`Overfitting` the Machine Learning Interview

Ideas Every Machine Learning Engineer Should Know

Tingke Shen

January 13, 2020

Disclaimer

Copyright

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook
(You are welcome to contribute!)

I’m tired of doing leetcode questions.

– Every software engineer ever

Preface

Machine learning and software development are two of the hottest jobs on the market right now. There are many reasons to apply to a machine learning engineer job over a software development one. Not doing (as many) leetcode questions is one of them. Why are candidates frustrated with spending hours doing practice algorithms for their coding interviews? One of the most common complaints is that what's tested on the interview is never the same as the work they do on the job. In machine learning this is called a *distribution mismatch*. After going through interviews at some of the largest machine learning companies in self-driving, retail, consulting and medical imaging, I've found that many of the same questions come up during interviews. There is a fundamental difference between software interviews at one of the FAANGs and machine learning interviews. The *distribution mismatch* between interview and job in machine learning is much lower.

For example, a lot of interviewers ask you to *interpret train/validation loss plots* — you will be doing exactly this on the job. One interviewer asked me if I can trust PCA'd feature inputs to a neural network to capture the important correlations from input to output. It just so happens the Youtube8M competition uses PCA'd video frame features.

Put simply, if you can answer ML interview questions, you will do your ML job better. This captures my intention when I set out to write this book. I wanted to provide a de-facto reference for practicing and aspiring machine learning engineers for the most fundamental ideas to their trade. I was motivated by the lack of good study materials online. Most resources cover too many concepts making it hard to know the most important ones, cover concepts in too much depth what is essential, or present material in Q&A format — the real type of *overfitting* you want to avoid. There is a timeless stratedgy to studying: read a textbook. But for those who are already under the time crunch of the interview cycle, I hope my book can help you prepare for your dream job.

T.S.

Detailed Contents

Figures

Tables

Introduction 1

1.1 Hi, My Name is The Overfitting ML Book

This book is meant to be your de-facto reference to the most fundamental ideas of machine learning. You're reading this if you're an experienced practitioner who has forgotten what *gradient boosted trees* are or an aspiring machine learning engineer who doesn't quite understand backpropogation. For those who are new to the field, the beauty of this book is that by studying these fundamental topics, you will not only improve your interviewing skills, you also be more prepared for day one on the job.

Each section will present a short and intuitive explanation of a topic followed by a list of thought-provoking interview questions. The expectation is you've seen the concepts before and require a pithy reminder. The purpose is not to teach you a topic from scratch. There are superb resources online if you are starting from square-one (textbooks, blog posts, and videos). I will link these at the end of each chapter.

1.2 Getting your Foot in the Door

A quick note on acquiring interviews. Online job postings are the obvious place to apply. However, machine learning (ML) conferences are hands down the best places to catch recruiters' attention. The best companies (those who can actually afford to deploy recruitment teams to conference) will be there. And the reason they put on cool demos and extravagant parties — is to attract machine learning talent. The conferences with the most industry presence are NeuRIPS, CVPR, ICCV, and ICML. Most conferences have a job board. For example, here is the one for CVPR `http://cvpr2019.thecvf.com/jobs`. One successful strategy is to email companies you're interested in a few days before the conference starts to arrange a meetup during the conference. This often leads to 1-on-1 interviews with team leads and a conference encounter is your warm opening to getting hired at your dream company.

1.3 Knowing What to Study

ML is still a relatively new field. As a result interviewers have not yet converged on the standard scope of interview questions. Therefore, as the candidate it can be hard to know exactly what to study because

there are a lot of relevant topics. Machine learning interview questions vary across these four high-level categories.

- technical question
- system design
- knowledge of literature
- explain a past project

This book will help you study the first point: technical questions. For system design and literature, unfortunately there is no substitute for real experience. There are many great resources online for successfully explaining your past projects in interviews.

Even within technical questions, there is a broad set of topics. Interviewers expect candidates to wear many hats.

- Math
- ML Theory
- ML Implementation
- Coding ability
- Algorithms

Every company tests for different things in a ML interview. In general, it's a good idea to look up the names of your interviewers beforehand (if they're given to you) to get a sense of interview questions you will be asked. Interviewers usually ask about topics they are experts in themselves. A ML researcher is likely to ask about deep learning and a ML software engineer is likely to ask a tricky algorithm question. It's also important to know each company's approach to the problem they are solving. One self driving company I onsited with asked exclusively algorithm and puzzle questions because they believed machine learning is just another tool that one can learn to use on the job.

One useful distinction to make is between companies that test algorithms (think leetcode questions) and companies that test coding ability (well structured code, sound approach). Companies that view machine learning as their product care more about directly evaluating your machine learning ability and care less about your ability to implement breadth first search. On the other hand, companies that view ML simply as another tool emphasis more on problem solving ability (I'm talking about the FAANGS). To be safe you could study Cracking the Coding Interview and spend some time on leetcode.

Despite the variance between ML interviews, I've highlighted some broad differences between different positions you could apply to.

Machine learning engineer

During a typical onsite with 4 technical interviews, 2 will be on "shallow" machine learning, 1 will be on deep learning, 1 will be coding and sometimes 1 will be on math (usually probability).

Machine learning scientist

The interview is likely more in-depth than ML engineer. In addition to the above 4 interviews, you will be tested on your knowledge of cutting-edge research. There may also be a challenging math problem.

Machine learning software developer

Software development interviews focus on algorithms. Usually algorithms relate to problems the company works on. For example, a self-driving company will ask about algorithms for processing images. Nevertheless, these problems tend to require the same skills you practice on leetcode - trees, graphs, dynamic programming, etc.

Data scientist

The main difference is data science positions require knowledge of big data technologies like Spark and Hadoop and comfort working with database query languages. There is also a focus on "shallow" machine learning and software development ability.

Math 2

I'll be honest with you, there is no shortcut to learning the math for machine learning. Every good machine learning engineer should have a solid grasp of linear algebra and probability. There is not enough time in this book to teach you either (see the end of the chapter for references). This chapter provides a basic overview of the math required to understand machine learning. Unfortunately a difficult onsite interview can ask a tricky probability question. For that, I suggest taking the time to study the resources at the end of this chapter.

2.1 Linear Algebra

Linear algebra underlies most of the math used in machine learning. You need it to understand backpropagation, logistic regression, PCA and many more algorithms. In fact, it's arguably impossible to understand anything in machine learning without linear algebra. This section reviews the core concepts required for topics covered in the other chapters of the book.

Vector spaces

In machine learning we usually talk about two different types of vector spaces, the **parameter vector space** and the **feature vector space**. Let's look at one of the most iconic equations of machine learning, the linear regression equation.

$$Xw = y \tag{2.1}$$

Here X is a $n \times p$ matrix where n is the number of samples and p is the number of features each sample has. $\mathbf{w}$ is a vector of learnable parameters or weights. It is of size $p \times 1$. $\mathbf{y}$ is a vector of labels, size $p \times 1$.

I'll be using capital letters for matrices throughout this book.

We say that w lies in a p-dimensional *parameter* vector space. This is also called the "solution" space. This is because we're given X and y and the goal is to find the solution w which solves equation 2.1.

We say that each sample x_i lies in a p dimensional *feature* (or input) vector space. Speaking about the input vector space can be useful, for example when Geoff Hinton says "natural images lie on a low dimensional manifold in feature space". Basically this means the pixels of an image are the features. And naturally occurring images only make up a small but well structured set of all possible pixel values.

Matrix inverse

If $p = n$ then X is a square matrix and we can solve 2.1 using the inverse matrix X^{-1}.

$$w = X^{-1}y \tag{2.2}$$

X^{-1} is also a $n \times n$ matrix called the inverse and defined as,

$$X^{-1}X = I \tag{2.3}$$

Unfortunately n is almost never equal to p and therefore the inverse does not exist and we cannot use equation 2.2. In fact, deep learning works because the number of samples n is usually much larger than the number of features p. As we will see in chapter **??** we can use gradient descent to solve 2.1 in the general case.

Dot product and Cosine Similarity

The dot product comes up a lot in deep learning. For example, to calculate the attention weights in transformer networks, you dot product the key vector with the value vector.

Within the context of machine learning, dot product is an unnormalized measure of the similarity between two vectors. Here is the definition.

$$a \cdot b = |a||b| \cos\theta \tag{2.4}$$

a and b are two arbitrary vectors. $|a|$ and $|b|$ are their magnitudes. θ is the angle between the two vectors. Often we care about the normalized dot product. This is the *cosine similarity* which we get by rearranging equation 2.4.

$$\cos\theta = \frac{a \cdot b}{|a||b|} \tag{2.5}$$

For example, cosine similarity is used as the training loss in learning image-language joint embeddings and for measuring the similarity between two words represented as *word2vec* vectors.

Norms

Norms are a way to measure the magnitude of a particular vector of the distance between two vectors. Typically we care about the 2-norm or *Euclidean distance*.

$$d_2(x) = \sqrt{\sum_{j=1}^{d}(x^j)^2} \tag{2.6}$$

x is an arbitrary vector in d-dimensional space. We can get its 2-norm by taking the square root of the sum of the squares of each dimension.

For example, the length of the hypotenuse of a triangle can be found by taking the square root of the sum of the squares of the length of the two sides.

The 1-norm is another commonly used norm in ML. The 1-norm can be computed by summing the absolute value of each dimension.

$$d_1(x) = \sum_{j=1}^{d} |x^j| \tag{2.7}$$

For the 2-norm we used powers of 2 and took the square root. For the 1-norm we used powers of 1 and took the first root (identity). In general, we can define a p-norm by taking the p^{th} power of each dimension, summing it, and taking the p^{th} root.

$$d_p(x) = \left[\sum_{j=1}^{d} (x^j)^p\right]^{1/p} \tag{2.8}$$

Gradients and the Rolling Ball Analogy

`\margintoc`.

Insert ball rolling down hill figure

Gradients describe the direction of largest change of a scalar quantity. There is a nice analogy to visualize gradients by describing it in terms of a ball rolling down a surface or hill. Suppose we have a set of parameters w and we would like to understand the gradients of w. Suppose w is 2-dimensional. w can be thought of as the *position* of an imaginary ball. Let $h(w)$ define a scalar function (*i.e.* a function which maps each position to a real value). We can interpret $h(w)$ as the height of an imaginary surface the imaginary ball is rolling on. The intuitive way to understand the gradient of h at position w is it is the vector describing the direction of greatest increase in elevation. A real ball would roll in the opposite direction of the gradient, the direction of greatest decrease in elevation, until it reaches the "bottom". Thinking about the gradients in terms of a rolling ball is very useful for understanding the machine learning algorithm gradient descent. As we will see $h(w)$ is the loss function for neural networks. Neural networks are trained by taking the *ball* to the lowest point on the surface $h(w)$. Notation wise, the gradient of a function is denoted by the ∇ symbol.

This is if we neglect the momentum of the ball.

$$g(w) = \nabla h(w) \tag{2.9}$$

Gradients have the same dimensionality as the parameters.

Here's a concrete example. Suppose you're buying a house and the price of the house depends on its square footage (area) A, location (represented as distance to the city center) L, and the number of rooms R.

$$P = w_1 A + w_2 L + w_3 R \tag{2.10}$$

w_1, w_2 and w_3 are weights (learned in a linear regression model) that scale how much each feature contributes to the price. We can find the gradient of the price with respect to the features.

$$g_{\text{features}}(A, L, R) = \nabla_{\text{features}} P(A, L, R) = \begin{bmatrix} \frac{dP}{dA} \\ \frac{dP}{dL} \\ \frac{dP}{dR} \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \tag{2.11}$$

The gradient tells us that if we want to increase the price of the house, we should change A, L, and R by the ratio $[w_1, w_2, w_3]$.

As we discussed in section 2.1, we can talk about two vector spaces: the **feature vector space** and the **parameter vector space**. In equation 2.11 we took the *gradient with respect to the feature variables* to get a gradient in the feature vector space. We can also take the *gradient with respect to the parameters* to get a gradient in the parameter vector space.

$$g_{\text{parameters}}(w_1, w_2, w_3) = \nabla_{\text{parameters}} P(w_1, w_2, w_3) = \begin{bmatrix} \frac{dP}{dw_1} \\ \frac{dP}{dw_2} \\ \frac{dP}{dw_3} \end{bmatrix} = \begin{bmatrix} A \\ L \\ R \end{bmatrix} \tag{2.12}$$

We would use equation 2.12 in linear regression to update the parameters using gradient descent. Notice that we've written the price function $P(w_1, w_2, w_3)$ to depend on the parameters instead of the input variables. This is notation trickery. To be explicit, P depends on all variables and parameters as so $P(w_1, w_2, w_3, A, L, R)$. However, it is typical to omit the variables or parameters that are held fixed.

Returning to the rolling ball analogy, the gradient actually tells us 2 things:

- g is the direction of greatest increase in elevation
- $\|g\|$ is the rate of increase in elevation if we moved in the direction g

If the magnitude of the gradient $\|g\|$ is small, then elevation does not change significantly in any direction. Hence, we are in a flat part of the vector space. A gradient of 0 means the ball has reached the top (or the bottom) of a hill.

The top of a hill is called a maxima and the bottom of a hill a minima. We can talk about the absolute highest point in the landscape *e.g.* Mount Everest the tallest mountain on Earth. This is called the *global maxima*. However, landscapes usually have other peaks that are lower than the global maxima. These are called *local maximas e.g.* Mount Fuji.

If we move in the direction perpendicular to the gradient, the elevation won't change. If you keep walking along the direction perpendicular to the gradient, you will traverse a "level curve". Level curves are paths in vector space where the elevation stays constant. Imagine flooding a mountain with water up to an elevation of 500 meters. The circumference where water meets dry mountain is the 500m level curve. As we will see, level curves are important in analyzing the behavior of loss functions in machine learning.

The Chain Rule

Chain rule is the backbone of the backpropagation algorithm. It specifies how we can take the derivative of the composition of two functions. Applied recursively, we can find the derivative of an arbitrary number of compositions of functions. In the one dimensional case, suppose f and g are two functions, x is the input and y is the output. We can find the change in the output with respect to change in the input using the chain rule (equation 2.14).

$$y = f(g(x)) \tag{2.13}$$

$$\frac{dy}{dx} = \frac{df}{dg}\frac{dg}{dx} \tag{2.14}$$

An easy want to remember the chain rule is it's a product of derivatives where the intermediate variables (in this case g) "cancel out" on the top and bottom.

$$\frac{dy}{dx} = \frac{df}{\cancel{dg}}\frac{\cancel{dg}}{dx} \tag{2.15}$$

In neural networks each layer has multiple inputs and multiple outputs. Therefore, we need to use the multi dimension version of chain rule. Conceptually this is exactly the same as the one dimensional case. Because we have multiple inputs and outputs, we need to consider the derivative of every output with respect to every input. What type of mathematical object allows us to store the derivatives of all input-output pairs? A matrix. In neural networks, each derivative is a matrix called the Jacobian. Chain rule is applied by matrix-multiplying Jacobian matrices. Here is an example. Let y and x be m and n dimensional vectors. G is a function that maps n to k dimensions and F is a function that maps k to m dimensions.

$$y = F(G(x)) \tag{2.16}$$

$$\frac{dy}{dx} = m\left\{\overbrace{\begin{bmatrix} \frac{dF}{dG} \end{bmatrix}}^{k}\right. k\left\{\overbrace{\begin{bmatrix} \frac{dG}{d\mathbf{x}} \end{bmatrix}}^{n}\right. \tag{2.17}$$

$\frac{dy}{dx}$ is a m by n matrix.

2.2 Probability

Basic Definitions

Probability measures the chance something will happen. For example if you're flipping a coin, you can talk about the probability of getting heads. Now suppose you're flipping an unfair coin meaning it doesn't land heads and tails with equal probability. To take an *empiricist's view* of probability, if the probability of heads is $P(\text{heads}) = 0.25$, then if I flip a coin many (infinite) times, the coin will come up heads 25% of the time. A probability *describes how frequent an outcome will occur an identical experiment is repeated infinitely many times*.

The sum of the probabilities of all possible outcomes of an event needs to add up to 1 (something must happen!). For example, with flipping coins

$$P(\text{heads}) + P(\text{tails}) = 1 \tag{2.18}$$

A **joint probability** is the probability of two events both happening. Suppose there are two events. Let A be the weather today and B whether I eat icecream (B means I do eat icecream and "not B" denoted $\overline{B}$ means I do not eat icecream).

$$P(A, B) = P(\text{it will be sunny today}, \text{I will eat icecream}) \tag{2.19}$$

$P(A, B)$ is the probability it will be sunny today and I will eat icecream. Sometimes I only care about one event, let's say B. In this case, I could **marginalize a joint probability of A and B with respect to A**. This means I'm going to consider all the values that A can take on and sum over them. Doing so gives me the **marginal probability** $P(B)$.

$$P(B) = \sum_{A} P(A, B) \tag{2.20}$$

We can interpret the marginal probability $P(B)$ as the "net" probability I'll eat icecream after accounting for all possible weather conditions. Specifically, if I want to find out $P(\text{I will eat icecream})$ I should add up my probabilities of eating icecream in the sun and in the rain (assuming sunny and rain are the only two possible weather conditions). Here is a colloquial way of writing equation 2.20.

$$\begin{aligned} P(\text{I will eat icecream}) = {} & P(\text{I will eat icecream}, \text{it will be sunny today}) \\ & + P(\text{I will eat icecream}, \text{it will rain today}) \end{aligned} \tag{2.21}$$

A conditional probability states the probability of something happening given something else has happened already. We can relate the conditional probability to the joint probability.

$$P(B|A) = \frac{P(A,B)}{P(A)} \tag{2.22}$$

One way to interpret equation 2.22 is "$P(A|B)$ is the probability both A and B occur (which is $P(A,B)$) discounted by $P(A)$ because A has already occurred.

Note that we can perform marginalization on the conditional probability as well.

$$P(B) = \sum_A P(B|A)P(A) \tag{2.23}$$

"The probability of eating icecream is the probability of eating given it's sunny plus eating given it's rainy."

If two events are **independent**, then one event occurring does not change the probability of the other event occurring. Mathematically,

$$P(B|A) = P(B) \tag{2.24}$$

In other words, icecream is equally enjoyed in the sun and the rain — the weather does not affect the probability I will eat icecream.

Another statement of independence is $P(A,B) = P(A)P(B)$. In words, we can find the joint probability simply by multiplying the probability of each event.

Baye's Rule

Baye's Rule shows up in a lot of probability interview questions. It is useful when we know one conditional probability $P(B|A)$ but we do not know the opposite conditional probability $P(A|B)$. Baye's Rule is,

e.g. it's easy to know if I'll eat icecream when it rains. It is harder to know the probability of rain given I ate icecream.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.25}$$

This equation is true because multiplying by $P(B)$ makes both the left-hand-side and right-hand-side equal to $P(A,B)$.

Usually $P(B)$ is not given in the problem so we must compute it ourselves by expanding and marginalizing the conditional probability. A' is a *dummy variable* for the sum.

$$P(A|B) = \frac{P(B|A)P(A)}{\sum_{A'} P(B|A')P(A')} \tag{2.26}$$

Mean, Variance, Expectation

Our definition of probability was how frequently we will get an outcome if we run an experiment infinitely many times. We might also be interested in the value of the average outcome. This is the mean.

$$\mu = \sum_X P(X)X \tag{2.27}$$

The mean captures my average icecream consumption, but this doesn't say how erratic my consumption is. On some days I may eat 10 icecreams and on others 0. Variance captures the tendency for individual experiments to deviate from the mean.

$$\text{var} = \sum_X P(X)(X-\mu)^2 \tag{2.28}$$

The square root of variance is standard deviation

$$\sigma = \sqrt{\text{var}} \tag{2.29}$$

We can bring mean and variance under the same framework of expected values or *expectations*. The mean is the expectation of the function X, this is written as $E[X]$. The variance is the expectation of the function $(X-\mu)^2$, written $E[(X-\mu)^2]$. In general we can take the expectation of an arbitrary function of X.

$$E[f(X)] = \sum_X P(X)f(X) \tag{2.30}$$

Common Probability Distributions

Bernoulli distribution describes the outcome of a logistic regression model. Uniform distributions are used to initialize the weights of neural networks. Gaussian distributions show up everywhere, from the loss function of a neural network to the latent distribution of a *variational autoencoder*.

Certain probability distributions always show up in machine learning. Here are the most common three.

Gaussian Distribution

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{2.31}$$

Uniform Distribution

$$P(x) = \frac{1}{b-a} \tag{2.32}$$

Bernoulli Distribution

$$P(X=1) = \theta \tag{2.33}$$

The multivariate Gaussian distribution is often used in higher dimensions.

$$P(x) \sim \exp(-\frac{1}{2}(\mathbf{x}-\mu)^{\intercal}\Sigma^{-1}(\mathbf{x}-\mu)) \tag{2.34}$$

μ is the mean vector and Σ is the covariance matrix.

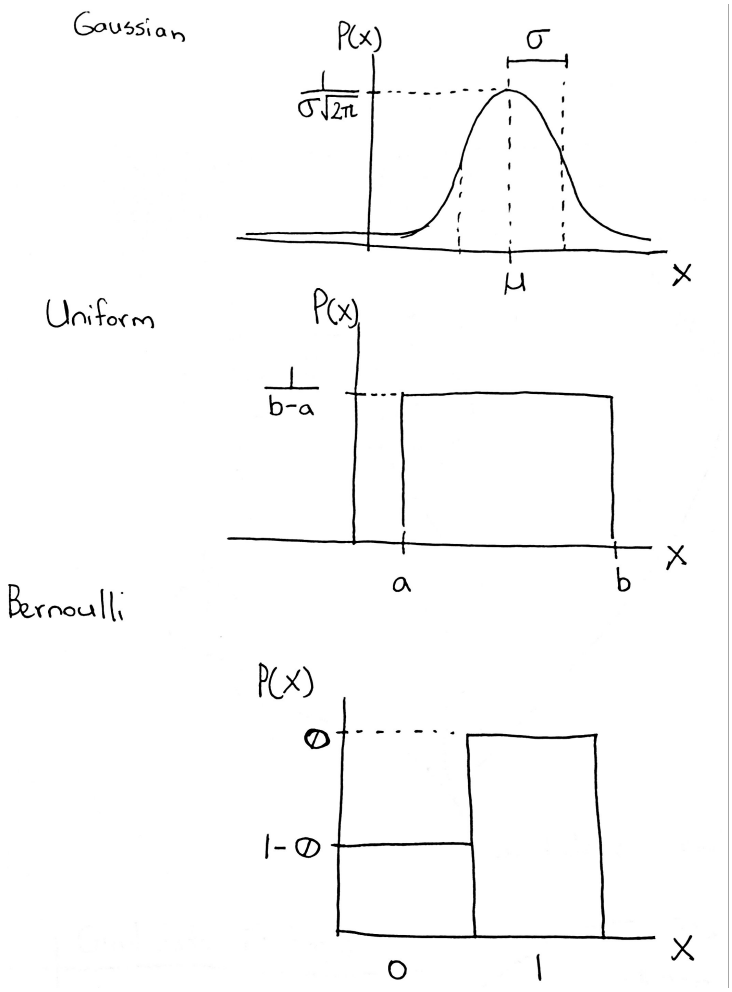

Figure 2.1: Some common probability distributions.

"Shallow" Machine Learning

3

It's a bit strange that we have a dichotomy between "shallow" machine learning and deep learning when deep learning makes up but a small fraction of machine learning. Shallow ML encompasses a suite of tools like logistic regression, decision trees, and much more while deep learning is arguably only a single algorithm. Deep learning receives considerably more attention today due to breakthroughs like AlphaGo and Deepfake that capture our imaginations. In fact, it's safe to wager that most readers of this book are starting a career in machine learning because of deep learning. However, it is almost certainly a bad idea to start studying for your machine learning interview by studying deep learning. Shallow machine learning is the foundation of deep learning. By studying it first, you'll have a much *deeper* understanding of deep learning.

3.1 Overview of Algorithms

We begin this chapter with an overview of the algorithms that will be discussed. They're grouped by the type of prediction task we're interested in:

1. **classification** — predicting one out of C choices/classes
2. **regression** — predicting a real valued number(s)

In machine learning we're usually concerned with two types of models: those that predict categories or classes (*e.g.* dogs, cats, bears) and those that predict real values (*e.g.* the price of a house). Some algorithms like decision trees can be adapted to do both.

Table **??** contains a list of algorithms covered in this chapter. Here's why we are reviewing each of these algorithms.

- Linear and logistic regression are the backbone of neural network models for binary classification and regression
- *Boosted* decision trees are still state-of-the-art on tabular data and used extensively in industry applications

Classification	Regression	Classification and Regression
Naive Bayes	Linear Regression	Decision Trees
k-Means		k-Nearest Neighbors
Logistic Regression		
SVM		

Acknowledgement: most of the sections in this chapter are derivatives of Mark Schmidt's machine learning course slides https://www.cs.ubc.ca/~schmidtm/Courses/340-F19/

- SVM's are also ubiquitous in applied work due to the availability of convenient libraries and nice theoretical properties
- k-nearest neighbors is a surprisingly hard baseline to beat for many machine learning tasks
- Naive Bayes and k-Means are common interview trivia questions

3.2 Decision Trees

Decision trees are a good "model algorithm" for us to study. They are conceptually simple, but are complex enough to train for us to dig into the nuances. Techniques like boosting and bagging (Chapter **??**) turn vanilla decision trees into the **state-of-the-art algorithm for tabular data**.

Decision tree is a machine learning algorithm built on a set of if-else statements. The intuition simple. For example, if I want to classify whether someone has a cold, I should ask myself whether sneezing=True. Of course only knowing whether I'm sneezing is not enough to decide if I'm sick; headaches and a fever may also be good indicators. Decision trees predict using a set of if-else statements organized into a tree. In fact, decision trees are exactly like the tree-algorithms people use in everyday life (figure 3.1), but with math.

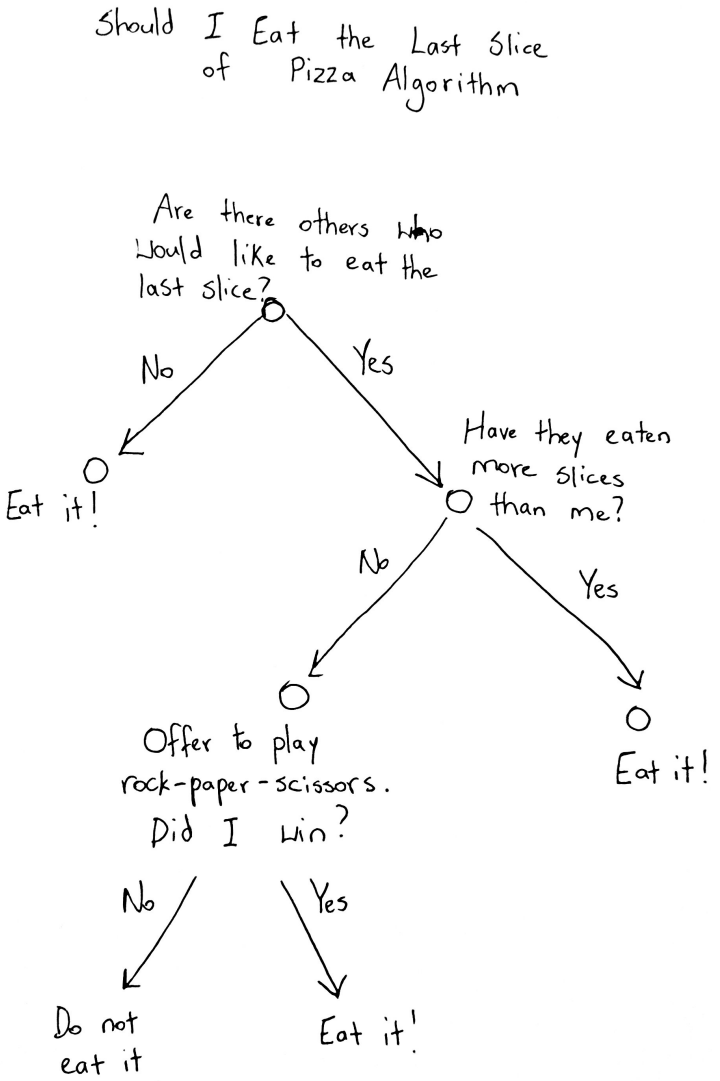

Figure 3.1: A practical algorithm for eating pizza.

Formally, consider an input matrix X of dimension $n \times p$. Let x be a

single sample and x^j denote the j^{th} feature of sample x. Here is a conceptual overview of the decision tree model:

1. An unbalanced binary tree.
2. There are **2 types of nodes, intermediate and leaf**.
3. **Intermediate nodes split data samples into two child nodes** based on whether samples satisfy a certain condition. Conditions are represented in the following form: $x^j > c$ where c is a constant threshold. j is called the *splitting feature*. At each intermediate node, the sample falls into the left or right child depending on whether it satisfies the node condition $x^j > c$.
4. **Child nodes contain a prediction** y. In the case of binary classification, $y = 1$ or $y = 0$.
5. At test time, a sample starts at the root node and falls like Plinko through the intermediate nodes, eventually landing in one of the leaf nodes. It receives the label $y_{\text{test}} = y_{\text{leaf}}$.

Let's look at a concrete example of the decision tree algorithm.

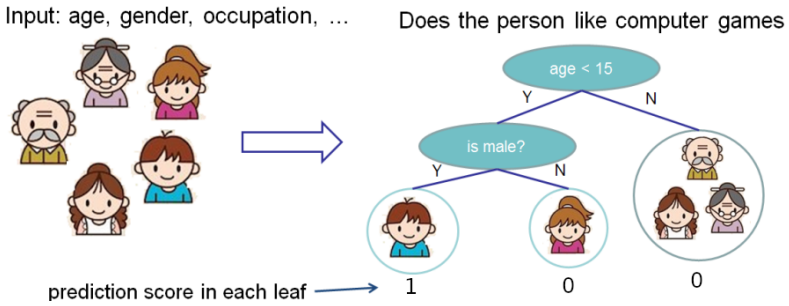

Figure 3.2: Example of a decision tree. Each person starts at the top of the tree and falls down it eventually ending up in a leaf node. Each person adopts the predicted value at the leaf node.

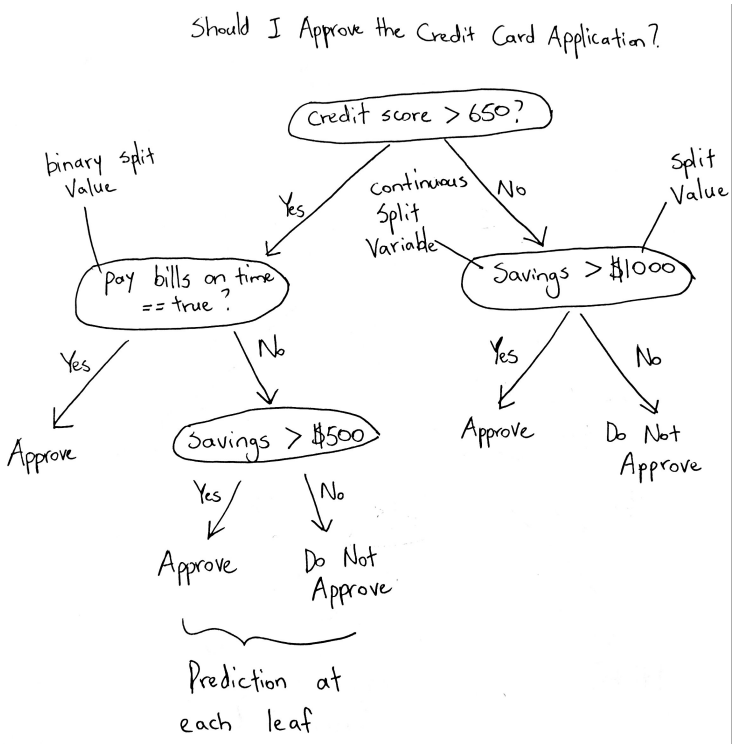

Figure 3.3: A more realistic application of decision trees to credit card approval.

The *splitting feature* at each intermediate node can be different. There are a few questions we haven't answered about decision trees.

1. How do we decide what the structure of the tree is?
2. How do we decide on the feature j to split on at each node?
3. How do we determine the threshold c for each node?
4. How do we determine the predictions at the leaves?

We will address these questions next.

Training

The reason the greedy splitting algorithm is used is because it's fast and works well empirically. What is the big-O complexity of finding the optimal tree? Even if we can compute the optimal tree, due to the bias-variance trade-off we'll discuss in the next sections, the **optimal tree may underperform the greedy tree** on the *test set*.

Training is the process of building the decision tree. In training we need to decide the structure, splitting features, thresholds, and leaf predictions. Greedy node addition is the de-facto algorithm used to build decision trees. It works as follows: starting at the root node, we will iteratively grow our tree by adding one node at a time. We choose the next node to add to our trees by considering all candidate leaf nodes, all splitting features, and all feature split thresholds. For each candidate node, feature, threshold, two child leaf nodes are created and we assign predictions to each based on maximizing the training accuracy. Then for each candidate we compute the training accuracy of the tree if we added the candidate split. Finally, the candidate which *increases the training accuracy of the tree the most* is actually added to the tree. This process is repeated until we decide the tree is large enough.

```
1
2  == Greedy algorithm for training a decision tree. ==
3
4  tree = root_node
5
6  \\ one "candidate" is a triplet of (node, feature, threshold)
       which specifies all the information for a split
7
8  while (True):
9      for all leaf nodes:
10         for all features:
11             for all possible thresholds for feature:
12                 candidate_score = compute_score(
13                                       split(node, feature, threshold
       ))
14
15     best_split = max(all_candidate_scores) \\ (node_best,
       feature_best, threshold_best)
16
17     tree = split(node_best, feature_best, threshold_best)
18
19     if (tree_is_large_enough):
20         break
```

A few questions still remain.

1. Q: How did we decide the set of possible thresholds to split a feature on? A: Take them to be all the values the feature takes on.
2. Q: How do we decide when to stop growing the tree? A: There are 2 common ways. We can limit the number of nodes or the depth of the tree. Limiting depth is more popular. The max depth is chosen experimentally by *searching for the best value against the validation set*. This is called hyperparameter tuning (more on this in section **??**).

```
1
2  function split(node, feature, threshold):
3
4      node.condition = (feature > threshold)
5
6      y_left =
       majority_label_of_train_samples_falling_into_left_child
7      y_right =
       majority_label_of_train_samples_falling_into_right_child
8
9      left_child = new_node(prediction=y_left)
10     right_child = new_node(prediction=y_right)
11
12     candidate_tree = tree.copy()
13     candidate_tree.add(node)
14     candidate_tree.add(left_child)
15     candidate_tree.add(right_child)
16
17     return candidate_tree
```

Splitting a node transforms it from an intermediate to a leaf. The split function replaces the prediction value with a splitting condition. The function also add two new children to the tree: *left_node* and *right_node*.

Testing

At test time we assume we have a fully specified tree with *conditions* at intermediate nodes and *predictions* at leaf nodes. We drop x_{test} into our root node and follow it down the tree based on the conditions it satisfies. The leaf node the samples falls in determines the prediction $\hat{y}_{\text{test}}$.

Limitations

The advantages of decision trees are they're **human intepretable and fast to train and test** (at least compared to neural networks). *Ensembles* of decision trees achieve state-of-the-art performance on tabular data. One disadvantage is the hyperparameters (*e.g.* max depth) can be hard to tune but this is often not a big problem in practice.

`\margintoc.`

This figure can be more illuminating

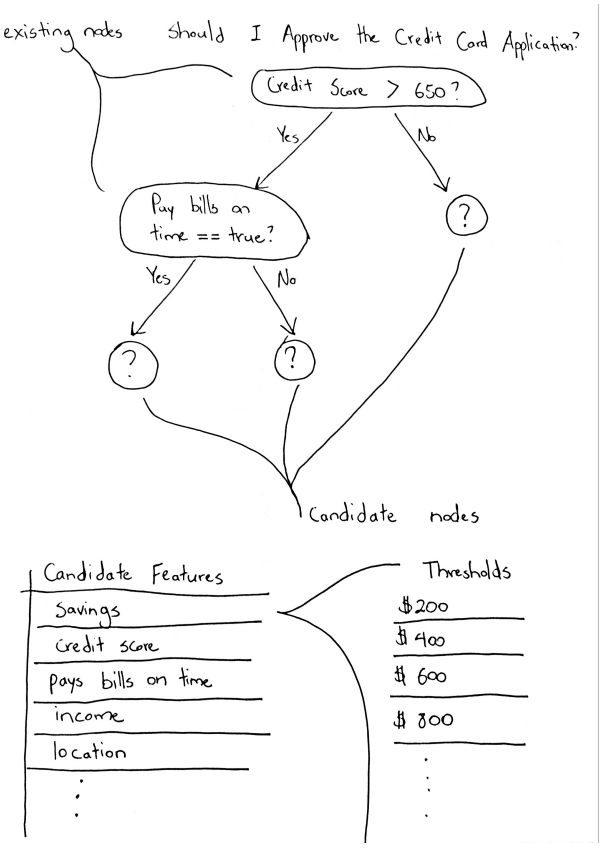

Figure 3.4: Adding the next node to the tree.

3.3 Naive Bayes

Naive Bayes is *tabular* algorithm (meaning the model is simply a lookup table) for binary classification. The idea is to predict based on the correlations between input features and label.

Interviewers like to ask about Naive Bayes not because it is a practical algorithm but because it tests the candidate's basic probability skills.

The derivation for the Naive Bayes algorithms begins with writing down the conditional probability of predicting label y given input x. As the name suggests, we expand the probability using Baye's rule.

$$P(y = 1|x) = \frac{P(x|y = 1)P(y = 1)}{P(x)} \tag{3.1}$$

Training

We're going to count the number of samples in our training set X having labels $y = 1$ and $y = 0$. This will give us $P(y = 1)$ and $P(y = 0)$ which we write down in our lookup table. Modelling $P(x|y = 1)$ is a bit harder. We're going to make the *simplifying assumption* that all features are independent of each other. This is the "naive" part of the algorithm.

$$P(x|y = 1) = \prod_{j=1}^{p} P(x_j|y = 1) \tag{3.2}$$

We model $P(x^j|y = 1)$ by counting the number of times $y = 1$ co-occur with different values of feature j in the training set. We insert $P(x^j|y = 1)$ into the lookup table. We then repeat the process with $P(x^j|y = 0)$. The reason we make the naive independence assumption is each sample x is very unlikely to occur multiple times if the number of features is large. Without assuming independence between features, the lookup table would have to keep track of $P(x|y = 1)$ and it would be very sparse. Notice we didn't bother to calculate $P(x)$, we will see why.

Testing

Given a new sample x_{test}, we use our lookup table to compute the probabilities:

1. $P(x_{\text{test}}|y = 1) = \prod_{j=1}^{p} P(x_{\text{test}}^j|y = 1)$
2. $P(x_{\text{test}}|y = 0) = \prod_{j=1}^{p} P(x_{\text{test}}^j|y = 0)$
3. $P(y = 1)$
4. $P(y = 0)$

We predict $y_{\text{test}} = 1$ if the conditional probability of $y = 1$ is greater than that of $y = 0$ (equation 3.3). Otherwise we predict $y_{\text{test}} = 0$. Notice

the denominator $P(x_{\text{test}}$ cancels out, this is why we did not need to compute it during training.

$$\frac{P(x_{\text{test}}|y=1)P(y=1)}{P(x_{\text{test}})} > \frac{P(x_{\text{test}}|y=0)P(y=0)}{P(x_{\text{test}})} \tag{3.3}$$

Limitations

The main limitation to Naive Bayes is our assumption that features are independent. For example, if our features were pixels of an image, then it would definitely be wrong to assume they were independent!

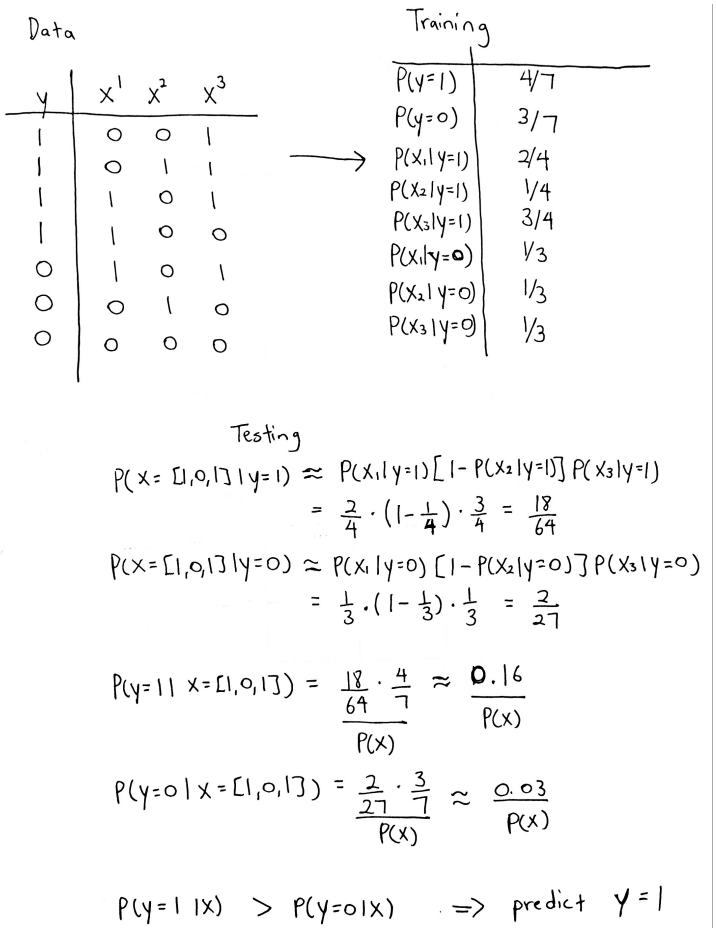

Figure 3.5: An example of the Naive Bayes algorithm.

3.4 k-Nearest Neighbors (kNN)

kNN is a simple algorithm but very scalable algorithm making it popular in industry. It is important for you to be able to explain the limitations of kNN to the interviewer. If a company works on recommendation systems with millions of samples, then they may be interested in scalable kNN algorithms. Resources are linked at the end of this chapter.

kNN can be used for both classification and regression. The intuition is simple. Samples that are *similar* to each other should have the same label. It turns out this is a pretty good assumption and kNN is a strong baseline algorithm in many applications.

Training

In the vanilla kNN algorithm there is nothing you need to do for training because the algorithm simply compares samples to each other. *Scalable kNN algorithms* precompute a data structure to quickly look up *neighbors* at test time.

Testing

At test time we define a function $\phi(x_i, x_j)$ which computes the distance of dissimilarity between two samples. For example we can use the Euclidean distance. The *k nearest neighbors* are the k training samples with the lowest distance values $\phi(x_{\text{test}}, x_j)$. We set y_{test} equal to the majority label of the neighbors. For classification, y_{test} is the most common label of the neighbors. For regression y_{test} is the average label of neighbors.

What is $\phi(x_i, x_j)$? For example, it's a bad idea to compute the similarity of two images in pixel-space. Typically images x_i and x_j are "encoded" using a convolutional neural network like Resnet. An encoding is a "semantic representation" of the image (*i.e.* it has the information "does a dog exist?" rather than "does pixel RGB=25,50,25 exist?"). ϕ is the dot product of the two image encodings.

Limitations

kNN is limited by the very assumption that defines the algorithm: samples that are similar to each other should have the same label. If we pick a bad function $\phi(x_i, x_j)$ to compare two samples, then neighboring samples will not have the same label. There are 2 other limitations. Inference is slow if the number of training samples is large since you need to compare against every sample to find the neighbors. Another issue is if the test sample is an outlier, its k closest points will be really far apart and assigning x_{test} the same labels as its neighbors will not be accurate.

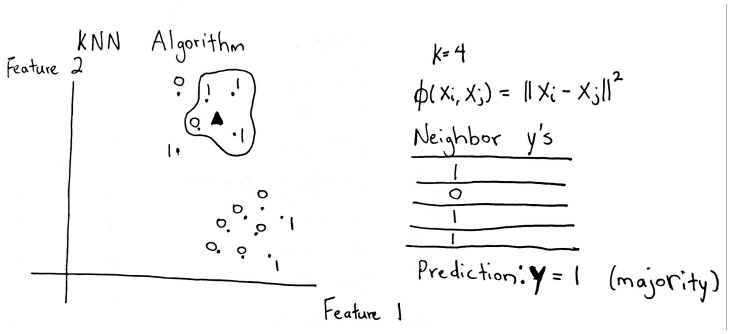

Figure 3.6: An example of kNN.

3.5 k-Means Clustering

Supervised vs Unsupervised Learning

So far all our algorithms have been **supervised learning** algorithms. These algorithms model the label given the input. Mathematically, the model captures the probability $P(y|x)$. **Unsupervised learning** doesn't care about making predictions y. Instead we would like to model the

While you should have a good grasp of supervised vs unsupervised learning, in-depth questions are unlikely in interviews. Interviewers usually ask "what is the difference between supervised and unsupervised learning?"

distribution of the inputs itself $P(x)$. Suppose you're interested in predicting the electricity usage based on the location of households (supervised learning). You may also be interested in finding clusters of households that are close to each other. The clusters you find would be towns or cities (unsupervised learning).

k-Means Clustering Algorithm

The goal of k-Means is to cluster training samples X into k clusters. Samples are grouped based on their similarity or distance to each other. More specifically, we define k clusters centers and each sample belongs to the closest cluster center to it.

Training

The goal of training is to decide the locations of the cluster centers. We begin by randomly initializing the cluster centers (perhaps by setting them to k samples). Each sample in the training set is assigned to the closest cluster. The algorithm proceeds by alternately assigning cluster centers to the average value of the samples belonging to them and updating each sample's cluster assignment to the closest cluster center.

\margintoc.

This is definitely not the best figure to draw for kmeans. Should try to improve on this.

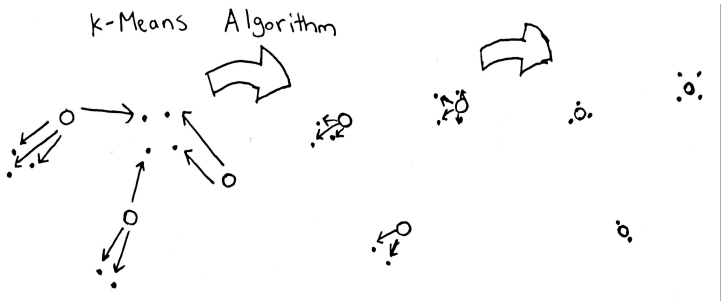

Figure 3.7: An example of k-means. Each point is assigned to the closest cluster center. The cluster centers are then updated to the average of points assigned to it. This can be visualized as the points pulling on the cluster centers.

```
==================== k-Means Algorithm ====================
\\ randomly initialize cluster centers

while (True):

    for all samples:
        assign closest cluster to sample

    for all clusters:
        set cluster center to average value of samples belonging
    to cluster

    if (cluster_centers_are_not_changing_much):
        break
```

Mathematically, suppose we've run the alternating algorithm for t steps and z_i^t is the cluster assignment of sample x_i at step t. Let m_j^t be the cluster center of cluster j at time t and S_j^t the set of points belonging to cluster j. The cluster assignments and cluster centers at the next time step can be computed from equations 3.4 and 3.5.

$$z_i^{t+1} = \arg\min_j \left\| x_i - m_j^t \right\|^2 \tag{3.4}$$

$$m_j^{t+1} = \frac{1}{|S_j^{t+1}|} \sum_{x_i \in S_j^{t+1}} x_i \tag{3.5}$$

Testing

The goal of k-Means at test time is to identify which cluster x_{test} belongs to. We simply assign x_{test} to the closet cluster center.

Limitations

One problem with the k-Means model is how to best choose k? Letting $k = n$ would lead to an absurd model where every sample is its own cluster. What does it mean for x_{test} to be part of a cluster with only one other sample!? Typically we regularize k by penalizing large values. We then find the optimal regularized value by tuning k against the validation set.

Another problem is k-Means is prone to finding sub-optimal or local minima solutions. One can rerun the k-Means algorithm with different random initializations and pick the best run.

3.6 Generative vs Discriminative Models

A common question that comes up is the difference between generative and discriminative model. To get a better intuition, I suggest reading about generative models like variational autoencoders and GANs (links at end of chapter). All of the models we've looked at so far with the exception of k-means have been discriminative models. Recall that k-means is an unsupervised model. Most of the time discriminative models are supervised learning models and generative models are unsupervised learning models. This could be an easy way to differentiate the two but keep in mind it is not always true!

Mathematically, discriminative models seek to learn the conditional probability of the label given the data.

$$P(y|x) \tag{3.6}$$

Examples include classification, regression, and object detection.

Generative models learn the joint distribution of the data and the label.

$$P(x, y) \tag{3.7}$$

Often times labels are not relevant (e.g. in k-means) so we're simply modelling the data distribution,

$$P(x) \tag{3.8}$$

Here's an analogy to help you remember the difference. In school, students write essays (generative model). Teachers grade essays (discriminative model).

3.7 Linear Regression and Gradient Descent

The most transferable ideas from shallow machine learning to deep learning are: bias-variance trade-off, *linear regression, logistic regression and gradient descent*.

Linear regression is the backbone of SVM, Logistic Regression and Neural Networks. Given its significance, interviewers want you to have a solid grasp on both how to train the model and its limitations.

Linear regression models the label y as the product of the data X and a set of weights w. Recall X is of size $n \times p$ and w is $p \times 1$.

$$Xw = y \tag{3.9}$$

It's *regression* because our labels y are real values. It's *linear* because our model says the label is a linear function of the features. The equation for a single sample is,

$$x^1 w_1 + x^2 w_2 + \cdots + x^p w_p = y \tag{3.10}$$

Training: Closed Form Solution

It would be nice if we can find a closed form solution to the linear regression equation. Doing so would give us the optimal w^* and at test time we can make the prediction $y_{\text{test}} = x_{\text{test}} w^*$. A first attempt would look something like this:

$$w^* = X^{-1} y \tag{3.11}$$

However, equation 3.11 doesn't make any sense because X is not a square matrix. In fact, there is usually no solution if $n > p$ because we're trying to make a line in p dimensions go through n different sample points. Therefore, it's more correct to say Xw is equal to the prediction of the model,

$$Xw = \hat{y} \tag{3.12}$$

And we want to minimize the difference between training predictions and training labels.

$$\frac{1}{2}\|\hat{y} - y\|^2 \tag{3.13}$$

In machine learning when we want to minimize the difference between the prediction and the label, we define a loss function. The loss function is a function of the *model parameters i.e.* the variables we can control to reduce the loss.

$$L(w) = \frac{1}{2}\|Xw - y\|^2 \tag{3.14}$$

$L(w)$ is our loss function and it tells us "given a particular set of parameters w, how well does the model do on the (training) set".

The goal of training is to find the parameters w^* which minimize the loss. It turns out we can write a closed-form solution for w^*. Recall X and y are the training set data and labels respectively. For closed-form training, all we need to do is perform the following matrix multiplications.

The derivation of equation 3.15 is included in the Appendix. For those applying to machine learning scientist positions, I highly recommend studying the derivation and being comfortable doing it during the interview.

$$w^* = (X^\intercal X)^{-1} X^\intercal y \tag{3.15}$$

Testing

It's easy to make a prediction once we've found w^*

$$\hat{y}_{\text{test}} = x_{\text{test}} w^* \tag{3.16}$$

Training: Gradient Descent

Great, we can use the exact formula equation 3.15 to find the optimal weights. Is this the end of the story? No, because equation 3.15 is $\mathcal{O}(np^2)$. If $n > 10,000$ and $p > 1000$, equation 3.15 is impossible to compute in practice. It is very common to come across problem where $n > 10,000$ and $p > 1000$, so we **almost never** use the closed form solution to solve linear regression!

Actually equation 3.15 is $\mathcal{O}(np^2 + p^3)$. $\mathcal{O}(np^2)$ for $X^\intercal X$, $\mathcal{O}(p^3)$ to invert $(X^\intercal X)^{-1}$ and $\mathcal{O}(np)$ to compute $X^\intercal \mathbf{y}$.

`\margintoc`.

Consider adding a figure to show gradient descent converging to minima

There is another way to find the solution to the linear regression problem using **gradient descent**. The idea is to iteratively update w and reduce the loss function rather than finding the optimal w in one-go. At each gradient descent step t, we reduce the loss using local, gradient information (recall from section **??** the gradient is the direction of greatest increase). More specifically, we go in the opposite direction of the gradient ($-\nabla$) so as to minimize the loss as much as possible.

Suppose w_t is the weights at step t. We set the weights at the next iteration to be:

$$w_{t+1} = w_t - \alpha \nabla L(w_t) \tag{3.17}$$

References to taking the gradient of the loss function in equation 3.18 is included at the end of this section.

$$\nabla L(w_t) = \nabla \frac{1}{2} \|Xw_t - y\|^2 = X^\intercal (Xw_t - y) \tag{3.18}$$

Here α is the learning rate. It scales the update to the weights at every iteration. A large α can be interpreted as putting more trust in the local information (gradient) to decrease the loss. α is usually tuned against the validation set.

How do we know how many interations to run gradient descent for? We usually stop when $L(w_t)$ is no longer decreasing/improving.

You can check $X^\intercal(Xw - y)$ is $\mathcal{O}(np)$. If we run gradient descent for t iterations, then the algorithm is $\mathcal{O}(npt)$ overall. This makes the solution computable for most problems in industry.

```
======================== GD Algorithm ========================

for t iterations:
    \\ loss_fnc is defined in eq. 3.14
    current_loss = loss_fnc(X, y, w)
    \\ grad_fnc is defined in eq. 3.18
    current_grad = gradient_fnc(current_loss)
    w = w - alpha * current_grad
```

Limitations

The biggest limitation of *linear* regression is it's linear. Unfortunately the output is not always well approximated by a linear function of the input. To get around this, practitioners transform the training matrix X into *non-linear* **engineered features** Z. The model $\hat{y} = Z\mathbf{w}$ is now non-linear in X (see section 3.10 for an example).

3.8 Logistic Regression

A common interview question is to "find the gradients of a single layer neural network with logistic loss". Therefore, it's important to familiarize yourself with backpropagation (in a later chapter) and taking the derivative of the logistic function. Interviewers also expect you to be able to explain advantages the logistic loss function over L2 loss for binary classification.

Logistic *regression* is a misnomer because it's actually an algorithm for (binary) classification. The reason it's called *regression* is because it is based off of linear regression. Actually to transform linear regression into logistic regression, we only need to make a few changes to the loss function.

$$Xw = \hat{y} \tag{3.19}$$

As before the prediction $\hat{y}$ is the product of Xw. Let x_i be the i^{th} sample. We introduce a "logistic function" which converts our predictions $\hat{y}$

into a probability. See figure 3.8, notice the logistic function is between 0 and 1, hence it can be interpreted as a probability.

$$\sigma(\hat{y}_i) = \frac{1}{1 + e^{-\hat{y}_i}} = \frac{1}{1 + e^{-x_i w}} \tag{3.20}$$

More specifically, it's called *logistic regression* because the model *regresses* to the *logits* (which are continuous real values). Wait, so is logistic classification a classification or regression model? The answer is it's a binary classification model where we do regression to the logits.

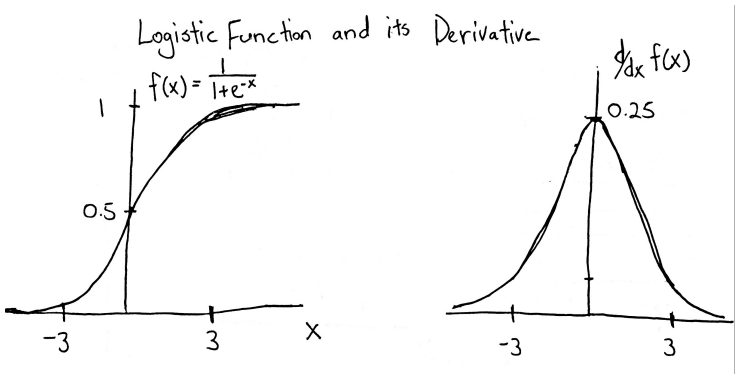

Figure 3.8: Logistic function and its derivative.

$\sigma(\hat{y}_i)$ can be interpreted as the "probability of the class 1 for sample i". We can write this mathematically as $P(y_i = 1)$. Clearly the "probability of the class 0 for sample i" is then $1 - \sigma(\hat{y}_i)$. The loss function for logistic regression is shown below (for a discussion of how this loss function is derived, see section 3.9).

$$L(w) = \sum_{i=1}^{n} y_i \log \sigma(\hat{y}_i) + (1 - y_i) \log(1 - \sigma(\hat{y}_i)), \quad \hat{y}_i = x_i w \tag{3.21}$$

Instead of writing the loss in matrix form, we've written it as a sum of the losses for each sample x_i. y_i is the groundtruth class. It takes on values 0 and 1. Therefore, keep in mind $y_i \log \sigma(\hat{y}_i) = 0$ when $y_i = 0$ and $(1 - y_i) \log(1 - \sigma(\hat{y}_i)) = 0$ when $y_i = 1$. Only one of the two terms is ever contributing to the loss.

$\hat{y}_i = x_i w$ is called the **logit**. Hence, we're using our linear regression model $Xw = \hat{y}$ to "regress to the logits".

Training

We train logistic regression the same way we train linear regression: using gradient descent. Here is the equation in matrix form for the gradient of the logistic loss function.

$$\nabla L(w_t) = X^\intercal(\sigma(\hat{y}) - y) \tag{3.22}$$

Refer to the Appendix for a full derivation of equation 3.22. I suggest knowing at least how to take the derivative of the logistic function.

$$w_{t+1} = w_t - \alpha \nabla L(w_t) \tag{3.23}$$

Testing

Predicting the label of a test sample is also very similar to linear regression. First we predict the logit.

$$\hat{y}_{\text{test}} = x_{\text{test}} w^* \tag{3.24}$$

Next we get the probability of class 1.

$$\sigma(\hat{y}_{\text{test}}) = \frac{1}{1 + e^{-\hat{y}_{\text{test}}}} \tag{3.25}$$

One reasonable thing to do at this point is to say if $\sigma(\hat{y}_{\text{test}}) > 0.5$ then we classify x_{test} as class 1. Else, class 0. Note: $\sigma(\hat{y}_{\text{test}}) > 0.5$ is mathematically equivalent to $\hat{y}_{\text{test}} > 0$. An alternative is to check if the logits are greater than zero.

Limitations

Like linear regression, logistic regression is limited by the linearity between prediction and input. See section 3.10 for an example of feature engineering non-linear features.

Logistic Loss

A common interview question is "why do we use the logistic loss for classification rather than the L2 loss?"

So far the discussion of logistic regression has been rather dogmatic; it's the same as linear regression but with a different loss function.

To understand logistic regression fully, we need to understand why we used *that* specific loss function. Let's consider what happens when we use the L2 (linear regression) loss.

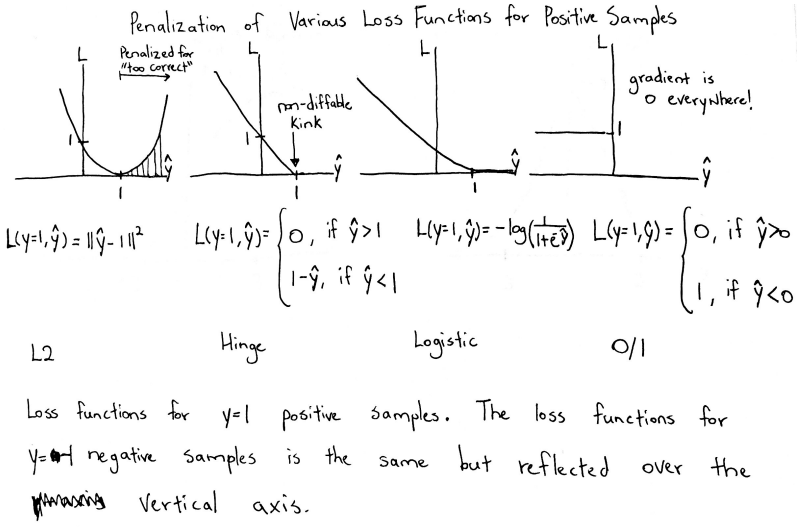

Figure 3.9: Candidate loss functions for binary classification.

A few other losses have been plotted for comparison.

- L2 loss

- 0/1 loss
- logistic loss
- hinge loss

L2 loss doesn't work because it *punishes* correct predictions (in the sense some correct predictions can actually increase the loss).

`\margintoc`.

Expand on this discussoin once diagram has been inserted.

The 0/1 loss doesn't punish correct predictions but notice its derivative (gradient) is 0 everywhere. This means gradient descent will always have an update of 0, making the model untrainable. Logistic loss seems like a good choice because it punishes only wrong predictions and the gradient is non-zero.

The hinge loss can also work. It's a "hard" version of the logistic loss meaning it abruptly changes from the function $y = 0$ to $y = x$ whereas the logistic loss gradually (smoothly) transitions from $y = 0$ to $y = x$. Hinge loss is a perfectly legitimate loss. Using it leads to the SVM model which we will discuss next.

3.9 Where Loss Functions come From

Where do loss functions come from? There are at least 3 answers:

1. Intuition into how the dissimilarity between prediction and ground truth should be minimized
2. Minimizing the cross entropy between prediction and ground truth
3. MLE and MAP methodologies (discussed in section **??**)

Sometimes the best way to construct loss functions is to use our intuition to decide what loss function will *make the model do what we want it to do*. For example in linear regression it makes intuitive sense to define the loss function as the L2 loss $\|\hat{y} - y\|^2$. Minimizing this loss would make predictions similar to ground truth. But notice the L2 loss penalizes the **square** of the difference. A difference of 10 is penalized 25 times more than a difference of 2. This means the model will work very hard to make sure the largest discrepancies between prediction and ground truth are minimized. Sometimes this is not a property of the model we want. Had we used the L1 loss $|\hat{y} - y|$, a difference of 10 is penalized 5 times more than a difference of 2 (directly proportional). L1 loss is used when we suspect there are outliers (difference is large) in our data and we do not want the model to *spend a large effort* reducing the difference for these samples.

There's a way to formalize this intuition of "reducing the difference between the prediction and ground truth" using probability. To derive the logistic regression loss, we first define a *groundtruth probability distribution* and *predicted probability distribution*. Dissimilarity (the loss) between the two probability distributions can be measured using **cross entropy**. Minimizing the cross entropy trains the model. The predicted probability distribution is simply the output of the logistic model. But wait, what is the *groundtruth probability distribution*? We have a single

value y_i but we can consider it as a one-hot probability distribution. This becomes clearer if we write y_i in vector form y_i:

$$y_i = \begin{bmatrix} P(y=0) \\ P(y=1) \end{bmatrix} = \left\{ \begin{array}{ll} \begin{bmatrix} 1 \\ 0 \end{bmatrix}, & \text{if } y_i = 1 \\ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, & \text{if } y_i = 0 \end{array} \right\} \tag{3.26}$$

In other words, the groundtruth probability distribution is 0% for $y_i = 0$ and 100% for $y_i = 1$ when y_i is 1. Likewise, it's 100% for $y_i = 0$ and 0% for $y_i = 1$ when y_i is 0. The predicted probability distribution is:

$$\hat{y}_i = \begin{bmatrix} P(\hat{y}=0) \\ P(\hat{y}=1) \end{bmatrix} = \begin{bmatrix} \sigma(\hat{y}_i) \\ 1-\sigma(\hat{y}_i) \end{bmatrix} \tag{3.27}$$

You should have a working understanding of cross entropy for the interview. See the Appendix for more details.

The loss is the cross entropy (denoted H) between $\hat{y}_i$ and y_i (the cross entropy measures the difference between two probability distributions). Intuitively, this loss function makes $\hat{y}_i$ more similar to y_i which is what we want.

$$\begin{aligned} L(\hat{y}_i, y_i) = H(\hat{y}_i, y_i) &= y_i[0]\hat{y}_i[0] + y_i[1]\hat{y}_i[1] && (3.28) \\ &= y_i[0]\sigma(\hat{y}_i) + y_i[1](1-\sigma(\hat{y}_i)) && (3.29) \end{aligned}$$

In general there's no *best* way to come up with loss functions. While cross entropy is the more *principled* approach, sometimes experienced practitioners can have a good intuition for what loss functions will work better without working out all the math.

3.10 Support Vector Machines

As we have discussed, SVM is what you get when you replace the logistic loss with the hinge loss. To make the math simpler, we typically denote the classes as 1 and −1 rather than 1 and 0 as in logistic regression. From figure **??** we see the hinge loss is:

$$L(w) = \left\{ \begin{array}{ll} w^\intercal x_i, & \text{if } y_i = -1 \text{ and } w^\intercal x_i > 1 \\ w^\intercal x_i, & \text{if } y_i = 1 \text{ and } w^\intercal x_i < -1 \\ 0, & \text{otherwise} \end{array} \right\} \tag{3.30}$$

It can be show this is mathematically equivalent to:

$$L(w) = \sum_{i=1}^{n} \max\{0, 1 - y_i w^\intercal x_i\} \tag{3.31}$$

We then add a L2 regularization term (more on regularization in the next chapter) wlhich encourages the parameters to take on small values.

$$L(w) = \overbrace{\sum_{i=1}^{n} \max\{0, 1 - y_i w^\intercal x_i\}}^{\text{hinge loss}} + \overbrace{\frac{1}{2}\|w\|^2}^{\text{L2 regularization}} \tag{3.32}$$

Great, we have our loss function and we can optimize it using gradient descent. Not so fast. Unfortunately the max operator makes the loss function *non-differentiable* meaning we cannot train a SVM model using gradient descent. Training SVMs is beyond the scope of this book and most ML interviews. Instead what we will focus on is a qualitative understanding of the advantages of SVM models.

Try understand SVMs, we need to make the simplifying assumption that the training samples are *linearly separable*. All this means is that for example in figure 3.10 the blue and green classes can be perfectly separated with a line. In a real dataset, linear separability is almost never satisfied. We make this assumption for the purpose of understanding.

We motivated SVM's as a potential alternative to the logistic regression model for binary classification (trained with a slightly different loss). Let's motivate the benefits of SVMs with a series of question-answers.

1. Q: SVMs are called a *max-margin* classifiers, Why? A: First, the margin is the distance between the closest members of two linearly separable classes. This is shown as the width of the yellow region in figure 3.10. Max-margin is the widest yellow region we can draw. For example if we tilted the red line (decision boundary) a bit, the yellow region would shrink. SVM finds the solution with the largest separation between the closest members of opposing classes.
2. Q: Why is max-margin good? A: Visually we can see that having a large gap means we can more reliably classify test points that are close to the decision boundary. An alternative interpretation is that it makes the least assumptions about where the "true decision boundary is" given we don't see any training samples within the yellow area.
3. Q: How do we know SVMs have *max-margins*? A: The regularization term $\|w\|^2$ tries to make w as small as possible. When w is small, the change to the prediction $\Delta y = w\Delta x$ is small (*i.e.* the model is robust). This is seen in figure 3.10 where it takes a large change in x (in fact the maximum change/margin) to go from one class to another.
4. Q: How do you compute the margin? A: As seen in figure 3.10 the margin is $2/\|w\|$. You can work this out by finding the distance between the two lines marking the boundary of the yellow region. This makes intuitive sense because as $\|w\|$ decreases, x needs to change a lot before the class changes (*i.e.* the margin is large).

Testing

At test time we simply compute $\hat{y}_{\text{test}} = w^\intercal x_{\text{test}}$. We predict class 1 if $\hat{y}_{\text{test}} > 0$, else class -1.

Training

Training SVM models gets quite mathematically involved and most interviewers will not expect you to explain in-depth. As seen from equation 3.32, the *SVM loss is non-differentiable* because of the max operator. This means we cannot optimize the loss using gradient descent as we had for linear and logistic regression. For reference, SVM's are optimized using *sub-gradient descent and coordinate descent algorithms*.

Limitations

SVM, like linear and logistic regression, is limited by its linearity. Let's address this now.

Gaussian Basis Functions

GBFs+SVM is usually one of the first models you should try for binary classification on *tabular data*. It frequently works better than SVM alone.

How do we convert regular SVM's into a near state-of-the-art algorithm? By using Gaussian Basis Functions (GBFs). Linear regression, logistic regression, SVM are all linear models because the predictions $\hat{y} = Xw$ is a linear function of both the weights and features. The motivation behind GBFs is to introduce non-linearity into the model (hence making it more "expressive" or increasing its capacity) by making the *features non-linear*. The idea is to transform the training feature matrix X into a transformed-feature matrix Z. The prediction is then:

$$\hat{y} = Zw \tag{3.33}$$

Specifically, we choose the below definition for Z. This is a case of **feature engineering**.

$$Z = \begin{bmatrix} e^{\|x_1 - x_1\|^2/\sigma^2} & e^{\|x_1 - x_2\|^2/\sigma^2} & \cdots & e^{\|x_1 - x_n\|^2/\sigma^2} \\ e^{\|x_2 - x_1\|^2/\sigma^2} & & & \\ \vdots & & \ddots & \\ e^{\|x_n - x_1\|^2/\sigma^2} & & & e^{\|x_n - x_n\|^2/\sigma^2} \end{bmatrix} \tag{3.34}$$

In other words:

$$Z_{ij} = \exp\left(\frac{-\left\|x_i - x_j\right\|^2}{\sigma^2}\right) \tag{3.35}$$

k-Nearest Neighbors where every point is a weighted neighbor. $\exp(-\|x_i - x_j\|^2/\sigma^2)$ is the similarity/weight between neighbors (weight in the sense of weighted, not weights w). w_i is the "vote" neighbor i contributes to the prediction. σ controls how to weigh votes from closer and farther neighbors.

$\exp\left(\frac{-\|\mathbf{x_i} - \mathbf{x_j}\|^2}{\sigma^2}\right)$ is called the **Gaussian kernel** and σ is called the **bandwidth** of the kernel. Notice now our feature vector Z is $n \times n$ and our weight vector w is $n \times 1$.

Why is $\exp\left(\frac{-\|x_i - x_j\|^2}{\sigma^2}\right)$ a good feature? The intuition is $\exp\left(\frac{-\|x_i - x_j\|^2}{\sigma^2}\right)$ computes the similarity between points x_i and x_j. Perhaps x_i is a prototypical sample for class 1. Therefore, proximity to point x_i is a good indicator of class 1. In this way, GBFs can be seen as an extension of k-Nearest Neighbors where we consider the similarity of every point to every other point.

Training

Training SVM+GBFs is the same as training just SVM but X is replaced by Z.

Testing

At test time we first construct z_{test} by computing the similarity of x_{test} to every point in the training set. Next, we compute $\hat{y}_{\text{test}} = w^\top z_{\text{test}}$.

$$z_{\text{test}} = \begin{bmatrix} e^{\|x_1 - x_{\text{test}}\|^2/\sigma^2} \\ e^{\|x_2 - x_{\text{test}}\|^2/\sigma^2} \\ \vdots \\ e^{\|x_n - x_{\text{test}}\|^2/\sigma^2} \end{bmatrix} \tag{3.36}$$

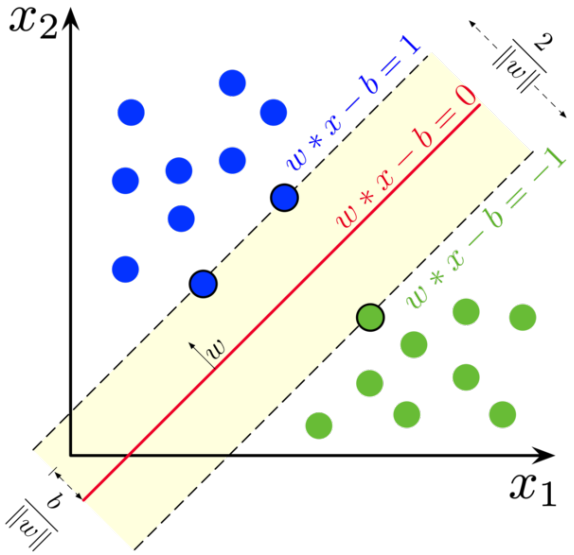

Figure 3.10: A diagram showing the *max-margin* property of the SVM algorithm. *Be careful in interpreting this visualization, the axes are two features x_1 and x_2 and red is the decision boundary.* SVM finds the solution with the largest margin or gap between points from different classes (imagine tilting the red line solution, the distance between the closest blue and green points would be reduced). Note, increasing $\|w\|$ would increase the width of the margin, not change its slope.

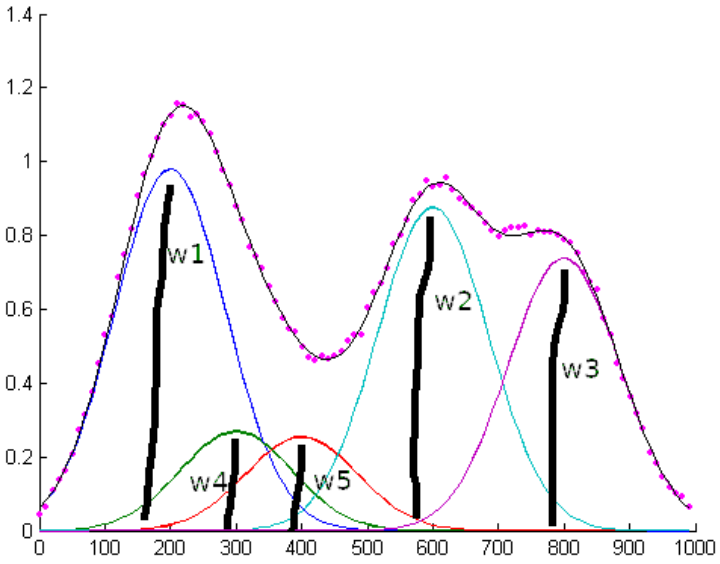

Figure 3.11: A visualization of linear regression with Gaussian RBF. Suppose we want to fit the purple samples. Five Gaussian functions from the RBF is shown. The learned weights are the heights of the Gaussians. Added together, the separate Gaussian RBFs make up the blue fitted line.

3.11 Parametric vs Nonparametric Models

GBFs+SVM is a **nonparametric model**. SVM alone is a **parametric model**. We won't go in-depth on the difference. For the machine learning interview, just know the difference is that nonparametric models have a number of parameters proportional to the number of training samples and parametric models have a number of parameters independent of the number of training samples. Recall that with GBF Z is a $n \times n$ matrix and our weight vector $\mathbf{w}$ is $n \times 1$. In contrast, $\mathbf{w}$ is $p \times 1$ if we train on X. The key difference is the number of parameters depends on n in the former case but not in the latter.

3.12 Model taxonomy

We round off this chapter with an overview of the models originating from linear regression.

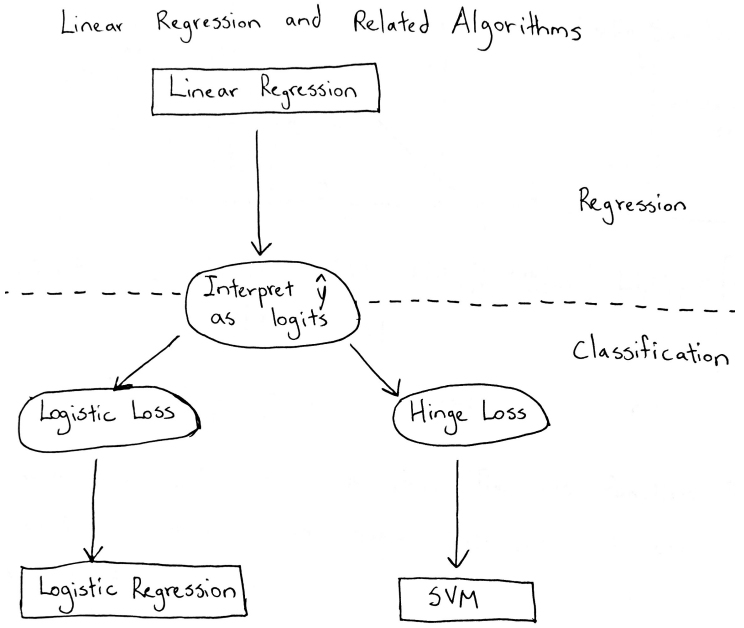

Figure 3.12: A taxonomy of linear regression, logistic regression and SVM.

Deep Learning 4

Deep learning (DL) has often been called a **black box** and optimizing deep learning systems is "more of an art than a science". DL has gained this reputation because practitioners spend most of their time choosing hyperparameters and model architectures seemingly arbitrarily. For this reason, interviewers appreciate it when candidates can shed some light onto the inner workings of DL by discussing the mathematics. This section will focus on the math of deep learning starting with backpropagation then moving on to convolution neural networks (CNNs) and recurrent neural networks (RNNs). Don't worry, despite our ambitious goals, I'll try to make the math as intuitive as possible.

Unfortunately it's impossible to cover all topics in deep learning. The literature for computer vision is very different from NLP which is very different from reinforcement learning, and so on. In this chapter, we will cover the most fundamental elements of deep learning that will help in any interview. I suggest supplementing your studies by reading papers and blogposts from the subfield of DL you're applying to.

4.1 Backpropagation

I assume you are familiar with the basic concepts of a neural network. To summarize, we have an input x which we pass through a sequence of "layers" to get prediction $\hat{y}$. A loss function $L(\hat{y}, y)$ defines the discrepancy between the model's prediction and groundtruth. We train the "layers" by minimizing the loss via backpropagation.

To make our intuition formal, let's define these variables.

1. activation function $f(\cdot)$ (*e.g.* ReLU, Sigmoid, Tanh)
2. weights of the model (also know as parameters) w_i
3. pre-activations $u_i = w_i z_{i-1}$
4. hidden units $z_i = f(u_i)$

We assume there are L layers in total and the activation functions are the same at each layer. For simplicity, we assume the hidden size of each layer is the same and equal to the number of features p. In other words our network has a constant width. Below I've written the series of steps or computations we take to get the prediction (and subsequently the loss) from the input.

The variables on the left side of the equations are known as the intermediate representations, neurons, or "hidden states". Although there seems to be a lot of math, the math is actually quite simple because it is a repetition of only a few types of equations. If we drop the indices, the only types of equations we have to worry about are,

The idea behind backpropagation is to use gradient descent to minimize the loss respect to the model weights, exactly the way we did it for linear equation.

$$w_{t+1} = w_t - \alpha \nabla L(w_t) \tag{4.1}$$

The only catch is the parameters we want to minimize are spread out amongst all the layers. There's a weight matrix w_t^l (l indexes the layer and t indexes the GD iteration) at every layer. Furthermore, we want each step of GD to update the entire network at once.

It turns out this is possible if we apply the chain rule to calculate dL/dw^l. But first, we'll take a detour to calculate dL/dx, the gradient of the output with respect to the input. As we will see, this calculation helps us greatly in finding dL/dw^l which is what we ultimately want.

To begin, recall the chain rule from section **??**.

$$\frac{dL}{dx} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dz_{L-1}} \frac{dz_{L-1}}{du_{L-1}} \frac{du_{L-1}}{dz_{L-2}} \frac{dz_{L-2}}{du_{L-2}} \frac{du_{L-2}}{dz_{L-3}} \cdots \frac{dz_1}{du_1} \frac{du_1}{dx} \tag{4.2}$$

As a sanity check, all intermediate variables "cancel out on the top and bottom". For example $d\hat{y}$ appears on the bottom in the first term and top on the second term. What we've done is we've used the chain rule to write the gradient of the loss L with respect to the input x as a product of the derivatives of each intermediate layer. Just as the "forward" calculation required us to compute something at each layer, backpropogation requires us to compute the derivative at every layer. These two processes are illustrated side by side below.

The left side of table **??** is called the **forward** pass. We can think of it as a *game of telephone* where the first person start with the input x, and the message is passed through several people until the final message $\hat{y}$. Every person in the chain modifies the message somehow. Loss describes the difference between the final message $\hat{y}$ and the expected final message y. The right side is called the **backward** pass. Here we're

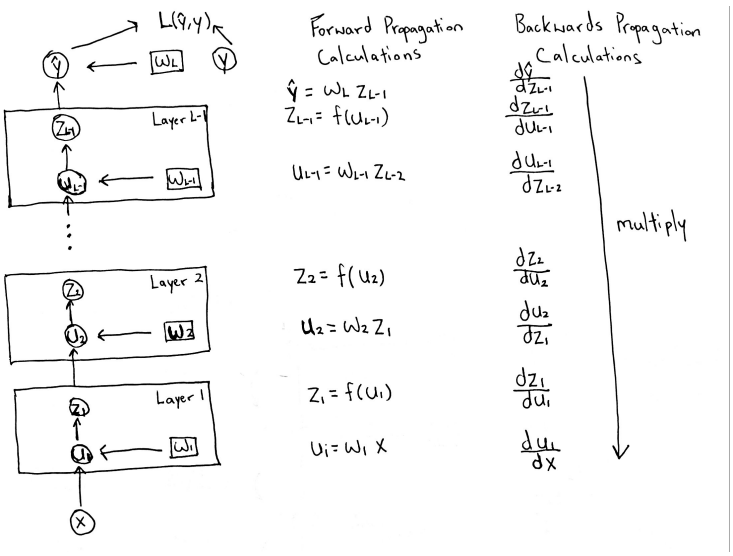

Figure 4.1: Visualization of a neural network, the forward and backwards passes.

running our game of telephone in reverse. The first person passes the message $dL/d\hat{y}$. Each subsequently person modifies the message by multiplying it with the gradient and passes it to to the next person.

The complication is u, z, and x are all multi-dimensional so the derivatives are actually *Jacobian* matrices. Jacobian matrices describe the gradient when there is more than one output (in this case each neuron in the previous layer feeds to multiple neurons in the next layer). For example:

$$J_{ij} = \frac{dy_i}{dx_j} \tag{4.3}$$

Therefore the first row of the Jacobian is the gradient of the first output y_0.

$$J_{0j} = \frac{dy_0}{x_j} = \nabla_x y_0 \tag{4.4}$$

\margintoc.

Not clear if we want to define Jacobians here, probably should move to math section but if we do we need a visual.

Again, it looks like there's a lot of math but the math is actually quite simple because it's a small set of equations repeated many times. I've enumerated all the types of equations we need to worry about below.

We can think of figuring out how to solve each of these derivatives as a subproblem to the main problem of finding dL/dx. Let's first focus on solving each subproblem. Then, we can bring these subproblems back together and find dL/dx.

$\frac{dL}{d\hat{y}}$ is a scalar value. You can also think of it as a 1×1 matrix which we can find by taking the derivative of the loss function.

$$\frac{dL}{d\hat{y}} = \frac{d}{d\hat{y}} L(\hat{y}, y) = L'(\hat{y}, y) \tag{4.5}$$

$\frac{d\hat{y}}{dz_{L-1}}$ is the $1 \times p$ matrix w_L.

$$\frac{d\hat{y}}{dz_{L-1}} = \frac{d}{dz_{L-1}} w_L z_{L-1} = w_L \tag{4.6}$$

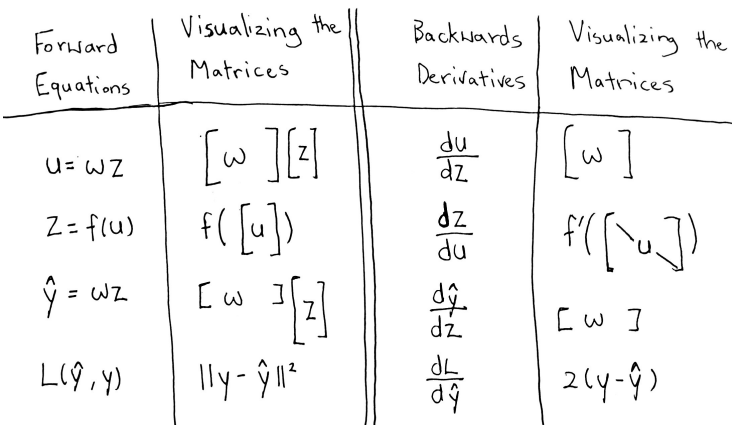

Figure 4.2: Types of equations we need to compute for forward and back propagation.

$\frac{dz_i}{du_i}$ is the $p \times p$ matrix $f'(u_i)$. We can compute $f'(u_i)$ by applying the derivative function f' element-wise to the *matrix* of u_i. u_i is actually a vector, so by "the matrix of u_i", we mean the diagonal matrix with the values of u_i along the diagonal.

$$\frac{dz_i}{du_i} = \frac{d}{du_i} f(u_i) = f'(u_i) \tag{4.7}$$

\margintoc.

Any chance for visuals here?

The derivative $\frac{du_i}{dz_{i-1}}$ is the $p \times p$ matrix w_i.

$$\frac{du_i}{dz_{i-1}} = \frac{d}{dz_{i-1}} w_i z_{i-1} = w_i \tag{4.8}$$

So far we've figured out how to find all of the different types of derivatives. We can now find the derivatives at each layer and compute $\frac{dL}{dx}$ by multiplying the derivatives together in a backwards message passing way starting with $\frac{dL}{d\hat{y}}$.

With $\frac{dL}{dx}$ computed, let's turn our sights on the original goal of finding $\frac{dL}{dw^l}$.

It turns out we can find $\frac{dL}{dw^l}$ by applying the chain rule one more time and passing the message *sideways*.

$$\frac{dL}{dw_i} = \frac{dL}{du_i} \frac{du_i}{dw_i} \tag{4.9}$$

One of the intermediate messages we computed when computing $\frac{dL}{dx}$ is $\frac{dL}{du_i}$. Therefore, we can get $\frac{dL}{dw^l}$ by multiplying the intermediate message with the derivative $\frac{du_i}{dw_i}$. Computing $\frac{dL}{dw_i}$ is the last subproblem we need to deal with before our telephone game description of backpropagation is complete. It turns out doing so is quite simple.

$$\frac{du_i}{dw_i} = \frac{d}{dw_i} w_i z_{i-1} = z_{i-1} \tag{4.10}$$

$\frac{du_i}{dw_i}$ is a $p \times p \times p$ **tensor**. A vector holds elements in a 1D array, a matrix in 2D array, and a 3-tensor holds elements in a 3D array. In other words we'd need a 3D cube to store all the entries in $\frac{du_i}{dw_i}$.

What is the shape of the matrix z_{i-1}? This is where the math gets a bit tricky and we'll fudge some things to simplify the situation. Notice that u_i is a p dimensional vector and w_i is $p \times p$. This means the "matrix" $\frac{du_i}{dw_i}$ should have a single entry telling us the derivative of each element of u_i with respect to each weight in w_i. This requires $p \times p \times p$ such entries. Looks complicated. We can bypass the math in exactly how $\frac{du_i}{dw_i}$ is represented (we know it is somehow equivalent to z_{i-1}) if we just consider finding $\frac{dL}{dw_i} = \frac{dL}{du_i} \frac{du_i}{dw_i}$.

It's easiest to visualize this special multiplication operation using a diagram (shown below). See the Appendix for a mathematical way of taking the product.

We've now understood the solution to every subproblem in the backwards telephone game. We understand how each person computes their own derivative and how these derivatives can be multiply to

pass on the message. Let's summarize the gradient descent algorithm for training neural networks.

\margintoc.

Consider notation change to W for weights in all layers.

```
==================== GD for Neural Network ====================

for t iterations:
    \\ pass telephone message forward (forward propagation)
    current_loss, Z, U = loss_fnc(X, y, W)

    \\ compute intermediate messages (Jacobians)
    jacobians = find_derivatives(current_loss, X, y, Z, U, W)

    \\ pass telephone message backward (backpropogation)
    dLdW = multiply_backwards(jacobians)

    \\ gradient descent step
    w = w - alpha * dLdW
```

\margintoc.

Is this algo unclear?

As with gradient descent for linear regression, we perform these steps for t iterations until *convergence*.

We have discussed training neural networks with gradient descent. Applying GD to neural networks requires us to recursively apply the chain rule in a telephone message passing fashion. This combination of chain rule and GD is called the backpropogation algorithm.

The specific neural network model we've studied is called the multilayer perceptron (MLP) or vanilla neural network (VNN). It is conceptually the simplest variant of neural networks. Backpropagation algorithms exist for other neural network models such as convolutional neural networks (CNN) or recurrent neural networks (RNN). The algorithms for these are slightly different but follow the same central idea of combining recursive chain rule with gradient descent.

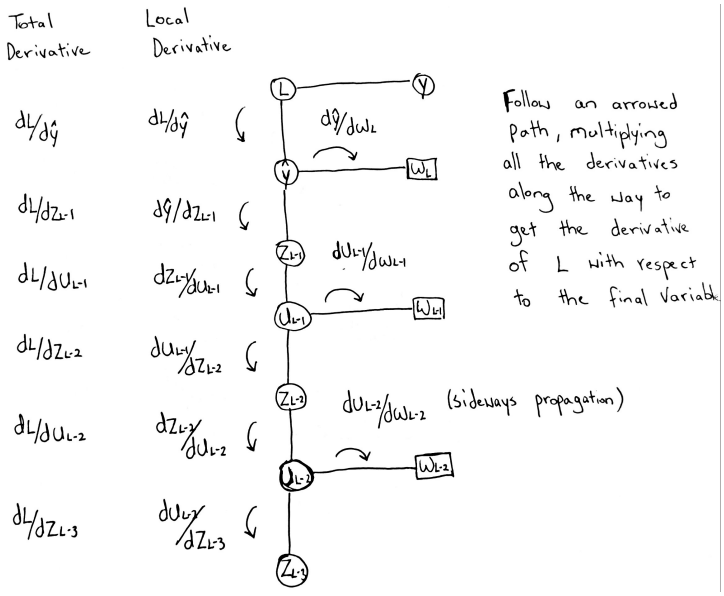

Figure 4.3: Illustration of propagating the gradient backwards.

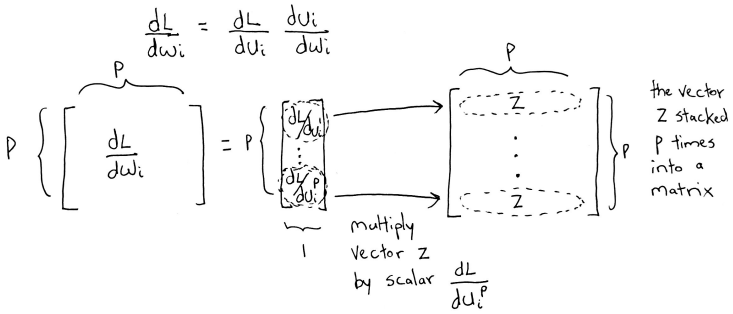

Figure 4.4: Special multiplication for $\frac{dL}{dw_i}$.

4.2 Deep Learning is Basically Linear Regression

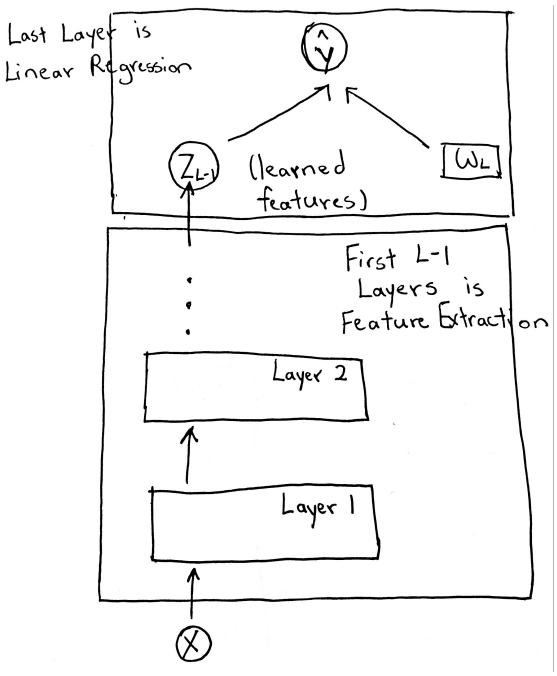

Figure 4.5: Neural network is linear regression stacked on top of learned features.

The astute reader would have noticed some striking similarities between neural network and linear regression models. Let's make these similarities more concrete. Table **??** compares the two algorithms.

The loss doesn't have to be the L2 loss. For example, it could be the logistic loss for binary classification. The point is, any loss used in linear regression can also be used in neural networks.

Let's take these similarities to their logical conclusion and state exactly how (little) neural networks differ from linear regression.

The first difference is GD can be applied directly on linear regression whereas neural networks require GD to applied with recursive chain rule (backpropogation).

The second difference is in how the prediction $\hat{y}$ is computed. It's not a concidence that we chose to use the symbol Z for both the hidden units in the neural network and the engineered features in the GBF section.

Recall from section 3.10 that we can replace the data matrix X with **feature engineered** features Z. We showed this for the logistic regression model but we can easily do the same for linear regression. The point of replacing X with Z was to introduce non-linearity into the features

	Linear Regression	Neural Network
Training algorithm	Gradient Descent	Gradient Descent with Chain Rule (BackPropoagation)
Loss	$\|\|\hat{y} - y\|\|$	$\|\|\hat{y} - y\|\|$
Prediction	$\hat{y} = Xw$ $\hat{y} = Zw$ (engineered features, *e.g.* GBF)	$\hat{y} = Zw$ (where Z is the hidden units at the last layer)

and hence the model. In section 3.10 we "hand picked" what Z should be (namely the GBFs).

$$\hat{y} = Zw \tag{4.11}$$

The above equation is also the equation we use to compute the prediction for the neural network. In that case, Z is the last layer of hidden units. Z is computed by the other $L-1$ layers in the neural network and it turns out that Z is also a non-linear function of X. We're now ready to make our insightful statement. **A neural network is a linear regression model with non-linear features Z which are computed by the first $L-1$ layers rather than "hand picked" (as in the case of GBFs).** The last layer of a neural network is exactly a linear regression model. What do we mean by "rather than hand picked"? The first $L-1$ layers are also *learned from the data* using backpropogation. This means over time the neural network learns how to convert the input into useful non-linear features, it does not need a machine learning expert to specify this transformation.

Just as there is an analogy between linear regression and neural network (with L2 loss), there is an analogy between logistic regression and neural network (with logistic loss). To be more precise, the last layer is exactly a logistic regression model and the first $L-1$ layers *learn* to compute non-linear features Z.

4.3 Multiclass Classification

Interviewers like to ask when you would use a Softmax loss versus a Logistic loss.

So far we've only looked at binary classification. What if there are more classes?

In binary classification, we had a single prediction $\hat{y}$. The simple fix is to make multiple predictions, specifically C of them for C classes. In this world, $\hat{y}$ is a C dimensional vector of predictions.

But before we jump the gun, I'd like to first introduce the notion of **class competition**. Class competition states whether an object can be labelled with two different classes (*e.g.* dog and golden retriever) or only one (*e.g.* alligator or elephant).

Specifically, we care about the competition between the predicted classes of a (neural network) model. Consider the task of classifying images. If class competition exists amongst the predictions, then each image can only have a single true class *e.g.* the classes are alligator, elephant, dog, cat. This makes sense because elephants cannot be alligators. If class competition doesn't exist amongst the predictions, then each image can have multiple true classes. *e.g.* the classes are dog, yellow dog, golden retriever. Our model should classify a picture of a golden retriever as "dog", "yellow dog" and "golden retriever". You wouldn't want class competition with these classes or the model will be learning that yellow dogs are not dog or golden retrievers are not yellow dogs, and so on.

Like we said, for multiclass classification we should predict a C dimensional vector $\hat{y}$, one dimension for each class. However, there are two loss functions we can use depending on whether we want to "teach the model" to have class competition amongst its predictions.

The *Softmax loss* **causes class competition**. It does this by defining the predicted probability of a particular class relative to the other classes. Suppose we make C predictions and $\hat{y}$ is a C dimensional vector of logits for each class. Let p be a vector storing the (predicted) probabilities of each class. The equation for the *Softmax function* is defined in equation 4.13.

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_C \end{bmatrix} \tag{4.12}$$

$$p_i = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{C} e^{\hat{y}_j}} \tag{4.13}$$

p_i is the predicted probability for class i. We can check that the sum of probabilities over all classes is 1.

The Softmax loss is,

$$L(p, y = c^*) = -\log(p_{c^*}) = -\log\left(\frac{e^{\hat{y}_{c^*}}}{\sum_{j=1}^{C} e^{\hat{y}_j}}\right) \tag{4.14}$$

c^* is the ground truth class for the image. Each image only has one ground truth class. Let's interpret the loss mathematically. Minimizing the negative log of something is the same as maximizing that something, in this case $\frac{e^{\hat{y}_{c^*}}}{\sum_{j=1}^{C} e^{\hat{y}_j}}$. When this happens, $e^{\hat{y}_{c^*}}$ in the numerator will increase meaning the ground truth class learns to be the more dominant prediction. More interestingly, $e^{\hat{y}_j}$ in the denominator will decrease for all j. Hence, the other classes learn to be less likely predictions. The correct class is "out-competing" the other classes during training.

The *multiclass logistic loss* **does not cause class competition**. It defines the predicted probabilities of classes independent of other classes. It differs from the binary logistic loss by having C predictions, one for each class. As before, let p be a vector storing the probabilities of each class. Each probability can be calculated from equation 4.16.

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_C \end{bmatrix} \tag{4.15}$$

$$p_i = \frac{1}{1 + e^{\hat{y}_i}} \tag{4.16}$$

You can check that the sum of probabilities over all classes does not add to 1 in general. This is because the probabilities are independent.

The multi-class Logistic loss is:

$$L(\mathbf{w}) = \sum_{i=1}^{C} y_i \log(p_i) + (1 - y_i) \log(1 - \sigma(p_i)) \tag{4.17}$$

What we've done is applied the binary logistic loss independently to each class. y_i is the ground truth label for each class (1 if the image is class i and 0 otherwise). A single image can have multiple ground truth labels. Let's interpret the loss mathematically. For each class i, increasing p_i has no effect on the other classes. This is because the equation for p_i depends only on class i.

We're now ready to put linear regression, neural networks, and the various losses they can take on under the same framework. Figure **??** relates all the models we've seen.

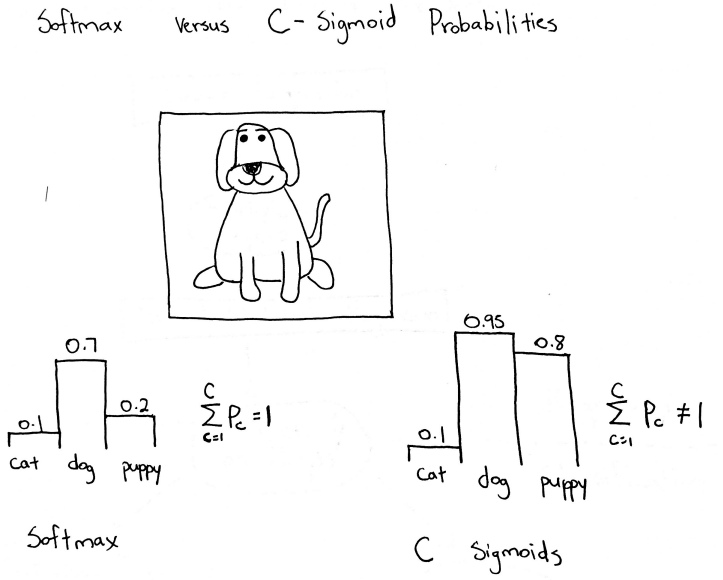

Figure 4.6: Prediction probability distributions of Softmax versus Sigmoid losses.

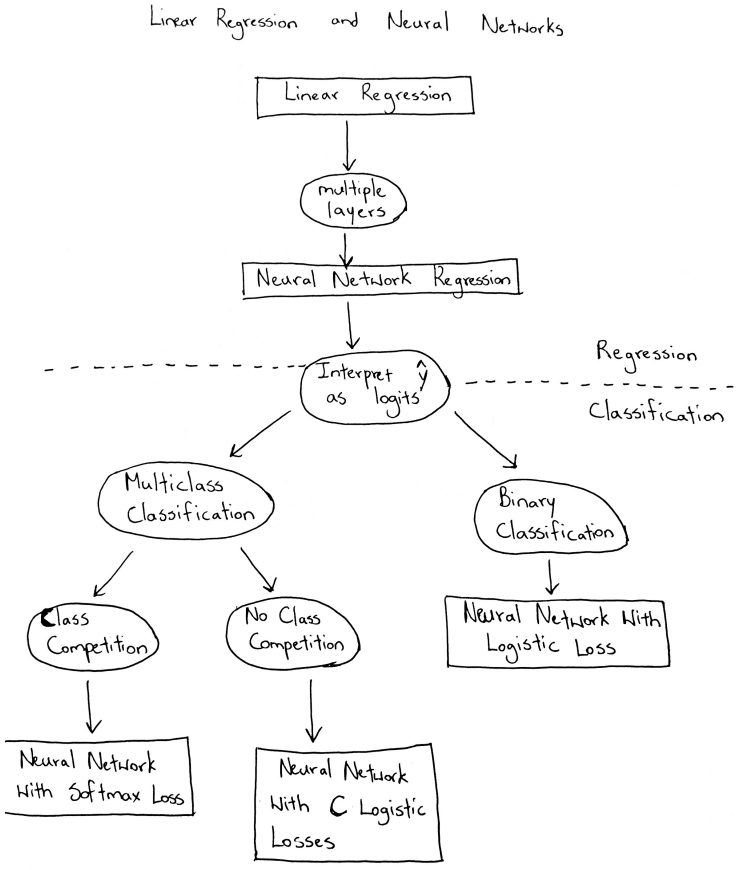

Figure 4.7: A taxonomy of linear regression and neural networks.

4.4 Convolutional Neural Networks

Convolutional neural networks are state-of-the-art not only on computer vision tasks but many others in NLP, audio, and sequential data. A common interview question is to explain the advantages of CNNs over vanilla neural networks.

Convolutional neural networks (CNNs) are the de-facto algorithm in computer vision. In this section, we'll take a look at why they have an advantage over *vanilla* neural networks as well as what limitations there are in training them.

Convolution

A derivation of the gradient is in the Appendix.

Convolutional neural networks follow the same high-level forward and backward propagation rules as vanilla neural networks. However, there is a key difference in how the input is represented. In CNNs x is a 2D grid (*e.g.* of pixels in computer vision). The hidden layers z_i are 2D as well. This changes how weights are multiplied to get the next layer's pre-activations.

To be specific, x is a $H \times W \times k$ tensor where H and W are the spatial dimensions height and width. k is the number of channels. *e.g.* $k = 3$ for RGB images. $k = 1$ for black and white images. z_l is similarly $H_l \times W_l \times k_l$. H_l and W_l are the height and width of the hidden units grid. They depend on the layer l. k_l is the number of channels which is the number of hidden neurons at each location on the grid. Under the CNN framework, a hidden layer in a VNN is simply a 1×1 grid of neurons, a special case.

CNNs are also trained using the forward propogation - backpropogation algorithm. Just like training VNN, we combine gradient descent with recursive chain rule. The difference is how we solve the subproblems in the forward and backward passes. Instead of a regular weight multiplication, each layer performs a convolution to compute the next layer. I won't go into CNNs in-depth because there are many great resources to learn about that online (links at the end of section). I will instead focus on the advantages and limitations of CNNs and how we can overcome them assuming you have a working knowledge of CNNs.

Advantages

The question we should ask for any neural network architecture is when and why should we use it over VNN? To answer the when, CNNs are used ubiquitously for images and sound. They're the model of choice whenever it makes sense to represent the input as a 2D grid. To answer the why, we should consider what happens when we process an image using a VNN.

We can process an image using a vanilla neural network in the following way. We stack the pixels together into a 1D vector by traversing the image left to right and top to bottom. Each element/pixel of that vector is a feature. We pass the features into a vanilla neural network.

Empirically this does not work well. Why? Because by stacking the pixels into a 1D vector the vanilla neural network receives no information about the relative positions of each pixel. *CNNs maintain the 2D dimensionality of the input and exploits it*.

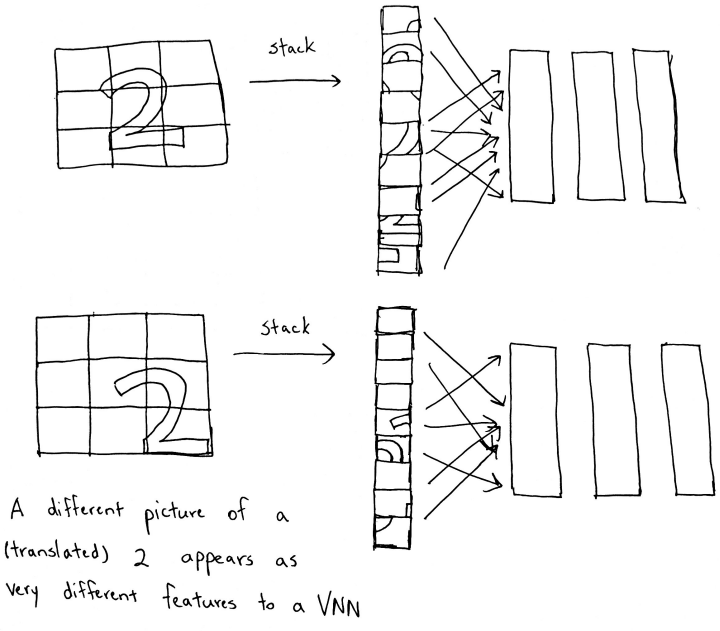

Figure 4.8: VNN is not translation invariant. This makes a shifted "2" looks like a completely new set of features to the VNN.

CNNs exploit location information by ensuring that the same concept is only learned once and can be applied at every location on the image. This is the idea of **translational invariance**. Suppose we're trying to detect dogs in an image. CNNs learn a dog *filter*. A filter is a hypothetical trained convolution kernel. Because of the convolution operation, this filter will look for dogs everywhere in the image. Contrast this with how a VNN would look for a dog. It would need to learn a dog filter at each spatial location.

In reality, there's no single dog filter inside a trained CNN. The processing of identifying dogs happens iteratively through the layers. First lower level "objects" are identified. Then these objects are combmined into higher level "objects". The smallest concepts are edges and rudimentary shapes. These only require local information to detect. As an image goes through the CNN, these low level concepts are combined to identify mouths, ears, tails, and so on and these features are combined to identify faces and bodies. In the final layer, the CNN decides if the image contains a dog. In this way the CNN composes large concepts from small concepts. This hierarchy of identified features is a consequence of the convolution operation. It is another advantage of the CNN model over VNN.

GPUs and Parallelization

CNNs are theoretically enticing but didn't work well in practice for a long time. As we will see in this section, this is because they are computationally expensive and the only (reasonable) way to train them is to exploit parallelization.

Being able to compute the big-O time of a neural network is generally a useful skill to have. It makes for a great interview question.

In this section we'll look at both the memory and time costs of CNNs. To simplify out analysis we'll only consider the convolution operations, ignoring any fully connected layers, ReLU, maxpooling, or any other operations. Suppose we have a CNN with L layers. To keep things simple, suppose all hidden layers have the same dimensions $H \times W \; timesk$. The kernel size is $p \times p$ in each layer. This is not realistic but it doesn't need to be if we're making ball-park estimates. Now suppose we're training the CNN using minibatches (section **??**) N samples large.

Memory

Each layer stores $H \times W \times k$ neurons as well as k^2p^2 weights (one kernel has p^2 weights and we need k^2 kernels to map every input channel to every output channel). There are L layers, making up $LHWp$ neurons Lk^2p^2 weights. However, each sample in the minibatch must store a set of neurons. In total, we have $NLHWp$ neurons in a minibatch.

$$M = \mathcal{O}(NLHWp + Lk^2p^2) \tag{4.18}$$

Some typical values are N = 10, L = 100, H = W = 100, p = 100, k = 3. Plugging these in gives,

$$M = 3 \times 10^7 + 9 \times 10^6 \tag{4.19}$$

It turns out the neurons dominate the weights in memory consumption. This makes sense since CNNs need to store neurons for each location on the hidden feature grid. Compare this to a VNN, which can be seen as a 1×1 grid, and stores only neurons at a single location.

Compute

To figure out the computation requirements, consider applying a single convolution kernel to neurons at one location. This requires p^2 multiplications. Now slide the kernel across the image. We have HWp^2 computations. Now repeat this process for all k^2 filters, giving us k^2HWp^2 computations. Repeating for all L layers and N samples in the minibatch gives us LNk^2HWp^2 computations for a CNN forward pass.

$$C = \mathcal{O}(NLHWk^2p^2) \tag{4.20}$$

Again, let's plug in typical values N = 10, L = 100, H = W = 100, p = 100, k = 3.

$$C = 9 \times 10^{11} \approx 10^{12} \tag{4.21}$$

Is 10^{12} fast or slow? Let's compare with VNN. Recall a VNN is a CNN with $H = 1$ and $W = 1$. Furthermore, the kernel size must be $p = 1$

because the grid is only 1×1. This leads us to the computation time for VNNs.

$$C = \mathcal{O}(NLk^2) \tag{4.22}$$

Plugging in typical values gives 10^7. Clearly CNNs are a lot slower than VNNs. CNNs would be prohibitive slow, except we can parallelize the forward pass. What variable can we parallelize over? Notice that the convolution kernel calculation at each location does not depend on the calculation at any other location. We can parallelize over H and W using a graphics processing unit (GPU). Removing H and W from the equation, we have $C = \mathcal{O}(NLk^2p^2) \approx 10^8$, a much more reasonable number compared to VNNs.

Summary of this Section

1. CNNs are state-of-the-art algorithms for computer vision. They improve over VNNs in two ways. First, by translational invariance and second by composing smaller concepts into larger concepts.
2. Memory costs $\mathcal{O}(NLHWp + Lk^2p^2)$. Neurons demoniate weights in cost.
3. Run time is $\mathcal{O}(NLHWk^2p^2)$ which is extremely slow but this is overcome by parallelizing computation across H and W using a GPU.

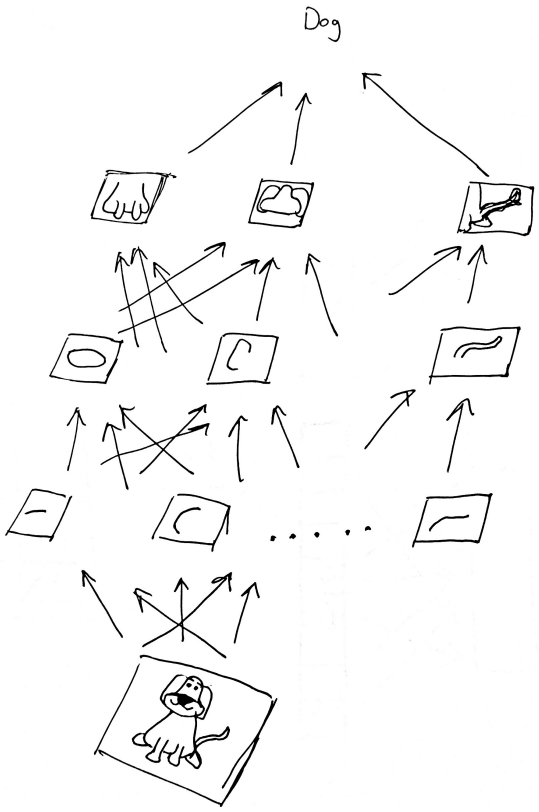

Figure 4.9: CNN learns a hierarchy of features, starting with edges and eventually higher level objects.

4.5 ReLU, Dropout, Batchnorm

In this section we'll look at some of the tools that take NNs from a theoretical curiosity to state-of-the-art machines. The concepts we cover are indispensable to any deep learning practitioner.

ReLU

So far we've put off prescribing a concrete activation function f. There are 3 common functions. They are Sigmoid, Tanh and ReLU. Their equations are plots are below.

$$f(x) = \left\{ \begin{array}{ll} \frac{1}{1+e^{-x}}, & \text{Sigmoid} \\ \frac{e^x - e^{-x}}{e^x + e^{-x}}, & \text{Tanh} \\ \max(0, x), & \text{ReLU} \end{array} \right\} \tag{4.23}$$

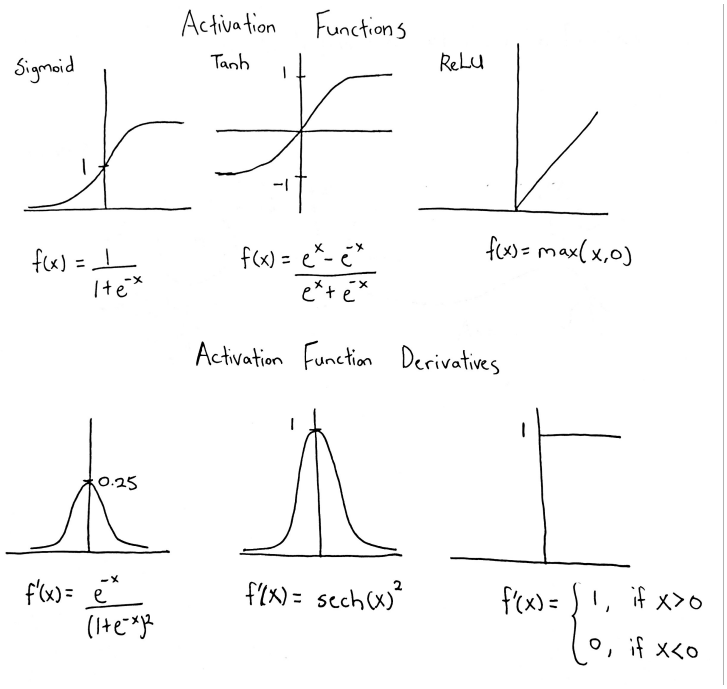

Figure 4.10: Activation functions.

Sigmoid and Tanh are biologically inspired activation functions. In biological neural circuits, neurons fire only if the input is sufficiently large. However, this firing saturates at some point. This phenomenon motivates the s-shape of Sigmoid and Tanh activations.

ReLU is different from the other two activations because it's exactly zero on the negative domain and unbounded on the positive domain (does not saturate). This would correspond to a cell that's either on or off (never a little bit on) and can flood the input of the next neuron. While the first property seems desirable the second does not. Nevertheless, ReLU has become the ubiquitous activation function for neural networks (Tanh is also used from time-to-time and Sigmoid is almost never used). Why is ReLU so effective?

Faster Training

The first advantage of ReLU is it has a large gradient in most parts of the input domain. The facilitates faster training and helps prevent the model from "getting stuck" (in local minimas). As seen from figure **??** the gradient of Sigmoid and Tanh saturate as x becomes very negative or very positive. In contrast, the gradient of ReLU is always 1 as long as $x > 0$. Saturating gradients are bad because they they're hard to "back out of" once you've entered the part of the domain with low gradient. ReLU causes sharp slopes in the landscape of the parameter vector space whereas Sigmoid and Tanh cause flat areas. Recalling our rolling ball analogy for gradient descent, it is much harder for a ball to get to the bottom if it's rolling in a landscape with many flat areas.

Fundamental Function

There's a second way to explain the effectiveness of ReLU. ReLU is in some sense more fundamental than Sigmoid and Tanh. We can compose a function that looks like Sigmoid or Tanh using two ReLU functions. But we cannot construct a function that looks like ReLU using any number of Sigmoid or Tanh functions. It's easy to see this since ReLU is unbounded while Sigmoid and Tanh are bounded.

When we use ReLU functions to approximate Tanh and Sigmoid, the resulting function has some nice properties. Unlike the real Tanh, the function created by ReLUs do not have saturating gradient problems. The gradient is either exactly 0 or 1. It seems like it would be impossible for us to "get unstuck" if we got into an area with 0 gradient. However, we are unlikely to get there in the first place.

Dropout

Dropout is a way to *regularization* neural networks. It is easily integrated into training and usually provides better generalization to the test set. Interviewers want candidates to have an intuitive understanding of how dropout regularizes, how it can be implemented and what some disadvantages may be.

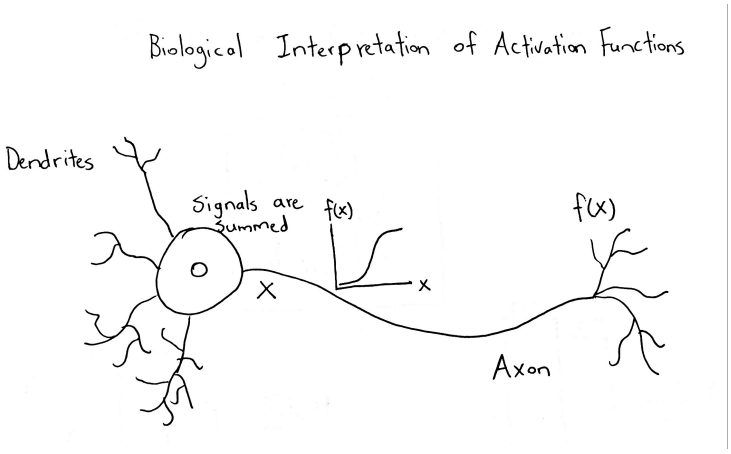

Figure 4.11: Biological motivation for the Sigmoid activation.

Dropout is conceptually simple. During every iteration of gradient descent, randomly turn off/kill p percent of the hidden neurons at every layer by setting their values to 0 (no gradients would flow through them either). In the next iteration, we turn all the neurons back on, kill another randomly chosen p percent and repeat. At test time, turn on all the neurons for prediction.

Why Dropout Works

It may seem like we're making it harder for the network to converge but most of the time, the model will still train. What's more, dropout reduces overfitting in the trained model. A typical value for p is 0.5. How can we intuitively explain why dropout reduces overfitting? By killing off some neurons randomly, we cause the neural network to learn *redundancy*. The network cannot rely on any subset of neurons to be on at all times and therefore must make the same prediction multiple ways. Redundancy leads to robustness of predictions.

This redundancy can be seen as a form of model ensembling, a concept we will discuss in-depth in section **??**. Ensembling is a general technique in machine learning where you combine the prediction of multiple models. The motivation being aggregating models will reinforce their individual strengths and help compensate for individual weaknesses. For each iteration of gradient descent, a subset of neurons get to train on the minibatch data. This neural *subnetwork* can be seen as one model in an ensemble. At test time all neurons are active, meaning all subnetworks are operating in tandem. The full network is the ensemble of all subnetworks. See diagram **??**.

`\margintoc`.

Insert subnetwork full network diagram

Test Time Adjustment

There's a mathematical problem with turning all the neurons on at test time. If we enable all the neurons at test time, neurons at the next layer will receive a much higher input than they're used to from

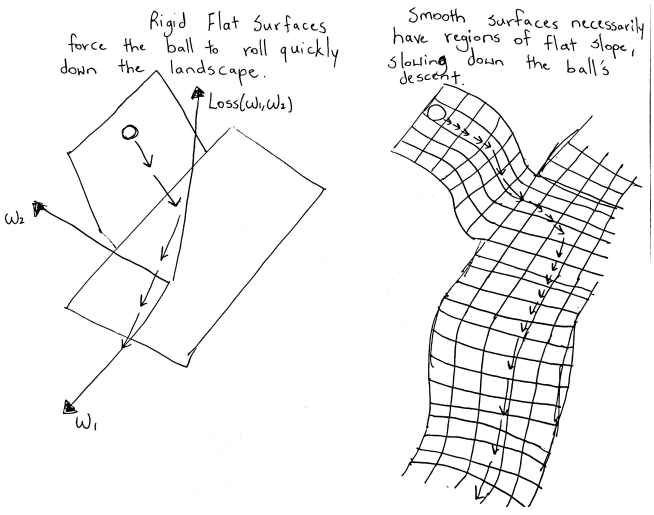

Figure 4.12: ReLU networks have a solution space assembled from hyperplanes. Sigmoid networks have a solution space with tapered surfaces. A ball rolls down hyperplanes quickly but gets stuck in tapered surfaces.

training when p% of the neurons are turned off and the input is $1 - p$ times the input at test. To compensate for this effect, we need to scale down the activations of layers by $\frac{1}{1-p}$. In other words, we multiply the activations by $1 - p$.

Most deep learning libraries use this train-time adjustment instead of test-time adjustment.

For edge-device applications, it's undesirable to introduce an extra computation at test time. There's a way to implement dropout at *training time*. The intuition is to *get the neurons used to the large inputs they will see at test time* during training. We do this by scaling up activations by $\frac{1}{1-p}$ at train time. At test time, the inputs will be on average larger by $\frac{1}{1-p}$ and no extra computation is required.

Disadvantages

One disadvantage of dropout is it slows down convergence during training. Dropout causes the neural network to learn redundancy. It's only natural that it'll take the model to learn redundantly than non-redundantly! Another way to see this is only $1 - p$% of neurons get trained in each iteration. In practice, 2× slow down in training with dropout versus without is typical. However, this is often worth the performance gain from reduced overfitting.

Batchnorm

Batch normalization or batchnorm for short, is another layer we add into neural networks to reduce overfitting and accelerate training.

The rough working principle is to normalize the activation units of each layer to zero mean and unit variance before passing it to the next layer. This is similar to **feature whitening**. The difference between batchnorm and feature whitening is we perform batchnorm at every layer and batchnorm is differentiable.

During training, each neuron is normalized according to its mean and variance calculating over the minibatch. At test time, we may want to predict on a single sample instead of a batch. Therefore, what we do is keep track of the average minibatch mean and variance and use these statistics to normalize the neuron during test time. The batchnorm equations are shown below.

$$\text{BN}(x) = \eta \frac{x - \mu_x}{\sigma_x} + \gamma \tag{4.24}$$

$$\mu_x = \sum_{i=1}^{N} x_i \tag{4.25}$$

$$\sigma_x = \sqrt{\sum_{i=1}^{N} (x_i - \mu_x)^2} \tag{4.26}$$

μ_x and σ_x are the minibatch mean and standard deviations for neuron x. The batchnorm layer first normalizes the neuron by subtracting μ_x

and dividing σ_x. Then two *learned* parameters η and γ are used to rescale and rebias the neuron (more on this in the next paragraphs).

Batchnorm Improves NN Training

Batch norm has multiple positive effects on training neural networks. They are,

- reduces overfitting
- accelerates training by allowing the network to converge in fewer iterations of gradient descent
- makes the model less sensitive to hyperparameter choices (hence requires less tuning) including weight initialization and learning rate

Why batchnorm causes all these good effects is an open research problem. However, we can provide some intuitive answers. In section **??** we will see that small weights provide a regularizing effect (*e.g.* L2 and L1 regularization), reducing overfitting. Small weights mean small changes to the output given changes to the input $\Delta y = \Delta x \Delta w$. Small activations provide a similar effect by causing the learned weights to be small and making the output less sensitive to input. Batchnorm accelerates training by improving the **condition number** of the Hessian of the parameters. Roughly this means if each neuron is normalized, then the learned parameters can be of similar magnitude. In the rolling ball picture of gradient descent, this means the landscape is about equal steepness in every direction, there are no sharp valleys for the ball to oscillate about. See figure **??**. With unnormalized neurons, some weights may need to be very large to capture the patterns in the data. Whether these weights can be successfully learned will be sensitive to how we initialize them. With neuron normalization, we don't need to worry about weight initialization as much.

η and γ Explained

Why do we normalize by μ_x and σ_x then rescale and rebias by η and γ? Is the net effect zero? No, there are two differences between η, γ and μ_x, σ_x. First, η, γ are learned parameters meaning the network learns what the best normalizatoin is based on the data. Second, η, γ are *global* mean and variances, meaning every minibatch is adjusted by them. μ_x, σ_x are *local* mean and variances, meaning each minibatch is adjusted by a unique μ_x and σ_x. The net effect is to make the distribution of neuron x (more) consistent over each batch. However, it's not consistently 0 mean and 1 variance. Instead, it's consistently η mean and γ^2 variance. This is better than not using batchnorm because the distribution of neuron x would be inconsistent in each batch. Why not make the distribution of the neuron 0 mean and 1 variance? It simply turns out this hurts the performance of the neural network because we've crippled its capacity too much.

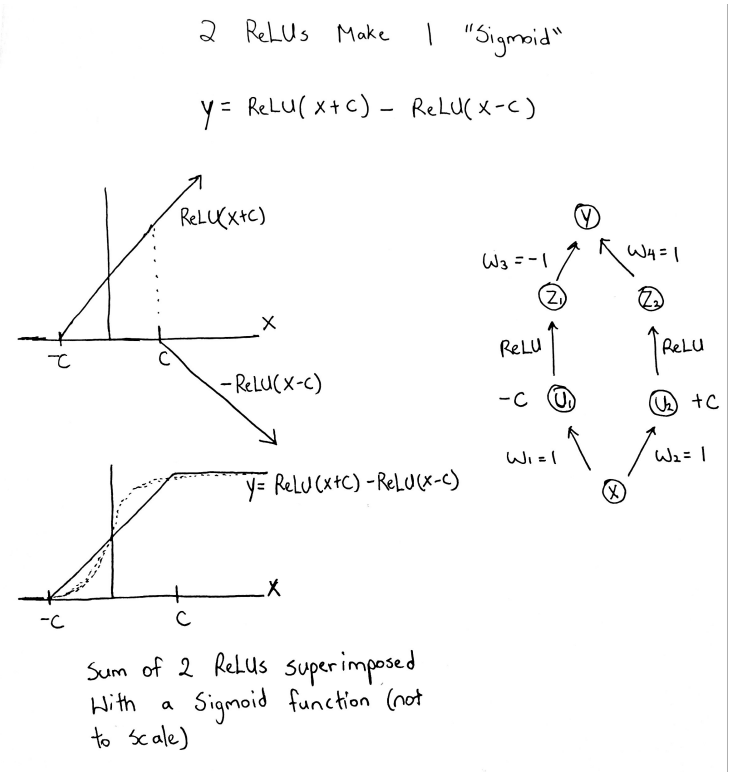

Figure 4.13: By playing "ReLU lego" we can construct the other two activation functions.

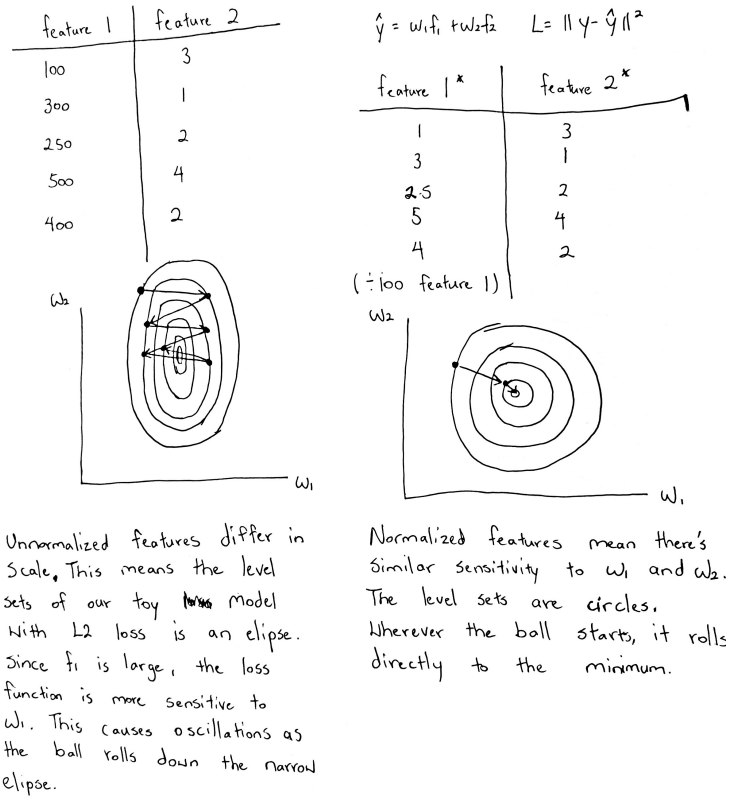

Figure 4.14: Normalizing features improves the condition number of the loss surface. This co responds to circular rather than elliptical level sets. The ball rolls directly to the bottom in the circular case. The ball oscillates in the elliptical case.

4.6 Resnet

The last trick I want to discuss deserves a section of its own. The residual connections network or Resnet for short, is a variant of CNN and one of the cornerstones of deep learning networks. It won the ILSVRC 2015 competition in 3 tasks: image classification, localization, and detection and is still considered today to be a state-of-the-art network. The architecture of Resnet is designed based on a trick called *residual connections*.

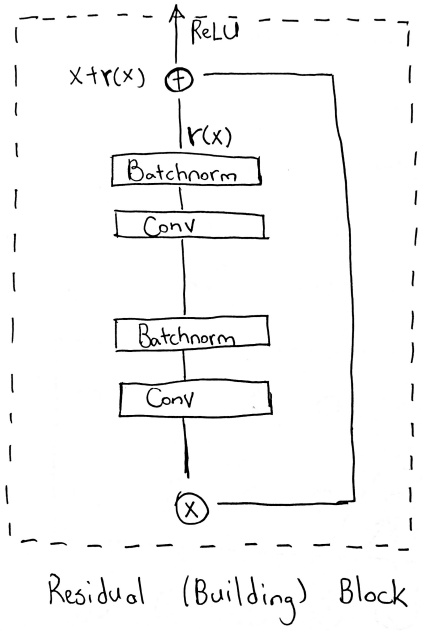

Figure 4.15: Residual connection block. A resnet is composed for many of these blocks stacked in series.

Resnet is popular not only because of its performance but also conceptual simplicity. Residual connections are a very natural and simple idea. The idea of residual connections is to add an information highway so information can go quicker from the input to the loss and vice versa. Why do we need an information highway? Overtime as people experimented with different CNN architectures, they found two things. One, neural networks with more layers (deeper) tend to generalize better onto the test set. Two, deeper neural networks are harder to train.

Specifically, deep CNNs are hard to train due to the vanishing gradient problem. Recall that backpropagation is analogous to a game of telephone. A training message needs to pass from the last person (the loss) to the first person (weights in earlier layers). The problem is if there are enough intermediate persons in the telephone line, then the initial message gets completely lost before it reaches the first persons. We can observe the vanishing gradient problem in the maths of backpropogation. Recall the recursive chain rule formula for backprop.

$$\frac{dL}{dx} = \frac{dL}{dz_N}\frac{dz_N}{dz_{N-1}}\cdots\frac{dz_1}{dx} \tag{4.27}$$

The problem is when there are too many layers, many derivatives in the product are less than 1. As you multiple all the derivatives

together, $\frac{dL}{dx}$ becomes nearly 0, meaning weights in the first layers barely get updated, and the neural network gets stuck during training. *e.g.* if the derivatives were each 0.9 then the net derivative goes 0.9^l (exponentially) to 0 with the number of layers l.

Resnet tackles this problem head-on by making the gradients closer to 1. The key idea is that if you have a lot of layers, then each layer only needs to make a small change to the hidden neurons of the previous layer. In other words, each layer only needs to predict the change in the hidden units, the residual. Mathematically, this means $z = f(x) + x$. The gradient is then $f'(x) + 1$. Resnet is a CNN that uses residual connections at each layer. It can be thought of as a highway network because the input x can pass straight from the input to the output without change if $f(x) = 0$ at every layer. Likewise, a backwards message can easily get from the last layer to the first because the gradient is exactly 1 if $f'(x) = 0$.

Resnet is shown in figure 4.16. It is a stack of residual "blocks". Each block contains convolutions, batchnorm, ReLU, and exploits the residual connection.

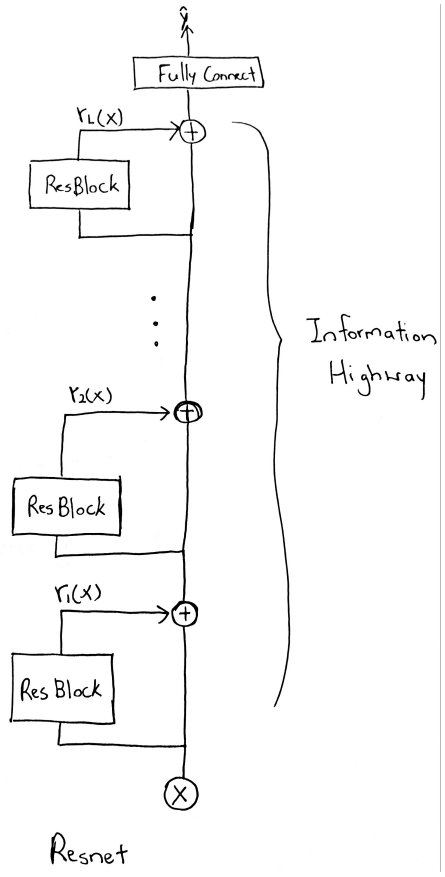

Figure 4.16: Resnet, a CNN with an information highway to prevent vanishing gradients.

4.7 Recurrent Neural Networks

Some recent works have shown that CNNs outperform RNNs on short sequences. Transformer networks tend to outperform RNNs on longer sequences. Therefore, RNNs are stilled used for sequential data but not as ubiquitously as they once were.

Recurrent neural networks or RNNs for short, exploit the notion of a memory in order to process sequential data. A memory allows the neural network to look backwards along the sequence of inputs and find

long range patterns within the data. RNNs are used widely in natural language processing and time series forecasting. RNNs are a staple of deep learning and I won't go into a full detail on their operations here. If you are unfamiliar with RNNs I recommend going through one of the fantastic resources linked at the end of the chapter.

What this chapter will focus on is understanding why recent advances have made RNNs work so well. RNNs were invented in the 1980s but did not work until LSTMs were introduced in the late 1990s. This was not simply a matter of insufficient computation. As we will see, RNNs suffer from the problem of vanishing gradients, the same problem that plague deep CNNs. We will examine how LSTMs solve this problem.

The math of backpropogation is very tedious for RNNs. It is conceptually similar to backpropogation for VNNs but is complicated by the memory/hidden state. There's no reason to go through it unless you're feeling particularly adventurous.

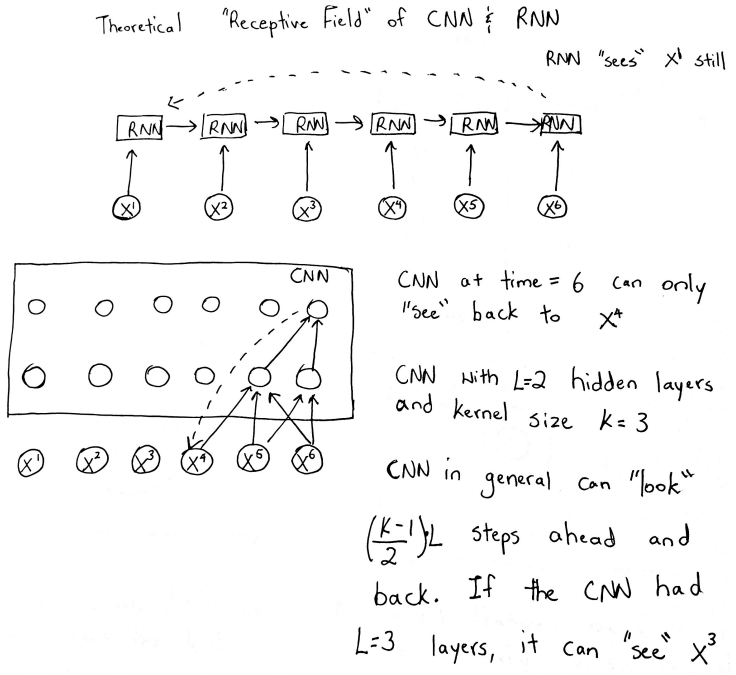

Figure 4.17: A CNN has a limited receptive field. A RNN has an infinite receptive field in theory, if it can retain information from many time steps ago.

There are two popular variants of RNNs: LSTM and GRU. I recommend familiarizing both for ML interviews. Another important concept in RNNs is bidirectionality. It turns out that machines understand language better if they read sentence backwards as well as forwards. A bidirectional RNN is simply two RNNs, one reading the input in forward mode and one reading the input in reverse mode, stacked on top of each other. See details in the additional resources.

The Vanishing Gradient Problem

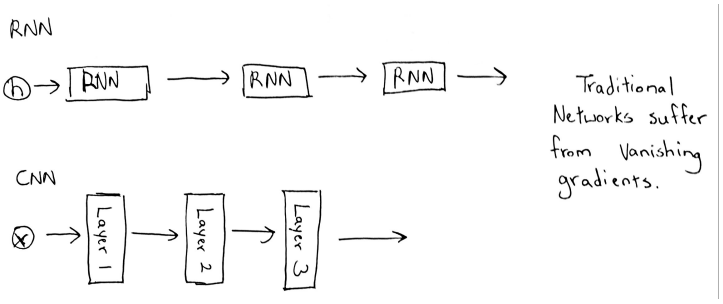

Figure 4.18: Both RNNs and CNNs suffer from vanishing gradients without an information highway.

We've seen how *deep* CNNs suffer from the vanishing gradients problems. This problem is alleviated by introducing residual connections that make up an *information highway*. The problem is very similar for RNNs. The original RNNs introduced in the 1980s suffer from vanishing gradients when they process long sequences. In figure **??** we've turned a CNN 90 degrees onto its side. Here we see how going deeper for a CNN is analogous to processing a longer input sequence. There are subtle differences between the vanishing gradient problems faced by CNNs and RNNs but at a high level it's the same. In both cases, the backpropogation of gradients must go through a long sequence of computations. Each computation diminishes the message that's passed back. Intuitively the RNN "can't remember what it did in the past". The RNN has a hard time making long range correlations between input tokens.

Below is the update equations for the hidden state of a RNN. We also consider the gradient going back along the hidden state. The magnitude of the gradient goes as W^t with the number of time steps. We have the same problem with vanishing gradients in CNNs where if $W < 1$ then the gradient diminishes exponentially quickly going back time steps.

$$h_t = \sigma(Wh_{t-1} + Ux + b) \tag{4.28}$$

$$\frac{dh_t}{dh_0} = \frac{dh_t}{dh_{t-1}}\frac{dh_{t-1}}{dh_{t-2}}\cdots\frac{dh_1}{dh_0} = \prod_{\tau=1}^{t} W\sigma'(Wh_\tau) = W^t \prod_{\tau=1}^{t} \sigma'(Wh_\tau) \tag{4.29}$$

LSTMs solve the vanishing gradient problem for RNNs much in the same way residual connections solve the vanishing gradient problem for CNNs. The idea is to introduce an information highway that runs along the direction of information loss. For CNNs this is the deepness or layer direction. For RNNs, this the sequence direction. For LSTMs, the cell state is the information "highway". The math is slightly different than residual networks but the effect is information is allowed to propagate efficiently back through the highway and the gradient is kept close to 1.

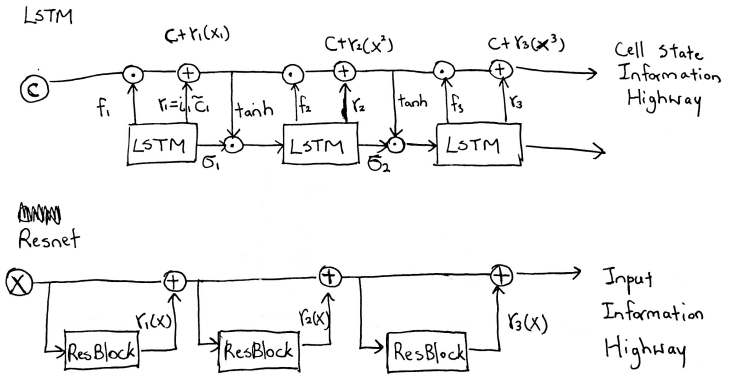

Figure 4.19: LSTM solves the vanishing gradient problem similar to CNN by introducing an information highway.

Mathematically the LSTM solves the vanishing gradient problem by using the cell state as an information highway. v_t here is the input to the

forget gate f_t. We see the gradient $\frac{dC_t}{dC_0}$ does not contain an exponential w^t term, meaning the gradient does not die off as quickly.

$$C_t = f_t C_{t-1} \tag{4.30}$$

$$f_t = \sigma(W v_t) \tag{4.31}$$

$$\frac{dC_t}{dC_0} = \frac{dC_t}{dC_{t-1}} \frac{dC_{t-1}}{dC_{t-2}} \cdots \frac{dC_1}{dC_0} = \prod_{\tau=1}^{t} \sigma(W v_\tau) \tag{4.32}$$

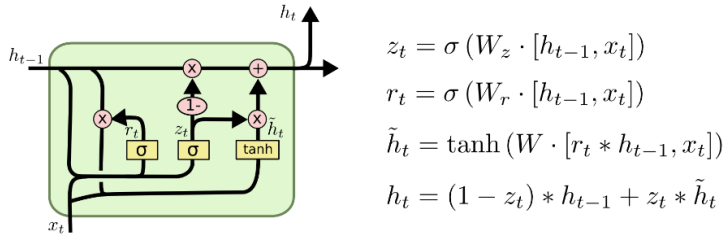

Figure 4.20: A LSTM cell.

I've included an explanation for why LSTMs and GRUs eventually face the vanishing gradient problem in the Appendix.

The holy grail of RNNs is to be able to find patterns within sequential data no matter how long-ranged they are. We're still quite far from this goal. LSTMs, GRUs can process sequences up to 30 units long (with tricks we can extend this number). These models start experiencing vanishing gradient problems on sequences longer than that.

4.8 Stochastic Gradient Descent

So far we've been talking about training neural networks using gradient descent but in reality no one uses GD. Neural networks are trained using *stochastic* gradient descent (SGD). Put simply, SGD is a variant of GD where you train using a minibatch of data instead of the entire training set in each iteration. The SGD algorithm is shown below. It's the same as GD but the loss at each iteration is computed using a small batch of training data rather than the the full set.

```
======================== SGD Algorithm ========================

for t iterations over X:

    shuffled_X = shuffle(X)
    while(shuffled_X not depleted):

        X_batch = extract_next_batch(shuffled_X, batch_size)
        \\ loss is computed with a batch of data instead of X
        current_loss = loss_fnc(X_batch, y, w)
        current_grad = gradient_fnc(current_loss)
        w = w - alpha * current_grad
```

SGD is better than GD

Why is SGD preferred over GD for training neural networks? There are at least 3 reasons. First, the entire dataset X often doesn't fit into memory. Second, empirically SGD trains faster than GD in the sense of fewer passes over the data. This is a somewhat intuitive because SGD allows the model to "make progress before seeing the entire dataset". The most surprising answer is practitioners have found that SGD almost always finds better solutions than GD in practice. This is counter-intutive because it seems like the randomness in SGD should not be good for training. If a particular batch contains only outliers to the data then it seems like the gradient step will hurt learning. It turns out these worst-case scenarios do not hurt training so much in practice. In fact, some researchers have proposed it is actually the noise caused by randomness which makes SGD better than GD. The exact reason SGD finds better solutions than GD is still an open research problem.

The hypothesis is noise from randomness helps the neural network get out of undesirable local minimas.

The Best Batch Size

What batch size should be used for the SGD algorithm? There are three factors to consider: performance, memory, and training speed. The best batch size to use is application dependent. However, most of the time batch size of 16, 32 or 64 give the highest performance. In some applications, it's not possible to use these batch sizes due to memory constraints. Batch sizes of 8, 4 or even 2 are used. In some applications, practitioners can take advantage of *parallelization* and distribute the computation for a single batch amongst multiple GPUs. Batch sizes of 128, 256, 512 or beyond are used. Practitioners almost always use a *batch size that is a power of 2*. This is to better utilize GPU memory, which is made with a power of 2 number of bytes.

4.9 ADAM

ADAM and vanilla SGD (with momentum) are used in 95% of deep learning problems. Which one works better depends on the particular problem. Most people start with ADAM because it requires less fine-tuning. Given enough patience SGD with momentum can often be fine-tuned to be better than ADAM.

For a full survey of SGD algorithms, see this blog post.

ADAM is the most popular variant of SGD. It is widely used because it requires little learning rate tuning and is robust to different parameter initializations. In this section I provide an overview of the ADAM algorithm and intuitive explanation for why it works well. ADAM combines two modifications to the SGD algorithm: momentum and adaptive learning rate.

Momentum

To understand ADAM, we need to first understand the idea of momentum. Recall we can view gradient descent as a ball rolling down a bumpy landscape. Intuitively a ball that's rolling should continue rolling in the same direction unless a force acts upon it. When this

happens, the new direction is the summation of the old direction and the velocity gained in the new direction. This is the idea of momentum. In the same way, a gradient descent algorithm should move in the same *historic* gradient direction until a new gradient is computed at which point the total gradient is calculate as the summation of the old and new. By giving the SGD algorithm momentum, we make it behave more like a physical ball. This helps the algorithm avoid getting stuck in valleys (shown below).

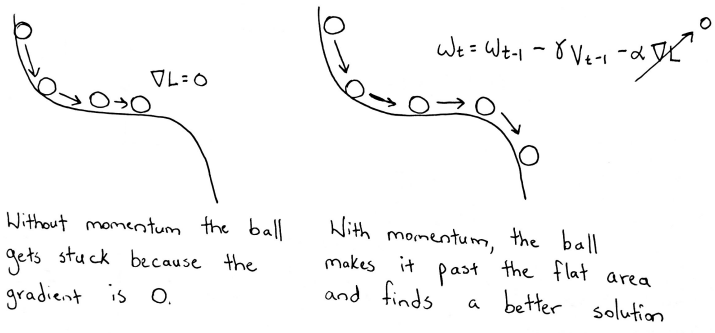

Figure 4.21: Momentum allows the ball to roll past the flat area and achieve a lower loss. Without momentum, the ball stops as soon as the gradient becomes zero.

SGD with mommentum has one additional equation. v_t is the velocity of the "ball" at iteration t. It is updated using the historic velocity v_{t-1} and the new velocity $\nabla L(w_t)$. γ is a *decay factor* (typically 0.9 or 0.99) which describes how long the historic momentum lingers. As we've said w_t can be interpreted as the position of the ball.

$$v_t = \gamma v_{t-1} + \alpha \nabla L(w_t) \tag{4.33}$$

$$w_t = w_{t-1} - v_t \tag{4.34}$$

Adaptive Learning Rate

The idea of adaptive learning rate is to use a personalized learning rate for each parameter rather than the same learning rate α for every parameter. How should we personalize the learning rates? The idea is to choose them automatically (*i.e.* adapt them) to each parameter. We will do so using the following policy: parameters that haven't been updated much historically should have a larger learning rate. One way to justify this is to have "better utilization" of all the weights. We don't want any weights sitting around doing nothing because they haven't been trained. These weights get a larger learning rate. More specifically, we'll adapt the learning rate at each iteration of SGD.

We will define the "history of how much a parameter has changed" by the magnitude of its gradients in the past h_t^j.

$$h_t^j = \beta h_{t-1}^j + (1-\beta)(g_t^j)^2 \tag{4.35}$$

t indexes the iteration number and j indexes the parameter. β is a decay factor that describes how long the contribution of an old gradient

should linger. One idea is to set the learning rate to $\alpha_j = \frac{1}{h_t^j}$ because parameters with small historical change should be updated more.

ADAM

ADAM combines momentum with adaptive learning rate. The ADAM algorithm equations without bells and whistle is shown below.

$$v_t^j = \gamma v_{t-1}^j + \alpha(\nabla L(w_t))^j \tag{4.36}$$
$$h_t^j = \beta h_{t-1}^j + (1-\beta)((\nabla L(w_t))^j)^2 \tag{4.37}$$
$$w_t^j = w_{t-1}^j - \frac{1}{h_t^j} v_t^j \tag{4.38}$$

We use the superscript j to indicate the gradient and learning rate are calculated separately for each parameter. The first line calculates the total velocity. The second line calculates the new gradient history. The third line combines the two for the parameter update.

`\margintoc.`

Should I add the precise ADAM equations? The notation for $(\nabla L(w_t))^j$ is pretty bad.

4.10 MLE and MAP

`\margintoc.`

I'm not sure how well this section fits here.

Let's revisit our question of where loss functions come from. It turns out there's often a way to ground loss functions in probability distributions of *Bayesian inference*. Doing so has two advantages. First, it allows us to interpret the loss functions we choose in a more mathematical way. Second, grounding loss functions in probability may help with choosing loss functions for novel applications.

Bayesian inference is a complex topic. It's unreasonable to do a full treatment of it here. Just keep in mind MLE and MAP are ideas from Bayesian inference. The language we use like *likelihood* is the language of Bayesian inference.

MLE and MAP are two methodologies from Bayesian inference that help us choose loss functions. We can think of MLE and MAP as black boxs which we feed (probability) assumptions into. Given the assumptions, the box churns out a loss function. We can also "run the black box in reverse". Given a loss function, the black box outputs some assumptions we "had to have made", allowing us to interpret the "latent maths" behind our decision to choosing a loss function. Common interview questions ask you to "interpret the L2 and L1 losses" under the MLE framework. This means we should run our black box in reverse and figure out what mathematical assumptions were made to decide to use the L2 or L1 loss.

The rest of this section opens up the black box. We will discuss what assumptions are inputs and how the assumptions are converted into a loss function.

Maximum Likelihood Estimation

Maximum likelihood estimation or MLE for short, takes a single probability assumption as input. Specifically, MLE requires us to specify what the probability distribution of the ground truth y is, given the prediction $\hat{y}$. This is called the *likelihood probability*. Here are a few equivalent ways to write the likelihood probability.

$$P(y|\hat{y}) = P(y|f_w(X)) = P(y|X, w) \tag{4.39}$$

The second equation assumes we're using a model f (*e.g.* neural network) to predict given the data X. w are the parameters/weights of the model. The third equation makes the model f implicit by conditioning only on X and w. At the end of the day, all three probabilities mean the same thing as long as the f we're talk about is a neural network, linear regression, etc.

Given we've decided the likelihood probability, MLE spits out a loss function. It does so in these steps.

\margintoc.

1. **Choose** a likelihood probability distribution $P(y|f_w(X))$.
2. **State** the goal of training model $f_w(X)$ is to maximize $P(y|f_w(X))$.
3. **Convert** the maximization of $P(y|f_w(X))$ into the minimization of the negative logarithm $NL = -\log P(y|f_w(X))$. NL is the MLE loss function.

Step 2. makes intuitive sense because the goal of training should be to change the model to make predictions that are likely given the ground truth. $P(y|f_w(X))$ swaps the point of reference related to the noise in the data, but we're effectively encouraging the same thing.

May need a paragraph describing what the likelihood function is. We hint that it's related to the noise in the data.

There's nothing special going on in step 3. Minimizing the negative log of a function is mathematically equivalent to maximizing the original function. Most of the time it's easier computationally to minimize the negative log rather than maximize the original function.

These seem like innocuous steps, but can actually be quite powerful in practice. Let's show this with an example.

How can we interpret the L2 loss within in the MLE framework? We will show the L2 loss comes from a Gaussian distribution assumption on the likelihood probability. For concreteness, we can think of $f_w(X)$ as a neural network model. Let's say the likelihood probability is,

$$P(y|f_w(X)) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{\|y - f_w(X)\|^2}{2\sigma^2}\right) \tag{4.40}$$

Why might $P(y|\hat{y})$ be Gaussian? We can justify this by saying the ground truth y is not exactly our prediction $\hat{y}$ because when the data (X, y) was generated or recorded, there was some noise in y. This noise

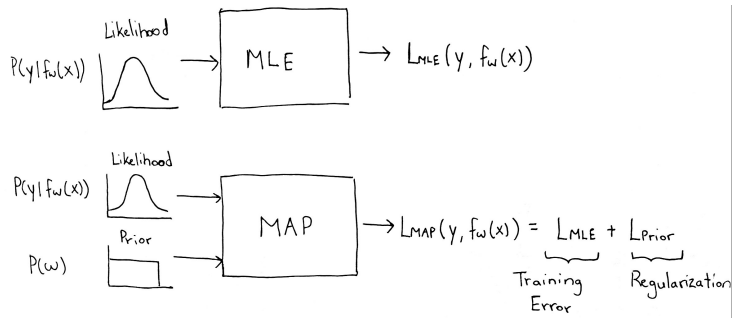

Figure 4.22: MLE and MAP can be seen as machines that create loss functions from probability assumptions or inputs.

follows a Gaussian distribution. That's step 1. Step 2 is to state the maximization of the likelihood.

$$\max_w \frac{1}{\sqrt{2\pi\sigma^2}} \exp(\frac{-\|y - f_w(X)\|^2}{2\sigma^2}) \tag{4.41}$$

Step 3 is to take the negative log and minimize it.

$$\min_w -\log\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp(\frac{-\|y - f_w(X)\|^2}{2\sigma^2})\right] \tag{4.42}$$

The log of a product is the sum of logs $\log(AB) = \log A + \log B$.

$$\min_w \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\|y - f_w(X)\|^2 \tag{4.43}$$

We can drop $\frac{1}{2}\log(2\pi\sigma^2)$ from the equation because changing w will not affect $\frac{1}{2}\log(2\pi\sigma^2)$.

$$\min_w \frac{1}{2\sigma^2}\|y - f_w(X)\|^2 \tag{4.44}$$

As promised, the MLE procedure produces the L2 loss (up to a scaling factor σ^2). To summarize, we've shown that the L2 loss corresponds to a Gaussian likelihood probability assumption. This means the noise in the data labels y follows a Gaussian distribution.

Maximum a Posterior

Maximum a posterior (MAP) is similar to MLE except MAP allows us to inject prior knowledge about "how the model should be" into the loss function. The prior knowledge is another input of the black box. It is called the *prior probability distribution*.

1. **Choose** a likelihood probability distribution $P(y|f_w(X))$.
2. **Choose** a prior probability distribution $P(w)$.
3. **State** the goal of training model $f_w(X)$ is to maximize $P(y|f_w(X))P(w)$.
4. **Convert** the maximization of $P(y|f_w(X))P(w)$ into the minimization of the negative logarithm $NL = -\log P(y|f_w(X)) - \log P(w)$. NL is the MAP loss function.

A couple of things have changed. In step 2, we define a prior distribution. This distribution states "what the weights ought to be like **before the model sees any data**". In step 3, we maximize $P(y|f_w(X))P(w)$ instead of $P(y|f_w(X))$ only. Where does this equation come from?

Understanding this questions requires us to take a quick peek through the doors of Bayesian inference. In MAP we care about maximizing another probability called the *A Posterior* distribution. Roughly it states "the probability of the weights **after the model sees data**". Its definition is,

$$P(w|y, \hat{y}) = P(w|y, f_w(X)) = P(w|X, y) \tag{4.45}$$

Prior Distribution	Regularization
$\frac{1}{\sqrt{2\pi(1/\lambda)}} e^{-\|w\|^2/2\lambda}$ (Gaussian)	$\frac{\lambda}{2}\|w\|^2$ (L2 Reg.)
$\frac{\gamma}{2} e^{-\gamma\|\theta\|}$ (Laplace)	$\gamma\|\theta\|$ (L1 Reg.)

What we care about is maximizing $P(w|y, f_w(X))$ but we can convert this into $P(y|f_w(X))P(w)$ using Baye's Rule.

$$P(w|y,X) = \frac{\overbrace{P(y|X,w)}^{\text{"likelihood"}}\overbrace{P(w)}^{\text{"prior"}}}{\underbrace{P(y)}_{\text{discardable}}} \tag{4.46}$$

$P(y)$ is discardable because $P(y)$ doesn't change as we change w.

$$\max_w P(w|y,X) = \max_w \frac{\overbrace{P(y|X,w)}^{\text{"likelihood"}}\overbrace{P(w)}^{\text{"prior"}}}{\underbrace{P(y)}_{\text{discardable}}} = \max_w P(y|X,w)P(w) \tag{4.47}$$

What type of prior knowledge would we want to inject into the loss? We may want to inject the knowledge that weights should be small. One way to translate this condition into a prior probability is to say the prior distribution of weights is a Gaussian with zero mean. As we will now show, the MAP framework converts a Gaussian prior distribution into L2 regularization.

Let's skip some steps because MAP is nearly the same as MLE. In step 4, we will **convert** the probabilities into a loss by minimizing $NL = -\log P(y|f_w(X)) - \log P(w)$. Suppose $P(w)$ is Gaussian with variance $1/\lambda$.

$$P(w) = \frac{1}{\sqrt{2\pi(1/\lambda)}} \exp(\frac{\|w\|^2}{2(1/\lambda)}) \tag{4.48}$$

$$\min_w P(w) = \frac{1}{2}\log(2\pi(1/\lambda)) + \frac{1}{2(1/\lambda)}\|w\|^2 \tag{4.49}$$

$$\min_w P(w) = \frac{\lambda}{2}\|w\|^2 \tag{4.50}$$

Bringing this back into the entire context of step 4,

$$\min_w \quad \underbrace{-\log P(y|f_w(X))}_{\text{MLE loss}} \quad + \quad \underbrace{\frac{\lambda}{2}\|w\|^2}_{\text{additional term from MAP}} \tag{4.51}$$

We see that a Gaussian prior distribution leads to L2 regularization in the loss. This is intuitive because both Gaussian priors and L2

regularization encourage the weights to be small. We see that in fact, they are equivalent. Table **??** shows some other common equivalences between prior distributions and regularization functions.

How do we determine whether to apply the MLE methodology or the MAP methodology? This is a design choice of the practitioner. MAP allows us to inject some prior knowledge we have about the model (weights) into the loss function while MLE stays agnostic to prior knowledge. If you, as the practitioner believe there's useful prior knowledge to inject, then choose MAP, otherwise choose MLE.

How to Make Machine Learning Models Work Better

5

As the title suggests, this chapter is concerned with making the machine learning models we've learned about work better. We'll start by discussing the bias-variance trade-off. This motivates ideas practitioners use such as ensembling and regularization which increase the performance of models. We then look at how to make models work when the labels are heavily biased.

5.1 Bias-Variance Trade-off

Usually it's unclear what the interviewer is looking for exactly when they say "explain the bias-variance trade-off". However, I've found that most interviewers will be happy with an answer that explains *why* it has to be a *trade-off* and what are some *ways* to "tune the knobs" of the *trade-off*. The bias-variance relationship is closely related to the underfitting and overfitting.

Interviewers love to ask you to "explain the bias-variance trade-off". Most people who've done some machine learning have an intuitive definition of bias and variance. It might sound something like this:

- "As you **increase the capacity** of a model, you **decrease the bias** while increasing the variance. As you **decrease the capacity** of the model, you **increase the bias** while decreasing the variance."
- "When you **regularize the model**, you **increase the bias** while reducing the variance."

The first point to understand is what *bias are we increasing? The bias of what?* Answer — we are increasing the model's bias to making predictions that are not represensitive of the training set. If the model is not representative of the training set, then what is it representative of? Answer — the practitioner's (that's us) prior assumptions. *e.g.* by using L2 regularization and decreasing the weights of the model, we prevent the model from fully learning the patterns inside the training data. We inject the prior assumption that *the model must be simple*.

Why might we want to increase the bias of the model? Because the *variance* may decrease and the model may improve overall. As the title of this section suggests, there is a trade-off between bias and variance. Both bias and variance are bad for the model but usually we can't decrease both of them to 0. Equation 5.1 summarizes the bias-variance trade-off.

This is one of those math equations that say 1 = 1 but turn out to be very useful!

$$L_{\text{test}} = \overbrace{\underbrace{L_{\text{train}}}_{\text{bias}}}^{\text{training loss}} + \overbrace{\underbrace{(L_{\text{test}} - L_{\text{train}})}_{\text{variance}}}^{\text{generalization error between test and train}} \tag{5.1}$$

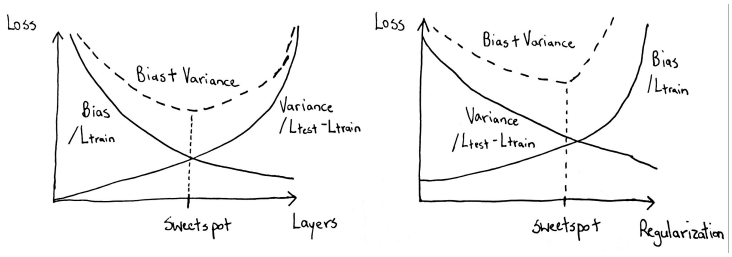

Figure 5.1: The bias-variance trade-off curve. The best model is the model with the lowest sum of bias and variance. This occurs at the sweetspot.

When we talk about "model performance" we usually talk about accuracy or another metric on the *test dataset*. Let's for a second take the metric we care about to be the test loss L_{test}. L_{test} can be described as the sum of the training loss and *generalization error*. Generalization error is how much worse our model performs on the test set versus training set (remember high loss is bad!). Training loss is the bias of our model *i.e.* tendency of our model to not be representative of the training set. The higher the training loss, the less representative. *Generalization error* is the variance of our model (how much does test loss vary from train loss). Unfortunately when we decrease bias/training loss, variance/generalization error usually increases (although not by the same amount!). Likewise when we regularize the model by

increasing bias, we hope variance decreases by more than that amount so there is a net decrease to L_{test}. We can't increase bias indefinitely because at some point variance won't decrease enough to compensate. Therefore, we usually look for a **sweet spot** in the bias-variance **trade-off**.

5.2 Overfitting and Underfitting

Bias-variance is closely related to the ideas of overfitting-underfitting.

Let's look at two similar quotes about bias-variance and overfitting.

- When we **decrease** bias (training loss), variance (generalization error) usually **increases**.
- When we increase the capacity of the model *too much*, it overfits. Bias (training loss) decreases but the variance (generalization error) increases more.

Overfitting and underfitting are situations where we've turned the knobs of bias-variance too far. The capacity of the model is *too* high, the bias we reduce is not worth the variance we acquire. We had a notion of a sweet spot for bias-variance (the minimum sum of bias and variance). If we pass the sweet-spot in the direction of decreasing bias, the model is said to overfit. If we pass the sweet-spot in the direction of increasing bias, the model is said to underfit. This is illustrated in figure **??**.

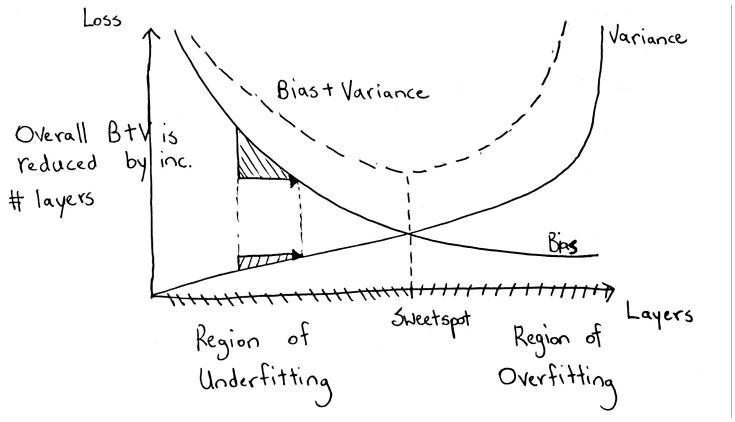

Figure 5.2: Regions of overfitting and underfitting shown on the bias-variance curve. In the underfitting region, it's worth it to increase the number of layers because the decrease in bias is more than the increases in variance.

When the model is overfitting, we should get closer to the bias-variance **sweet spot** by reducing the capacity of the model. *e.g.* use L2 or L1 regularization.

When the model is underfitting, we should get closer to the bias-variance **sweet spot** by increasing the capacity of the model. *e.g.* add layers to a neural network or increase the number of hidden units.

Why does overfitting occur when we increase the capacity of the model? There are two reasons. Overfitting means worsening performance on the test set. This could happen because there are quarks/patterns in the training set that don't exist in the test set. A high capacity model is more capable of picking up these quarks and making

predictions based on them, leading to bad test set predictions. Another reason is high capacity models tend to "memorize" the training set instead of learning the *patterns* of the training set. There is a subtle difference.

For example, take the equation of the line $y = 5x$. Suppose a model learns to predict y from x given the points $(1, 5), (2, 10)$. One model may get 0 training error by memorizing to always output $y = 10$ when $x = 2$ and $y = 5$ when $x = 1$. Something bizarre may happen when the model predicts for $x = 3$. On the other hand, a linear regression model will capture the equation $y = 5x$ exactly. It will predict $y = 15$ when $x = 3$.

Exceptions to the Word "Trade-off"

Is it always a trade-off between bias-variance? Is our model doomed to do no better than the **sweet spot** of the bias-variance curve?

Not exactly. Bias-variance trade-offs are unique for each model. Different models have difference trade-off curves (figure figure **??**). We can achieve a better trade-off by navigating a particular bias-variance curve. We can also achieve a better trade-off by changing models and "jumping" bias-variance curves. We can "jump the curve" in two ways.

1. Choosing a different model class (*e.g.* neural network versus random forest).
2. Ensembling models (of the same or different classes).

For one reason or another, some models do a better job of learning certain datasets than other models. *e.g.* CNNs work better than VNNs for images. The bias-variance **sweet spot** is better for CNNs than VNNs. Model ensembling and how it improves model performance is discussed in section **??**.

5.3 The Deep Learning Recipe

This section contains more practical advise on training neural networks. I present a basic *deep learning recipe*. This recipe is the result of conventional wisdom on how to approach NN training to get the best bias-variance trade-off. As we go through this section, a useful analogy to keep in mind is seeing overfitting and underfitting as two sides of a see-saw.

The Deep Learning Recipe

1. Begin by underfitting. A simple model (fewer layers, small hidden sizes) will not achieve the best possible performance but it should train quickly. This allows you to find any bugs you have in the code.

2. Add capacity by increasing the number of layers, hidden size, etc. until the model overfits. Do this until training loss is significantly lower than the validation loss.
3. Reduce the complexity of the model using regularization (*e.g.* L2, L1, dropout, batchnorm). Do this until you've maximized validation loss.

The motivation for step 1 is to get the model working quickly. A smaller model trains faster. The motivation for step 2 is to use a overly large model which captures all the patterns in the training data. Step 3 ensures some of the spurious patterns captured by the model are pruned off.

Most of the time *a well trained neural network will have higher validation loss than training loss*. This is an expected result, it simply means variance is non-zero. Variance is typically not zero at the sweet-spot of the bias-variance curve. If it is zero, your neural network most likely has underfitted. An intuitive way to understand this is it's better to learn all the important patterns plus some spurious ones in the training set rather than not learn all the important patterns.

5.4 Ensembling

`\margintoc`.

What's a reasonable value to state here? It won't go up 5 - 10% if performance is at 98% already.

Ensembling is a technique that always boosts performance by 5 – 10%, even after all other machine learning "tricks" have been employed. The intuition is as follows. Suppose you want to build a medical organization of doctors to classify different diseases. It's a much better idea to compose your organization of many specialists (*e.g.* cancer, heart disease, liver disease, etc.) than to find a single generalist who knows a bit about all the diseases. By analogy, ensembling is a machine learning method whereby you train several different models and aggregate their predictions. The hope is that each model learns to be a specialist on a particular part of the data. By aggregating their predictions, you reinforce individual model strengths and compensate for weaknesses. *The problem with ensembling is training and testing multiple models takes longer*. This is the main reason model ensembling is sometimes not used for real-time applications.

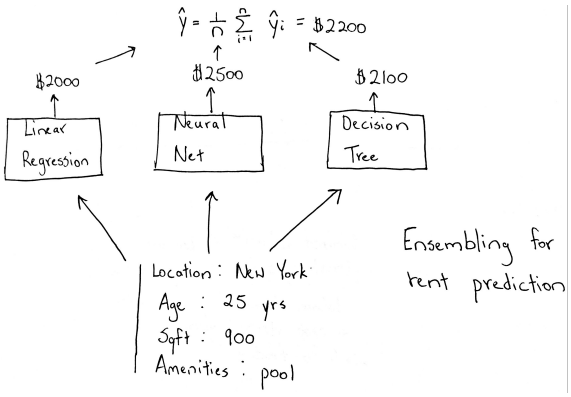

Figure 5.3: Test time ensembling. Each model is trained independently. Their predictions are aggregated (averaging depicted here).

Sometimes we want to ensemble models that are very different. For example, Kaggle winners usually ensemble decision trees with neural networks with SVMs in their solution. Each model is trained **independently** and the predictions are combined. The surprising thing is even if you ensemble a weak model with a strong model, the combined performance can be better than the strong model's performance alone.

Other times we can ensemble similar models by building the ensembling process into the training algorithm. We've already seen an example with dropout. Some neurons are killed in each iteration of gradient descent. The remaining neurons form a sub network, which can conceptually be considered another model different from the full network. In this version of ensembling, the different models are trained **in tandem** rather than independently.

Test-time Ensembling

More sophisticated aggregation methods exist such as training a *meta/manager model* to weigh the predictions of all the other models.

Test-time ensembling is the simplest form of assembling. N models are trained in isolation and their predictions are aggregated at test-time. In classification tasks, predictions are aggregated by taking the most common prediction. In regression tasks, the average prediction is used. For example, it's common to ensemble neural networks, SVMs, decision trees, and logistic regression for Kaggle contests. A second type of ensembling makes all the models aware of each other in the sense that the training procedure of one model depends on models. We will see examples in the next section.

5.5 Bagging and Boosting

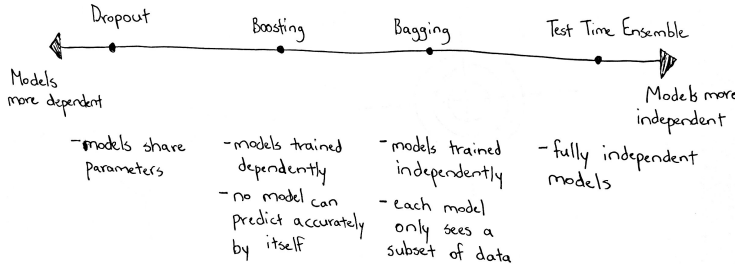

Figure 5.4: A spectrum of ensembling algorithms. Some algorithms like dropout and boosting impose strong dependency between each of the models.

Boosting and Bagging are popular ensembling methods. Boosted and Bagged *decision trees* are used widely in real world applications. In this section I'll go over the high-level ideas of boosting and bagging and contrast the two. Because boosting is slightly more mathematically involved, I've saved a detail description of the algorithm for the Appendix.

Take a look at figure **??**. The dropout algorithm has the strongest dependencies between model. Each model is a sub network of the full neural network and hence share parameters with each other. Test-time ensembling has no dependencies between models. Models are trained in isolation and predictions are aggregated at test-time. Each

model makes a prediction independent of the other models. Bagging and boosting methods lie between dropout and test-time ensembling because in bagging and boosting, each model is trained dependent on other model but models do not share parameters.

Bagging

\margintoc.

Consider adding a figure to explain bagging

Recall the high-level idea of ensembling is to train N specialized models rather than one general model. In Bagging, we impose each model to be a specialist in a subset of the training data. The algorithm is simple. We create N data subsets by uniformly sampling from the training set *with replacement* (which just means a sample can be in a data subset twice). N different models are assigned one data subset each. All models are trained in isolation and predictions are aggregated at test-time.

```
==================== Bagging Algorithm ====================
\\ N: # models
\\ n: size of train dataset
\\ m: size of data subsets

data_subsets = train_set.sample_uniformly_with_replace(N, m)

for k from 1 to N:
    new_model = train(model_type, data_subsets[k])
    all_models.add(new_model)

\\ Test time
predict_and_aggregate(all_models, X_test)
\\ e.g. aggregate by averaging predictions
```

How is Bagging different from test-time ensembling? Although each model is trained in isolation, the data set it trains on is as ubset of the full dataset. Therefore, each model is specialized on a fraction of the data and no model understands the entire data distribution. In this way, the models are dependent on each other. A single model could make a prediction by itself, but it will likely be an inaccurate compared to the prediction of the ensemble.

Random Forest and Bias-Variance Tradeoff

Random Forest is a Bagging algorithm where each model is a decision tree. Because we want each model to be an expert on a data subset, *each decision trees has high depth (they're tall trees) and hence high capacity to learn the patterns of the subset fully*.

Recall that models with high capacity are low bias, high variance models and tend to overfit. Overfitting seems bad but the philosophy of Random Forest is to exploit this fact. By combining low bias, high variance models, we decrease the variance of the ensembled model.

The ensembled model has *low bias and low variance model*. The intuitive way to understand this is each tree is overfitting on a different part of the input domain. Therefore, they will compensate for each other's weaknesses when they're aggregated.

A last note about Random Forests. The implementation you will find in most libraries also randomly samples a subset of features in addition to subset of samples for each tree. In other words, each model is an expert of certain features as well as certain samples. No model understands all features, nor all samples. This implementation of Random Forest has been found to work better, likely because it gives us a better tradeoff between bias and variance.

Boosting

`\margintoc.`

Consider adding a figure to explain boosting

In Boosting, each model becomes a specialist on the mistakes of its peers. The models live in a symbiotic society where the total prediction is the sum of the predictions of the members and new members try to correct the mistakes of the existing members. The society starts with zero members. Models are added to the society one at a time, until all N models have been added. Suppose $k < N$ models have been added so far. The $k + 1^{\text{th}}$ model is trained to correct the mistakes of the previous k models (on the train set). By, this we mean the labels for the $k + 1^{\text{th}}$ model is $r_{k+1} = y - \hat{y}_k$. r_{k+1} is the mistake or often also called the **residual**. The $k + 1^{\text{th}}$ model is trained on the data (X, r_{k+1}). When we're training the first model, the "mistakes" are simply defined as the labels of the training set $r_1 = y$. The first model has a regular training objective, it attempts to fit the data (X, y).

```
==================== Boosting Algorithm ====================
\\ N: # models
\\ X, y: training data and labels

r = y

for k from 1 to N:
    \\ add a new model to the ensemble
    new_model = train(model_type, (X, r))
    all_models.add(new_model)

    \\ calculate the new residuals
    predictions = predict(all_models, X)
    r = compute_residuals(predictions, y)

\\ Test time
predict(all_models, X_test)
```

For implementation details of Boosting algorithms, specifically what *mistakes* are and how *find_mistakes* is implemented, see the Appendix.

Boosting has a much stronger dependency between models than Bagging does. It doesn't make sense to predict only with the k^{th} model (added to the society) because that model would output r_k given X. The only exception may be the first model ($k = 1$) which is actually

trained on data $(X, r_1 = y)$ instead of (X, r_k). The models trained by boosting need to predict as an ensemble.

Boosted Decision Trees and Bias-Variance

As the name suggests, Boosted decision trees (BDT) is an ensembled model where each model is a decision tree. Unlike Random Forest, in BDT each tree is short and low capacity. The reason is as follows. If the trees were deep, then the first tree added to the ensemble would achieve very good predictions by itself. It would make few mistakes. In other words the residuals would be small. This ruins the point of having a symbiotic society where each member meaningfully corrects the mistakes of the other members. Therefore, we use short trees that tend to make mistakes and leave large residuals for the next tree to correct. But of course, the residuals are still decreasing over time.

Because the trees are short, they tend to underfit. They are *high bias, low variance* models. This is the opposite idea from Random Forests (Bagged trees) where we aggregated *low bias, high variance* models. Underfitting seems bad but BDT's exploit this fact. By combining high bias, low variance models, we decrease the bias of the ensembled model. The ensembled model has both *low bias and low variance*. The intuitive way to understand this is each tree is decreasing the residual of the ensemble. The residual is related to the training loss which is how we defined bias in section **??**.

At the time of writing, a variant of BDT called **xgboost** is the state-of-the-art on *tabular data*. Xgboost consistently outperforms neural networks, SVMs, logistic regression, and other popular models, making it a popular model for Kaggle contests. It is also fast to train and do inference.

Summary of Bagging and Boosting

We summarize our discussion of bagging and boosting by comparing and contrasting them.

1. Bagging trains models that are experts of a fraction of the training data. Boosting trains models that are experts of mistakes of other models in the ensemble.
2. Bagging assigns data *subsets* to models and trains them independently while Boosting builds an ensemble one model at a time, each model learning from the residuals of the previous models.
3. Bagging combines high capacity models to reduce their overfitting. Boosting combines low capacity models to reduce their underfitting. In other words, Bagging ensembles low bias, high variance models while Boosting ensembles high bias, low variance models. The result of both Bagging and Boosting is a combined low bias, low variance model.

5.6 Regularization

Take-aways from this section: (1) how L2 and L1 regularization work, (2) why L1 and L2 reg. reduce overfitting, (3) L1 causes parameter sparsity where some weights are set to 0, (4) L1 reg is useful for *feature selection*.

Regularization is a general term for techniques we can use to reduce overfitting. Recall to reduce overfitting means to reduce to reduce *variance*. This is often at the cost of *increased bias*. Some techniques like dropout are specific to particular models (neural networks) while others like weight decay (a general name for L2 and L1 regularization) are much more general and apply to multiple model types.

This section focuses on L2 and L1 regularization. They are in principle very simple to implement. We can simply add $\frac{\lambda}{2}\|w\|$ or $\gamma|w|$ to the loss function. We have already seen a probabilistic interpretation of L2 and L1. Under the MAP framework, they are Gaussian and Laplace prior distributions on the weights of the model. In other words, they reflect a prior belief that weights of the model should be small. In this section, we'll try to understand exactly why small weights are beneficial. More specifically, how small weights reduce overfitting. Lastly, we will also look at a special property of L1 regularization to cause sparsity in the weights.

L2 Regularization

Recall the linear regression model. L2 regularization adds an additional $\frac{1}{2}\lambda\|\mathbf{w}\|^2$ term to the loss function.

$$L(w) = \overbrace{\frac{1}{2}\|Xw - y\|^2}^{\text{training loss}} + \overbrace{\frac{1}{2}\lambda\|w\|^2}^{\text{L2 regularization loss}} \tag{5.2}$$

"L2" refers to the norm we're penalizing the weights with. L2 regularization penalizes the Euclidean length of a weight vector while L1 regularization penalizes the Manhattan length. Both cause the model to learn smaller weights during training.

λ is a scaling factor that controls the strength of the L2 regularization loss relative to the training loss. As we increase λ, the model will learn smaller and smaller weights. Increasing λ reduces the variance of the model and increases the bias. If we continue increasing λ, at some point the model will be too constrained and underfit. If λ is too small, the model may remain overfitted.

L2 and L1 regularization are sometimes called "weight decay" because their gradients always point in a direction of decreasing weight magnitude. The gradient for L2 reg. is shown below. We see a dependence on w_t in the reg. gradient. This means the larger that $\|w_t\|$ is, the faster it will decay.

$$\begin{aligned} w_{t+1} &= w_t - \nabla L(w_t) \\ &= w_t - \underbrace{X^\intercal(Xw_t - y)}_{\text{loss gradient}} - \underbrace{\lambda w_t}_{\text{L2 reg gradient}} \end{aligned} \tag{5.3}$$

Smaller Weights Reduce Overfitting

We've seen one reason why smaller weights reduce overfitting. The predictions of a linear model is $\hat{y} = xw$. Given a small change to the input Δx, the output will change by $\Delta\hat{y} = w\Delta x$. Imagine what would happen if the model saw a sample x_{test} which was different from all training samples. More specifically, suppose x_{test} was Δx different from the *most similar* training sample. This difference causes a change in prediction equal to $\Delta\hat{y} = w\Delta x$. If w is small, then the change to the input will only cause a small change to output. On the other hand, if w is big, then a small change to the input will cause a drastic change to the output. This increases the chance the prediction $\hat{y}_{\text{test}}$ is inaccurate, decreasing test-time performance.

The problem is amplified for neural networks which have multiple layers of cascading difference-propagation. $\Delta z_1 = w_1\Delta x, \Delta z_2 = w_2\Delta z_1, \Delta z_3 = w_3\Delta z_2, \ldots$ If all the w's are large, then $\hat{y}_{\text{test}}$ can deviate drastically from a small deviation to the input x_{test}.

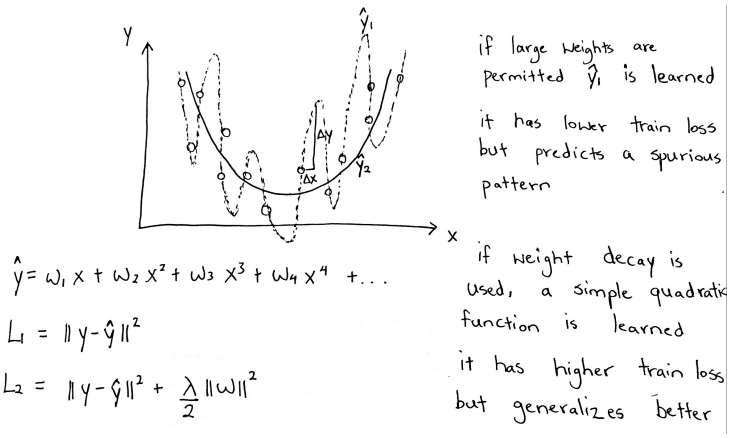

Figure 5.5: Given a high capacity model, in this case a high order polynomial, permitting large weights lead to overfitting on spurious patterns. Regularization prevents a high capacity model from using large weights. This forces the model to learn a much lower order (quadratic) solution.

Figure **??** explains this intuition. If given the chance, a high capacity model will learn a very complex function that fits the data samples almost perfectly. However, this causes problems for interpolation and extrapolation. The model makes bad predictions between training samples. The prediction changes drastically for just a small change in the input. The model also makes bad predictions extrapolating away from the training samples. The predictions change too quickly. Bad interpolation and extrapolation leads to bad test accuracy.

On the other hand, small weights learn a simple function. This leads to smooth interpolation between training samples. This means at test time if we get a sample that is "in between" two training sample, our model will make a reasonable prediction.

In our example, the reason large weights learn erratic functions and small weights learn smooth functions is because for the high capacity model (higher order polynomial) to go through all the training points, it needs to have large coefficients. On the other hand, the low capacity (lower order polynomial) fits the training points well with small weights.

L1 Regularization

L1 regularization adds a $\gamma|w|$ term to the training loss.

$$L(w) = \overbrace{\frac{1}{2}\|Xw - y\|^2}^{\text{training loss function}} + \overbrace{\gamma|w|}^{\text{L1 regularization loss function}} \tag{5.4}$$

γ is similar to λ for L2 regularization. It scales the severity of the L1 loss. A larger γ means the model will learn smaller weights.

L1 regularization has a similar effect on training as L2 regularization. A slight difference is L2 reg. penalizes the squared distance while L1 penalizes the absolute value. To see what this means, let's suppose our model has two weights $w_1 = 10$ and $w_2 = 0$. L2 inflicts a loss of $10^2 + 0 = 100$ while L1 inflicts a loss of $10 + 0 = 10$. If instead $w_1 = w_2 = 5$, L2 loss is $5^2 + 5^2 = 50$ while L1 is $5 + 5 = 10$. L2 reg. reduced drastically when the weights were both medium sized rather than one large and one small. L1 reg. stayed the same. We see that *L2 reg. prefers all the weights to be about the same magnitude* (it hates having any one weight be very large). *L1 reg. doesn't care about relative weight sizes as long as the total size of weights is the same.*

A fascinating property of L1 regularization is it causes some weights to go to exactly 0 during training. In other words, L1 regularization causes the weights to be **sparse**. This is very useful for *feature selection*. In many applications, the data we use is "dirty". It contains many features, some are relevant to the machine learning model and others are not. It's desirable to remove the irrelevant features so they don't cause slowdown during training and inference. One way to do this is using L1 regularization. Suppose we're training a linear regression model with L1 regularization. After training, some weights will be exactly 0. These correspond to the features the model has learned to be irrelevant for predicting the output. These are irrelevant features we can remove.

L1 Regularization and Weight Sparsity

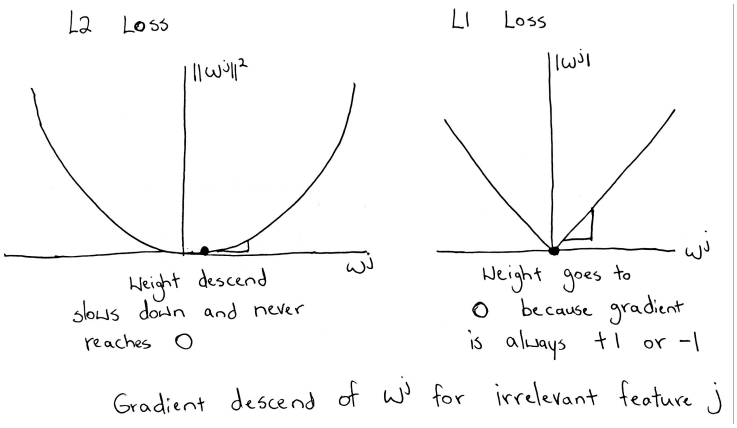

Figure 5.6: L1 loss maintains a large gradient, causing weights of irrelevant features to go exactly to 0.

A common question interviewers ask is why L1 regularization causes weight sparsity. There are at least two ways to understand it.

Considering training a linear regression model with L2 or L1 regularization. Let's consider just a single feature x^i and see how L1 reg. causes sparsity while L2 reg. does not. The regularization gradient is plotted against the weight w^i for both L2 and L1. For L2, the gradient is proportional to w^i. As w^i gets closing to 0 (for example, this may happen if feature i is an irrelevant feature to the prediction) the gradient gets smaller and smaller. w^i never reaches exactly 0. For L1 regularization, the magnitude of the gradient remains constant at γ. It is large enough to carry w^i to exactly 0.

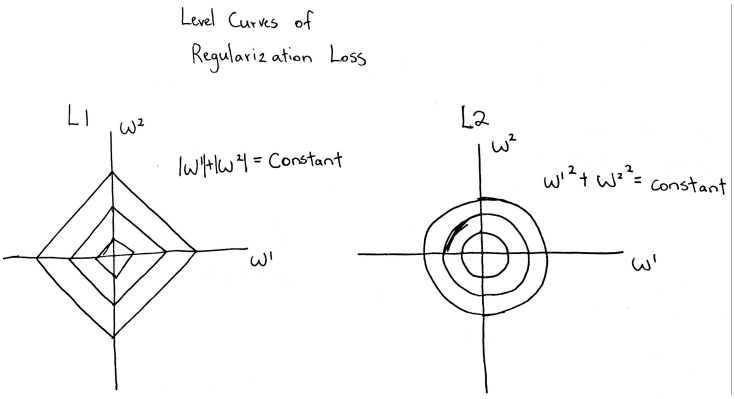

Figure 5.7: Level curves of L1 and L2 loss.

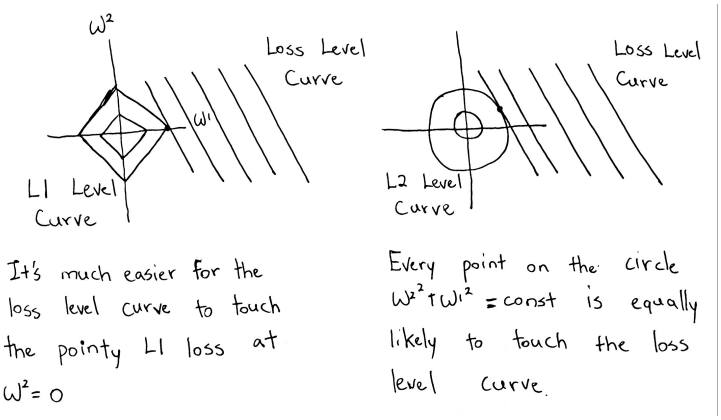

Figure 5.8: The "pointiness" of the L1 level curves means it's more like to meet the loss function's level curves at one of the points (where one of the weights is exactly 0). On the other hand, the level curves of the L2 loss is round which means every point is equally likely to touch the loss function's level curves.

We can also understand weight sparsity using level-curves. Level-curves of a loss function are contours where the loss function is constant. In figure **??**, we've plotted the level-curves of the training and regularization losses. It turns out the combined training and regularization loss function achieves a minimum value when the level curves of the two losses touch. As seen in figure **??**, L1 regularization has "pointy" level curves while L2 regularization has round level curves. It is very likely for the training level-curve to touch the L1 level-curve at one of the points (we could prove this mathematical but it seems intuitive from the figure). These points correspond to solution where one of the weights is exactly 0. On the other hand, every point on the level-curve of the L2 loss is equally likely to touch the training level-curve. Hence, L2 regularization does not cause the model to find solutions with sparse weights.

5.7 Dealing with Class Imbalance

We round off this chapter with a discussion of class imbalance. Class imbalance is one of the most common problems for machine learning system in the real world because class immbalance is a fact of life. There are more regular emails than spam emails, more healthy patients than sick patients, and more cars than bicycles on the road. A problem with perfectly balanced classes is the exception rather than the norm.

Many machine learning models do undesirable things when the training data is imbalanced. Suppose we trained a neural network to do binary classification of spam versus not-spam emails. Also suppose we have a ratio of 99 not-spam emails to every spam email because not-spam emails are much easier to get a hold of. Naively training the neural network will cause it to *always predict "not-spam"*. This is because the neural network can achieve 99% accuracy with this policy! However, this is a useless model. The intent of building a model is to detect spam, a model which gets 99% accuracy but does not detect a single spam email is useless. How can we correct for this class imbalance?

There are two problems here. First, raw accuracy is clearly a bad way to measure the success of the model. We'll need a new way to evaluate the model that captures the importance of making "spam" predictions. Second, we need some way to "encourage" the model to actually predict "spam". Let's address these problems next.

Precision and Recall

In this section, we'll introduce two new metrics that measure the success of the model predicting on imbalanced data. Before we begin, we need to define some things. Let a positive example be the interesting class we're trying to detect (spam) and a negative example be the normal class (not-spam). Let the "dominant" class be the one with more samples in the training set, and the "underrepresented" class the one with fewer samples.

The problem with measuring model success with accuracy is that it does not account for the importance of the positive, spam class. Let's introduce a metric which directly measures the accuracy on the positive class.

$$R = \frac{\text{\# True Positives in Predicted Positives}}{\text{\# True Positives in Dataset}} \tag{5.5}$$

Recall is a number from 0 to 100% and it measures how well our model "recalled" the positive samples in the dataset. In other words, how well did we do to find all the spam emails. A recall of 1 means all positive samples were identified and a recall of 0 means no positive samples were identified. It's insufficient to only measure the success of the model on recall because a model which only predicts "spam" will find all the spam emails (and achieve R=1) while making a lot of

wrong predictions as well. We can use another metric called precision to make sure the model is not making too many false predictions.

$$P = \frac{\text{\# True Positives in Predicted Positives}}{\text{\# Predicted Positives}} \tag{5.6}$$

Precision is also a number between 0 and 100% and it measures the accuracy of the positive predictions. In other words, out of all the spam predictions, how many of them did the model get right?

There is an inherent trade-off between recall and precision. On the one extreme, a model that always predicts "spam" will achieve 100% recall but low precision. On the other hand, a model that never predicts "spam" will achieve 0 recall and undefined precision. Our goal is to build a model that does something between the two extremes. We want to build a model which makes positive and negative predictions intelligently so that both recall and precision are close to 100%.

F1 Score

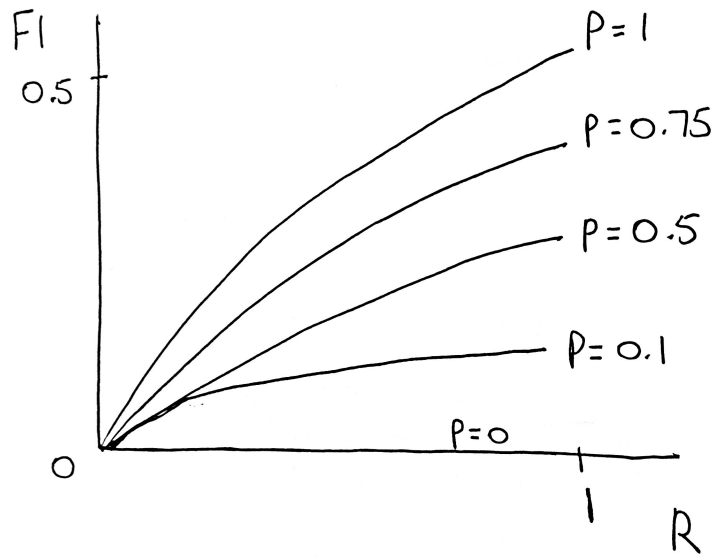

Figure 5.9: F1 score as a function of recall at different precision values.

Since we care about both recall and precision, we can define an aggregate score which captures the overall success of our model. The F1 score combines recall and precision:

$$F1 = \frac{PR}{P + R} \tag{5.7}$$

F1 is a number between 0, the worst possible score, and 0.5, the best possible score. When both precision and recall are 1, F1 is 0.5. If either recall or precision is 0, then F1 is 0. This captures the fact that a successful model should have both high recall and precision. F1 is dragged if either of the two scores are low.

There is a "weighted" F1 score for applications where you care more about recall than precision, or vice versa.

Solving Class Imbalance

Now that we have a good way to measure model success, let's focus on the problem of all "not-spam" predictions. The root of the problem is the model sees a disproportionately high number of "not-spam" examples. The solution is to simply equalize the number of positive and negative at training time. There are several ways to do this which we will discuss. There's a problem with equalizing the training set though. Suppose we had some way to equalize our 99-to-1 training set to a 1-to-1 training set. This means our model will predict "spam" on about 50% of the samples. At test time, only 1% of emails will actually be spam. This means there will be a lot of false-positives (safe emails that we predict as spam). To solve this, we'll need to calibrate the model after training to deal with realistic data distributions.

Our solution consists of two steps. First, we will use some method to equalize the number of positive and negative samples at training time. Next, we will calibrate the model to predict well at test time by reverting the bias we've imposed by balancing the training set. Below are some ways the training set can be balanced.

1. Undersample (*i.e.* discard some samples from) the dominant class until the class ratio is 1-to-1.
2. Oversample (*i.e.* train using the same sample twice in one epoch) the underrepresented class until the class ratio is 1-to-1.
3. Apply data augmentation to the underrepresented class until the class ratio is 1-to-1. For each sample in the underrepresented class, generate multiple samples by adding random transformations to the original. This is similar to oversampling but usually works better because the model sees more diverse data. For example, images can be augmented by adding random rotations, cropping, flipping, and uniform noise.
4. Weigh the loss differently for the dominant and underrepresented classes. Multiply the loss for the dominant class by small number.

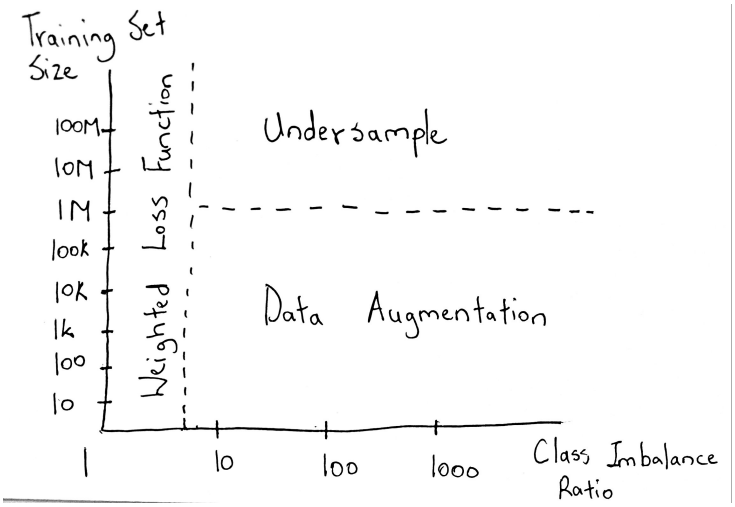

Figure 5.10: Rough guidelines for when to use each class balancing method.

Which method one should pick depends on two things: the imbalance ratio (number of dominant class samples for each underrepresented

class sample) and the total number of training samples. The simplest thing to do is to weigh the loss function. This usually works up to an imbalance ratios of 5. If the data is more imbalanced than that, then undersampling or data augmentation should be applied. Undersampling is a good solution when we have a lot of data and samples from the dominant class likely look very similar so it's okay if we drop some of them. Another reason we may choose undersampling is to speed up training by reducing the number of samples. Data augmentation works better if we don't have the luxury of dropping samples from the training set.

The second step of fixing class imbalance is to calibrate the model to deal with imbalance data at test time. Let's look at a specific example. Suppose we're training a logistic regression model. Usually we predict $\hat{y} = 1$ if $P(Y = 1) > 0.5$. This works during training because we've balanced the dataset to have 50% positive and negative samples. At test time, there are a lot more "not-spam" emails. In other to deal with this, we adjust our definition of a positive prediction. A prediction is positive $\hat{y} = 1$ if $P(Y = 1) > \tau$ for some threshold τ. We'd want τ to be a big number (maybe 0.9?) so that we can raise the bar on what samples are deemed spam. Only the most confident predictions are considered spam.

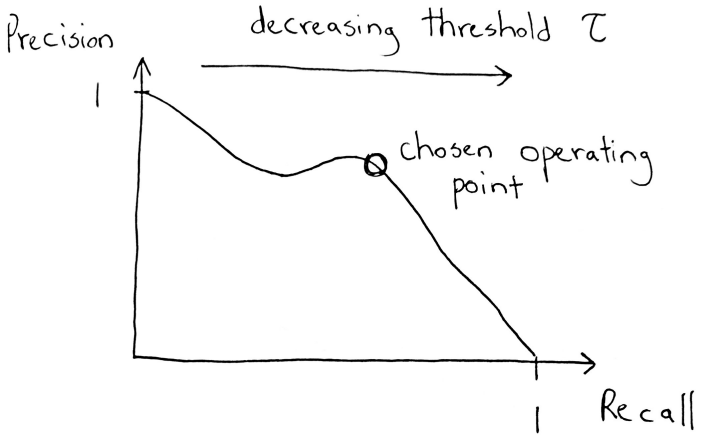

Figure 5.11: Precision-recall curve as we vary the prediction threshold τ. At test time, we choose a value of τ according to the trade-off between recall and precision we would like.

How do we set a precise value for the threshold? Notice that every value of τ we choose will trade-off recall with precision. For example, if τ is 0, then every sample will be predicted as $\hat{y} = 1$ and recall will be 100% but precision is close to 0. On the other hand, if τ is 0.999, then precision may be 100% (we make a single positive prediction and it's correct) but recall will be close to 0. Increasing τ means we make fewer positive predictions. This always decreases recall. However, it's not clear what happens to precision. Most of the time making fewer positive predictions means the ones that are left are more likely to be correct. Other times, we're unlucky and the remaining positive predictions are less accurate. Figure 5.11 shows a typical trade-off between recall and precision as we vary τ.

Ultimately, the "best value" of τ is decided based on how the practitioner wants to use the model. An algorithm which detects cancer

should have high recall so that all at-risk patients are flagged for further diagnosis. Predicting a patient with cancer as "healthy" means the patient will not receive further tests and the cancer will grow undetected. Therefore, the recall should be close to 100%. On the other hand, low precision is tolerable because doing some extra tests is mostly harmless. On the other hand, an email spam detector probably should not have low precision. Users will be annoyed if too many emails are flagged as "spam". It is typical for practitioners to build the recall-precision curve seen in figure 5.11 by computing the recall and precision at different values of τ on the *validation set*. The practitioner then decides where on the curve the model should operate based on the specific needs of the application.

Practical Machine Learning | 6

This chapter deals with the nitty gritty of training machine learning models. Experience is hard to fake but I've tried to distill the basic skills into a concrete set of topics. For most applications the model life cycle is,

1. Collect and label data
2. Choose data splits
3. Train model
4. Intrepret results
5. Debug
6. Tune hyperparameters
7. Collect more data or utilize pretraining to increase model performance

In this chapter, we will look some common practices for choosing data splits, tuning hyperparameters, and figuring out what's going on when things aren't working. We'll also look at pretraining as a way to boost the performance of our model when we have little data.

6.1 Train, Validation, Test

There are 3 data splits in machine learning: train, val and test. Train is of course the dataset we train on. Val is the dataset we use to evaluate your model to tune hyperparameters. Test is the final evaluation, a true test of the model's performance. The reason we need both a val and a test set is because by tuning hyperparameters, we run the risk of overfitting on the val set. This is bad because performance of your model on val is no longer representative of performance during deployment.

The purpose of the test set is to simulate your model running in deployment. Using the test set in any way during training ruins the simulation. In the real world you don't get to travel forward in time and see what users you'll have on your app or what patients will be visiting your hospital. If you use the test set, you run the risk of overfitting on it and hurting the performance of your model in deployment (you've made your test set another validation set).

It's surprising how often machine learning practitioners break the zeroth law. It's very tempting to take a peak at the test set so you can report a higher number to your manager or conference reviewers. When you do so you're only cheating yourself.

Sometimes the simulation is unnecessary. For example, if we plan on monitoring the performance of our model in deployment and we promise to retrain our model once its performance degrades, then we can get away with just train and validation sets. However, research papers do not have the luxury of monitoring performance in the real world. That is why they usually use 3 splits with the test set simulating performance of the model in the wild.

However, if we do choose to use a test set then we must respect the **zeroth law of machine learning**: lock your test set away in a safe and throw away the key. Only retrieve the key once you've finalized your model!

```
// Hyperparameter tuning and reporting test scores

train_set, val_set, test_set = data.split()
best_val_score = 0
best_hyperparams = null

while (tuning_hyperparameters):
    train(model, train_set, hyperparameters)
    model_score = evaluate(model, val_set, hyperparameters)

    if model_score > best_val_score:
        best_val_score = model_score
        best_hyperparams = hyperparameters

// When we're sure we're finished training
test_score = evaluate(model, test_set, best_hyperparams)
```

Total n. Samples	% Train	% Validation	% Test
1,000	50	50	-
5,000	50	25	25
10,000	50	20	20
100,000	70	15	15
1,000,000+	80	10	10

Recommended data splits as a function of total samples. 1,000 samples is not enough to meaningfully differentiate validation and test sets. As the number of total samples increases, we can get away with using a smaller percent (still larger number) of samples for val and test.

6.2 Data Splitting

Below is a rough guide for dividing data into train, validation and test splits. The main idea is we want our *validation and test sets to be large enough to be representative of the samples the model will see in production*. Hence when the dataset is small, we need to use a larger fraction of the dataset for val and test. When the dataset is large, we can get away with using a smaller fraction as val and test because the absolute number of samples is still large. The appropriate number of val and test samples is also application dependent. If the data samples have large variations or the task is difficult, then we would want to use larger val and test sets. On the other hand, if the data is more homogeneous or the task is easy, we may get away with smaller val and test sets.

6.3 k-Fold Cross Validation

Suppose we split the dataset into train/val/test splits. We may think the validation set is being "wasted" because we're tuning hyperparameters but not training on it. k-fold cross validation is a method which allows us to utilizes the data in the val set for training without contaminating the validation score.

k-fold cross validation is seldom used in deep learning do to the computation costs. Nevertheless, interviewers expect you to be able to explain k-fold CV.

The idea of k-fold CV is to divide the combined train and val datasets into k sets. We train the model k times. During each round of training, each of the k sets takes turn being the val set while the other k-1 sets are the train sets. To evaluate the performance of the model, we average its performance over the k rounds of training.

k-fold cross validation is commonly used with smaller models like linear and logistic regression and when there is little data.

`\margintoc`.

Consider adding a figure to show k-fold CV

What k-fold cross validation is doing is it allows us to train on every data sample in the train+val set across the k rounds. However, no sample is ever used for both training and validation in any particular round. Therefore, we get more samples to train on but still properly validate our model. Another benefit is each sample in train+val is evaluated on once. Therefore, we get a more reliable validation than had we evaluate just on the val set. The price we pay is extra computation. We need to train the model k times to evaluate it once. For this reason, k-fold cross validation is not popular in deep learning where models

can take hours or even days to train once. k-fold cross validation is used for simpler models like linear or logistic regression. It excels when we have little data or training is fast.

```
// k-fold cross validation

k_sets = data.split(k)
best_val_score = 0

while (tuning_hyperparameters):

    k_scores = []
    for i from 1 to k:
        train_set = k_sets.get(all sets but set i)
        val_set = k_sets.get(set i)

        train(model, train_set, hyperparameters)
        model_score = evaluate(model, val_set, hyperparameters)

        k_scores.add(model_score)

    avg_score = k_scores.average()

    if avg_score > best_val_score:
        best_val_score = avg_score
```

6.4 Reading Loss Curves

It's common for interviewers to print out loss curves and ask candidates to interpret them.

Reading loss curves is a basic skill for machine learning engineers. Sometimes it's difficult to know exactly what went wrong. Debugging machine learning systems can feel like detective work. Loss curves are like the thermometers of machine learning models. They tell us when something is wrong and help us diagnose the problem. Loss curves track the performance of the model during training. Specifically, they plot the number of training iterations (*e.g.* gradient descent steps) on the x-axis versus the loss on the y-axis. Often times we track both train and val losses. By comparing the two, we can deduce model pathologies such as overfitting.

I can't enumerate all the possible curves that will come up in the real world. I can't even enumerate all curves that will come up during an interview. However, I'll go over some of the most common cases of when things going wrong.

Not Converging

If your model is not converging, the *training loss* curve will not be decreasing. Some common causes include bad choice of hyperparameters and bugs in your code.

Underfitting

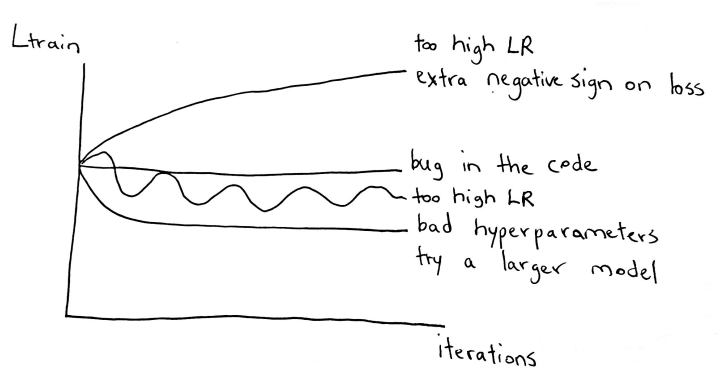

Figure 6.1: Possible training curves when the model is not converging and some potential causes.

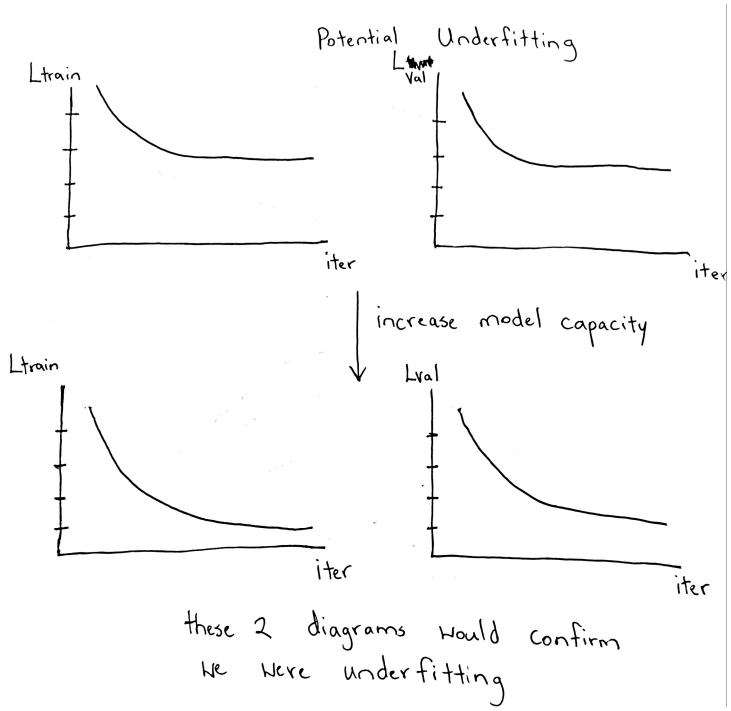

Figure 6.2: Loss curves when the model is underfitting.

Underfitting can be tricky to diagnose. When your model is underfitting, the train and val curves will look very similar. However, similar train and val curves is a necessary but insufficient condition for underfitting. There are datasets where the train and val curves always look similar (*e.g.* samples in train and val sets look very similar). The only way to truly know if the model is overfitting is by increasing the capacity and seeing whether the val loss increases or decreases.

Overfitting

Machine learning models overfit if they have enough capacity and we train them for long enough. The sure-sign of overfitting is decreasing train loss and increasing validation loss. What's happening is the model is starting to memorize quarks of the training set. These patterns don't exist in the val set, causing the model to make erroneous predictions. When your model is overfitting, you should employ **early**

stopping. This means we stop training as soon as the val loss start to increase significantly. Early stopping is usually implemented retrospectively by saving model weights at each epoch and choosing the epoch with lowest val loss. Another way to address overfitting is to use regularization.

Validation Loss is too High/Low

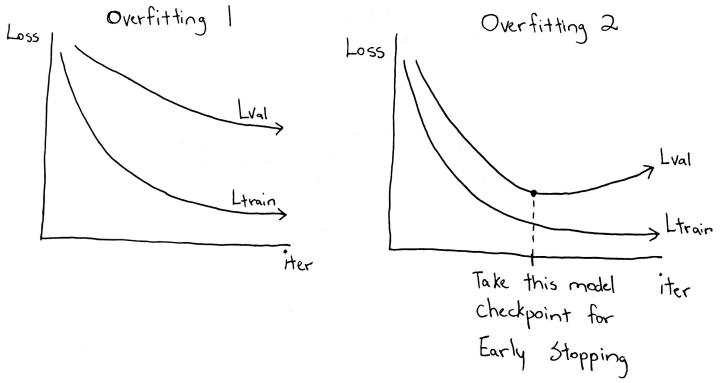

Figure 6.3: Loss curves when the model is overfitting.

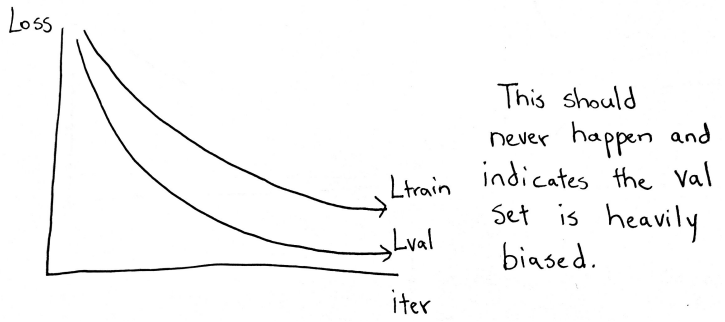

Figure 6.4: The val loss should never be lower than the train loss. If this happens, then either there's a bug in the code or the val dataset is highly biased.

What does it mean if the validation loss does not reflect the training loss at all? Here are some possibilities:

1. There's a bug in your evaluation code for the validation set.
2. Your validation set is biased. Some possible biases.

Bias in the validation set is usually a result of improper data splitting. For example, if we forget to randomly shuffle the dataset before splitting, the validation set may contain systematically different samples than the training set. Often, we need to be careful to fully represent all classes in the val and test datasets. Having only certain classes in the validation set can artificially inflate or deflate the validation loss. Deflation occurs when the val set contains mostly minority classes that the model is under-performing on. Inflation occurs when the val set contains mostly majority classes that the model is performing very well on.

6.5 Tuning Hyperparameters

Hyperparameter tuning is the process where we train a model several times to look for the best hyperparameters. Being good at tuning hyperparameters requires a combination of intuition and computing resources. The easiest way to speed up hyperparameter tuning is to buy more computers. This allows us to try more hyperparameters in parallel. We could also use our experience to narrow down the range of hyperparameters to try. Research papers are a good place to look for settings that have worked for others in the past.

There are 3 high-level strategies to tune hyperparameters.

1. Random search
2. Grid search
3. Optimization based search

Random search simply means randomly picking a value for each hyperparameter (within some range) at each round. Grid search means incrementing hyperparameters from a start value to an end value. Optimization based search tries to use the information gained from the previous training rounds to inform which hyperparameters to try in the next round. Choosing a good search range requires experience.

In practice, most practitioners use random search due to its simplicity of implementation and also because random search often works just as well (and sometimes even better) than the other two methods.

Grid search works poorly in practice. This is because out of all the hyperparameters you are tuning, only a subset of them will actually be important to the model's performance. Grid search only modifies a single hyperparameter at each round and test its importance. This means it wastes rounds adjusting a hyperparameter that does not affect performance much. On the other hand, random search modifies all hyperparameters in every round. This means no rounds are wasted tuning unimportant hyperparameters.

Researchers are actively exploring Bayesian optimization and reinforcement learning for automatic hyperparameter tuning. Unfortunately in the current state of research, random search often works just as well as these fancier methods and it's not worth bothering with the extra complexity.

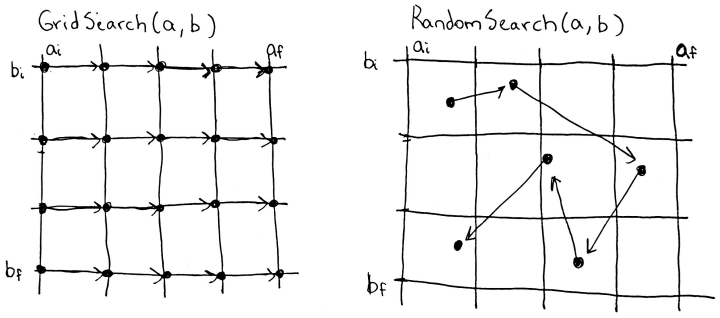

Figure 6.5: Visualization of grid and random search.

```
\\ Grid search
a_range = 1.0 to 8.0 in increments of 1.0
b_range = 0.1 to 10 in increments of 10x

for a in a_range:
    for b in b_range:
        train(model, train_set, hyperparameter=[a, b])
        model_score = evaluate(model, val_set, hyperparameter=[a,
    b])


\\ Random search
a_range = 1.0 to 8.0
b_range = 0.1 to 10

while (tuning_hyperparameters):
    a = random_value(a_range)
    b = random_value(b_range)

    train(model, train_set, hyperparameter=[a, b])
    model_score = evaluate(model, val_set, hyperparameter=[a, b])
```

6.6 Pretraining

Often times we don't have enough data when we start building machine learning applications. Neural networks require hundreds of thousands, if not millions of data samples to train. It's often very difficult to collect and very expensive to label such a large dataset. Luckily, there is a way to achieve respectable performance using only a few thousand samples.

The idea of **pretraining** is to utilize a large, existing, dataset that is similar to our own dataset to teach your model generally useful patterns. We only need to collect a much smaller **finetuning** dataset to convert these general patterns into the specific ones required for our task. To be more concrete, suppose we're building a model to classify different species of cats. It's a good idea to teach our model how to classify general objects like humans, tables, cars, and other animals. While learning to detect these other objects does not directly improve cat classification, it helps the model learn to pick up patterns that may be useful. At a low-level, the model will learn to detect edges, circles and other rudimentary shapes. At a higher-level, the model will learn to detect legs, faces and other components that will transfer to cat classification. The intuition is a model that knows how to detect human faces is a much better starting point to learn to detect cat faces than a model that doesn't know how to detect at all. After training on the general objects, we can finetune the model on a smaller cat dataset to convert the lessons learned to cat specific expertise.

The Imagenet dataset containing 14.2 million images and image class label. It is a popular pretraining dataset for many computer vision tasks. Practitioners often train their models on Imagenet or simply use part of a trained model. This allows them to train "downstream"

Pretraining Dataset	Downstream Tasks
Imagenet (image classification)	Object detection, image segmentation, sytle transfer.
Wiki-12M	Sentiment analysis, question-answering, text completion.

tasks like object detection and image segmentation on much smaller datasets. In table **??** I've listed some popular pretraining datasets for other domains.

\margintoc.

Add more rows to table. Need to verify text dataset, add audio, etc.

6.7 Debugging

\margintoc.

This section needs work. I'm not sure if I should include it since there are no concrete interviewe questions. I'll probably also need to do a survey of debugging methods with peers.

Design and Additional Features

Appendix

A

Heading on Level 0 (chapter)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.1 Heading on Level 1 (section)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 2 (subsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 3 (subsubsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text,

you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 4 (paragraph) Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.2 Lists

Example for list (itemize)

- First item in a list
- Second item in a list
- Third item in a list
- Fourth item in a list
- Fifth item in a list

Example for list (4*itemize)

- First item in a list
  - First item in a list
    - First item in a list
      - First item in a list
      - Second item in a list
    - Second item in a list
  - Second item in a list
- Second item in a list

Example for list (enumerate)

1. First item in a list
2. Second item in a list
3. Third item in a list
4. Fourth item in a list
5. Fifth item in a list

Example for list (4*enumerate)

1. First item in a list
    a) First item in a list
        i. First item in a list
            A. First item in a list
            B. Second item in a list
        ii. Second item in a list
    b) Second item in a list
2. Second item in a list

Example for list (description)

First item in a list
Second item in a list
Third item in a list
Fourth item in a list
Fifth item in a list

Example for list (4*description)

First item in a list
    First item in a list
        First item in a list
            First item in a list
            Second item in a list
        Second item in a list
    Second item in a list
Second item in a list

Notation

The next list describes several symbols that will be later used within the body of the document.

c Speed of light in a vacuum inertial frame

h Planck constant

Greek Letters with Pronounciation

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.