sum` can be called using different symbols, like `sum(x,y)`. The same goes for these math definitions—here z is the symbol we chose to use when defining f, but later we might refer to $f(x)$.

Now, consider running gradient descent starting from $x_0 = (5, 10)$ and using a step size of $\alpha = 0.01$. What is the next point, x_1? See if you can work this out on your own before continuing reading. The first step is to compute the gradient:

$$\nabla f(z) = \frac{df(z)}{dz} = \left(\frac{\partial f(z)}{\partial z_1}, \frac{\partial f(x)}{\partial z_2}\right). \tag{5.5}$$

Again, in the above expression you could choose to use any symbol as the input to f, even writing $\nabla f(☺)$ and $\frac{\partial f(☺)}{\partial ☺_1}$. The point is that we require the gradient of f—the partial derivatives of f with respect to each of its inputs. We solve for each partial derivative:

$$\frac{\partial f(z)}{\partial z_1} = 3z_1 - 7z_2 \tag{5.6}$$

and

$$\frac{\partial f(z)}{\partial z_2} = -7z_1 + 28z_2. \tag{5.7}$$

So, the gradient of f at $z = (z_1, z_2)$ is

$$\nabla f(z) = (3z_1 - 7z_2, -7z_1 + 28z_2). \tag{5.8}$$

Simply changing the symbol we use as input to f from z to x_k, we obtain the gradient:

$$\nabla f(x_k) = (3x_{k,1} - 7x_{k,2}, -7x_{k,1} + 28x_{k,2}). \tag{5.9}$$

This expression tells us the direction of steepest *ascent* of the function f at any point x_k. For example, the direction of steepest ascent of f at the location $(1, 2)$ is $(-11, 49)$. This is a "direction" in that we should decrease the first element (since -11 is negative) and increase the second element (since 49 is positive). Furthermore, for every 11 units we decrease the first element, we should increase the second element by 49. So, a small step in the direction $(-11, 49)$ from the point $(1, 2)$ could result in a new point like $(1 + 0.01 \times -11, 2 + 0.01 \times 49) = (0.89, 2.49)$. However, this would be a step in the direction of the gradient—a step to *increase* f.

In this example problem, we are running gradient *descent* not *ascent*. Starting from the location $(5, 10)$, can you work out the direction of steepest *descent*? The answer to this question will be given soon.

The gradient descent update is:

$$x_{k+1,1} = x_{k,1} - \alpha\,(34x_{k,1} - 7x_{k,2}) \tag{5.10}$$
$$x_{k+1,2} = x_{k,2} - \alpha\,(-7x_{k,1} + 28x_{k,2})\,. \tag{5.11}$$

Plugging in $x_0 = (5, 10)$ and $\alpha = 0.01$, we can solve for x_1:

$$x_{1,1} = 5 - 0.01((34 \times 5) - (7 \times 10)) = 4 \tag{5.12}$$
$$x_{1,2} = 10 - 0.01((-7 \times 5) + (28 \times 10)) = 7.55. \tag{5.13}$$

So, starting from $x_0 = (5, 10)$, the direction of steepest descent of f is $(-100, -245)$, and a small step in this direction using a step size of $\alpha = 0.01$ results in $x_1 = (4, 7.55)$. Can you run one more iteration of gradient descent? If you did so properly, you should get $x_2 = (3.1685, 5.716)$.

5.2 Gradient Descent for Least Squares Linear Regression

To complete the least squares regression algorithm using a linear model, we must compute the partial derivatives, $\partial l(w_k)/\partial w_{k,j}$ used in Algorithm 4.2:

$$\frac{\partial l(w_k)}{\partial w_{k,j}} = \frac{\partial}{\partial w_{k,j}} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5.14}$$
$$= \frac{\partial}{\partial w_{k,j}} \sum_{i=1}^{n} (y_i - f_{w_k}(x_i))^2 \tag{5.15}$$
$$= \sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i)) \frac{\partial}{\partial w_{k,j}} (y_i - f_{w_k}(x_i)) \tag{5.16}$$
$$= -\sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i)) \frac{\partial}{\partial w_{k,j}} f_{w_k}(x_i) \tag{5.17}$$
$$= -\sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i)) \frac{\partial}{\partial w_{k,j}} \sum_{\beta=1}^{m} w_{k,\beta} x_{i,\beta} \tag{5.18}$$
$$= -\sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i)) \frac{\partial}{\partial w_{k,j}} w_{k,j} x_{i,j} \tag{5.19}$$
$$= -\sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i)) x_{i,j}. \tag{5.20}$$

Note that some sources include a factor of $1/2$ in the least squares loss function, specifically so that it cancels with the 2 in these partial derivatives. This scaling factor scales the entire gradient, and so it can actually be viewed as a modification of the step size.

Bringing together the pieces that we have developed, we obtain Algorithm 5.1, which uses gradient descent to search for weights that minimize the least squares loss function. Note that this algorithm hard-codes some hyperparameters, like the decision to use a linear model and the

least squares loss function. Do you think these should be viewed as hyperparameters of this algorithm?

Algorithm 5.1: Linear Least Squares Regression Using Gradient Descent

Input : Data set $(x_i, y_i)_{i=1}^n$, where each $x_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$, additional input x_{n+1}.
Output : A prediction, $\hat{y}_{n+1}$, of the label for x_{n+1}.
Hyperparameters : Initial input w_0, step size $\alpha \in \mathbb{R}_{>0}$, stopping criterion `stop`.

$k \leftarrow 0$;
while *stopping criterion* `stop` *not satisfied* **do**
  `/* Compute next point in the sequence. */`
  for $j \leftarrow 1$ **to** m **do**
    $w_{k+1,j} = w_{k,j} + \alpha \sum_{i=1}^{n} 2(y_i - f_{w_k}(x_i))x_{i,j}$;
  end
  $k = k + 1$;
end
return $f_{w_k}(x_{n+1})$;

Regression 5 | 6

6.1 Convergence of Gradient Descent

Recall that the direction of steepest descent is a *local* direction. When standing on the side of a valley, if too large of a step is taken in the direction of steepest descent, one might step all the way over the valley and up the other side, resulting in upwards movement—an increase in the loss function. To counter this, we suggested that the step size should be small. However, what if the steps are too small? Recall that the gradient approaches zero as the solution approaches a local optimum. Since the step size is multiplied by the gradient, if the step size is small and the gradient converges to zero, will the steps shrink so fast that the algorithm is unable to reach the goal?

Convergence proofs aim to answer this question, providing us with conditions under which gradient descent converges. There are many different proofs that gradient descent converges, with different required conditions and different types of convergence guarantees (which answer the question: converges to what?).

Common conditions for convergence include various subsets of the following conditions:

- The function l must be differentiable everywhere, since otherwise $\frac{\partial l(w_k)}{\partial w_{k,j}}$ may not be defined.
- The function l must be **Lipschitz continuous** with constant L. Given the above condition, this equates to requiring the slope of l to be at most L.
- The step size is sufficiently small. For example, one condition is that $\alpha < 1/L$, where L is the Lipschitz constant discussed above. An alternative is to decay the step size, α, after each step. Let α_k be the step size during the k^{th} step (iteration) of gradient descent. Then, a common assumption is that the step size sequence is square summable (which ensures that the step size decays to be sufficiently small):
$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty, \tag{6.1}$$
but also that the step size sequence is not summable (which ensures that the step sizes are big enough to allow the algorithm to move as far as needed to reach an optimum):
$$\sum_{k=1}^{\infty} \alpha_k = \infty. \tag{6.2}$$
A common step size sequence that is square summable but not summable is $\alpha_k = \frac{1}{k}$.
- The *gradient*, ∇l, is Lipschitz continuous.

- The loss function, l, is convex, quadratic, strongly convex, or satisfies some other similar property.

Before giving examples of common convergence guarantees, recall that in this setting with a smooth objective function, a **global minimum** is any weight vector that actually minimizes the loss function—any element of $\arg\min_w l(w)$. A **local minimum** is any weight vector from which there are no descent directions—all directions of change to the weights result in a local increase (or no change) to the objective function. That is, w is a local minimum of l if $\nabla l(w)$ is the zero vector. Notice that every global minimum is a local minimum, but not every local minimum is necessarily a global minimum. Also, there can be multiple global minima.

Common types of convergence guarantees include:

- Convergence to a global minimum w^*, i.e., $w_k \to w^\star \in \arg\min_w l(w)$.
- Convergence to a local minimum, i.e., $\nabla l(w_k) \to 0$.
- Convergence to a local minimum ($\nabla l(w_k) \to 0$) or divergence of the loss function to $-\infty$, i.e., $l(w_k) \to -\infty$.

For example, in our case the gradient of the loss function is Lipschitz continuous and the loss function is bounded below by zero. So, if the step size sequence is square summable but not summable, then $\nabla l(w) \to 0$ [14]. Since the least squares loss function using a linear parametric model is convex, this local minimum is necessarily a global minimum, and so we have that $w_k \to w^\star \in \arg\min_w l(w)$.

[14]: Bertsekas et al. (2000), 'Gradient convergence in gradient methods with errors'

6.2 Convergence Intuition

Some of the assumptions might appear mystical. In this section we show how Lipschitz assumptions and assumptions relating the step size to the Lipschitz constant can occur. However, we do not provide a complete convergence proof.

Consider the one-dimensional setting ($m = 1$). We make the following assumptions

1. l is continuous and twice differentiable. That is, $\frac{\partial^2 l(w)}{\partial w^2}$ exists for all w.
2. $\nabla l(w)$ is Lipschitz with constant L. That is, the rate of change of the gradient (first derivative) is bounded by L. Put differently, the magnitude of the slope of the slope is bounded by L. Saying the same thing one other way, for all w, $\left|\frac{\partial^2 l(w)}{\partial w^2}\right| \leq L$, where $|\cdot|$ denotes absolute value.
3. $\alpha = 1/L$.

We will provide an intuitive argument (not a formal proof) that the gradient descent update:

$$w_{k+1} = w_k - \alpha \nabla l(w_k), \tag{6.3}$$

cannot cause an increase in the loss function. That is, it cannot occur that $l(w_{k+1}) > l(w_k)$. This alone is not a complete proof of convergence to a local or global optimum—for example setting $\alpha = 0$ also satisfies this

property but clearly does not result in convergence to a local or global optimum.

We consider two cases:

- Case 1: $\nabla l(w_k) = 0$. In this case the update $w_{k+1} = w_k - \alpha \nabla l(w_k)$ degenerates to $w_{k+1} = w_k$, and so $l(w_{k+1}) = l(w_k)$, satisfying the desired condition. Intuitively, this says that, if gradient descent is already at a local optimum it will not move off of the local optimum.
- Case 2: $\nabla l(w_k) \neq 0$. Recall that $-\nabla l(w_k)$ is a descent direction. This means that a "small enough" step in this direction will decrease the loss function. However, how small is "small enough"?

  First, notice that for a step to be "too big," resulting in the loss function *increasing*, the step must be big enough that somewhere between w_k and w_{k+1} the gradient changed sign. If we think of the function as being a surface and w_k as being a point on the side of a valley, the downhill step could be too big, taking us up the other side of the valley. However, for this to happen the step must cross the bottom of the valley.

  This means that for $l(w_{k+1})$ to be greater than $l(w_k)$, between w_k and w_{k+1} the slope must change sign (it might change sign multiple times—stepping over multiple valleys, but it must change sign at least once). Notice that the assumption that $\nabla l(w_k)$ is Lipschitz with constant L means that the maximum rate of change of the slope is L. If the current slope (gradient) is $\nabla l(w_k)$, and the slope can change at a rate of at most L, how far can we move away from w_k before the gradient could be reduced all the way to zero? The answer is $|\nabla l(w_k)|/L$.

  Notice that with $\alpha = 1/L$, the gradient descent update is:

$$w_{k+1} = w_k - \frac{\nabla l(w_k)}{L}. \tag{6.4}$$

  Hence, the change from w_k to w_{k+1} is precisely $\nabla l(w_k)/L$—precisely the distance necessary for it to be possible for the magnitude of the slope to be reduced to zero (but not far enough for it to cross zero). So, with the step size that we use it is not possible for the gradient to change sign, and so we could not have stepped past a local optimum.

Regression 6 | 7

7.1 Objective Function Scaling

Consider again the least squares loss function:

$$l(w) = \sum_{i=1}^{n} \left(y_i - \hat{f}_w(x_i) \right)^2, \tag{7.1}$$

and think about the GPA example wherein student GPAs are predicted from entrance exam scores. For this use case, is a loss of 100 good or bad? That is, if we have model parameters w such that $l(w) = 100$, are the model parameters good or bad?

To answer this question, we need to know n, the number of points. If $n = 10$, then $l(w)$ corresponds to an average squared residual of 10, which is enormous given that GPAs are on a 4.0 scale. However, if $n = 100{,}000$, then the average squared residual is 0.001, which is minuscule and means that the predictions are outstanding.

Also consider the gradient descent update:

$$\forall j \in \{1, 2, \ldots, m\}, \quad w_{k+1,j} = w_{k,j} - \alpha \frac{\partial l(w_k)}{\partial w_{k,j}} \tag{7.2}$$

$$= w_{k,j} - \alpha \sum_{i=1}^{n} \frac{\partial}{\partial w_{k,j}} \left(y_i - f_{w_k}(x_i) \right)^2. \tag{7.3}$$

This form shows that the magnitude of the gradient scales with the number of points, n. Even though the magnitude of the gradient changes, the desired length of a step from w_k to w_{k+1} does *not* change with the magnitude of the loss function. So, when n is large, we will need to use a smaller step size to counteract the larger gradients.

To make the magnitude of the loss function more intuitive (without having to think about n), and to avoid having to consider n when picking a step size, we often rescale the loss function. For the least squares loss, we might compute the *average* squared residual rather than the sum of the squared residuals:

$$l(w) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2, \tag{7.4}$$

or, including the $1/2$ factor that cancels with the exponent when taking the gradient,

$$l(w) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{7.5}$$

With either of these rescaled loss functions, is a loss of $l(w) = 10$ good or bad for the GPA prediction example? Clearly it is bad, as it corresponds to a mean residual of more than 3.0 per student, which is huge given

that GPAs are on a 4.0 scale. Similarly, regardless of the value of n, a loss of $l(w) = 0.001$ would correspond to model parameters w that result in extremely accurate GPA predictions.

Remember that the choice of loss function is a hyperparameter. There are no *wrong* choices—different choices will result in the algorithm behaving differently, and it is up to you to set the hyperparameters to obtain the behavior you want. Also, notice that these changes simply rescale the loss function and its gradient, which is equivalent to rescaling the step size in (7.2).

7.2 Input Normalization

Consider the gradient descent update for the least squares objective in (7.2). Using a linear parametric model, this update becomes:

$$\forall j \in \{1, 2, \ldots, m\}, \quad w_{k+1,j} = w_{k,j} + \alpha 2 \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right) x_{i,j}. \quad (7.6)$$

Notice the red term, $x_{i,j}$, which corresponds to the j^{th} feature of the i^{th} input. What happens if the j^{th} feature is rescaled by a factor of 2? To obtain the same predictions from the linear parametric model, the weight on this feature would be halved. However, this is not the only difference: the magnitude of the update for the j^{th} feature would also be doubled (the update is scaled by this red term), and so the step size is effectively twice as large! Similarly, if the j^{th} feature is scaled to be small, then the step size has effectively been shrunken. To avoid these scaling issues, we might rescale the inputs, just like we did with the loss function.

However, the input scaling is more nefarious than the scaling of the loss function as a whole: what happens if one feature is scaled to be large and another is scaled to be small? Unlike the other rescalings that we have discussed so far, this scaling issue cannot be repaired by changing the step size. A step size that is small enough for the weight multiplied by the input feature that has a large magnitude will be far too small of a step for the weight multiplied by other input feature, which has a small magnitude. That is, one single step size will not be effective. While this can be remedied by using different step sizes for each weight, w_j, we will then have m step size hyperparameters rather than just one, making the algorithm even harder to tune.

So, instead of using different step sizes for each weight, we normalize all of the input features so that they are of roughly the same scale. This way, one step size should be effective for all of the weights. This raises the question: how should we normalize input features so that they are all roughly the same magnitude? Yet again, this decision is a hyperparameter. Two common choices are:

1. Normalize the inputs so that the average value of the j^{th} input feature is zero and the standard deviation is one. This is achieved by taking the original value of the input feature, subtracting the average value of the feature, and dividing by the standard deviation of the feature. This procedure is provided as pseudocode in Algorithm 7.1.

2. Normalize the inputs to be in some range, $[a, b]$, where usually $b = 1$ and $a \in \{-1, 0\}$. If $x_j^{\min}$ and $x_j^{\max}$ are lower and upper bounds on the j^{th} input feature, then this normalization (with $a = 0$) can be achieved with the following change to each $x_{i,j}$:

$$x_{i,j} \leftarrow \frac{x_{i,j} - x_j^{\min}}{x_j^{\max} - x_j^{\min}} . \tag{7.7}$$

Algorithm 7.1: Input normalization to mean zero, variance one.

Input : Data set inputs $(x_i)_{i=1}^n$, where each $x_i \in \mathbb{R}^m$ and $n \geq 2$.
Output : Modified data set where each feature has mean zero and variance one.
Hyperparameters: None!

for $j = 1$ **to** m **do**
  /* Compute the average of the j^{th} feature. */
  $\bar{x}_j \leftarrow \frac{1}{n} \sum_{i=1}^n x_{i,j}$;
  /* Compute the variance of the j^{th} feature. */
  $\hat{\sigma}_j^2 \leftarrow \frac{1}{n-1} \sum_{i=1}^n (x_{i,j} - \bar{x}_j)^2$;
end
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    /* Normalize the j^{th} feature of the i^{th} input. */
    $x_{i,j} \leftarrow \frac{x_{i,j} - \bar{x}_j}{\sqrt{\hat{\sigma}_j^2}}$;
  end
end
/* Return the normalized inputs. */
return $x_1, \ldots, x_n$;

7.3 Basis Functions

You might be thinking: linear parametric models are too simple to be useful. For many interesting regression problems, we should not assume that the predictions scale linearly with the input. However, this argument is flawed because linear parametric models are linear functions *of the model parameters*, but are not necessarily linear functions of the input!

For example, in the $1D$ case (i.e., $m = 1$) the following model *is a linear parametric model*:

$$f_{w_k}(x_i) = w_{k,1} x_{i,1}^3 + w_{k,2} x_{i,1}^2 + w_{k,3} x_{i,1} + w_{k,4}. \tag{7.8}$$

However, this parametric model can represent any third degree polynomial, not just lines!

A function $f(x)$ is **linear** if its derivative is a constant, i.e., the value of $\partial f(x)/\partial x$ does not depend on x [wiki]. For the parametric model in (7.8), the derivative of $f_{w_k}(x_i)$ with respect to x_i is $3w_{k,1}x_{i,1}^2 + 2w_{k,2}x_{i,1} + w_{k,3}$, which depends on the value of $x_{i,1}$, and so $f_{w_k}(x_i)$ is not linear with respect to x_i. However, the derivative of $f_{w_k}(x_i)$ with respect to w_k is the vector $(x_{i,1}^3, x_{i,1}^2, x_{i,1}, 1)$, which does *not* depend on the value of w_k, and

so $f_{w_k}(x_i)$ *is* linear with respect to w_k. So, even though this parametric model can represent more than just lines, it is still called a linear model.

Another perspective of this same idea is that we could write a program that preprocesses the data. In the example above with $m = 1$ and the parametric model in (7.8), this program could read in the data set and output a new data set with $m = 4$, where the first feature in the new data set is the cube of the only feature from the original data set, the second feature is the square, etc. Using this new data set and the perspective from before this section, wherein linear parametric models have one weight per input feature (and perhaps a y-intercept weight), we obtain precisely the parametric model in (7.8).

To formalize this notion of expanding the set of features used by the linear parametric model, we use m to denote the number of input features in the original data set and m' to denote the number of features in the data set after it has been preprocessed. It is not uncommon for m' to be *much* larger than m (e.g., $m = 10$ and $m' = 10{,}000$). In these cases, the preprocessed data set might be enormous (in terms of memory usage) in comparison to the original data set. To avoid this, we can use the original data set without any preprocessing by applying the preprocessing to the input, x_i, any time that it is used.

That is, let $\phi : \mathbb{R}^m \to \mathbb{R}^{m'}$ be a function that takes as input a vector from the original data set, x_i, and produces as output a vector of m' features to use with the linear parametric model. To help remember this symbol, recall that ϕ is pronounced fee, as in *fea*tures. Also, to differentiate between the inputs before applying ϕ and those after, we call x_i the **input vector** and $\phi(x_i)$ the **feature vector**.

Incorporating ϕ into the algorithms that we have developed thus far is trivial: simply replace all occurrences of x_i with $\phi(x_i)$, and m with m'.

The function ϕ is usually chosen to be nicely scaled. As a result, input normalization is typically performed prior to applying ϕ, or ϕ can apply an input normalization scheme itself. For example, the Fourier basis is one option for ϕ that typically assumes the inputs have been normalized to the interval $[-1, 1]$ or $[0, 1]$, depending on the form of the Fourier basis.

How should ϕ be selected? You guessed it—it is yet another hyperparameter. Sometimes ϕ is chosen together by a team of machine learning researchers and experts in the application area (e.g., experts in medicine for medical applications). In other cases, fixed application-independent choices of ϕ (like the polynomial basis or Fourier basis)[1] can be effective. However, common choices for ϕ like the polynomial basis and Fourier basis have their own hyperparameters that set how large m' is. Often this hyperparameter is called the *degree* (polynomial basis) or *order* (Fourier basis). In the coming chapters, we will discuss neural networks, which can be thought of as a way for the agent to *learn* ϕ.

When using a linear parametric model, ϕ is called a **basis** (hence why the two examples were the polynomial *basis* and Fourier *basis*). Though the precise definition of a basis is beyond the scope of this course, you can think of a basis as the coordinate frame for a space—from the space of real numbers to the space of all colors, for example. In our case, ϕ is

1: For a description of the Fourier basis see this paper [15]. Note that this paper discussed *value function approximation*, which is a topic that we will cover in the second part of this course, which covers reinforcement learning. However, the basis functions presented apply to both supervised and reinforcement learning. The *value function approximation* in the referenced paper corresponds to the parametric model here.

the "coordinate frame" for the space of all possible functions that the parametric model can represent.

More precisely, if a basis has elements x, y, and z, then the space that it parameterizes is the set of all linear combinations of x, y, and z. That is: $ax + by + cz$ for any real-valued a, b, and c. For example, $(0,0,1), (0,1,0), (1,0,0)$ is a basis for 3D space. The point $(3,4,5)$ is $5 \times (0,0,1) + 4 \times (0,1,0) + 3 \times (1,0,0)$ (where here $\times$ denotes that the constant is multiplied by each element of the vector). Notice that $(0,0,2), (0,1,0), (3,0,0)$ is also a basis for 3D space. Even more mind-bending, $(2,1,0), (0,1,0), (0,3,-1)$ is *also* a basis for 3D space!

Consider the case where $m = 1$ and where we are using the polynomial basis of degree 2, i.e., $\phi(x_i) = (x_{i,1}^2, x_{i,1}, 1)$. One can view $f_{w_k}(x_i)$ as taking a linear combination of the three functions $x_{i,1}^2$, $x_{i,1}$, and 1. Visually, any function represented by $f_{w_k}(x_i)$ will be the sum of these three functions, each rescaled by the corresponding weights, $w_{k,1}$, $w_{k,2}$, and $w_{k,3}$.

8 Perceptron

In the previous chapter, we discussed how basis functions can be used to allow linear models to represent nonlinear functions of the input. We now take a different approach to representing nonlinear functions of the input: using nonlinear models. We will begin with a simple model called a **perceptron**, which is an extremely crude simulation of a single neuron. Next, we will discuss how multiple perceptrons can be connected into a network, called an *artificial neural network*.

8.1 Biological Inspiration

Recall that neurons can roughly be thought of as having three main components: dendrites, a cell body, and an axon. These three main parts are depicted in Figure 8.1. The dendrites serve as the *input* to the neuron, receiving signals from other neurons and conveying them to the cell body. The cell body, or *soma*, processes these inputs. Lastly, the axon can be thought of as the *output* of the neuron, connecting to the inputs (dendrites) of other neurons.

Every so often the neuron *spikes* (more technically, this spiking is called an action potential). The rate at which these action potentials occur can increase or decrease depending on the recent activation patterns received by the dendrites. Many different properties of the neuron control which activation patterns of the inputs result in frequent action potentials, e.g., dendritic morphology (the physical shape and structure of dendrites).[1] One can think of these various properties as model parameters. Evolution and animal learning have achieved the remarkable result of tuning these model parameters so that a large network of neurons produces intelligent behavior.

If our linear parametric models are too simple to represent complicated functions, but we know that collections of neurons can produce human-level intelligence, this suggests that we might try to mimic nature: creating parametric models inspired by the brain and neurons. Before creating a network of simulated neurons, we focus on how we might create a parametric model that resembles a single neuron. There are many simplified models of neurons that resemble real neurons to various degrees. In this course, our goal is *not* to study neurons, and so we will not attempt to create or use the most realistic model of a neuron. Rather, we will use one of the crudest neuron models—one so crude that I would suggest not even thinking of it as a model of a neuron, but rather as a parametric model *inspired* by neurons. Though this model barely resembles real neurons, it can be efficiently implemented on modern computer hardware and is the parametric model used by many of the most sophisticated ML systems today.

1: Long ago, I took a course on computational neuroscience, and for the course project, my group used evolutionary algorithms to search for dendritic morphologies that enable a single simulated neuron to solve a supervised learning task. Though this use of a neuron likely does not mirror biology, it was an example of how dendritic morphology alone can result in significant amounts of computation within a *single* neuron. Note: We used a neuron model called a *Hodgkin-Huxley model*, which is more realistic than the models we will use in this course. To learn about these more sophisticated neuron models, see this book [17].

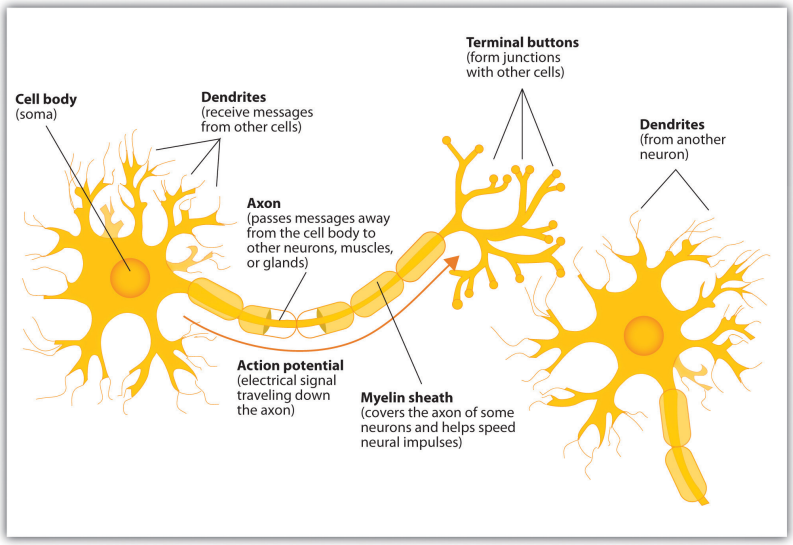


Figure 8.1: Diagram of a neuron [16]. We focus on three high-level structures: the dendrites, cell body, and axon.

8.2 Perceptron Parametric Model

With this model, called a **perceptron**, we use real numbers to characterize the output of the neuron. Intuitively, you can think of this number as encoding the frequency of action potentials (how rapidly the neuron is firing). Using this perspective, the input to a perceptron is a vector with each element corresponding to the firing rate of a neuron connected to the input (dendrite) of the current neuron. In our case, we will use just one perceptron, and so the inputs will be the vector of input features, $x_i \in \mathbb{R}^m$, and the output of the neuron will be the prediction, $\hat{y}_i$, of the label y_i.

Formally, the perceptron operates as follows. First, the vector of inputs is transformed into a single real number using a linear combination. That is:

$$\text{in}_i = \sum_{j=1}^{m} w_{k,j} x_{i,j}, \tag{8.1}$$

where w_k is a vector of model parameters just like before, scaling the impact that the j^{th} input has on the perceptron's activation. This should look familiar—this is just the regular linear parametric model that we have dealt with so far!

Next, the neuron must determine whether or not it will fire. One way to do this would be to pick a *threshold*, τ. If $\text{in}_i \geq \tau$, then the neuron fires (outputting a 1), otherwise the neuron does not fire (it outputs 0). However, this view requires us to reason about another parameter, τ. To avoid this, we use the same trick that we used to include a y-intercept in our linear models: we create an additional input, $x_{i,m+1}$, that is always one. If we then test whether $\text{in}_i \geq 0$ rather than $\text{in}_i \geq \tau$, we are effectively using $-w_{k,m+1}$ as the threshold, τ. Hereafter, we assume this additional input is included, and that m is the total number of inputs (including this additional input).

The above logic can be represented without an if-else statement using the **Heaviside function** $H : \mathbb{R} \to 0, 1$, which is zero for negative numbers, and one for nonnegative numbers (positive numbers and zero), as

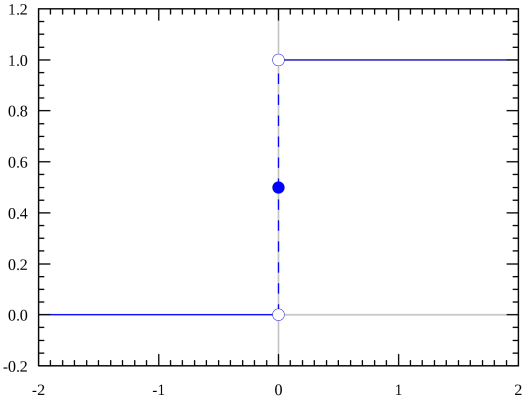


Figure 8.2: Heaviside function [wiki].

depicted in Figure 8.2.[2] That is:

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{8.2}$$

Let a_i be the output of the perceptron (this is the same as $f_{w_k}(x_i)$ when there is only one perceptron, but when we have many perceptrons we will need to differentiate between the output of an internal perceptron and the output of the model). We then have that

$$\begin{aligned} a_i &= H(\text{in}_i) && (8.3) \\ &= H\left(\sum_{j=1}^{m} w_{k,j} x_{i,j}\right) && (8.4) \\ &= \begin{cases} 1 & \text{if } \sum_{j=1}^{m} w_{k,j} x_{i,j} \geq 0 \\ 0 & \text{otherwise.} \end{cases} && (8.5) \end{aligned}$$

In general, the function applied to in_i (the weighted sum of inputs) to obtain a_i (the final output of the perceptron) is called the **activation function**. One problem with using the Heaviside function as an activation function is that it does not work well with gradient descent. Its gradient is zero everywhere (not a very useful gradient!) except at zero, where the gradient is undefined (even less useful!).

One way to overcome this limitation is to smooth out the activation function: rounding the corners and slanting the flat lines so that the gradient always exists and is never zero. There are *many* such functions. **Sigmoids** are a class of functions that are shaped like a rounded version of the Heaviside function—almost an "S" shape. The **logistic function**, $\sigma : \mathbb{R} \to \mathbb{R}$ is one such sigmoid. In fact, the logistic function is so commonly used that it is sometimes called *the* sigmoid function. In the remainder of this text, we will refer to the logistic function as *the* sigmoid function, though keep in mind that technically sigmoids are a class (set) of functions, not just the logistic function.

The sigmoid function (logistic function), σ, is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{8.6}$$

2: The Heaviside function is one example of a *step function*.

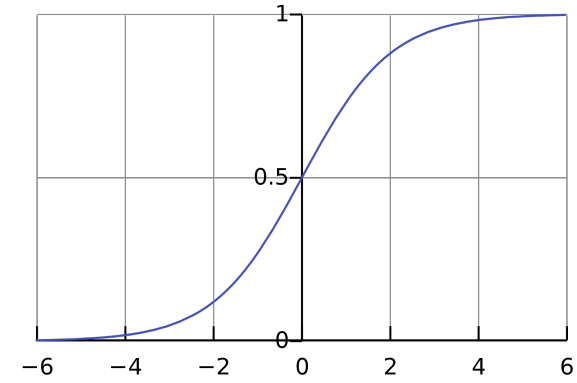


Figure 8.3: The sigmoid function (technically, the logistic function, which is one example of a sigmoid function).

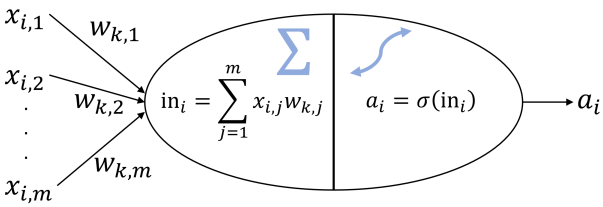


Figure 8.4: Diagram of the perceptron parametric model. The left side, marked with Σ, computes a weighted sum of the input, while the right side, marked with a sigmoid shape, puts this weighted sum through an activation function to determine the final output of the perceptron. This figure is inspired by a figure from a popular textbook [18].

and is depicted in Figure 8.3. Notice that using the sigmoid instead of the Heaviside function, the output of the perceptron can span the range $[0, 1]$, rather than only taking the extreme values of 0 or 1.

Incorporating the sigmoid function into the perceptron, we obtain the final equations for specifying a perceptron model:

$$f_{w_k}(x_i) = a_i \tag{8.7}$$

$$a_i = \sigma(\text{in}_i) \tag{8.8}$$

$$\text{in}_i = \sum_{j=1}^{m} w_{k,j} x_{i,j}, \tag{8.9}$$

where σ is defined in (8.6). Recall that $f_{w_k}(x_i) = a_i$ for now since there is only one perceptron—later, with many perceptrons, a_i will correspond to the output of the perceptrons, which may not be the output of the parametric model. Figure 8.4 depicts the general structure of the perceptron model. Written in one big equation, this gives:

$$f_{w_k}(x_i) = \frac{1}{1 + e^{-\left(\sum_{j=1}^{m} w_{k,j} x_{i,j}\right)}}. \tag{8.10}$$

8.3 Gradient Descent using a Perceptron Parametric Model

To perform gradient descent on the least squares loss function, we must derive an expression for $\frac{\partial l(w_k)}{\partial w_{k,j}}$. In this section, we will derive an expression for this partial derivative, leveraging the **chain rule**. Recall that the least

squares loss function is:

$$l(w_k) = \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right)^2. \tag{8.11}$$

One way to differentiate this function would be to expand it completely and then differentiate:

$$\begin{aligned} l(w_k) &= \sum_{i=1}^{n} (y_i - \sigma(\text{in}_i))^2 && (8.12)\\ &= \sum_{i=1}^{n} \left(y_i - \frac{1}{1+e^{-\text{in}_i}}\right)^2 && (8.13)\\ &= \sum_{i=1}^{n} \left(y_i - \frac{1}{1+e^{-\sum_{j=1}^{m} w_{k,j} x_{i,j}}}\right)^2. && (8.14) \end{aligned}$$

While we *could* differentiate this expression directly, it would be quite tedious. Instead, we will break the derivative of $l(w_k)$ into chunks that are easier to differentiate, leveraging the chain rule. This strategy will be even more important when we have many perceptrons connected together in a network.

First, we determine how to write the partial derivative of the loss function in terms of the derivative of the parameterized model:

$$\begin{aligned} \frac{\partial l(w_k)}{\partial w_{k,j}} &= \frac{\partial}{\partial w_{k,j}} \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right)^2 && (8.15)\\ &= -2 \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right) \frac{\partial}{\partial w_{k,j}} f_{w_k}(x_i). && (8.16) \end{aligned}$$

Next, we solve for the partial derivative of the parameterized model (note that the red terms are the same):

$$\frac{\partial}{\partial w_{k,j}} f_{w_k}(x_i) = \frac{\partial f_{w_k}(x_i)}{\partial \text{in}_i} \frac{\partial \text{in}_i}{\partial w_{k,j}}, \tag{8.17}$$

by the chain rule. We can obtain expressions for the two terms on the right hand side as follows:

$$\begin{aligned} \frac{\partial f_{w_k}(x_i)}{\partial \text{in}_i} &= \frac{\partial}{\partial w_{k,j}} \sigma(\text{in}_i) && (8.18)\\ &= \sigma(\text{in}_i)(1 - \sigma(\text{in}_i)), && (8.19) \end{aligned}$$

since the derivative of $\sigma(x)$ is $\sigma(x)(1-\sigma(x))$.[3] For the blue term:

$$\begin{aligned} \frac{\partial \text{in}_i}{\partial w_{k,j}} &= \frac{\partial}{\partial w_{k,j}} \sum_{\beta=1}^{m} w_{k,\beta} x_{i,\beta} && (8.20)\\ &= x_{i,j}. && (8.21) \end{aligned}$$

The derivation of the blue term above should be familiar—it is the derivative of a linear parametric model.

Putting the yellow and blue terms together to obtain a complete expression for the red term (the partial derivative of the parametric model with

3: For a derivation of the derivative of the sigmoid, see this Wikipedia page.

respect to the j^{th} weight), we obtain:

$$\frac{\partial}{\partial w_{k,j}} f_{w_k}(x_i) = \sigma(\text{in}_i)(1 - \sigma(\text{in}_i))x_{i,j}. \tag{8.22}$$

Recall that the slope of a line is a constant—it does not depend on the position on the line. Similarly, the slope of a linear function is a constant. Notice that the derivative of the perceptron model *does* depend on w_k, since in_i is a function of w_k. Hence, the perceptron is *not* a linear parametric model (recall that linearity of a model is determined based on whether it is a linear function of the model parameters, not based on whether it is a linear function of the inputs).

While one might implement the derivative of the loss function with separate functions for the red, yellow, and blue terms, we can put the pieces together to obtain a complete expression for the partial derivative of the loss function with respect to the j^{th} weight:

$$\frac{\partial l(w_k)}{\partial w_{k,j}} = -2 \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right) \sigma(\text{in}_i)(1 - \sigma(\text{in}_i))x_{i,j}. \tag{8.23}$$

Plugging this into the gradient descent update, we obtain the following update for the model parameters:

$$w_{k+1,j} = w_{k,j} + \alpha 2 \sum_{i=1}^{n} \left(y_i - f_{w_k}(x_i)\right) \sigma(\text{in}_i)(1 - \sigma(\text{in}_i))x_{i,j}. \tag{8.24}$$

9 Artificial Neural Networks

In the last chapter, we presented a nonlinear parametric model that is a crude simulation of a neuron; we called this a *perceptron*. In this chapter, we study a more sophisticated nonlinear parametric model: a network of perceptrons. These networks are called **artificial neural networks** (ANNs) as they can be viewed as extremely crude models of networks of neurons. When dealing with ANNs, we will call an individual perceptron within the network a **node**, since the network can be viewed as a graph with perceptrons as nodes and input/output connections as edges. Alternatively, nodes are sometimes called **units**. There are many ways that perceptrons can be connected together to form a network (graph), and the structure of a network is called its **network architecture**.

We will focus on a network architecture wherein the nodes are arranged in **layers** and each node in a layer takes as input the output of all of the nodes in the previous layer, as depicted in Figure 9.1. This architecture is called a **fully connected feedforward** network architecture. The nodes in the input layer are special—their output values are set to the values of the inputs, $x_{i,1}, \ldots, x_{i,m}$. The output of the output layer, which has a single node, is the output of the parametric model, i.e., $f_{w_k}(x_i)$. The layers between the input and output layers are called **hidden layers**. The network is "fully connected" in that each node in a layer takes as input the output of all nodes in the previous layer. The network is "feedforward" in that there are no backwards connections (the graph is *acyclic*). Network architectures with backwards connections (networks that form cyclic graphs) are called **recurrent** networks.

The choice of network architecture is yet another hyperparameter. Even if you select fully connected feedforward networks, you must still select the number of hidden layers, the number of nodes in each hidden layer (which may vary per layer), and the activation function to use for each node.

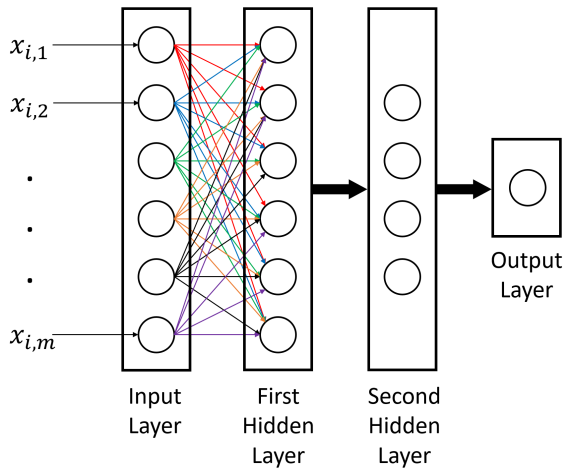


Figure 9.1: Fully connected feedforward network architecture with four layers. Arrows indicate that the output of a node is an input to another node. To avoid drawing large numbers of arrows and nodes, layers are often depicted using just the rectangles with arrows between the rectangles (often without even the circles) indicating fully connected layers, like in the right portion of this diagram.

9.1 New Notation

Before proceeding, we will establish a new indexing notation for model parameters (weights). Consider how we might reference one particular weight in one particular node during one particular iteration of gradient descent. This might use *four* indices—something like $w_{k,j}^{l,o}$ might denote the j^{th} weight of the o^{th} perceptron in the l^{th} layer at the k^{th} iteration of gradient descent, reserving the symbol i for indexing which data point the network is run on. While this works, it becomes difficult to track all of the various indices.

Instead, we adopt the notation of Russell and Norvig [18]. First, we consider a single specific input vector, $x \in \mathbb{R}^m$, freeing up the symbol "i". That is, $x \in \mathbb{R}^m$, and x_j now denotes the j^{th} element of the input x. Next, we consider some *current layer* and its *previous layer*, without using a symbol or index to specify the actual layer number, and we use i to indicate indices related to the previous layer and j to indicate indices related to the current layer. We can then write $w_{i,j}$ to denote the weight from the i^{th} node in the previous layer to the j^{th} node in the current layer. Next, we write a_i and a_j to denote the outputs of the i^{th} node in the previous layer and j^{th} node in the current layer respectively. Finally, we add an additional 0^{th} node to each layer that always outputs one, which serves as a threshold (y-intercept) parameter.

[18]: Russell et al. (2003), *Artificial Intelligence: A Modern Approach*

With this new notation, any particular perceptron in the network can be characterized by the equations:

$$\text{in}_j = \sum_{i=0}^{m} w_{i,j} a_i \tag{9.1}$$

$$a_j = \sigma(\text{in}_j), \tag{9.2}$$

where σ is the activation function—recall that we will use the sigmoid (logistic function).[1] The input to the network is handled via a special case, where the outputs of the nodes in the input layer are defined to be the inputs: $a_j = x_j$ for the input layer. Figure 9.2 depicts a perceptron with this new notation.

1: Our notation differs from that of Russell and Norvig, who denote the activation function by g.

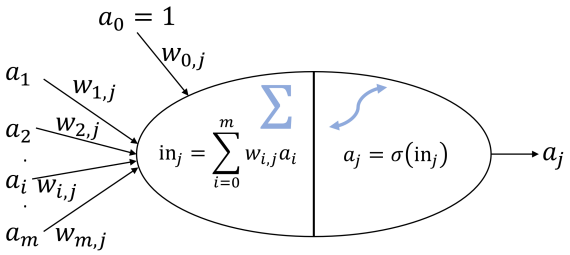


Figure 9.2: Diagram of a perceptron parametric model using the new notation. The inputs, a_i, are the outputs of perceptrons in the previous layer, and the output a_j is the output of the entire parametric model if the perceptron is in the output layer, or it is one of the inputs to perceptrons in the next layer if the perceptron is in a hidden layer.

9.2 Forward Pass

The process of running all of the perceptrons in an ANN to determine the output for some input, $x \in \mathbb{R}^m$, is called a **forward pass**. A forward pass for a fully connected feedforward network is straightforward: the input nodes are initialized to the input vector, and then all of the perceptrons

in the hidden layers and output layer are run in order. Pseudocode for this procedure is provided in Algorithm 9.1.

Algorithm 9.1: Forward pass

Input : Input vector x, model parameters w.
Output : Output of the parametric model, $f_w(x)$.
Hyperparameters : Network architecture (number of layers, how layers are connected, activation functions, etc.). Here we assume a fully connected feedforward network.

```
/* Load input layer                                    */
for each input x_j in x do
    a_j ← x_j;
end
/* Run Hidden Layers                                   */
for each hidden layer do
    for each node j in current hidden layer do
        in_j = ∑_i w_{i,j} a_i // i sums over nodes in previous layer
        a_j ← σ(in_j);
    end
end
/* Return prediction of the label for x                */
return output of the single node in the output layer;
```

10 Backpropagation

Recall from (8.16) that the gradient of the least squares loss can be written in terms of the partial derivatives of the output of the parametric model with respect to each weight (the red term in (8.16)).[1] So, in order to run gradient descent using an ANN as the parametric model, we must derive an expression for the partial derivative of the output of the network with respect to any particular weight, $w_{i,j}$. We can express this derivative using the chain rule as:

$$\frac{\partial f_w(x)}{\partial w_{i,j}} = \frac{\partial f_w(x)}{\partial \text{in}_j} \frac{\partial \text{in}_j}{\partial w_{i,j}}. \tag{10.1}$$

1: We do not reproduce this derivation here, since this point is best understood using our old notation, and this chapter uses the new notation established in the previous chapter.

The first (pink) term is a special term that we will reference repeatedly later, and so we give it its own symbol:

$$\Delta_j = \frac{\partial f_w(x)}{\partial \text{in}_j}. \tag{10.2}$$

Going back to (10.1) and applying the chain rule to Δ_j (the pink term), we obtain:

$$\frac{\partial f_w(x)}{\partial w_{i,j}} = \underbrace{\frac{\partial f_w(x)}{\partial a_j} \frac{\partial a_j}{\partial \text{in}_j}}_{=\Delta_j} \frac{\partial \text{in}_j}{\partial w_{i,j}}. \tag{10.3}$$

The latter two terms should be straightforward given the derivatives that we have worked out previously:

$$\frac{\partial a_j}{\partial \text{in}_j} = \frac{\partial}{\partial \text{in}_j} \sigma(\text{in}_j) \tag{10.4}$$

$$= \sigma(\text{in}_j)(1 - \sigma(\text{in}_j)), \tag{10.5}$$

assuming σ is the logistic function (sigmoid). Next, the yellow term:

$$\frac{\partial \text{in}_j}{\partial w_{i,j}} = a_i. \tag{10.6}$$

The tricky term is then the red term, $\partial f_w(x)/\partial a_j$. Recall that this term represents the rate that the output of the parametric model, $f_w(x)$, will change if the output of the current node, a_j, increases.

To see how to compute this term, first consider the case where the next layer only includes one node, as depicted in Figure 10.1. Notice that here we use k to index terms in the *next* layer (while still using j to index terms from the current later and i to index terms in the previous layer). Using the chain rule, we can break the red term into two partial derivatives:

$$\frac{\partial f_w(x)}{\partial a_j} = \underbrace{\frac{\partial f_w(x)}{\partial \text{in}_k}}_{\text{(a)}} \underbrace{\frac{\partial \text{in}_k}{\partial a_j}}_{\text{(b)}}. \tag{10.7}$$

If we were to update the nodes in the *reverse* of the order that the nodes were run (from the output layer back to the input), then the **(a)** term is one that we would already have computed: it is Δ_k (the pink term in (10.1), but for the node in the next layer rather than the current layer). So, if we compute these partial derivatives backwards, from the output to the input, we would already have computed the term **(a)**, so we can simply store its value when we computed it, allowing us to look up its value when updating the current node.[2] This leaves term **(b)**:

$$\frac{\partial \text{in}_k}{\partial a_j} = w_{j,k}. \tag{10.8}$$

2: Notice that the output layer is an exception: For the output layer the red term is one, since $f_w(x)$ is a_j.

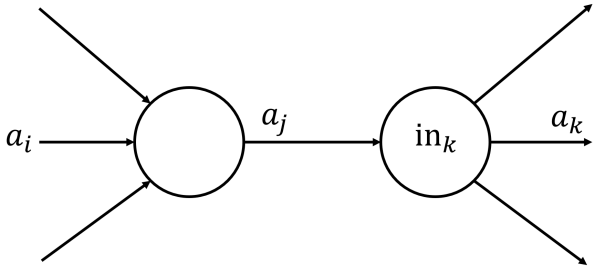


Figure 10.1: Simplified network structure wherein the current node's output is only used by one node in the subsequent layer.

Revisiting (10.3), we can now plug in the expressions for each term (keeping the same colors for each term) to obtain:

$$\frac{\partial f_w(x)}{\partial w_{i,j}} = \underbrace{\underbrace{\frac{\partial f_w(x)}{\partial \text{in}_k}}_{=\Delta_k} w_{j,k}\sigma(\text{in}_j)(1-\sigma(\text{in}_j))\, a_i}_{=\Delta_j}. \tag{10.9}$$

However, we derived this using the simplifying assumption that the output of the current node is only used by a single node in the next layer. In multilayer networks, the output of one node is frequently used as an input to *multiple* other nodes in the network, as depicted in Figure 10.2. In this case, the output of the current node influences $f_w(x)$ through every node in the next layer.

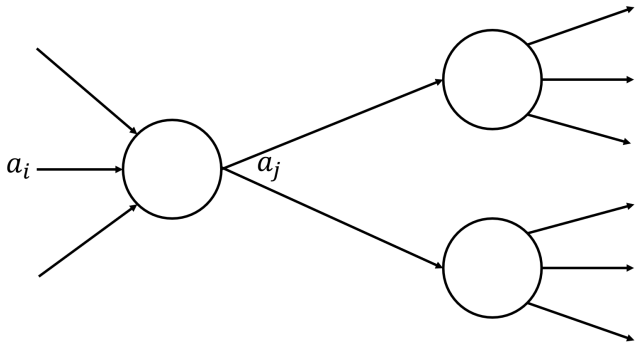


Figure 10.2: Network structure wherein the current node's output is only used by two nodes in the subsequent layer. The output of the current node, a_j, impacts the output of the network through both of the subsequent nodes. So, the impact that a_j has on $f_w(x)$ (the network output) is the sum of its impact through nodes in the next layer. This trend carries over to any number of nodes in the next layer—not just two.

That is,

$$\frac{\partial f_w(x)}{\partial a_j} = \sum_k \frac{\partial f_w(x)}{\partial \text{in}_k} w_{j,k}, \tag{10.10}$$

where the summation over k indicates summation over all nodes in the next layer. That is, the impact that a change to a_j has on $f_w(x)$ is the sum of the impacts that changes to a_j will have on $f_w(x)$ via each of the nodes in the next layer.[3]

Using this form for the red term, we can finally express (10.3) in expanded form as:

$$\frac{\partial f_w(x)}{\partial w_{i,j}} = \underbrace{\sum_k \underbrace{\frac{\partial f_w(x)}{\partial \text{in}_k}}_{=\Delta_k} w_{j,k}\sigma(\text{in}_j)(1-\sigma(\text{in}_j))\, a_i}_{=\Delta_j}. \tag{10.11}$$

Plugging this into the gradient descent update (see (8.16))

$$w_{i,j} \leftarrow w_{i,j} + \alpha 2 \sum_{(x,y) \text{ in data set}} (y - f_w(x)) \frac{\partial f_w(x)}{\partial w_{i,j}}, \tag{10.12}$$

results in gradient descent on the least square objective using a neural network as the parametric model.

However, consider what happens when there are many (millions) of points in the data set: Each step of gradient descent will require a large amount of computation, making training slow. When training ANNs, it is therefore common to use a variant of gradient descent called **incremental gradient methods**. In this variant of gradient descent, the weights are updated using an estimate of the gradient computed from a single data point. That is, for each data point (x, y) in the data set, for each weight $w_{i,j}$ in the network,

$$w_{i,j} \leftarrow w_{i,j} + \alpha 2 (y - f_w(x)) \frac{\partial f_w(x)}{\partial w_{i,j}}. \tag{10.13}$$

That is, rather than summing over all of the "(x, y) in data set," the algorithm uses a single (x, y) pair (input-label pair) to perform the weight update and loops over the data set (perhaps multiple times) to select which training point to use. Intuitively, you can think of this approach as using estimates of the gradient of the loss function to perform each weight update. The convergence properties of this gradient descent variant are nearly identical to those of regular gradient descent [14].

Even with the improved efficiency of incremental gradient descent, we should be careful to compute the updates in an efficient way. This is achieved by computing each Δ_j term once, and storing its value for reference later. The algorithm for efficiently computing these derivatives is called **backpropagation** because the gradient information is propagated backwards through the network via the stored Δ_j terms. Furthermore, the gradient update loops over the layers in reverse order, and so this process is sometimes called a **backwards pass**. Pseudocode for backpropagation is provided in Algorithm 10.1.

3: We have stated (10.10) and provided intuition for its meaning, but have not given a complete derivation from first principles. For the interested reader, this derivation can be found on page 745 of the third edition of Russell and Norvig's book [18].

[14]: Bertsekas et al. (2000), 'Gradient convergence in gradient methods with errors'

Algorithm 10.1: Gradient descent on least squares loss using a fully connected feedforward neural network and backpropagation for gradient computation. Recall from Section 8.2 that each perceptron implements a threshold for firing by including an extra input that is always equal to one.

Input : Data set consisting of many input-label pairs, (x, y)

Output : Weights, w, that approximately minimize the least squares loss for a fully connected feedforward artificial neural network, with the logistic function as the activation function.

Hyperparameters : Number of hidden layers, number of nodes in each hidden layer, initial weight distribution p, step size $\alpha \in \mathbb{R}_{>0}$, stopping criterion `stop`.

```
// Initialize weights w by sampling i.i.d. from p
for each weight w_{i,j} in w do
    w_{i,j} ~ p;
end
while stopping criterion stop not satisfied do
    /* Train on each point in the data set once                    */
    for each data point (x, y) in the data set do
        /* Get the predicted label using Algorithm 9.1             */
        ŷ = ForwardPass(x, w);
        /* Run a backward pass to obtain Δ_j values. Start
           by computing Δ_j for the output layer as a
           special case (recall that the red term in (10.3)
           is one in this case).                                   */
        for each node j in the output layer do
            Δ_j ← σ(in_j)(1 − σ(in_j));
        end
        /* Loop over the layers in reverse order,
           computing the Δ_j values for all nodes.                 */
        for each hidden layer from the last (near output) to first (near input)
         do
            for each node j in the current layer do
                /* The summation over k below is over nodes
                   in the next layer.                              */
                Δ_j ← Σ_k Δ_k w_{j,k} σ(in_j)(1 − σ(in_j));
            end
        end
        /* Perform incremental gradient step: (10.13).            */
        for each weight w_{i,j} in w do
            w_{i,j} ← w_{i,j} + 2α(y − ŷ)Δ_j a_i;
        end
    end
end
return w;
```

11 Vanishing Gradients

11.1 Vanishing Gradients

Recall the partial derivative of the model output with respect to a particular weight in an ANN (this is a reproduction of (10.11)):

$$\frac{\partial f_w(x)}{\partial w_{i,j}} = \underbrace{\sum_k \underbrace{\frac{\partial f_w(x)}{\partial \text{in}_k}}_{=\Delta_k} w_{j,k}\sigma(\text{in}_j)(1-\sigma(\text{in}_j))\, a_i}_{=\Delta_j}. \tag{11.1}$$

Notice again that the expression for Δ_j is written in terms of Δ_k (Δ_k is the Δ_j terms from the *next* layer in the network toward the output). What happens to the magnitude of Δ_j as we progress farther and farther back within the network toward the input? In (11.1), $\sigma(\text{in}_j)$ is between zero and one, and so the blue term is maximized when $\sigma(\text{in}_j) = 0.5$, resulting in the blue term being 0.25. As a consequence, Δ_j tends to have a smaller magnitude than Δ_k (even though Δ_j sums over multiple Δ_k). As this progresses back through the network, the values of Δ_j continue to shrink.

This means that the partial derivative, $\partial f_w(x)/\partial w_{i,j}$ tends to be much smaller for weights $w_{i,j}$ near the input layer. This tendency is called **vanishing gradients** because, as we progress further and further back within the network (towards the input layer), the gradients "vanish"—decreasing in magnitude towards zero. This causes issues with the gradient descent update: weights near the input layer hardly change, and so the network learns slowly. This should also make sense intuitively: weights near the output of the network likely have a larger influence on the output of the network, and so the derivative of the network output with respect to those weights will be relatively large. Similarly, weights near the input of the network likely have a smaller influence on the output of the network (since their output is filtered through the entire remainder of the network), and so the derivative of the network output with respect to the weight there will be relatively small.

A first thought might be to fix this by increasing the step size. However, the gradient is *not* poorly scaled near the output layer, and so a large step size will result in too big of a step for weights near the output layer. Two of the most common and effective fixes for this issue are **1)** changing the activation function and **2)** using an adaptive step size. We will discuss adaptive step sizes later within this chapter. With regard to changing the activation function: notice that the troublesome terms that make the values of Δ_j shrink as we progress farther back within the network are the blue terms in (11.1). These terms correspond to $\partial\sigma(\text{in}_j)/\partial\,\text{in}_j$. That is, they come from the derivative of the activation function!

This suggests an idea: could we swap out the sigmoid activation function for one where this partial derivative is *not* always less than 0.5? One popular alternative activation function, the **rectified linear unit** (ReLU), achieves exactly this:

$$g(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{otherwise.} \end{cases} \tag{11.2}$$

While we could easily spend multiple lectures discussing just activation functions, here we simply point out that the derivative of the ReLU activation function is often one (though sometimes zero) and that this has been found to be an effective strategy for mitigating the problem of vanishing gradients.

12 Other Topics

This lecture covers many topics at a high level, providing you with exposure to topics that you will learn more about if you pursue continued machine learning education.

12.1 Weight Initialization

While we have focused heavily on how the weights (model parameters) of an ANN can be updated, we grazed over the question of how the weights should be initialized. It turns out that the way that weights are initialized can have a significant impact on how well the network will be able to learn. If weights are too large, then (even with the problem of vanishing gradients!) the gradients tend to be large, resulting in divergence of the network (sometimes called *exploding gradients*). If the weights are too small, then it can exacerbate the problem of vanishing gradients—notice that weight terms show up in the expression for Δ_j in (11.1)—and so if the weights are all small (significantly less than one), it will also cause vanishing gradients!

While there are *many* options for initializing the weights of an ANN, here we present two of the most well-known to give you a taste for what these weight initialization schemes look like. First, Xavier initialization [19] aims to achieve two properties: the mean activation (in_j) should be zero and the variance of the activations should be the same across all layers. A scheme that achieves these two properties, and which has come to be known as **Xavier initialization**, is to sample each weight in layer l from a normal distribution with mean zero and variance $1/n_{l-1}$, where n_{l-1} is the number of nodes in the *previous* layer. This means that a node that has many inputs will be initialized with smaller weights, on average, than a node with fewer inputs.

[19]: Glorot et al. (2010), 'Understanding the difficulty of training deep feedforward neural networks'

Subsequently, **He initialization** [20] was proposed as an alternative to Xavier initialization and tends to be particularly effective when using ReLU activation functions. He initialization is identical to Xavier initialization, except that the weights have variance $2/n_{l-1}$ rather than $1/n_{l-1}$.

[20]: He et al. (2015), 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification'

12.2 Adaptive Step Sizes

Selecting a value for the step size hyperparameter, α, in the backpropagation algorithm can be quite challenging. For example, in the last two sections, we discussed different ways that the magnitude of the step size may need to be different for different weights in the network. Additionally, independent of any scaling issues of different weights, the shape of the loss function can cause issues. That is, the slope of the loss

function may be large (steep) for some settings of the model parameters and small (shallow) for others. Because the gradient update,

$$w_{k+1,k} \leftarrow w_{k,j} - \alpha \frac{\partial l(w_k)}{\partial w_{k,j}}, \tag{12.1}$$

includes multiplication by $\frac{\partial l(w_k)}{\partial w_{k,j}}$, when the gradient is small (slope is shallow), the algorithm will make very small changes to the model parameters, and when the gradient is large (slope is steep), the algorithm will make very large changes to the model parameters.

Adaptive step size methods are algorithms for automatically changing the length of a gradient descent step (and sometimes even the direction of the step!) with the goal of making it easier to tune the step size hyperparameter and/or to speed up the learning process by allowing for larger steps across parts of the loss function where the gradient is relatively small (the slope is shallow). There are *many* different adaptive step size techniques, which range from simple approaches that introduce a notion of momentum to more sophisticated approaches that reason about the curvature of the loss function. Perhaps the most common adaptive step size methods are adaptive gradient (AdaGrad), root mean square propagation (RMSProp), and adaptive moment estimation (Adam). While a review of these methods is beyond the scope of this course, this Wikipedia page provides a nice summary for the curious reader.

12.3 Classification

Classification problems are like regression problems, but where the set of possible labels (values for each y_i) is finite, and typically the labels do *not* correspond to numbers. For example, consider the problem of predicting whether or not an image is a picture of a cat—in this case, the output of the model would be "yes" or "no". Similarly, a system that maps images of handwritten letters to the corresponding letter would have labels $\{a, b, \ldots, z\}$.

This raises the question: how can we modify our parametric models, like ANNs, so that they output values like "a", "b", "yes", or "no"? The most common strategy is to change the model to have one output per label, and to drop the activation function from the output layer of the network (so that the outputs range from $-\infty$ to ∞ rather than from 0 to 1. Next, we might define the output of the parametric model to be the label corresponding to the output with the largest value. That is, if the third output corresponds to the label "c" and the third output is the largest of the outputs, then the model's prediction is "c".

A problem with this approach is that the max operator is not differentiable, and so the derivative of the model output with respect to any particular weight may not exist (and when it does exist, it is zero!). This precludes the use of backpropagation or gradient descent to train the parametric model.

One way to overcome this limitation is to replace the maximum with a softened form of maximum called **softmax**. Softmax is essentially a randomized version of the regular max operator. Specifically, if the

outputs of the network are $a_1, a_2, \ldots, a_m$, then the label actually output by the network is sampled according to:

$$\Pr\left(\text{The } j^{\text{th}} \text{ label is output}\right) = \frac{e^{a_j}}{\sum_{j'} e^{a_{j'}}}. \tag{12.2}$$

You can think of the exponentiation as mapping each a_j (which may be any real number) to e^{a_j}, which is positive. Next, to make the positive numbers into probabilities, they must be normalized so that they sum to one—this is the purpose of the denominator in (12.2).

Notice that the use of softmax changes the parametric model in a fundamental way: it now includes some stochasticity (randomness). When presented with the same input multiple times, it can produce various different outputs.

The remaining challenge to tackle for classification setting is to define a loss function that characterizes how (in)accurate a parametric model is. There are *many* choices of loss functions, just like in the regression setting. One common choice is the **cross entropy loss**.

We will present this loss function in a simplified but very common setting: binary classification. **Binary classification** problems are classification problems where there are two possible labels. Typically these labels are either 0 and 1 or -1 and 1. Binary classification problems are common when determining whether or not the input has a property of interest, like whether an image of a tumor corresponds to a benign or malignant tumor, or whether an image taken from a car includes a pedestrian or not.

In the binary classification setting, the cross entropy loss can be written as:

$$l(w) = -\frac{1}{n}\sum_{i=1}^{n} \ln\left(\Pr\left(f_w(x_i) = y_i\right)\right) \tag{12.3}$$

While we will not present a detailed derivation of this loss function from first principles, notice that it makes sense intuitively. $\Pr(f_w(x_i) = y_i)$ is the probability that the parametric model produces the correct label for the i^{th} point in the data set. The $\frac{1}{n}\sum_{i=1}^{n}$ simply averages over all of the data points, and the negative in the front indicates that larger probabilities of producing the correct label correspond to smaller loss. Finally, the ln operator is monotonic, and so it simply changes the emphasis on how important it is to achieve various probabilities of the correct label. That is, $\ln(x) \approx x - 1$ when $x \approx 1$, but when $x \to 0$, $\ln(x) \to -\infty$. Hence, if the probability of giving the correct label is near zero for one point, it produces a massive loss relative to the loss incurred when the correct label is given a reasonably large (but not close to one) probability for multiple points.

12.4 Generalization Bounds

We know what loss the models that we train achieve on the available training data. However, even when the loss is low on the available data, why do we expect the model to be useful for making predictions for points that are not in the training data? Could it be that the training data

we have are not actually representative of the data the model will face when deployed in the real world? Why do we expect a model trained from some data to be effective for any other data, when we know that future data points will be different from the training points (e.g., we are unlikely to see the exact same handwritten letter twice).

Without any additional knowledge about how the training data was collected, we cannot say anything about how effective the trained model will be when applied to new data points. However, we *can* reason about how effective our trained model will be if we assume that the training data was sampled probabilistically. This view of the training data as a random sample from some population lies at the foundation of ML and is the reason that ML and statistics are so closely related. To see why, we briefly discuss a concept from statistics before returning to discuss how this concept allows us to reason about how effective a trained model will be for data points not in the training data.

Hoeffding's Inequality

You likely have heard of the **law of large numbers**, which states that (under certain mild conditions) if you take many independent samples from some distribution and average the samples, this sample average will "converge"[1] to the actual mean of the distribution as you obtain more and more samples. Hoeffding's inequality is an inequality that characterizes how unlikely it is that the sample mean will deviate from the actual mean by a given amount.

To present Hoeffding's inequality, we begin with a few definitions. Let $Z_1, \ldots, Z_n$ be n independent and identically distributed random variables (n independent samples from some distribution). Let $\bar{Z}_n = \frac{1}{n}\sum_{i=1}^{n} Z_i$ be the sample mean of these n samples. Lastly, let $\mu = \mathbf{E}[Z_i]$ be the actual mean of the distribution.[2]

Hoeffding's inequality characterizes how unlikely it is that the sample mean, $\bar{Z}_n$, differs from the actual mean, μ, by more than some constant, $c \in \mathbb{R}_{>0}$. That is, it provides an upper bound on the probability that $|\bar{Z}_n - \mu| \geq c$. Specifically, Hoeffding's inequality states that:[3]

$$\Pr\left(|\bar{Z}_n - \mu| \geq c\right) \leq 2e^{-\frac{2nc^2}{r^2}}, \tag{12.4}$$

where r is the range of each Z_i—the range of possible values of Z_i.[4]

Now, consider our loss function (using our old notation):

$$l(w_k) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{12.5}$$

This loss function is computed from the available data—n samples from some larger population. If these n samples are sampled i.i.d. from the population, then we can apply Hoeffding's inequality with Z_i being $(y_i - \hat{y}_i)^2$ from (12.5). This gives:

$$\Pr\left(|l(w_k) - \mu| \geq c\right) \leq 2e^{-\frac{2nc^2}{r^2}}, \tag{12.6}$$

1: Technically this is not convergence, but a probabilistic form of convergence. Different assumptions about the random variable (like whether or not it has finite variance) produce different forms of probabilistic convergence, like almost sure convergence or convergence in probability.

2: It does not matter which $i \in \{1, \ldots, n\}$ is used in the definition of μ because all of the Z_i are identically distributed and therefore have the same expected value.

3: Often the constant c is called t when discussing Hoeffding's inequality.

4: More precisely this inequality requires $Z_1, \ldots, Z_n$ to be i.i.d. random variables where $\Pr(Z_1 \in [a, b]) = 1$ for some constants a and b, and our expression uses $r = b - a$. The most general form of Hoeffding's inequality slightly loosens these requirements by only requiring the random variables to be independent, but not necessarily identically distributed.

where μ is the expected loss (the true loss over the entire population), c is any positive constant, and r is the range of the squared losses.[5] This expression indicates that term on the right is an upper bound for the probability that the quality of predictions (measured using the loss function) on the entire population differs from the that on the available data by c or more. Notice that the term on the right decreases as n increases, indicating that as we obtain more data we can become increasingly confident that our model will be accurate when applied to new (possibly previously unseen) data points.

5: For example, if $y_i \in [0, 1]$ and $\hat{y}_i \in [0, 1]$ (like when using a sigmoid in the last layer of an ANN), $r = 1$, since the largest possible residual is 1, and $1^2 = 1$.

Hence (12.6) provides a sort of **generalization bound**, since it characterizes how well a learned model will generalize to unseen data.

Due to subtle reasons beyond the scope of this course, (12.6) only applies when $l(w_k)$ is computed using data that was *not* used to learn the weights w_k. For an excellent and more complete discussion of generalization bounds, we refer the reader to the beginning of "Learning From Data" by Yaser Abu-Mostafa [21] [Amazon].

[21]: Abu-Mostafa et al. (2012), *Learning from Data*

12.5 Overfitting

How can we ensure that a learned model is safe to use? One idea would be to check whether the final loss, $l(w_k)$ for the final value of k, is small. If this value is small, it suggests that the model is making accurate predictions.

This proposed strategy has at least two flaws. First, the loss function does not differentiate between different types of errors. For example, when predicting how far a landslide will travel in order to decide how far from a slope to build a house [10], underestimating the distance the landslide will travel is significantly worse than overestimating the distance. So, we may wish to consider a modified objective that focuses on the types of errors critical to ensuring safety. For this landslide example, this might mean placing a larger weight on errors where the model underpredicts, or even entirely ignoring errors due to overprediction.

Even if the loss function is carefully designed to characterize the safety of the learned model, using the final loss to measure safety is still irresponsible due to the second flaw: that the model will tend to **overfit** the training data. That is, it will tend to be accurate for the training data, but inaccurate for any new data points not included in the training data.

To see why this happens, consider the extreme case of using linear regression (without basis functions) to fit a line to two data points from the GPA data set. Given *any* two data points, one can always fit a line that goes through them. This means that the prediction error for each of the points will be zero, and so the sum of the squared errors (the loss) will also be zero.

However, this does not mean that the line that we have fit will have approximately zero error when the model is applied to new training points! In a sense, the model has "memorized" part of the training data. It therefore produces accurate predictions for the training data, but fails to generalize as well to points that were not seen during training.

This same problem arises when using many points and complicated parametric models. The more complicated the model (generally, the more model parameters there are), the more the model will be able to overfit (think, partially memorize) the training data. In our GPA example using a linear parametric model, if we use all 43,303 data points, there is far too much data for such a small parametric model to memorize, and so overfitting will not be a significant concern. However, for most real problems, either data is extremely limited or the parametric model will be so complex that it can begin to overfit even large data sets.

To visualize overfitting, consider Figure 12.1. The horizontal axis denotes the training time (one might measure this using the number of weight updates or the number of passes that gradient descent takes over the entire data set). The red curve depicts the loss computed on the training data. If the step size is tuned properly, then we expect the red curve to have this general shape—decreasing over time (though not necessarily reaching a loss of zero, as that may not be possible). However, if we compute the loss for data points not provided to the algorithm, the loss will follow a trend like the green curve—initially decreasing as the parametric model learns the correct general shape, but eventually increasing when the parametric model begins overfitting to the training data.

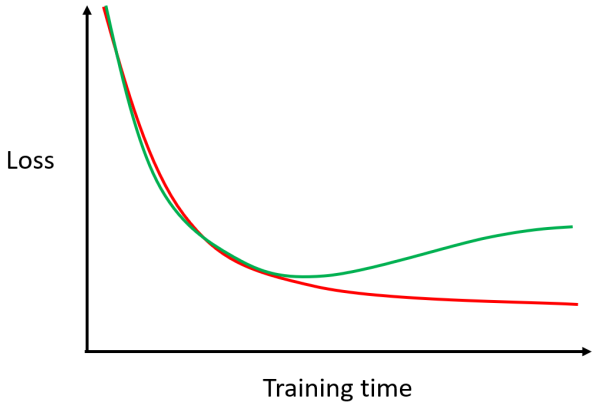


Figure 12.1: Example depicting overfitting. The red curve is the loss function computed using the training data, and the green curve is the loss function when computed using data not provided to the agent during training. Overfitting occurs when the green curve begins increasing in value while the red curve continues decreasing.

There are many ways to overcome overfitting (for another approach not discussed in this introductory course, I recommend studying *regularization*). One strategy is to split the available data into two sets: training data and testing data. The model can then be trained using only the training data. After training, we can compute the loss using the testing data. Since the ML algorithm never saw this data before, the model cannot overfit to it, and so the computed loss will be a reasonable characterization of what will happen if the model is applied to new data points.[6]

6: Recall when discussing generalization techniques that we indicated Hoeffding's inequality must be applied using data not used to train the model—Hoeffding's inequality can be applied using the test set.

However, splitting into training and testing sets only fixes part of the problem (it fixes the overestimation of how well the model will perform). The ML algorithm will still tend to overfit to the training data, resulting in low accuracy on the test data (and future data points for which predictions are required). To overcome this, we partition the training data set into two sets: one still called the training data, and the other called the validation data. The ML model is trained using only the training data. However, during training we also track the loss function computed from the validation data. When the loss on the validation begins to increase,

we can stop training as further training will result in overfitting to the training data.

This train-validation-test splitting of data is common and allows for various techniques for improving the reliability and accuracy of ML models. In general, the training data is used to train the model, the validation data is used to determine which settings of hyperparameters are most effective (like when to stop training to avoid overfitting), and the test data is used after the final model has been selected to estimate how well the model will perform when applied to new data points.

Reinforcement Learning

What is Reinforcement Learning 13

Reinforcement learning is an area of machine learning, inspired by behaviorist psychology, concerned with how an agent can learn from interactions with an environment.

–Wikipedia, Sutton and Barto [22], Phil

[22]: Sutton et al. (1998), *Reinforcement Learning: An Introduction*

This process of learning via interactions with an environment is often depicted using a **agent-environment diagram** like Figure 13.1. The agent makes some **observations** about the state of the environment (the state of the world around it). More specifically, the **state** of the environment is any complete description of the environment at a given moment. For us humans as agents, the environment is the universe and the state is a complete characterization of the universe at a given moment.

Clearly we do not know, nor will we ever know, the exact state of the entire universe—the positions and velocities of every single particle. However, the state does exist. Similarly, the agent may not know the complete state of the environment, but this state does exist. The agent's **observation** of the state is typically both incomplete (for example, at the moment I am unable to observe the position of the moon or whether a person is moments away from hitting my doorbell) and noisy (since almost all sensors introduce some noise or uncertainty in their measurements).

Although observations and states are different, for the purposes of this introductory course we will treat them as interchangeable. For simplicity, you might assume for now that the agent's observations of the state of the environment are complete and noise-free. However, the methods that we will present are effective even when the observations are incomplete and imperfect.

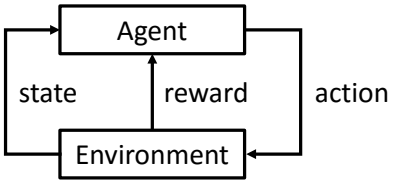


Figure 13.1: Agent-environment diagram. Examples of **agents** include a child, dog, robot, program, etc. Examples of **environments** include the world, lab, software environment, etc.

RL has two key concepts that differentiate it from supervised learning: evaluative feedback and sequential decisions.

Evaluative Feedback: Rewards convey how "good" an agent's actions are, not what the best actions would have been. If the agent was given instructive feedback (what action it should have taken) this would be a *supervised learning* problem, not a reinforcement learning problem.

Sequential: The entire sequence of actions must be optimized to maximize the "total" reward the agent obtains. This might require forgoing immediate rewards to obtain larger rewards later. Also, the way that the agent makes decisions (selects actions) changes the distribution of states that it sees. This means that RL problems aren't provided as fixed data

sets like in supervised learning, but instead as code or descriptions of the entire environment.

Question: If the agent-environment diagram describes a child learning to walk, what exactly is the "Agent" block? Is it the child's brain, and its body is part of the environment? Is the agent the entire physical child? If the diagram describes a robot, are its sensors part of the environment or the agent?

Answer: Either perspective is valid. It is up to you to determine the perspective that best fits your use of RL. If you are studying high-level animal behavior, it is likely more useful to view the entire animal as the agent. If you are using RL to balance a robot, it is likely more useful to view the physical actuators and their complex dynamics as part of the environment.

Neuroscience and *psychology* ask how animals learn. Those fields are the study of some examples of learning and intelligence. RL asks how we can make an agent that learns. It is the study of learning and intelligence in general (animal, computer, matchboxes, purely theoretical, etc.). In this course, we may discuss the relationship between RL and computational neuroscience in one lecture, but, in general, we will *not* concern ourselves with how animals learn (other than, perhaps, for intuition and motivation).

There are many other fields that are similar and related to RL. Separate research fields often do not communicate much, resulting in different language and approaches. Other notable fields related to RL include operations research and control (classical, adaptive, etc.). Although these fields are similar to RL, there are often subtle but impactful differences between the problems studied in these other fields and in RL. Examples include whether the dynamics of the environment are known to the agent *a priori* (they are not in RL) and whether the dynamics of the environment will be estimated by the agent (many, but not all, RL agents do not directly estimate the dynamics of the environment). There are also many less impactful differences, like differences in notation (in control, the environment is called the *plant*, the agent the *controller*, the reward the (negative) *cost*, the state the *feedback*, etc.).

A common misconception is that RL is an alternative to supervised learning—that one might take a supervised learning problem and convert it into an RL problem in order to apply sophisticated RL methods. For example, one might treat the state as the input to a classifier, the action as a label, and the reward as -1 if the label is correct and 1 otherwise. Although this is technically possible and a valid use of RL, it *should not be done*. In a sense, RL should be a last resort—the tool that you use when supervised learning algorithms cannot solve the problem you are interested in. If you have labels for your data, do *not* discard them and convert the feedback from instructive feedback to evaluative feedback (telling the agent if it was right or wrong). The RL methods will likely be far worse than standard supervised learning algorithms. However, if you have a sequential problem or a problem where only evaluative feedback is available (or both!), then you cannot apply supervised learning methods and you should use RL.

MENACE, Notation, and Problem Formulation

14

14.1 Machine Educable Naughts and Crosses Engine (MENACE)

We will begin our study of RL algorithms with one of the earliest RL agents, the Machine Educable Noughts and Crosses Engine (**MENACE**), which was built by Donald Michie in 1961 and published in 1963. Without access to a computer, Michie decided to make a computational system that learns to play the game noughts and crosses (tic-tac-toe in the United States) using matchboxes and beads. Readers not familiar with noughts and crosses (tic-tac-toe) should read about it on Wikipedia [link], as we will not review the game rules here.

Michie determined that there were 304 possible board configurations that MENACE could possibly faced with.[1] He therefore obtained 304 matchboxes and associated each with one board state (e.g., you can imagine the board state being drawn on the matchbox, though that was not how it was actually implemented). Next, he used beads of nine different colors to represent each possible move. He filled each matchbox with beads of the colors corresponding to legal moves from that state.

1: There are actually significantly more possible board configurations, but many are equivalent. For example, a board with only an X in the top left corner and a board with only an X in the top right corner are effectively equivalent—they are simply rotations of each other, and rotations make no difference to the game.

To obtain a move from MENACE, one can select the matchbox corresponding to the current board state, randomly select a bead from the box, and take the move that corresponds to the bead. When MENACE is learning, the selected beads are left out of the box until the end of the game. At the end, the beads are replaced or discarded depending on the outcome of the game.

If MENACE won, then the beads are returned to the matchboxes along with an extra three beads of the same color. This makes it significantly more likely that these moves, which resulted in a win, will be played again during future games. If MENACE lost, then the beads are not returned to the matchboxes. This reduces the probability of MENACE making these moves in future games. If the game was a draw (sometimes called a *cat's game*), then the beads are returned to the matchboxes along with one extra bead of the same color. This makes it slightly more likely that these moves will be played again during future games.[2]

2: This reinforcing of moves that result in a draw makes sense for this game because if both players play optimally the outcome will be a draw.

Exploration Versus Exploitation

Notice that MENACE does not always select the move that it thinks is best (this would happen if the most common bead color is always selected from each matchbox). The behavior of RL agents can roughly be classified as either **exploration** or **exploitation**. Exploration is when the agent selects an action that it does not think is optimal in order to learn more about the action's outcome—perhaps it is better than the agent thinks, and the only way to determine this is to try the action.

Exploitation is when the agent selects an action that it thinks is optimal in order to obtain the resulting large (expected) sum of rewards.

To see why both exploration and exploitation are necessary, consider trying to find the fastest route to work after moving to a new city (without using a service that reports fastest routes while accounting for traffic). You expect the commute to take 30 minutes, but on the first day, you arrive in just 20 minutes. As the route you took was better than what you expect any route to be, you go the same way every day. What if there were an alternative route that you didn't try and which actually takes 10 minutes on average? In order to try this route, you must sometimes explore other routes, rather than only exploiting the one good one you found. Moreover, it could be that the first time you try a faster route it happens to be slower due to random traffic, so you may need to try each path many times. Hence, an exploitation-only learning strategy can cause the agent to get stuck taking suboptimal behaviors because it fails to try out other, potentially better, behaviors.

Similarly, an exploration-only learning strategy wouldn't be reasonable: at some point, we should leverage the knowledge we have gained from all the exploration. If you randomly select routes to work every morning, you will have a very good idea about what the fastest route is, having thoroughly explored all of the possibilities. However, if you keep selecting paths (uniformly) randomly, you won't be taking advantage of this knowledge. Hence, exploration and exploitation are a trade-off: agents must balance how much they explore (trying out actions they think are, or might be, suboptimal) with how much they exploit (taking the actions they think are best). Within MENACE, exploration is captured by the random process of selecting a bead from the matchbox.

14.2 Operant Conditioning

At their core, RL algorithms are a form of **operant conditioning**, a learning mechanism wherein animals reinforce (make more likely) behaviors that result in rewards and avoid behaviors that do not. [Wiki]

Consider MENACE as an example. Essentially, MENACE waits until the end of the game and observes the outcome of its actions (a win, loss, or draw). When the outcome is "good," MENACE increases the probability that it will play the chosen actions in future games, and if the outcome is "bad," MENACE decreases that probability. This simple concept underlies nearly every RL algorithm, and in this part of the course, we will formalize and refine this idea to produce effective RL algorithms.

14.3 Notation

To reason more formally about RL agents and problems, we begin by establishing symbols for the various terms like states, actions, and rewards.

1. Time progresses in a sequence of discrete **time steps**, indexed by $t \in \{0, 1, 2, \dots\}$.

2. Let S_t be the **state** of the environment at time t. The set of possible states can be any set. Here, we will focus on the case where the state is a vector of numbers (features).
3. Let A_t be the **action** chosen by the agent at time t. For simplicity, we will assume that there are a small finite number of possible actions. However, the concepts that we describe generally carry over to the settings where actions are continuous or hybrid (a combination of discrete and continuous).
4. Let R_t be the reward provided to the agent at time t, as a result of the agent taking action A_t in state S_t and the environment transitioning to state S_{t+1}. The reward is a real number, that is, $R_t \in \mathbb{R}$.

We call any way that the agent can select actions based on the state (observation) a **policy**, π. Because we want the agent to be able to explore, the policy is stochastic (it has some randomness), and so it doesn't directly map states to actions. Instead, a policy is a distribution over the set of possible actions, conditioned on the state. That is, $\pi(s, a) = \Pr(A_t = a | S_t = s)$. We assume that the distribution of the actions only depends on the state, not the time step t, and so the left and right hand sides are equal for all values of t.[3]

3: To make actions depend on the time step, simply include the time step within your definition of the state.

There are many (uncountably infinite) policies, some "good" and some "bad." But, what exactly does "good" or "bad" mean? How do we measure the quality of a policy? Intuitively, the agent should search for a policy that maximizes the total amount of rewards that it receives. We can try to encode this within an objective function J:[4]

$$J(\pi) \not\models \sum_{t=0}^{\infty} R_t, \tag{14.1}$$

4: We call J the "objective function" rather than the "loss function" because the agent's goal is the *maximize* J, not the minimize it. A loss function is an objective function that is to be minimized, while an objective function could be one that an agent aims to minimize or maximize.

where we write $\not\models$ to indicate that this is *not* the actual definition of J, but a stepping stone towards the full definition of J.

While (14.1) captures the general intuition behind maximizing the total reward that the agent receives, it has a few flaws that we will fix. First, if the agent uses the same exact policy twice, the resulting rewards can be different due to stochasticity in the state transitions, rewards, and chosen actions. Even if you drive the same route to work every day, the amount of time that it takes will vary slightly. Similarly, even if you use the same exact formula for determining how much insulin a type 1 diabetic should inject prior to eating a meal, the resulting blood sugar levels for the day will vary. To handle this, we consider the *expected* sum of rewards:

$$J(\pi) \not\models \mathbf{E}\left[\sum_{t=0}^{\infty} R_t\right]. \tag{14.2}$$

Next, notice that $J(\pi)$ as defined in (14.2) could be $\pm\infty$, for example, if $R_t = 1$ always. While this can be handled in many ways, one common strategy ensures that $J(\pi)$ is always finite: reward discounting. This strategy is also appealing because it captures another aspect of how animals make decisions. Specifically, a reward is worth more to an agent the sooner it occurs. When presented with the opportunity to have one cookie right now versus two cookies next year, many people will select one cookie right now. To capture this preference for rewards that occur

in the near future relative to those that occur in the distant future, we can introduce a hyperparameter $\gamma \in [0, 1]$. The utility of a reward to the agent is then $\gamma^t R_t$. For example, if $\gamma = 0.5$, then a reward of 1 at $t = 0$ is worth 1 to the agent, while a reward of 1 at time $t = 1$ is worth 0.5 to the agent and a reward of 1 at time $t = 2$ is worth just 0.25 to the agent. Typically values of γ are around 0.9, 0.95, 0.99, or 1.0.[5]

5: It is acceptable for $\gamma = 1$, which results in no discounting of rewards, if other assumptions are used to ensure that $J(\pi)$ is finite, like assuming that the sequence of decisions has a bounded length.

This results in the updated objective function definition:

$$J(\pi) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right]. \tag{14.3}$$

While this definition is the correct and full definition of the objective function, J, often it is written differently to emphasize *how* the right side is a function of π. That is, notice that π is the argument to J on the left side of (14.3), but doesn't appear on the right side. This is because the distributions of the rewards R_t (for all t) depend on the actions chosen by the agent, and the distribution over actions is defined by the policy. So, π influences the distribution of R_t. To make this explicit, authors sometimes write:

$$J(\pi) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \middle| \pi\right], \tag{14.4}$$

where conditioning on π indicates that $A_t \sim \pi(S_t, \cdot)$, i.e., the action A_t is sampled according to the policy π given the current state S_t.

While (14.4) is the most common definition of J, I support a change of notation within the field. The problem with (14.4) is that it looks like a *conditional expectation*, but it is *not* a conditional expectation. That is, π is not an event that is being conditioned on. The curious reader may work out how this is a problem, for example, by writing out the definition of the conditional expectation and then applying Bayes' theorem to the conditional probability. This will result in an expression that is absolute nonsense, showing that (14.4) is an abuse of notation (it uses the notation of conditional expectations when it is not really one). To fix this, we will use a semicolon rather than the "given" symbol:

$$J(\pi) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t ; \pi\right], \tag{14.5}$$

where $; \pi$ is a reminder that π influences the distribution of one or more random variables in the expected value (R_t in this case).

We can now express the agent's goal in math: find an **optimal policy** π^*, which is any policy that satisfies:

$$\pi^* \in \arg\max_{\pi} J(\pi). \tag{14.6}$$

Notice that there may be multiple optimal policies. For example, if R_t is always equal to a constant regardless of the actions chosen by the agent, then all policies are optimal. Also, while (14.6) characterizes the goal, in practice we often cannot hope to find a policy that is actually optimal, and instead aim to find policies that are better than some currently used policy (as measured by J). For example, we likely will never know the truly optimal insulin doses to give a person with type 1 diabetes before

eating a particular meal. However, we can search for insulin doses that will be more effective than the current insulin-dosing policy.

Episodes and Additional Notation

15

15.1 Episodes

Often RL problems can be broken into multiple independent trials. For example, each game that MENACE plays can be viewed as one trial. While we assume that these trials share the same environment dynamics (the same rules and opponent strategy), the actions selected by the agent during one trial do not influence the state transitions or rewards during a future trial. Similarly, when modeling bolus insulin dosing as an RL problem, researchers often assume that each day is an independent trial with three actions (the injections prior to breakfast, lunch, and dinner).

In RL, these trials are called **episodes**. Each episode begins at time $t = 0$. Episodes are not necessarily the same length, and individual episodes may even be infinitely long (for example, if episodes end when the agent causes the environment to enter a state called a *goal state*, but the agent's learning mechanism fails, resulting in the agent never selecting actions that would cause the environment to enter the goal state).

15.2 Trials

While we referred to episodes as "trials" in the previous section, a **trial** in RL actually refers to something different: an entire agent lifetime. Imagine creating a plot where the horizontal axis is the number of episodes that have passed (e.g., the number of games of tic-tac-toe), and the vertical axis is the discounted sum of rewards that the agent received during each episode. If the agent is effective at learning, then as it plays more games it will win more games, resulting in larger rewards. So, we might hope that this plot will have an upwards trend as the number of episodes increases.

However, due to the random nature of S_t, A_t, and even sometimes R_t, the points may not form a smooth trend—they may be noisy samples around some true trend. To get a better picture of how effective the learning algorithm used by the agent is, we might run this entire experiment multiple times and average the results. This can even be done in parallel: create K different agents and environments that are identical except for the random seed used. Next, run each agent for a **lifetime**, which corresponds to some fixed number of episodes. In RL, K is referred to as the number of *trials* or *runs*.

Having run K trials, we can update the plot that we described previously. At position x on the horizontal axis (which corresponds to the x^{th} episode), plot the *average* discounted sum of rewards that the K agents received during their x^{th} episodes. This gives a clear impression of how the agent learns on average. As an example, see Figure 5b here. Notice that the

vertical axis is labeled **return**: the "return" is a brief way to refer to the discounted sum of rewards.

15.3 Reward Design

You might be wondering: who defines R_t? The answer to this differs depending on how you are trying to use RL. If you are using RL to solve a specific real-world problem, then it is up to you to define R_t to cause the agent to produce the behavior you want. It is well known that we as humans are bad at defining rewards that cause optimal behavior to be what we want. Often, you may find that you define rewards to produce the behavior you want, train an agent, think the agent is failing, and then realize that the agent has in a way outsmarted you by finding an unanticipated way to maximize the expected discounted return via behavior that you do not want.

Consider an example, where you want to give an RL agent (represented by the dog) rewards to get it to walk along the sidewalk to a door (which ends the episode) while avoiding a flowerbed: How would you assign

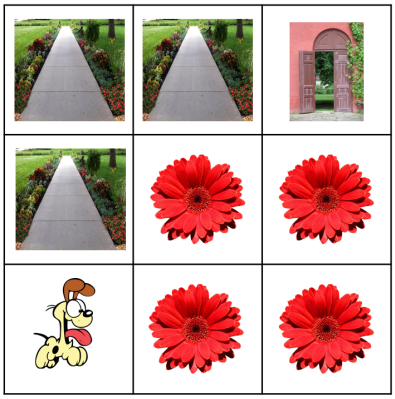


Figure 15.1: A simple "gridworld" environment. The dog is the agent, starting in the bottom left cell. It has four actions: move up, down, left, or right. When it reaches the door in the top right, the episode ends.

rewards to states in order to get the dog to go to the door? Humans frequently assign rewards in a way that causes undesirable behavior for this example. One mistake is to give positive rewards for walking on the sidewalk—in that case the agent will learn to walk back and forth on the sidewalk gathering more and more rewards, rather than going to the door where the episode ends. In this case, optimal behavior is produced by putting negative rewards on the flowerbed, a positive reward at the door, and zero reward along the sidewalk (other solutions exist, like putting negative rewards everywhere with larger negative rewards on the flowers).

This provides a general rule of thumb when designing rewards: give rewards for what you want the agent to achieve, not for how you think the agent should achieve it. Rewards that are given to help the agent quickly identify what behavior is optimal are related to something called *shaping rewards* [23]. When done properly, shaping rewards can be designed such that they will not change the optimal policy. However, when we simply make up shaping rewards (like putting a positive constant on the sidewalk states in the above example), they often will change optimal behavior in an undesirable way.

Back to the earlier question: who defines R_t? So far we have discussed how *you* can choose R_t when applying RL to a problem. Some researchers study other ways that rewards can be defined. For example, *inverse reinforcement learning* (IRL) involves observing the behavior of an optimal agent and trying to infer what definition of rewards the agent's behavior optimizes. This is useful because it enables people to give examples of the behavior they want an agent to learn (e.g., how should a doctor trade-off side-effects with improved outcomes when deciding which drugs to prescribe?), and IRL methods can then infer the corresponding definition of rewards. For example, a person could give an example to a robot of placing dishes in a dishwasher. Ideally it would infer the goal (place dishes in dishwasher without breaking them) and would then be able to use RL to find a policy that allows it to successfully load the dishwasher. IRL is related to the area of *value alignment*, which focuses on ensuring that the goals of agents (RL and otherwise) are aligned with our goals.

15.4 How to Represent π?

Notice that (14.6) calls for optimizing the agent's policy. To implement this, we need a way to represent a policy on a computer. The most common way to do this is to use a **parameterized policy** π_θ, which is parameterized just like f_w was a parameterized model in the supervised learning chapters. That is, θ is a vector of parameters of the policy, and changing θ changes the policy (how actions are selected).

Recall that the models used in the supervised learning section were typically deterministic: always mapping inputs to the same labels. However, the models used for classification were stochastic, producing a distribution over possible labels. Parameterized policies are generally constructed identically to parametric models for classification: there is one output per possible action, and softmax is used to convert these outputs into probabilities of each action. Specifically, if o_i is the i^{th} output of the network, then

$$\pi(s,a) = \Pr(A_t = a | S_t = s) = \frac{e^{o_a}}{\sum_{a'} e^{o_{a'}}}. \tag{15.1}$$

While π could be represented with an ANN, for many RL problems it is easier, faster, and just as effective to use linear policies with basis functions. That is, let $\phi(s)$ be a vector of features computed from s (just like $\phi(x_i)$ in Section 7.3). For example, ϕ could be the Fourier basis [15]. Next, notice that there should be one output for each possible action. Let θ^i be the vector of policy parameters used for the i^{th} output. That is, $\theta^i = (\theta^i_1, \theta^i_2, \dots)$, where each θ^i_j is a real number—the weight on the j^{th} input when computing the i^{th} output.

[15]: Konidaris et al. (2011), 'Value function approximation in reinforcement learning using the Fourier basis'

Combining this with softmax action selection as in (15.1), we obtain the following equation for computing action probabilities:

$$\pi(s,a) = \frac{e^{\sum_j \theta^a_j \phi_j(s)}}{\sum_{a'} e^{\sum_j \theta^{a'}_j \phi_j(s)}}, \tag{15.2}$$

where $\phi_j(s)$ denotes the j^{th} feature output by ϕ.

15.5 From Supervised Learning to RL

Even though many of the concepts we described in supervised learning carry over to the RL setting, they are often referenced using different terminology or notation. In this section, we provide a list of notation and terminology changes.

1. **model→policy**: In supervised learning an agent tries to train a model, which maps inputs to the agent to outputs of the agent. In RL, an agent tries to train a policy. While models for regression are often deterministic, models for classification and RL are typically stochastic.
2. **model parameters→policy parameters**: Whereas w were the model parameters in supervised learning, we refer to θ as the **policy parameters** in RL. However, the two are similar: they are the parameters or weights that are learned by the agent.
3. $f_w \rightarrow \pi_\theta$: In supervised learning, we wrote f_w to denote the parametric model with parameters w, while in RL we write π_θ to denote the parameterized policy with parameters θ.
4. $l \rightarrow -J$: The goal in supervised learning is to minimize the loss function, l, while in RL the agent's goal is to maximize the objective function J, which is equivalent to minimizing $-J$.

16 Policy Gradient Part 1

16.1 A Simple RL Algorithm

We will now present a simple RL algorithm, beginning with high-level intuitive pseudocode and then refining this pseudocode into a precise algorithm. First, Algorithm 16.1 describes the general flow of the algorithm, which mimics MENACE.

Algorithm 16.1: A simple RL algorithm inspired by MENACE.

```
for each episode do
    // Run one episode (play one game).
    for each time t in the episode do
        Agent observes state S_t;
        Agent selects action A_t according to the current policy, π_θ;
        Environment responds by transitioning from state S_t to state
          S_{t+1} and emitting reward R_t;
    end
    // Learn from the outcome of the episode.
    if Σ_{t=0}^{∞} γ^t R_t is big then
        for each time t in the episode do
            Make action A_t more likely in state S_t;
        end
    end
    if Σ_{t=0}^{∞} γ^t R_t is small then
        for each time t in the episode do
            Make action A_t less likely in state S_t;
        end
    end
end
```

The outer for-loop iterates over episodes (for example, games of tic-tac-toe). Each episode is then handled in two phases. During the first phase, the episode is run using the current policy, π_θ. During the second phase, the agent changes its policy based on the outcome of the game. If the agent did well (in terms of the total discounted reward, $\sum_{t=0}^{\infty} \gamma^t R_t$), then the actions that it chose are reinforced—they are made more likely (this corresponds to putting multiple beads back into each MENACE matchbox, matching the color of the beads to the bead color that was chosen during the game). If the agent did poorly, then the actions that it chose are made less likely (in MENACE, this corresponds to not replacing the chosen beads).

To refine this algorithmic outline into a specific algorithm, we first consider how to implement the lines "Make action A_t more likely in state S_t" and "Make action A_t less likely in state S_t."

Make Action A_t More Likely in State S_t

If f is a function that takes two inputs and produces a real number, then recall that

$$\frac{\partial}{\partial y} f(x, y) \tag{16.1}$$

is an indication of how y should be changed to increase $f(x, y)$. That is, if this partial derivative is positive, then increasing y will increase $f(x, y)$. Similarly, if this partial derivative is negative, then increasing y will decrease $f(x, y)$. If f is a smooth function, then locally (around the current value of y), the partial derivative being negative means that while increasing y will decrease $f(x, y)$, then *decreasing* y will necessarily increase $f(x, y)$.

Recall that $\pi_\theta(s, a) = \Pr(A_t = a | S_t = s; \theta)$. Furthermore, the θ subscript is just another way of indicating that the output of π depends on θ, and so we could write $\pi_\theta(s, a)$ as $\pi(s, a, \theta)$.

With these two properties, it is clear how to change θ to make action A_t more likely in state S_t: We change each policy parameter in the direction of the partial derivative of $\pi_\theta(s, a)$ with respect to the policy parameter. That is, if $\theta = (\theta_1, \theta_2, \dots)$,[1] then we can make the following update to each policy parameter θ_i:

$$\theta_i \leftarrow \theta_i + \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}. \tag{16.2}$$

1: We often store policy parameters in different structures, like matrices or arrays of arrays. For example, recall that during the discussion of linear softmax policies we kept a different policy parameter vector, θ^a, for each action a. However, even when using these other policy representations, the math is easiest if we view θ as one long vector. So, hereafter, we assume that $\theta = (\theta_1, \theta_2, \dots)$, where each θ_i is a real number. Notice that we can easily convert between these different perspectives by keeping the same policy parameters, but changing how they are stored.

The update in (16.2) *might* work, but it has a problem: It could make very large changes to the policy parameters. This is undesirable for at least two reasons. First, we may not want to make massive changes to the agent's behavior based on the outcome of a single episode, especially for applications where the outcomes of episodes include significant stochasticity (randomness). Second, the partial derivative is an inherently *local* quantity. For nonlinear policy representations like ANNs, if $\partial \pi_\theta(S_t, A_t)/\partial \theta_i$ is positive, it means that locally, close to the current value of θ_i, increasing θ_i will increase $\pi_\theta(S_t, A_t)$. However, as θ_i continues to increase, this may change, and continued increases to θ_i could actually *decrease* $\pi_\theta(S_t, A_t)$. That is, steps that are too large could actually *decrease* the probability of action A_t in state S_t.

To remedy these two problems, we insert a **step size** α, which is typically a small constant.[2] The resulting update is:

$$\theta_i \leftarrow \theta_i + \alpha \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}. \tag{16.3}$$

2: I've seen anywhere from $\alpha = 0.000001$ to 1.0. A general trend when using linear function approximation is that one over the total number of features tends to be a good starting point when selecting step sizes.

This step size, α, is yet another hyperparameter, scaling how much the agent updates its behavior from the outcome of a single episode. For some problems, α should be small—if you bite your tongue once while chewing, you shouldn't stop chewing your food for a week. For other problems, α should be large—if you eat food that makes you sick, you are likely to be repulsed by it for weeks! Furthermore, if α is too large, the agent can run into issues with the local nature of the partial derivative, resulting in the algorithm's divergence.

Make Action A_t Less Likely in State S_t

Recall that $\partial\pi_\theta(S_t, A_t)/\partial\theta_i$ indicates how to change θ_i to *increase* the value of $\pi_\theta(S_t, A_t)$. Also, recall that when we zoom in far enough on a smooth function, it will appear flat (planar). For a planar function, if one direction points uphill, then the other direction necessarily points downhill. So, to obtain a direction of change to θ_i that will *decrease* $\pi_\theta(S_t, A_t)$, we need only reverse the direction of change used to increase $\pi_\theta(S_t, A_t)$. That is, we multiply the partial derivative by -1, resulting in the update:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}. \tag{16.4}$$

A Simple RL Algorithm v2.0

Replacing "Make action A_t more likely in state S_t" and "Make action A_t less likely in state S_t" with the updates that we have derived, we obtain Algorithm 16.2.

Algorithm 16.2: A simple RL algorithm inspired by MENACE, Version 2.0

```
for each episode do
    // Run one episode (play one game).
    for each time t in the episode do
        Agent observes state S_t;
        Agent selects action A_t according to the current policy, π_θ;
        Environment responds by transitioning from state S_t to state
         S_{t+1} and emitting reward R_t;
    end
    // Learn from the outcome of the episode.
    if Σ_{t=0}^{∞} γ^t R_t is big then
        for each time t in the episode do
            ∀i, θ_i ← θ_i + α ∂π_θ(S_t,A_t)/∂θ_i;
        end
    end
    if Σ_{t=0}^{∞} γ^t R_t is small then
        for each time t in the episode do
            ∀i, θ_i ← θ_i − α ∂π_θ(S_t,A_t)/∂θ_i;
        end
    end
end
```

Is The Discounted Sum of Rewards Big or Small?

Next, we focus on the if-statements testing whether $\sum_{t=0}^{\infty} \gamma^r R_t$ is big or small (whether the outcome was good or bad, respectively). Consider how we might model a board game like tic-tac-toe as an environment that emits rewards. One option would be to give a reward of +3 when the agent wins and a reward of -1 when the agent loses. However, another option would be to give the agent a reward of -1 when it wins and -100 when it loses. So, if the agent receives a reward of -1 is that good or bad? The agent doesn't know!

To fix this, we entirely avoid checking whether the outcome was good or bad—whether $\sum_{t=0}^{\infty} \gamma^t R_t$ was big or small. Instead, we *weight* the update by the discounted sum of rewards, $\sum_{t=0}^{\infty} \gamma^t R_t$:

$$\forall i,\ \theta_i \leftarrow \theta_i + \alpha \left(\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'} \right) \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}. \tag{16.5}$$

To understand this update, first consider the case where a "good" outcome means that $\sum_{t=0}^{\infty} \gamma^t R_t = 4$ and a "bad" outcome means that $\sum_{t=0}^{\infty} \gamma^r R_t = -1$. This resembles MENACE, which returns four beads total if MENACE wins, and removes one bead if MENACE loses. In this case, (16.5) places a positive weight on the partial derivative when the actions should be made more likely and a negative weight when the actions should be made less likely—exactly the behavior we want.

Now, consider a more subtle case. What if a "good" outcome corresponds to $\sum_{t=0}^{\infty} \gamma^r R_t = 10$ and a "bad" outcome corresponds to $\sum_{t=0}^{\infty} \gamma^r R_t = 9$? In this case, it seems like (16.5) does the wrong thing—increasing the probability of actions in both cases. However, consider what happens on average. Imagine that there were only one possible state and two possible actions, a_1 and a_2, where a_1 produces the "good" outcome and a_2 produces the "bad" outcome (quite a boring game with only one state and two actions!). When the agent takes action a_2 (resulting in a loss), (16.5) *does* increase the probability of action a_2. However, when the agent takes action a_1 (resulting in a win), (16.5) increases the probability of action a_1 even more, since it is given a larger weight (10 as opposed to 9). Since probability distributions always sum to one, we know that the probabilities of actions a_1 and a_2 must sum to one. If we constantly try to increase both probabilities, but try to increase the probability of action a_1 more than that of a_2, then on average a_1 will win out—it will become more likely and a_2 will necessarily become less likely (since the probability of a_1 plus the probability of a_2 must always be one). This general trend carries over to settings with any number of states and actions: Actions that result in larger expected discounted sums of rewards will tend to be made more likely than actions that result in smaller sums of rewards.

Plugging this update in for both of the cases where the discounted sum of rewards is big or small, we obtain Algorithm 16.3.

Algorithm 16.3: A simple RL algorithm inspired by MENACE, Version 3.0

for *each episode* **do**
  `// Run one episode (play one game).`
  for *each time t in the episode* **do**
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
  end
  `// Learn from the outcome of the episode.`
  for *each time t in the episode* **do**
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \left(\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'} \right) \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$;
  end
end

17 Policy Gradient Part 2

17.1 Improving the MENACE-Like Algorithm

We can further improve the update in Algorithm 16.3. Consider what happens after the episode has been completed and the agent is in the process of updating its policy parameters—at time $t = 5$ in the for-loop on Line 9 of Algorithm 16.3. The update to θ_i is given by:

$$\forall i,\ \theta_i \leftarrow \theta_i + \alpha \left(\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'} \right) \frac{\partial \pi_\theta(S_5, A_5)}{\partial \theta_i}. \tag{17.1}$$

Notice that (17.1) is simply (16.5) but with t replaced with 5. Recall that $\partial \pi_\theta(S_5, A_5)/\partial \theta_i$ is the direction of change to θ_i that increases the probability of action A_5 in state S_5. Finally, notice that the weight given to this direction, $\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'}$, considers all of the rewards, including $R_1, R_2, \ldots, R_4$.

This inclusion of all rewards in the weight is a bit odd. If we assume that decisions made in the future cannot have a causal impact on rewards in the past (a natural assumption), then why do these rewards have any impact on how we update the decision made at time $t = 5$? We can fix this by only considering the rewards that happen *after* action A_t is chosen by the agent. These are rewards $R_t, R_{t+1}, R_{t+2}, \ldots$.

There are two ways that we could do this. We could replace $\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'}$ with

$$\sum_{t'=t}^{\infty} \gamma^{t'} R_{t'} = \gamma^t \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{17.2}$$

or

$$\sum_{k=0}^{\infty} \gamma^k R_{t+k}. \tag{17.3}$$

Both of these only consider the rewards starting with R_t. The difference is how the rewards are discounted. Notice that (17.2) discounts R_t by γ^t while (17.2) discounts R_t by γ^0.

Taking the agent's perspective at time t, its goal is to maximize the expected discounted sum of rewards that it obtains in the future, in which case the reward R_t should not be discounted, as it is the "next" reward from the agent's perspective when selecting action A_t in state S_t. Given this point of view, we use (17.3) rather than (17.2). This results in a further refinement of our MENACE-like algorithm, presented in Algorithm 17.1.

If we were to use (17.2), which results in an extra γ^t before the sum over k, and if we were to replace $\frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$ with $\frac{\partial \ln(\pi_\theta(S_t, A_t))}{\partial \theta_i}$, this would result in Algorithm 17.2, which is a common implementation of REINFORCE [24], which is very close to being stochastic gradient ascent on $J(\theta)$.[1]

1: Though it is beyond the scope of this course, for completeness we point out that to modify our implementation of REINFORCE to truly be stochastic gradient ascent, when updating the policy the loop over time steps should be converted into a summation within the update, resulting in the update: $\theta_i \leftarrow \theta_i + \alpha \sum_{t=0}^{\infty} \gamma^t (\sum_{k=0}^{\infty} \gamma^k R_{t+k}) \frac{\partial \ln(\pi_\theta(S_t, A_t))}{\partial \theta_i}$. The small difference this makes is that when computing $\partial \pi_\theta(S_t, A_t)/\partial \theta_i$ at time $t > 0$, the actual gradient ascent update uses the policy parameters θ that were used when running the episode, whereas our REINFORCE implementation uses the parameters θ that have already been updated using the data up to time t from the current episode.

Algorithm 17.1: A simple RL algorithm inspired by MENACE, Version 4.0

for *each episode* **do**
  `// Run one episode (play one game).`
  for *each time t in the episode* **do**
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
  end
  `// Learn from the outcome of the episode.`
  for *each time t in the episode* **do**
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right) \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$;
  end
end

Algorithm 17.2: REINFORCE

for *each episode* **do**
  `// Run one episode (play one game).`
  for *each time t in the episode* **do**
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
  end
  `// Learn from the outcome of the episode.`
  for *each time t in the episode* **do**
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \gamma^t \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right) \frac{\partial \ln(\pi_\theta(S_t, A_t))}{\partial \theta_i}$;
  end
end

17.2 Value Functions and Updating during Episodes

So far our MENACE-like algorithm has two phases. During the first phase, the agent interacts with the environment for one full episode using its current policy. During the second phase, the agent learns from its experiences during the episode, changing its policy.

It would be better if the agent could also learn during each episode, not just between episodes. To achieve this, we will refine our algorithm to be a little different from MENACE. MENACE was built on the idea that actions should be made more likely when they result in a good outcome (a win at the end of the game/episode). However, we do not need to wait for an outcome (or even a reward) before we can start learning.

As an example, imagine that you play the lottery and learn that you have won. In the future, you will obtain the prize money, which may result in actual rewards (in the form of increased comfort and pleasure). However, you will likely be happy and celebrate, perhaps learning that you should play the lottery more, all *before* you actually collect any prize money or see the actual impact that it has on your life. This learning isn't due to any

rewards—you have received no actual rewards yet. Instead, this learning is due to your expectations of future rewards.

When you realize that you have won the lottery, you recognize that the expected outcome was better than you were previously expecting, and this is sufficient for you to learn. That is, instead of making actions more likely when they result in an observed desirable outcome, the agent can make actions more likely when it makes observations that cause it to believe it will obtain more reward than previously expected. In order to do so, the agent must have some notion of how much reward it is expecting to get at any given moment. This is captured by a function called a *value function*, v^π.

The **value function** v^π takes as input any state s, and produces as output the expected discounted sum of rewards that the agent would receive if it were in state s. That is,

$$v^\pi(s) = \mathbf{E}\left[\sum_{k=0}^{\infty} \gamma^t R_{t+k} \middle| S_t = s; \pi\right]. \tag{17.4}$$

Notice that the right hand side of (17.4) depends on t, which is not specified on the left side. This is because, given a common assumption called the **Markov assumption**, which we are not discussing in this course, one can prove that the value of the right hand side is the same for any value of t. Intuitively, given the state of the environment (S_t) and how the agent will select actions (π), the expected discounted sum of rewards that the agent will obtain in the future is completely specified—it does not further depend on *when* the environment is in state S_t.[2]

2: For problems wherein the current time *does* matter, one can embed the current time within the agent's observations; that is, t can be included within S_t.

17.3 Temporal Difference Error

Later, we will discuss how the agent can learn (estimate) v^π based on its experiences. First, let's explore how the agent could use v^π (or an estimate of v^π). So, for now we will assume that the agent has access to v^π for its current policy π.

If the agent knows v^π, how could it update after one time step—after observing the transition from S_t to S_{t+1} due to action A_t and with the resulting reward R_t? The core idea is that the agent should make the action A_t more likely if it causes the agent to expect that it will obtain more reward starting from S_t than it had predicted.

When in state S_t, the agent expects to receive a discounted sum of rewards equal to $v^\pi(S_t)$.[3] It then observes an immediate reward of R_t, followed by state S_{t+1}. From state S_{t+1}, it expects to obtain a discounted sum of rewards equal to $v^\pi(S_{t+1})$.

3: Notice that $v^\pi(s)$ is the value of any state s when using policy π. Since S_t is the state at time t, $v^\pi(S_t)$ denotes the value of the particular state that happens to occur at time t.

So, *before* observing R_t and S_{t+1}, the agent expects to receive a total discounted sum of rewards, starting from time t, equal to $v^\pi(S_t)$. *After* observing R_t and S_{t+1}, the agent expects to receive a total discounted sum of rewards, starting from time t, of $R_t + \gamma v^\pi(S_{t+1})$, since it obtained a reward of R_t at time t and $v^\pi(S_t)$ captures how much reward it expects to get thereafter. Taking the difference between the agent's expectations before and after observing R_t and S_{t+1} gives a measure of much the events

that transpired during the current time step differ from its expectations. We call this difference the **temporal difference error** or **TD error**, δ_t:

$$\delta_t = \underbrace{R_t + \gamma v^\pi(S_{t+1})}_{\text{(a)}} - \underbrace{v^\pi(S_t)}_{\text{(b)}}, \tag{17.5}$$

where term **(a)** is the agent's expectations after the current events (the transition to S_{t+1} and receipt of reward R_t) and **(b)** is the agent's expectations before the current events.

So, if δ_t is positive, it means that the outcome (of one time step) was better than the agent expected. Similarly if δ_t is negative, it means that the outcome (of one time step) was worse than the agent expected. In both cases, the "outcome" incorporates the agent's expectations about how much reward it will receive in the future.

So, a positive TD error δ_t indicates that the events of the current time step (the transition to S_{t+1} and receipt of reward R_t) resulted in the agent now expecting to receive more reward than it did previously. Hence, A_t turned out *better* than the agent expected, and so perhaps A_t should be made more likely. Similarly, if the TD error δ_t is negative, it indicates that the events of the current time step resulted in the agent now expecting to receive less reward than it did previously. Hence, A_t turned out *worse* than the agent expected, and so perhaps A_t should be made less likely.

This intuition can be included in our update by replacing the discounted sum of rewards, $\sum_{k=0}^{\infty} \gamma^k R_{t+k}$, which indicated how good the outcome was at the end of the episode, with δ_t, which indicates how "good" the outcome of one time step was relative to the agent's expectations. That is, we replace

$$\forall i,\ \theta_i \leftarrow \theta_i + \alpha \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k} \right) \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}, \tag{17.6}$$

with

$$\forall i,\ \theta_i \leftarrow \theta_i + \alpha \delta_t \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}, \tag{17.7}$$

where

$$\delta_t = R_t + \gamma v^\pi(S_{t+1}) - v^\pi(S_t). \tag{17.8}$$

Notice that the earlier expression, (17.6), made the change to make action A_t more or less likely in state S_t dependent on $R_t, R_{t+1}, R_{t+2}, \ldots$, and therefore could only be performed at the end of the episode when these rewards had all been observed by the agent. However, the new expression, (17.7), only depends on S_t, A_t, R_t, and S_{t+1}, and so it can be performed as soon as S_{t+1} is observed. Hence learning can be interleaved with action selection, instead of requiring the agent to act for an entire episode before switching to a learning phase. Algorithm 17.3 incorporates this change.

Algorithm 17.3: A simple RL algorithm inspired by MENACE, Version 5.0

for *each episode* **do**
  // Run one episode (play one game).
  for *each time t in the episode* **do**
    // Execute one time step of agent-environment interaction
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
    // Learn from the outcome of this one time step
    $\delta_t \leftarrow R_t + \gamma v^\pi(S_{t+1}) - v^\pi(S_t)$;
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \delta_t \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$;
  end
end

18 Policy Gradient Part 3

18.1 An Actor-Critic Algorithm

Notice that Algorithm 17.3 uses the value function, v^π, which is not known to the agent. To fix this limitation, the agent must also estimate the value function, v^π. Notice that this problem is a supervised learning problem. For every time t, we can view S_t as the input and the observed discounted sum of rewards, $\sum_{k=0}^{\infty} \gamma^k R_{t+k}$, as the label (output) for a regression problem.

When describing regression algorithms previously, we wrote $f_w(x)$ to denote a parametric model with weights w applied to input x. When estimating the value function, we write v_w for the parametric model, rather than f_w, and the input is a state, s. Hence, $v_w(s)$ is the agent's estimate of $v^\pi(s)$—the value of state s using weights w.

From each episode, we can construct many input-output pairs. The first input is S_0, and the first output is $\sum_{k=0}^{\infty} \gamma^k R_{0+k}$. The second input is S_1, and the second output is $\sum_{k=0}^{\infty} \gamma^k R_{1+k}$. In general, the t^{th} input is S_t and the t^{th} output is $\sum_{k=0}^{\infty} \gamma^k R_{t+k}$.

We simply apply a variant of least squares regression to this data set—specifically (10.13) but with the scaling factor of two removed (this scaling factor can be viewed as a change to the step size, α). The resulting update, assuming that $w = (w_1, w_2, \dots)$ is a vector of real-valued weights, $w_j \in \mathbb{R}$:

$$\forall t, \forall j, \quad w_j \leftarrow w_j + \alpha \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k} - v_w(S_t) \right) \frac{\partial v_w(S_t)}{\partial w_j}. \tag{18.1}$$

Incorporating this into our MENACE-like algorithm, we obtain Algorithm 18.1. Notice that we have changed the step size from the symbol α to the symbol β when updating the value function estimate. This is because the step sizes used for updating the policy and value function need not be the same (in most cases, these algorithms work better when the step sizes are not the same). So, α is the step size for updating the policy and β is the step size for updating the value function estimate. This type of algorithm is called an **actor-critic** because it has two components: the policy π_θ (actor) and the value function estimate, v_w (critic).

Notice that the critic update in Algorithm 18.1 happens after the end of the episode because it relies on all of the rewards. To change this, shifting the critic update to happen after each time step, we must change the labels associated with each state, S_t. Instead of using $\sum_{k=0}^{\infty} \gamma^k R_{t+k}$, which depends on future rewards, we desire a label that only depends on the terms available at the end of time step t. That is, we desire a label that is defined in terms of S_t, R_t, and S_{t+1}.

Algorithm 18.1: A simple RL algorithm inspired by MENACE, Version 6.0

for *each episode* **do**
  `// Run one episode (play one game).`
  for *each time t in the episode* **do**
    `// Execute one time step of agent-environment interaction`
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
    `// Learn from the outcome of this one time step`
    $\delta_t \leftarrow R_t + \gamma v_w(S_{t+1}) - v_w(S_t)$;
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \delta_t \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$;
  end
  `// Update the critic after episode.`
  for *each time t in the episode* **do**
    $\forall j, w_j \leftarrow w_j + \beta(\sum_{k=0}^{\infty} \gamma^k R_{t+k} - v_w(S_t)) \frac{\partial v_w(S_t)}{\partial w_j}$;
  end
end

Notice that if the value function estimate is accurate, then $\sum_{k=0}^{\infty} \gamma^k R_{t+k}$ is approximately equal to $R_t + \gamma v_w(S_{t+1})$. So, we can use $R_t + \gamma v_w(S_{t+1})$ as the label for input S_t. This results in the update:

$$\forall t, \forall j, \quad w_j \leftarrow w_j + \beta(R_t + \gamma v_w(S_{t+1}) - v_w(S_t)) \frac{\partial v_w(S_t)}{\partial w_j}. \tag{18.2}$$

Notice that the term inside the parentheses is simply the TD error, δ_t! So, we can write this update as:

$$\forall t, \forall j, \quad w_j \leftarrow w_j + \beta \delta_t \frac{\partial v_w(S_t)}{\partial w_j}. \tag{18.3}$$

Incorporating this change, and moving the update to happen during the episode (rather than after the episode), we obtain Algorithm 18.2, which is a common and effective RL algorithm.

Algorithm 18.2: An Actor-Critic Algorithm

for *each episode* **do**
  // Run one episode (play one game).
  for *each time t in the episode* **do**
    // Execute one time step of agent-environment interaction
    Agent observes state S_t;
    Agent selects action A_t according to the current policy, π_θ;
    Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
    // Learn from the outcome of this one time step
    $\delta_t \leftarrow R_t + \gamma v_w(S_{t+1}) - v_w(S_t)$;
    $\forall i,\ \theta_i \leftarrow \theta_i + \alpha \delta_t \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$ // Actor update
    $\forall j,\ w_j \leftarrow w_j + \beta \delta_t \frac{\partial v_w(S_t)}{\partial w_j}$ // Critic update
  end
end

Bibliography

References in citation order:

[1] Nils Johan Nilsson. *Artificial Intelligence: A New Synthesis*. San Francisco, CA: Morgan Kaufmann, 1998 (cited on page 1).

[2] Wikipedia contributors. *Intelligent Agent*. [Wikipedia: accessed January 24, 2021]. 2021. URL: https://en.wikipedia.org/wiki/Intelligent_agent (cited on page 1).

[3] Tom Mitchell. *Machine Learning*. Burr Ridge, IL: McGraw Hill, 1997 (cited on page 3).

[4] Merriam-Webster. *Learning*. [Merriam-Webster.com: accessed January 24, 2021]. 2021. URL: https://en.wikipedia.org/wiki/Data (cited on page 3).

[5] Wikipedia contributors. *Artificial General Intelligence*. [Wikipedia: accessed January 24, 2021]. 2021. URL: https://en.wikipedia.org/wiki/Artificial_general_intelligence (cited on page 5).

[6] Lauren Weber. *Your Résumé vs. Oblivion*. [The Wall Street Journal: accessed February 4, 2021]. Jan. 2012. URL: https://www.wsj.com/articles/SB10001424052970204624204577178941034941330 (cited on page 8).

[7] Julia Carneiro. *Brazil's universities take affirmative action*. [BBC News: accessed February 4, 2021]. Aug. 2013. URL: https://www.bbc.com/news/business-23862676 (cited on page 8).

[8] Carlo Rovelli. *The Order of Time*. New York: Riverhead Books, 2018 (cited on page 9).

[9] Wikipedia contributors. *k-d tree*. [Wikipedia: accessed February 7, 2021]. 2021. URL: https://en.wikipedia.org/wiki/K-d_tree (cited on page 10).

[10] Randall W Jibson. 'Regression models for estimating coseismic landslide displacement'. In: *Engineering Geology* 91.2–4 (2007), pp. 209–218 (cited on pages 20, 62).

[11] Christopher M Bishop. *Pattern Recognition and Machine Learning*. New York: Springer, 2006 (cited on page 21).

[12] Ahmed M Abdel-Zaher and Ayman M Eldeib. 'Breast cancer classification using deep belief networks'. In: *Expert Systems with Applications* 46 (2016), pp. 139–144 (cited on page 23).

[13] Nikolaus Hansen. 'The CMA evolution strategy: a comparing review'. In: *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*. Ed. by Jose. A. Lozano et al. Berlin: Springer, 2006, pp. 75–102 (cited on page 27).

[14] Dimitri P Bertsekas and John N Tsitsiklis. 'Gradient convergence in gradient methods with errors'. In: *SIAM Journal on Optimization* 10.3 (2000), pp. 627–642 (cited on pages 36, 54).

[15] George Konidaris, Sarah Osentoski, and Philip Thomas. 'Value function approximation in reinforcement learning using the Fourier basis'. In: *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*. 2011, pp. 380–385 (cited on pages 41, 75).

[16] Wikimedia Commons. *File:Components of neuron.jpg*. 2021. URL: https://en.wikipedia.org/wiki/File:Components_of_neuron.jpg (cited on page 44).

[17] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. Cambridge, MA: MIT Press, 2007 (cited on page 43).

[18] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Upper Saddle River, NJ: Pearson, 2003 (cited on pages 46, 50, 54).

[19] Xavier Glorot and Yoshua Bengio. 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256 (cited on page 58).

[20] Kaiming He et al. 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification'. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1026–1034 (cited on page 58).

[21] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from Data*. Vol. 4. New York: AMLBook, 2012 (cited on page 62).

[22] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998 (cited on page 66).

[23] Andrew Y Ng, Daishi Harada, and Stuart Russell. 'Policy invariance under reward transformations: Theory and application to reward shaping'. In: *Icml*. Vol. 99. 1999, pp. 278–287 (cited on page 74).

[24] Ronald J Williams. 'Simple statistical gradient-following algorithms for connectionist reinforcement learning'. In: *Machine learning* 8.3 (1992), pp. 229–256 (cited on page 82).

[25] Jim Hefferon. *Linear Algebra*. 2020. URL: https://hefferon.net/linearalgebra/ (cited on page 93).

Notation

List of common symbols in alphabetical order.

$\hat{y}_i$	Prediction of the label associated with the i^{th} data point (supervised learning).
$\mathbb{N}_{>0}$	The set of natural numbers excluding zero. That is, $\{1, 2, \dots\}$.
$\mathbb{N}_{\geq 0}$	The set of natural numbers including zero. That is, $\{0, 1, 2, \dots\}$.
$\mathbb{R}$	The set of real numbers, which does not include $-\infty$ or ∞.
$\mathbb{R}^+$	The set of extended real numbers, which is the same as $\mathbb{R}$ but includes $-\infty$ and ∞.
$\mathbb{R}_{>0}$	The set of real numbers that are more than zero.
$\mathbb{R}_{\geq 0}$	The set of real numbers that are at least zero.
$\mathbf{E}$	Expected value.
$\Pr(A)$	The probability of event A.
f_w	Parametric model with model parameters w (supervised learning).
$f_w(x_i)$	Prediction of the label associated with x_i, generated using the parametric model f_w (supervised learning).
l	Loss function (supervised learning).
$l(w)$	Value of the loss function for model parameters w (supervised learning).
m	The number of features associated with each data point (supervised learning).
n	The number of data points (supervised learning).
w	Model parameters (supervised learning).
w_j	The j^{th} weight of w (supervised learning, early chapters).
w_k	The k^{th} vector of model parameters in a sequence of model parameters (supervised learning).
$w_{k,j}$	The j^{th} weight of the k^{th} vector of model parameters in a sequence of model parameters (supervised learning).
x_i	Features associated with the i^{th} data point (supervised learning).
y_i	Label associated with the i^{th} data point (supervised learning).

Greek Letters with Pronunciation

Character	Name	Character	Name
α	alpha *AL-fuh*	ν	nu *NEW*
β	beta *BAY-tuh*	ξ, Ξ	xi *KSIGH*
γ, Γ	gamma *GAM-muh*	o	omicron *OM-uh-CRON*
δ, Δ	delta *DEL-tuh*	π, Π	pi *PIE*
ϵ	epsilon *EP-suh-lon*	ρ	rho *ROW*
ζ	zeta *ZAY-tuh*	σ, Σ	sigma *SIG-muh*
η	eta *AY-tuh*	τ	tau *TOW (as in cow)*
θ, Θ	theta *THAY-tuh*	υ, Υ	upsilon *OOP-suh-LON*
ι	iota *eye-OH-tuh*	ϕ, Φ	phi *FEE, or FI (as in hi)*
κ	kappa *KAP-uh*	χ	chi *KI (as in hi)*
λ, Λ	lambda *LAM-duh*	ψ, Ψ	psi *SIGH, or PSIGH*
μ	mu *MEW*	ω, Ω	omega *oh-MAY-guh*

Capitals shown are the ones that differ from Roman capitals.
This table is from Jim Hefferon's linear algebra text [25].

Index