CHAPTER 1 Introduction

The main focus of machine learning (ML) is making decisions or predictions based on data. There are a number of other fields with significant overlap in technique, but difference in focus: in economics and psychology, the goal is to discover underlying causal processes and in statistics it is to find a model that fits a data set well. In those fields, the end product is a model. In machine learning, we often fit models, but as a means to the end of making good predictions or decisions.

As ML methods have improved in their capability and scope, ML has become arguably the best way–measured in terms of speed, human engineering time, and robustness–to approach many applications. Great examples are face detection, speech recognition, and many kinds of language-processing tasks. Almost any application that involves understanding data or signals that come from the real world can be nicely addressed using machine learning.

One crucial aspect of machine learning approaches to solving problems is that human engineering plays an important role. A human still has to frame the problem: acquire and organize data, design a space of possible solutions, select a learning algorithm and its parameters, apply the algorithm to the data, validate the resulting solution to decide whether it's good enough to use, try to understand the impact on the people who will be affected by its deployment, etc. These steps are of great importance.

The conceptual basis of learning from data is the problem of induction: Why do we think that previously seen data will help us predict the future? This is a serious long standing philosophical problem. We will operationalize it by making assumptions, such as that all training data are so-called i.i.d.(independent and identically distributed), and that queries will be drawn from the same distribution as the training data, or that the answer comes from a set of possible answers known in advance.

In general, we need to solve these two problems:

- estimation: When we have data that are noisy reflections of some underlying quantity of interest, we have to aggregate the data and make estimates or predictions about the quantity. How do we deal with the fact that, for example, the same treatment may end up with different results on different trials? How can we predict how well an estimate may compare to future results?
- generalization: How can we predict results of a situation or experiment that we have never encountered before in our data set?

We can describe problems and their solutions using six characteristics, three of which characterize the problem and three of which characterize the solution:

1. Problem class: What is the nature of the training data and what kinds of queries will be made at testing time?
2. Assumptions: What do we know about the source of the data or the form of the solution?

3. Evaluation criteria: What is the goal of the prediction or estimation system? How will the answers to individual queries be evaluated? How will the overall performance of the system be measured?
4. Model type: Will an intermediate model of the world be made? What aspects of the data will be modeled in different variables/parameters? How will the model be used to make predictions?
5. Model class: What particular class of models will be used? What criterion will we use to pick a particular model from the model class?
6. Algorithm: What computational process will be used to fit the model to the data and/or to make predictions?

Without making some assumptions about the nature of the process generating the data, we cannot perform generalization. In the following sections, we elaborate on these ideas.

1.1 Problem class

There are many different problem classes in machine learning. They vary according to what kind of data is provided and what kind of conclusions are to be drawn from it. Five standard problem classes are described below, to establish some notation and terminology.

In this course, we will focus on classification and regression (two examples of supervised learning), and we will touch on reinforcement learning, sequence learning, and clustering.

1.1.1 Supervised learning

The idea of supervised learning is that the learning system is given inputs and told which specific outputs should be associated with them. We divide up supervised learning based on whether the outputs are drawn from a small finite set (classification) or a large finite ordered set or continuous set (regression).

1.1.1.1 Regression

For a regression problem, the training data D_n is in the form of a set of n pairs:

$D_n = {(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})}$

where $x^{(i)}$ represents an input, most typically a d-dimensional vector of real and/or discrete values, and $y^{(i)}$ is the output to be predicted, in this case a real-number. The y values are sometimes called target values.

The goal in a regression problem is ultimately, given a new input value $x^{(n+1)}$, to predict the value of $y^{(n+1)}$.

Regression problems are a kind of supervised learning, because the desired output $y^{(i)}$ is specified for each of the training examples $x^{(i)}$.

1.1.1.2 Classification

A classification problem is like regression, except that the values that $y^{(i)}$ can take do not have an order. The classification problem is binary or two-class if $y^{(i)}$ (also known as the class) is drawn from a set of two possible values; otherwise, it is called multi-class.

1.1.2 Unsupervised learning

Unsupervised learning doesn't involve learning a function from inputs to outputs based on a set of input-output pairs. Instead, one is given a data set and generally expected to find some patterns or structure inherent in it.

1.1.2.1 Clustering

Given samples $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$, the goal is to find a partitioning (or "clustering") of the samples that groups together similar samples. There are many different objectives, depending on the definition of the similarity between samples and exactly what criterion is to be used (e.g., minimize the average distance between elements inside a cluster and maximize the average distance between elements across clusters). Other methods perform a "soft" clustering, in which samples may be assigned 0.9 membership in one cluster and 0.1 in another. Clustering is sometimes used as a step in the so-called density estimation (described below), and sometimes to find useful structure or influential features in data.

1.1.2.2 Density estimation

Given samples $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$ drawn i.i.d. from some distribution $\text{Pr}(X)$, the goal is to predict the probability $\text{Pr}(x^{(n+1)})$ of an element drawn from the same distribution. Density estimation sometimes plays a role as a "subroutine" in the overall learning method for supervised learning, as well.

1.1.2.3 Dimensionality reduction

Given samples $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^D$, the problem is to re-represent them as points in a d-dimensional space, where $d < D$. The goal is typically to retain information in the data set that will, e.g., allow elements of one class to be distinguished from another.

Dimensionality reduction is a standard technique that is particularly useful for visualizing or understanding high-dimensional data. If the goal is ultimately to perform regression or classification on the data after the dimensionality is reduced, it is usually best to articulate an objective for the overall prediction problem rather than to first do dimensionality reduction without knowing which dimensions will be important for the prediction task.

1.1.3 Sequence learning

In sequence learning, the goal is to learn a mapping from input sequences $x_0, \ldots, x_n$ to output sequences $y_1, \ldots, y_m$. The mapping is typically represented as a state machine,

with one function f_s used to compute the next hidden internal state given the input, and another function f_o used to compute the output given the current hidden state.

1.1.4 Reinforcement learning

In reinforcement learning, the goal is to learn a mapping from input values (typically assumed to be states of an agent or system) to output values (typically we want control actions). However, we need to learn the mapping without a direct supervision signal to specify which output values are best for a particular input; instead, the learning problem is framed as an agent interacting with an environment, in the following setting:

- The agent observes the current state s_t
- It selects an action a_t
- It receives a reward, r_t, which typically depends on s_t and possibly a_t
- The environment transitions probabilistically to a new state, s_{t+1}, with a distribution that depends only on s_t and a_t
- The agent observes the current state, s_{t+1}
- ...

The goal is to find a policy π, mapping s to a, (that is, states to actions) such that some long-term sum or average of rewards r is maximized.

1.1.5 Other settings

There are many other problem settings. Here are a few.

In semi-supervised learning, we have a supervised-learning training set, but there may be an additional set of $x^{(i)}$ values with no known $y^{(i)}$. These values can still be used to improve learning performance (if they are drawn from $\text{Pr}(X)$ that is the marginal of $\text{Pr}(X, Y)$ that governs the rest of the data set).

In active learning, it is assumed to be expensive to acquire a label $y^{(i)}$ (imagine asking a human to read an x-ray image), so the learning algorithm can sequentially ask for particular inputs $x^{(i)}$ to be labeled, and must carefully select queries in order to learn as effectively as possible while minimizing the cost of labeling.

In transfer learning (also called meta-learning), there are multiple tasks, with data drawn from different, but related, distributions. The goal is for experience with previous tasks to apply to learning a current task in a way that requires decreased experience with the new task.

1.2 Assumptions

The kinds of assumptions that we can make about the data source or the solution include:

- The data are independent and identically distributed (i.i.d.).

- The data are generated by a Markov chain (i.e. outputs only depend only on the current state, with no additional memory).
- The process generating the data might be adversarial.

• The "true" model that is generating the data can be perfectly described by one of some particular set of hypotheses.

The effect of an assumption is often to reduce the "size" or "expressiveness" of the space of possible hypotheses and therefore reduce the amount of data required to reliably identify an appropriate hypothesis.

1.3 Evaluation criteria

Once we have specified a problem class, we need to say what makes an output or the answer to a query good, given the training data. We specify evaluation criteria at two levels: how an individual prediction is scored, and how the overall behavior of the prediction or estimation system is scored.

The quality of predictions from a learned model is often expressed in terms of a loss function. A loss function $L(g, a)$ tells you how much you will be penalized for making a guess g when the answer is actually a. There are many possible loss functions. Here are some frequently used examples:

• 0-1 Loss:

$L(g, a) = \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$

• Squared loss:

$L(g, a) = (g - a)^2$

• Absolute loss:

$L(g, a) = |g - a|$

• Asymmetric loss:

$L(g, a) = \begin{cases} 1 & \text{if } g = 1 \text{ and } a = 0 \\ 10 & \text{if } g = 0 \text{ and } a = 1 \\ 0 & \text{otherwise} \end{cases}$

Any given prediction rule will usually be evaluated based on multiple predictions and the loss of each one. At this level, we might be interested in:

• Minimizing expected loss over all the predictions (also known as risk)

• Minimizing maximum loss: the loss of the worst prediction

• Minimizing or bounding regret: how much worse this predictor performs than the best one drawn from some class

• Characterizing asymptotic behavior: how well the predictor will perform in the limit of infinite training data

• Finding algorithms that are probably approximately correct: they probably generate a hypothesis that is right most of the time.

There is a theory of rational agency that argues that you should always select the action that minimizes the expected loss. This strategy will, for example, make you the most money in the long run, in a gambling setting. As mentioned above, expected loss is also sometimes called risk in ML literature, but that term means other things in economics or other parts of decision theory, so be careful...it's risky to use it. We will, most of the time, concentrate on this criterion.

1.4 Model type

Recall that the goal of a ML system is typically to estimate or generalize, based on data provided. Below, we examine the role of model-making in machine learning.

1.4.1 Non-parametric models

In some simple cases, in response to queries, we can generate predictions directly from the training data, without the construction of any intermediate model, or more precisely, without the learning of any parameters.

For example, in regression or classification, we might generate an answer to a new query by averaging answers to recent queries, as in the nearest neighbor method.

1.4.2 Parametric models

This two-step process is more typical:

1. "Fit" a model (with some a-prior chosen parameterization) to the training data

2. Use the model directly to make predictions

In the parametric models setting of regression or classification, the model will be some hypothesis or prediction rule $y = h(x; \Theta)$ for some functional form h. The term hypothesis has its roots in statistical learning and the scientific method, where models or hypotheses about the world are tested against real data, and refined with more evidence, observations, or insights. Note that the parameters themselves are only part of the assumptions that we're making about the world. The model itself is a hypothesis that will be refined with more evidence.

The idea is that Θ is a set of one or more parameter values that will be determined by fitting the model to the training data and then be held fixed during testing.

Given a new $x^{(n+1)}$, we would then make the prediction $h(x^{(n+1)}; \Theta)$.

The fitting process is often articulated as an optimization problem: Find a value of Θ that minimizes some criterion involving Θ and the data. An optimal strategy, if we knew the actual underlying distribution on our data, $\text{Pr}(X, Y)$ would be to predict the value of y that minimizes the expected loss, which is also known as the test error. If we don't have that actual underlying distribution, or even an estimate of it, we can take the approach of minimizing the training error: that is, finding the prediction rule h that minimizes the average loss on our training data set. So, we would seek Θ that minimizes

$E_n(h; \Theta) = \frac{1}{n}\sum_{i=1}^n L(h(x^{(i)}; \Theta), y^{(i)})$

where the loss function $L(g, a)$ measures how bad it would be to make a guess of g when the actual value is a.

We will find that minimizing training error alone is often not a good choice: it is possible to emphasize fitting the current data too strongly and end up with a hypothesis that does not generalize well when presented with new x values.

1.5 Model class and parameter fitting

A model class $\mathcal{M}$ is a set of possible models, typically parameterized by a vector of parameters Θ. What assumptions will we make about the form of the model? When solving a regression problem using a prediction-rule approach, we might try to find a linear function $h(x; \theta, \theta_0) = \theta^T x + \theta_0$ that fits our data well. In this example, the parameter vector $\Theta = (\theta, \theta_0)$.

For problem types such as classification, there are huge numbers of model classes that have been considered...we'll spend much of this course exploring these model classes, especially neural networks models.

We will almost completely restrict our attention to model classes with a fixed, finite number of parameters. Models that relax this assumption are called "non-parametric" models.

How do we select a model class? In some cases, the ML practitioner will have a good idea of what an appropriate model class is, and will specify it directly. In other cases, we may consider several model classes and choose the best based on some objective function. In such situations, we are solving a model selection problem: model-selection is to pick a model class $\mathcal{M}$ from a (usually finite) set of possible model classes, whereas model fitting is to pick a particular model in that class, specified by (usually continuous) parameters Θ.

1.6 Algorithm

Once we have described a class of models and a way of scoring a model given data, we have an algorithmic problem: what sequence of computational instructions should we run in order to find a good model from our class?

For example, determining the parameter vector which minimizes the training error might be done using a familiar least-squares minimization algorithm, when the model h is a function being fit to some data x. Sometimes we can use software that was designed, generically, to perform optimization. In many other cases, we use algorithms that are specialized for ML problems, or for particular hypotheses classes.

Some algorithms are not easily seen as trying to optimize a particular criterion. In fact, a historically important method for finding linear classifiers, the perceptron algorithm, has this character.

CHAPTER 2

Regression

Regression is an important machine-learning problem that provides a good starting point for diving deeply into the field.

2.1 Problem formulation

A hypothesis h is employed as a model for solving the regression problem, in that it maps inputs x to outputs y,

$x \rightarrow h \rightarrow y$

where $x \in \mathbb{R}^d$ (i.e., a length d column vector of real numbers), and $y \in \mathbb{R}$ (i.e., a real number). Real life rarely gives us vectors of real numbers; the x we really want to take as input is usually something like a song, image, or person. In that case, we'll have to define a function $\phi(x)$, whose range is $\mathbb{R}^d$, where ϕ represents features of x, like a person's height or the amount of bass in a song, and then let the $h: \phi(x) \rightarrow \mathbb{R}$. In much of the following, we'll omit explicit mention of ϕ and assume that the $x^{(i)}$ are in $\mathbb{R}^d$, but you should always have in mind that some additional process was almost surely required to go from the actual input examples to their feature representation, and we'll talk a lot more about features later in the course.

Regression is a supervised learning problem, in which we are given a training dataset of the form

$D_n = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$

which gives examples of input values $x^{(i)}$ and the output values $y^{(i)}$ that should be associated with them. Because y values are real-valued, our hypotheses will have the form

$h: \mathbb{R}^d \rightarrow \mathbb{R}$

This is a good framework when we want to predict a numerical quantity, like height, stock value, etc., rather than to divide the inputs into discrete categories.

What makes a hypothesis useful? That it works well on new data; that is, that it makes good predictions on examples it hasn't seen. But we don't know exactly what data this hypothesis might be tested on when we use it in the real world. So, we have to assume a connection between the training data and testing data – typically, the assumption is that they (the training and testing data) are drawn independently from the same probability distribution.

To make this discussion more concrete, we have to provide a loss function, to say how unhappy we are when we guess an output g given an input x for which the desired output was a.

Given a training set D_n and a hypothesis h with parameters Θ, we can define the training error of h to be the average loss on the training data:

$E_n(h; \Theta) = \frac{1}{n}\sum_{i=1}^n L(h(x^{(i)}; \Theta), y^{(i)})$ (2.1)

The training error of h gives us some idea of how well it characterizes the relationship between x and y values in our data, but it isn't the quantity that we most care about. What we most care about is test error:

$E(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} L(h(x^{(i)}), y^{(i)})$

on n' new examples that were not used in the process of finding the hypothesis.

For now, we will try to find a hypothesis with small training error (later, with some added criteria) and try to make some design choices so that it generalizes well to new data, meaning that it also has a small test error.

2.2 Regression as an optimization problem

Given data, a loss function, and a hypothesis class, we need a method for finding a good hypothesis in the class. One of the most general ways to approach this problem is by framing the machine learning problem as an optimization problem.

We begin by writing down an objective function $J(\Theta)$, where Θ stands for all the parameters in our model (i.e., all possible choices over parameters). We often write $J(\Theta; D)$ to make clear the dependence on the data D.

The objective function describes how we feel about possible hypotheses Θ: we will generally look for values for parameters Θ that minimize the objective function:

$\Theta^* = \arg\min_{\Theta} J(\Theta)$

A very common form for a machine-learning objective is

$J(\Theta) = \underbrace{\frac{1}{n}\sum_{i=1}^n L(h(x^{(i)}; \Theta), y^{(i)})}_{\text{loss}} + \underbrace{\lambda}_{\text{non-negative constant}} R(\Theta)$ (2.2)

The loss tells us how unhappy we are about the prediction $h(x^{(i)}; \Theta)$ that Θ makes for $(x^{(i)}, y^{(i)})$. Minimizing this loss makes the prediction better. The regularizer is an additional term that encourages the prediction to remain general, and the constant λ adjusts the balance between reproducing seen examples, and being able to generalize to unseen examples.

2.3 Linear regression

To make this discussion more concrete, we have to provide a hypothesis class and a loss function.

We will begin by picking a class of hypotheses $\mathcal{H}$ that we think might provide a good set of possible models of the relationship between x and y in our data. We will start with a very simple class of linear hypotheses for regression. It is both simple to study and very powerful, and will serve as the basis for many other important techniques (even neural networks!).

In linear regression, the set $\mathcal{H}$ of hypotheses has the form

$h(x; \theta, \theta_0) = \theta^T x + \theta_0$ (2.3)

with model parameters $\Theta = (\theta, \theta_0)$. In one dimension $(d = 1)$ this has the same familiar slope-intercept form as $y = mx + b$; in higher dimensions, this model describes the so-called hyperplanes.

We define a loss function to describe how to evaluate the quality of the predictions our hypothesis is making, when compared to the "target" y values in the data set. The choice of loss function is part of modeling your domain. In the absence of additional information about a regression problem, we typically use squared loss:

$L(g, a) = (g - a)^2$

where $g = h(x)$ is our "guess" from the hypothesis, and a is the "actual" observation (in other words, here a is being used equivalently as y). With this choice of squared loss, the average loss as generally defined in 2.1 will become the so-called mean squared error (MSE), which we'll study closely very soon.

The squared loss penalizes guesses that are too high the same amount as it penalizes guesses that are too low, and has a good mathematical justification in the case that your data are generated from an underlying linear hypothesis with the so-called Gaussian-distributed noise added to the y values.

Our objective in linear regression will be to find a hyperplane that goes as close as possible, on average, to all of our training data.

Applying the general optimization framework to the linear regression hypothesis class of Eq. 2.3 with squared loss and no regularization, our objective is to find values for $\Theta = (\theta, \theta_0)$ that minimize the MSE:

$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2$ (2.4)

resulting in the solution:

$\theta^*, \theta_0^* = \arg\min_{\theta,\theta_0} J(\theta, \theta_0)$ (2.5)

For one-dimensional data $(d = 1)$, this becomes the familiar problem of fitting a line to data. For $d > 1$, this hypothesis may be visualized as a d-dimensional hyperplane embedded in a $(d + 1)$-dimensional space (that consists of the input dimension and the y dimension).

A richer class of hypotheses can be obtained by performing a non-linear feature transformation before doing the regression, as we will later see (in Chapter 5), but it will still end up that we have to solve a linear regression problem.

2.4 A gloriously simple linear regression algorithm

Okay! Given the objective in Eq. 2.4, how can we find good values of θ and θ_0? We'll study several general-purpose, efficient, interesting algorithms. But before we do that, let's start with the simplest one we can think of: guess a whole bunch (k) of different values of θ and θ_0, see which one has the smallest error on the training set, and return it.

```

RANDOM-REGRESSION(D, k)

1 For i in 1...k: Randomly generate hypothesis θ^(i), θ_0^(i)

2 Let i = arg min_i J(θ^(i), θ_0^(i); D)

3 Return θ^(i), θ_0^(i)

```

This seems kind of silly, but it's a learning algorithm, and it's not completely useless.

Study Question: If your data set has n data points, and the dimension of the x values is d, what is the size of an individual $\theta^{(i)}$?

Study Question: How do you think increasing the number of guesses k will change the training error of the resulting hypothesis?

2.5 Analytical solution: ordinary least squares

One very interesting aspect of the problem of finding a linear hypothesis that minimizes mean squared error is that we can find a closed-form formula for the answer! This general problem is often called the ordinary least squares (OLS).

Everything is easier to deal with if we assume that all of the $x^{(i)}$ have been augmented with an extra input dimension (feature) that always has value 1, so that they are in $d + 1$ dimensions, and rather than having an explicit θ_0, we let it be the last element of our θ vector, so that we have, simply,

$y = \theta^T x$

In this case, the objective becomes

$J(\theta) = \frac{1}{n}\sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2$ (2.6)

We approach this just like a minimization problem from calculus homework: take the derivative of J with respect to θ, set it to zero, and solve for θ. There are additional steps required, to check that the resulting θ is a minimum (rather than a maximum or an inflection point) but we won't work through that here. It is possible to approach this problem by:

• Finding $\partial J/\partial \theta_k$ for k in $1, \ldots, d$,

• Constructing a set of k equations of the form $\partial J/\partial \theta_k = 0$, and

• Solving the system for values of θ_k

That works just fine. To get practice for applying techniques like this to more complex problems, we will work through a more compact (and cool!) matrix view. Along the way, it will be helpful to collect all of the derivatives in one vector. In particular, the gradient of J with respect to θ is following column vector of length d:

$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_1 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix}$

We can think of our training data in terms of matrices X and Y, where each column of X is an example, and each "column" of Y is the corresponding target output value:

$X = \begin{bmatrix} x^{(1)}_1 & \cdots & x^{(n)}_1 \\ \vdots & \vdots & \vdots \\ x^{(1)}_d & \cdots & x^{(n)}_d \end{bmatrix} \qquad Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(n)} \end{bmatrix}$

In most textbooks, they think of an individual example $x^{(i)}$ as a row, rather than a column. So that we get an answer that will be recognizable to you, we are going to define a new matrix and vector, $\tilde{X}$ and $\tilde{Y}$, which are just transposes of our X and Y, and then work with them:

$\tilde{X} = X^T = \begin{bmatrix} x^{(1)}_1 & \cdots & x^{(1)}_d \\ \vdots & \vdots & \vdots \\ x^{(n)}_1 & \cdots & x^{(n)}_d \end{bmatrix} \qquad \tilde{Y} = Y^T = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$

Now we can write

$J(\theta) = \frac{1}{n}\underbrace{(\tilde{X}\theta - \tilde{Y})^T}_{1\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} = \frac{1}{n}\sum_{i=1}^n \left(\sum_{j=1}^d \tilde{X}_{ij}\theta_j - \tilde{Y}_i\right)^2$

and using facts about matrix/vector calculus, we get

$\nabla_\theta J = \frac{2}{n}\underbrace{\tilde{X}^T}_{d\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1}$ (2.7)

Setting $\nabla_\theta J$ to 0 and solving, we get:

$\frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) = 0$

$\tilde{X}^T\tilde{X}\theta - \tilde{X}^T\tilde{Y} = 0$

$\tilde{X}^T\tilde{X}\theta = \tilde{X}^T\tilde{Y}$

$\theta = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T\tilde{Y}$

And the dimensions work out!

$\theta = \underbrace{(\tilde{X}^T\tilde{X})^{-1}}_{d\times d}\underbrace{\tilde{X}^T}_{d\times n}\underbrace{\tilde{Y}}_{n\times 1}$

So, given our data, we can directly compute the linear regression that minimizes mean squared error.

2.6 Regularization

The objective function of Eq. 2.2 balances (training-data) memorization, induced by the loss term, with generalization, induced by the regularization term. Here, we address the need for

regularization specifically for linear regression, and show how this can be realized using one popular regularization technique called ridge regression.

2.6.1 Regularization and linear regression

If all we cared about was finding a hypothesis with small loss on the training data, we would have no need for regularization, and could simply omit the second term in the objective. But remember that our ultimate goal is to perform well on input values that we haven't trained on! It may seem that this is an impossible task, but humans and machine-learning methods do this successfully all the time. What allows generalization to new input values is a belief that there is an underlying regularity that governs both the training and testing data. One way to describe an assumption about such a regularity is by choosing a limited class of possible hypotheses. Another way to do this is to provide smoother guidance, saying that, within a hypothesis class, we prefer some hypotheses to others. The regularizer articulates this preference and the constant λ says how much we are willing to trade off loss on the training data versus preference over hypotheses.

For example, consider what happens when $d = 2$, and x_2 is highly correlated with x_1, meaning that the data look like a line. Thus, there isn't a unique best hyperplane.

Such correlations happen often in real-life data, because of underlying common causes; for example, across a population, the height of people may depend on both age and amount of food intake in the same way. This is especially the case when there are many feature dimensions used in the regression. Mathematically, this leads to $\tilde{X}^T\tilde{X}$ close to singularity, such that $(\tilde{X}^T\tilde{X})^{-1}$ is undefined or has huge values, resulting in unstable models.

A common strategy for specifying a regularizer is to use the form

$R(\Theta) = \|\Theta - \Theta_{\text{prior}}\|^2$

when we have some idea in advance that Θ ought to be near some value Θ_{prior}.

Here, the notion of distance is quantified by squaring the l_2 norm of the parameter vector: for any d-dimensional vector $v \in \mathbb{R}^d$, the l_2 norm of v is defined as,

$\|v\| = \sqrt{\sum_{i=1}^d |v_i|^2}$

In the absence of such knowledge a default is to regularize toward zero:

$R(\Theta) = \|\Theta\|^2$

2.6.2 Ridge regression

There are some kinds of trouble we can get into in regression problems. What if $(\tilde{X}^T\tilde{X})$ is not invertible?

Study Question: Consider, for example, a situation where the data-set is just the same point repeated twice: $x^{(1)} = x^{(2)} = [1 \quad 2]^T$. What is $\tilde{X}$ in this case? What is $\tilde{X}^T\tilde{X}$? What is $(\tilde{X}^T\tilde{X})^{-1}$?

Another kind of problem is overfitting: we have formulated an objective that is just about fitting the data as well as possible, but we might also want to regularize to keep the hypothesis from getting too attached to the data.

We address both the problem of not being able to invert $(\tilde{X}^T\tilde{X})^{-1}$ and the problem of overfitting using a mechanism called ridge regression. We add a regularization term $\|\theta\|^2$ to the OLS objective, with a non-negative scalar value λ to control the tradeoff between the training error and the regularization term.

Here is the ridge regression objective function:

$J_{\text{ridge}}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2 + \lambda\|\theta\|^2$

Larger λ values pressure θ values to be near zero. Note that we don't penalize θ_0; intuitively, θ_0 is what "floats" the regression surface to the right level for the data you have.

If we decide not to treat θ_0 specially (so we add a 1 feature to our input vectors as discussed above), then we get:

$\nabla_\theta J_{\text{ridge}} = \frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta$

Setting to 0 and solving, we get:

$\frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta = 0$

$\frac{1}{n}\tilde{X}^T\tilde{X}\theta - \frac{1}{n}\tilde{X}^T\tilde{Y} + \lambda\theta = 0$

$\frac{1}{n}\tilde{X}^T\tilde{X}\theta + \lambda\theta = \frac{1}{n}\tilde{X}^T\tilde{Y}$

$\tilde{X}^T\tilde{X}\theta + n\lambda\theta = \tilde{X}^T\tilde{Y}$

$(\tilde{X}^T\tilde{X} + n\lambda I)\theta = \tilde{X}^T\tilde{Y}$

$\theta = (\tilde{X}^T\tilde{X} + n\lambda I)^{-1}\tilde{X}^T\tilde{Y}$

So the solution is:

$\theta_{\text{ridge}} = (\tilde{X}^T\tilde{X} + n\lambda I)^{-1}\tilde{X}^T\tilde{Y}$ (2.8)

and the term $\tilde{X}^T\tilde{X} + n\lambda I$ becomes invertible when $\lambda > 0$.

2.7 Evaluating learning algorithms

In this section, we will explore how to evaluate supervised machine-learning algorithms. We will study the special case of applying them to regression problems, but the basic ideas of validation, hyper-parameter selection, and cross-validation apply much more broadly.

We have seen how linear regression is a well-formed optimization problem, which has an analytical solution when ridge regularization is applied. But how can one choose the best amount of regularization, as parameterized by λ? Two key ideas involve the evaluation of the performance of a hypothesis, and a separate evaluation of the algorithm used to produce hypotheses, as described below.

2.7.1 Evaluating hypotheses

The performance of a given hypothesis h may be evaluated by measuring test error on data that was not used to train it. Given a training set D_n, a regression hypothesis h, and if we choose squared loss, we can define the OLS training error of h to be the mean square error between its predictions and the expected outputs:

$E_n(h) = \frac{1}{n}\sum_{i=1}^n [h(x^{(i)}) - y^{(i)}]^2$

Test error captures the performance of h on unseen data, and is the mean square error on the test set, with a nearly identical expression as that above, differing only in the range of index i:

$E(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} [h(x^{(i)}) - y^{(i)}]^2$

on n' new examples that were not used in the process of constructing h.

In machine learning in general, not just regression, it is useful to distinguish two ways in which a hypothesis $h \in \mathcal{H}$ might contribute to test error. Two are:

Structural error: This is error that arises because there is no hypothesis $h \in \mathcal{H}$ that will perform well on the data, for example because the data was really generated by a sine wave but we are trying to fit it with a line.

Estimation error: This is error that arises because we do not have enough data (or the data are in some way unhelpful) to allow us to choose a good $h \in \mathcal{H}$, or because we didn't solve the optimization problem well enough to find the best h given the data that we had.

When we increase λ, we tend to increase structural error but decrease estimation error, and vice versa.

2.7.2 Evaluating learning algorithms

A learning algorithm is a procedure that takes a data set D_n as input and returns an hypothesis h from a hypothesis class $\mathcal{H}$; it looks like

$D_n \rightarrow \text{learning alg}(\mathcal{H}) \rightarrow h$

Keep in mind that h has parameters. The learning algorithm itself may have its own parameters, and such parameters are often called hyperparameters. The analytical solutions presented above for linear regression, e.g., Eq. 2.8, may be thought of as learning algorithms, where λ is a hyperparameter that governs how the learning algorithm works and can strongly affect its performance.

How should we evaluate the performance of a learning algorithm? This can be tricky. There are many potential sources of variability in the possible result of computing test error on a learned hypothesis h:

• Which particular training examples occurred in D_n

• Which particular testing examples occurred in $D_{n'}$

• Randomization inside the learning algorithm itself

2.7.2.1 Validation

Generally, to evaluate how well a learning algorithm works, given an unlimited data source, we would like to execute the following process multiple times:

• Train on a new training set (subset of our big data source)

• Evaluate resulting h on a validation set that does not overlap the training set (but is still a subset of our same big data source)

Running the algorithm multiple times controls for possible poor choices of training set or unfortunate randomization inside the algorithm itself.

2.7.2.2 Cross validation

One concern is that we might need a lot of data to do this, and in many applications data is expensive or difficult to acquire. We can re-use data with cross validation (but it's harder to do theoretical analysis).

```

CROSS-VALIDATE(D, k)

1 divide D into k chunks $D_1$, $D_2$, ..., $D_k$ (of roughly equal size)

2 for i = 1 to k
```

3 train h_i on $D \setminus D_i$ (withholding chunk D_i as the validation set)

4 compute "test" error $E_i(h_i)$ on withheld data D_i

5 return $1/k \sum_{i=1}^{k} E_i(h_i)$

```

It's very important to understand that (cross-)validation neither delivers nor evaluates a single particular hypothesis $h$. It evaluates the learning algorithm that produces hypotheses.

2.7.2.3 Hyperparameter tuning

The hyper-parameters of a learning algorithm affect how the algorithm works but they are not part of the resulting hypothesis. So, for example, $\lambda$ in ridge regression affects which hypothesis will be returned, but $\lambda$ itself doesn't show up in the hypothesis (the hypothesis is specified using parameters $\theta$ and $\theta_0$).

You can think about each different setting of a hyper-parameter as specifying a different learning algorithm.

In order to pick a good value of the hyper-parameter, we often end up just trying a lot of values and seeing which one works best via validation or cross-validation.

CHAPTER 3

Gradient Descent

In the previous chapter, we showed how to describe an interesting objective function for machine learning, but we need a way to find the optimal $\Theta^* = \arg\min_\Theta J(\Theta)$, particularly when the objective function is not amenable to analytical optimization. For example, this can be the case when $J(\Theta)$ involves a more complex loss function, or more general forms of regularization. It can also be the case when there is simply too much data for it to be computationally feasible to analytically invert the required matrices.

There is an enormous and fascinating literature on the mathematical and algorithmic foundations of optimization, but for this class, we will consider one of the simplest methods, called gradient descent.

Intuitively, in one or two dimensions, we can easily think of $J(\Theta)$ as defining a surface over $\Theta$; that same idea extends to higher dimensions. Now, our objective is to find the $\Theta$ value at the lowest point on that surface. One way to think about gradient descent is that you start at some arbitrary point on the surface, look to see in which direction the "hill" goes down most steeply, take a small step in that direction, determine the direction of steepest descent from where you are, take another small step, etc.

3.1 Gradient descent in one dimension
```

We start by considering gradient descent in one dimension. Assume $\Theta \in \mathbb{R}$, and that we know both $J(\Theta)$ and its first derivative with respect to Θ, $J'(\Theta)$. Here is pseudo-code for gradient descent on an arbitrary function f. Along with f and its gradient $\nabla_\Theta f$ (which, in the case of a scalar Θ, is the same as its derivative f'), we have to specify some hyperparameters. These hyper-parameters include the initial value for parameter Θ, a step-size hyper-parameter η, and an accuracy hyper-parameter ϵ.

The hyper-parameter η is often called learning rate when gradient descent is applied in machine learning. For simplicity, η may be taken as a constant, as is the case in the pseudo-code below; and we'll see adaptive (non-constant) step-sizes soon. What's important to notice though, is that even when η is constant, the actual magnitude of the change to Θ may not be constant, as that change depends on the magnitude of the gradient itself too.

```

1D-GRADIENT-DESCENT(Θ_init, η, f, f', ϵ)

1 Θ^(0) = Θ_init

2 t = 0

3 repeat

4     t = t + 1

5     Θ^(t) = Θ^(t−1) - η f'(Θ^(t−1))

6 until |f(Θ^(t)) - f(Θ^(t−1))| < ϵ

7 return Θ^(t)

```

Note that this algorithm terminates when the change in the function f is sufficiently small. There are many other reasonable ways to decide to terminate, including:

• Stop after a fixed number of iterations T, i.e., when $t = T$.

• Stop when the change in the value of the parameter Θ is sufficiently small, i.e., when $|\Theta^{(t)} - \Theta^{(t-1)}| < \epsilon$.

• Stop when the derivative f' at the latest value of Θ is sufficiently small, i.e., when $|f'(\Theta^{(t)})| < \epsilon$.

Theorem 3.1.1. Choose any small distance $\tilde{\epsilon} > 0$. If we assume that f has a minimum, is sufficiently "smooth" and convex, and if the step size η is sufficiently small, gradient descent will reach a point within $\tilde{\epsilon}$ of a global optimum point Θ.

However, we must be careful when choosing the step size to prevent slow convergence, non-converging oscillation around the minimum, or divergence.

The following plot illustrates a convex function $f(x) = (x-2)^2$, starting gradient descent at $x_{\text{init}} = 4.0$ with a step-size of $1/2$. It is very well-behaved!

[Plot axes showing x from -1 to 6 and f(x) from 0 to 4]

If f is non-convex, where gradient descent converges to depends on x_{init}. First, let's establish some definitions. Suppose we have analytically defined derivatives for f.

we say that f has a local minimum point or local optimum point at x if $f'(x) = 0$ and $f''(x) > 0$, and we say that $f(x)$ is a local minimum value of f. More generally, x is a local minimum point of f if $f(x)$ is at least as low as $f(x')$ for all points x' in some small area around x. We say that f has a global minimum point at x if $f(x)$ is at least as low as $f(x')$ for every other input x'. And then we call $f(x)$ a global minimum value. A global minimum point is also a local minimum point, but a local minimum point does not have to be a global minimum point.

If f is non-convex (and sufficiently smooth), one expects that gradient descent (run long enough with small enough step size) will get very close to a point at which the gradient is zero, though we cannot guarantee that it will converge to a global minimum point.

There are two notable exceptions to this common sense expectation: First, gradient descent can get stagnated while approaching a point x which is not a local minimum or maximum, but satisfies $f'(x) = 0$. For example, for $f(x) = x^3$, starting gradient descent from the initial guess $x_{\text{init}} = 1$, while using step size $\eta < 1/3$ will lead to $x^{(k)}$ converging to zero as $k \to \infty$. Second, there are functions (even convex ones) with no minimum points, like $f(x) = \exp(-x)$, for which gradient descent with a positive step size converges to $+\infty$.

3.2 Multiple dimensions

The extension to the case of multi-dimensional Θ is straightforward. Let's assume $\Theta \in \mathbb{R}^m$, so $f: \mathbb{R}^m \to \mathbb{R}$. The gradient of f with respect to Θ is

$\nabla_\Theta f = \begin{bmatrix} \partial f/\partial \Theta_1 \\ \vdots \\ \partial f/\partial \Theta_m \end{bmatrix}$

The algorithm remains the same, except that the update step in line 5 becomes

$\Theta^{(t)} = \Theta^{(t-1)} - \eta\nabla_\Theta f(\Theta^{(t-1)})$

and any termination criteria that depended on the dimensionality of Θ would have to change. The easiest thing is to keep the test in line 6 as $|f(\Theta^{(t)}) - f(\Theta^{(t-1)})| < \epsilon$, which is sensible no matter the dimensionality of Θ.

3.3 Application to regression

Recall from the previous chapter that choosing a loss function is the first step in formulating a machine-learning problem as an optimization problem, and for regression we studied the mean square loss, which captures loss as $(guess - actual)^2$. This leads to the ordinary least squares objective

$J(\theta) = \frac{1}{n}\sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2$ (3.1)

We use the gradient of the objective with respect to the parameters,

$\nabla_\theta J = \frac{2}{n}\underbrace{\tilde{X}^T}_{d\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1}$ (3.2)

to obtain an analytical solution to the linear regression problem. Gradient descent could also be applied to numerically compute a solution, using the update rule

$\theta^{(t)} = \theta^{(t-1)} - \eta\frac{2}{n}\sum_{i=1}^n [(\theta^{(t-1)})^T x^{(i)} - y^{(i)}]x^{(i)}$ (3.3)

3.3.1 Ridge regression

Now, let's add in the regularization term, to get the ridge-regression objective:

$J_{\text{ridge}}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2 + \lambda\|\theta\|^2$

Recall that in ordinary least squares, we finessed handling θ_0 by adding an extra dimension of all 1's. In ridge regression, we really do need to separate the parameter vector θ from the offset θ_0, and so, from the perspective of our general-purpose gradient descent method, our whole parameter set Θ is defined to be $\Theta = (\theta, \theta_0)$. We will go ahead and find the gradients separately for each one:

$\nabla_\theta J_{\text{ridge}}(\theta, \theta_0) = \frac{2}{n}\sum_{i=1}^n (\theta^T x^{(i)} + \theta_0 - y^{(i)})x^{(i)} + 2\lambda\theta$

$\frac{\partial J_{\text{ridge}}(\theta, \theta_0)}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^n (\theta^T x^{(i)} + \theta_0 - y^{(i)})$

Note that $\nabla_\theta J_{\text{ridge}}$ will be of shape $d \times 1$ and $\partial J_{\text{ridge}}/\partial \theta_0$ will be a scalar since we have separated θ_0 from θ here.

Let's put everything together with our gradient descent algorithm for ridge regression:

```

RR-GRADIENT-DESCENT(θ_init, θ₀_init, η, ϵ)

1 θ^(0) = θ_init

2 θ₀^(0) = θ₀_init

3 t = 0

4 repeat

5     t = t + 1

6     θ^(t) = θ^(t−1) - η(1/n ∑_{i=1}^n [(θ^(t−1))^T x_i^(i) + θ₀^(t−1) - y_i^(i)]x_i^(i) + λθ^(t−1))

7     θ₀^(t) = θ₀^(t−1) - η(1/n ∑_{i=1}^n [(θ^(t−1))^T x_i^(i) + θ₀^(t−1) - y_i^(i)])

8 until |J_ridge(θ^(t), θ₀^(t)) - J_ridge(θ^(t−1), θ₀^(t−1))| < ϵ

9 return θ^(t), θ₀^(t)

```

3.4 Stochastic gradient descent

When the form of the gradient is a sum, rather than take one big(ish) step in the direction of the gradient, we can, instead, randomly select one term of the sum, and take a very small step in that direction. This seems sort of crazy, but remember that all the little steps would average out to the same direction as the big step if you were to stay in one place. Of course, you're not staying in that place, so you move, in expectation, in the direction of the gradient.

Most objective functions in machine learning can end up being written as a sum over data points, in which case, stochastic gradient descent (SGD) is implemented by picking a data point randomly out of the data set, computing the gradient as if there were only that one point in the data set, and taking a small step in the negative direction.

Let's assume our objective has the form

$f(\Theta) = \sum_{i=1}^n f_i(\Theta)$

where n is the number of data points used in the objective (and this may be different from the number of points available in the whole data set). Here is pseudocode for applying SGD to such an objective f; it assumes we know the form of $\nabla_\Theta f_i$ for all i in $1 \ldots n$:

```
```

STOCHASTIC-GRADIENT-DESCENT(Θ_init, η, f, ∇_Θf₁, ..., ∇_Θf□, T)

1 $\Theta^{(0)}$ = Θ_init

2 for t = 1 to T

3 randomly select i ∈ {1, 2, ..., n}

4 $\Theta^{(t)} = \Theta^{(t-1)} - \eta^{(t)} \nabla_\Theta f_i(\Theta^{(t-1)})$

5 return $\Theta^{(t)}$

```

Note that now instead of a fixed value of $\eta$, $\eta$ is indexed by the iteration of the algorithm, $t$. Choosing a good stopping criterion can be a little trickier for SGD than traditional gradient descent. Here we've just chosen to stop after a fixed number of iterations $T$.

For SGD to converge to a local optimum point as $t$ increases, the step size has to decrease as a function of time. The next result shows one step size sequence that works.

**Theorem 3.4.1.** If $f$ is convex, and $\eta^{(t)}$ is a sequence satisfying

$\sum_{t=1}^{\infty} \eta^{(t)} = \infty$ and $\sum_{t=1}^{\infty} (\eta^{(t)})^2 < \infty$

then SGD converges with probability one to the optimal $\Theta$.

Why these two conditions? The intuition is that the first condition, on $\sum\eta^{(t)}$, is needed to allow for the possibility of an unbounded potential range of exploration, while the second condition, on $\sum(\eta^{(t)})^2$, ensures that the step sizes get smaller and smaller as $t$ increases. One "legal" way of setting the step size is to make $\eta^{(t)} = 1/t$ but people often use rules that decrease more slowly, and so don't strictly satisfy the criteria for convergence.

There are multiple intuitions for why SGD might be a better choice algorithmically than regular GD (which is sometimes called batch GD (BGD)):

• BGD typically requires computing some quantity over every data point in a data set. SGD may perform well after visiting only some of the data. This behavior can be useful for very large data sets – in runtime and memory savings.

• If your $f$ is actually non-convex, but has many shallow local optimum points that might trap BGD, then taking samples from the gradient at some point $\Theta$ might "bounce" you around the landscape and away from the local optimum points.

• Sometimes, optimizing $f$ really well is not what we want to do, because it might overfit the training set; so, in fact, although SGD might not get lower training error than BGD, it might result in lower test error.
```

CHAPTER 4 Classification

4.1 Classification

Classification is a machine learning problem seeking to map from inputs $\mathbb{R}^d$ to outputs in an unordered set. Examples of classification output sets could be {apples, oranges, pears} in contrast to a continuous real-valued output, as we saw for linear regression if we're trying to figure out what type of fruit we have, or {heartattack, noheartattack} if we're working in an emergency room and trying to give the best medical care to a new patient. We focus on an essential simple case, binary classification, where we aim to find a mapping from $\mathbb{R}^d$ to two outputs. While we should think of the outputs as not having an order, it's often convenient to encode them as {−1, +1}. As before, let the letter h (for hypothesis) represent a classifier, so the classification process looks like:

$x \rightarrow h \rightarrow y$

Like regression, classification is a supervised learning problem, in which we are given a training data set of the form:

$D_n = {(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})}$

We will assume that each $x^{(i)}$ is a $d \times 1$ column vector. The intended meaning of this data is that, when given an input $x^{(i)}$, the learned hypothesis should generate output $y^{(i)}$.

What makes a classifier useful? As in regression, we want it to work well on new data, making good predictions on examples it hasn't seen. But we don't know exactly what data this classifier might be tested on when we use it in the real world. So, we have to assume a connection between the training data and testing data; typically, they are drawn independently from the same probability distribution.

In classification, we will often use 0-1 loss for evaluation (as discussed in Section 1.3). For that choice, we can write the training error and the testing error. In particular, given a training set D_n and a classifier h, we define the training error of h to be:

$E_n(h) = \frac{1}{n}\sum_{i=1}^n \begin{cases} 1 & \text{if } h(x^{(i)}) \neq y^{(i)} \ 0 & \text{otherwise} \end{cases}$

For now, we will try to find a classifier with small training error (later, with some added criteria) and hope it generalizes well to new data, and has a small test error:

$E(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} \begin{cases} 1 & \text{if } h(x^{(i)}) \neq y^{(i)} \ 0 & \text{otherwise} \end{cases}$

on n' new examples that were not used in the process of finding the classifier.

4.2 Linear classifiers

We begin by introducing the hypothesis class of linear classifiers (Section 4.2) and then define an optimization framework to learn linear logistic classifiers (Section 4.3).

4.2.1 Linear classifiers: definition

A linear classifier in d dimensions is defined by a vector of parameters $\theta \in \mathbb{R}^d$ and scalar $\theta_0 \in \mathbb{R}$. So, the hypothesis class H of linear classifiers in d dimensions is parameterized by the set of all vectors in $\mathbb{R}^{d+1}$. We'll assume that θ is a $d \times 1$ column vector.

Given particular values for θ and θ_0, the classifier is defined by:

$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$

Remember that we can think of θ, θ_0 as specifying a d-dimensional hyperplane. But this time, rather than being interested in that hyperplane's values at particular points x, we will focus on the separator that it induces. The separator is the set of x values such that $\theta^T x + \theta_0 = 0$. This is also a hyperplane, but in $d-1$ dimensions!

We can interpret θ as a vector that is perpendicular to the separator. (We will also say that θ is normal to the separator.)

For example, in two dimensions (d = 2) the separator has dimension 1, which means it is a line, and the two components of $\theta = [\theta_1, \theta_2]^T$ give the orientation of the separator, as illustrated in the following example.

Example: Let h be the linear classifier defined by $\theta = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, $\theta_0 = 1$.

What is θ_0? We can solve for it by plugging a point on the line into the equation for the line. It is often convenient to choose a point on one of the axes, e.g., in this case, $x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, for which $\theta^T \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \theta_0 = 0$, giving $\theta_0 = 1$.

In this example, the separator divides $\mathbb{R}^d$, the space our $x^{(i)}$ points live in, into two half-spaces. The one that is on the same side as the normal vector is the positive half-space, and we classify all points in that space as positive. The half-space on the other side is negative and all points in it are classified as negative.

Example: Let h be the linear classifier defined by $\theta = \begin{bmatrix} -1.5 \\ 1 \end{bmatrix}$, $\theta_0 = 3$.

Let $x^{(1)} = \begin{bmatrix} 3 \ 2 \end{bmatrix}$ and $x^{(2)} = \begin{bmatrix} -4 \ 1 \end{bmatrix}$.

$h(x^{(1)}; \theta, \theta_0) = \text{sign}(\begin{bmatrix} -1.5 & 1 \end{bmatrix} \begin{bmatrix} 3 \ 2 \end{bmatrix} + 3) = \text{sign}(3) = +1$

$h(x^{(2)}; \theta, \theta_0) = \text{sign}(\begin{bmatrix} -1.5 & 1 \end{bmatrix} \begin{bmatrix} -4 \ 1 \end{bmatrix} + 3) = \text{sign}(-2.5) = -1$

Thus, $x^{(1)}$ and $x^{(2)}$ are given positive and negative classifications, respectively.

4.3 Linear logistic classifiers

Given a data set and the hypothesis class of linear classifiers, our goal will be to find the linear classifier that optimizes an objective function relating its predictions to the training data. To make this problem computationally reasonable, we will need to take care in how we formulate the optimization problem to achieve this goal.

For classification, it is natural to make predictions in {+1, −1} and use the 0-1 loss function, L_{01}, as introduced in Chapter 1:

$L_{01}(g, a) = \begin{cases} 0 & \text{if } g = a \ 1 & \text{otherwise} \end{cases}$

However, even for simple linear classifiers, it is very difficult to find values for θ, θ_0 that minimize simple 0-1 training error:

$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n L_{01}(\text{sign}(\theta^T x^{(i)} + \theta_0), y^{(i)})$

This problem is NP-hard, which probably implies that solving the most difficult instances of this problem would require computation time exponential in the number of training examples, n.

[Note: The "probably" here is not because we're too lazy to look it up, but actually because of a fundamental unsolved problem in computer science theory, known as "P vs. NP."]

What makes this a difficult optimization problem is its lack of "smoothness":

• There can be two hypotheses, (θ, θ_0) and (θ', θ_0'), where one is closer in parameter space to the optimal parameter values (θ^*, θ_0^*), but they make the same number of misclassifications so they have the same J value.

• All predictions are categorical: the classifier can't express a degree of certainty about whether a particular input x should have an associated value y.

For these reasons, if we are considering a hypothesis θ, θ_0 that makes five incorrect predictions, it is difficult to see how we might change θ, θ_0 so that it will perform better, which makes it difficult to design an algorithm that searches in a sensible way

through the space of hypotheses for a good one. For these reasons, we investigate another hypothesis class: linear logistic classifiers, providing their definition, then an approach for learning such classifiers using optimization.

4.3.1 Linear logistic classifiers: definition

The hypotheses in a linear logistic classifier (LLC) are parameterized by a d-dimensional vector θ and a scalar θ_0, just as is the case for linear classifiers. However, instead of making predictions in {+1, −1}, LLC hypotheses generate real-valued outputs in the interval (0, 1). An LLC has the form:

$h(x; \theta, \theta_0) = \sigma(\theta^T x + \theta_0)$

This looks familiar! What's new?

The logistic function, also known as the sigmoid function, is defined as:

$\sigma(z) = \frac{1}{1 + e^{-z}}$

and is plotted below, as a function of its input z. Its output can be interpreted as a probability, because for any value of z the output is in (0, 1).

What does an LLC look like? Let's consider the simple case where d = 1, so our input points simply lie along the x axis. Classifiers in this case have dimension 0, meaning that they are points. The plot below shows LLCs for three different parameter settings: $\sigma(10x + 1)$, $\sigma(-2x + 1)$, and $\sigma(2x - 3)$.

But wait! Remember that the definition of a classifier is that it's a mapping from $\mathbb{R}^d \rightarrow \{-1, +1\}$ or to some other discrete set. So, then, it seems like an LLC is actually not a classifier!

Given an LLC, with an output value in (0, 1), what should we do if we are forced to make a prediction in {+1, −1}? A default answer is to predict +1 if $\sigma(\theta^T x + \theta_0) > 0.5$ and −1 otherwise. The value 0.5 is sometimes called a prediction threshold.

4.3.2 Learning linear logistic classifiers

Optimization is a key approach to solving machine learning problems; this also applies to learning linear logistic classifiers (LLCs) by defining an appropriate loss function for optimization. A first attempt might be to use the simple 0-1 loss function L_{01} that gives a value of 0 for a correct prediction, and a 1 for an incorrect prediction.

For learning LLCs, we'd have a class of hypotheses whose outputs are in (0, 1), but for which we have training data with y values in {+1, −1}. How can we define an appropriate loss function? We start by changing our interpretation of the output to be the probability that the input should map to output value 1.

Intuitively, we would like to have low loss if we assign a high probability to the correct class. We'll define a loss function, called negative log-likelihood (NLL), that does just this. In addition, it has the cool property that it extends nicely to the case where we would like to classify our inputs into more than two classes.

In order to simplify the description, we assume that (or transform our data so that) the labels in the training data are $y \in \{0, 1\}$.

We would like to pick the parameters of our classifier to maximize the probability assigned by the LLC to the correct y values, as specified in the training set. Letting guess $g^{(i)} = \sigma(\theta^T x^{(i)} + \theta_0)$, that probability is:

$\prod_{i=1}^n \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{otherwise} \end{cases}$

under the assumption that our predictions are independent. This can be cleverly rewritten, when $y^{(i)} \in \{0, 1\}$, as:

$\prod_{i=1}^n (g^{(i)})^{y^{(i)}} (1 - g^{(i)})^{1-y^{(i)}}$

Because the log function is monotonic, the θ, θ_0 that maximize the quantity above will be the same as the θ, θ_0 that maximize its log, which is the following:

$\sum_{i=1}^n [y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log(1 - g^{(i)})]$

Finally, we can turn the maximization problem above into a minimization problem by taking the negative of the above expression, and write in terms of minimizing a loss:

$\sum_{i=1}^n L_{\text{nll}}(g^{(i)}, y^{(i)})$

where L_{nll} is the negative log-likelihood loss function:

$L_{\text{nll}}(\text{guess}, \text{actual}) = -((\text{actual} \cdot \log(\text{guess}) + (1 - \text{actual}) \cdot \log(1 - \text{guess}))$

[Note: You can use any base for the logarithm and it won't make any real difference. If we ask you for numbers, use log base e.]

What is the objective function for linear logistic classification? We can finally put all these pieces together and develop an objective function for optimizing regularized negative log-likelihood for a linear logistic classifier. In fact, this process is usually called "logistic regression," so we'll call our objective J_{lr}, and define it as:

$J_{\text{lr}}(\theta, \theta_0; D) = \left(\frac{1}{n}\sum_{i=1}^n L_{\text{nll}}(\sigma(\theta^T x^{(i)} + \theta_0), y^{(i)})\right) + \lambda \|\theta\|^2$ (4.1)

What role does regularization play for classifiers? This objective function has the same structure as the one we used for regression, Eq. 2.2, where the first term (in parentheses) is the average loss, and the second term is for regularization. Regularization is needed for building classifiers that can generalize well (just as was the case for regression). The parameter λ governs the trade-off between the two terms as illustrated in the following example.

4.4 Gradient descent for logistic regression

Now that we have a hypothesis class (LLC) and a loss function (NLL), we need to take some data and find parameters! We need derivatives with respect to both θ_0 (the scalar component) and θ (the vector component) of Θ. Explicitly, they are:

$\nabla_\theta J_{\text{lr}}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n (g^{(i)} - y^{(i)})x^{(i)} + 2\lambda\theta$

$\frac{\partial J_{\text{lr}}(\theta, \theta_0)}{\partial \theta_0} = \frac{1}{n}\sum_{i=1}^n (g^{(i)} - y^{(i)})$

Note that $\nabla_\theta J_{\text{lr}}$ will be of shape $d \times 1$ and $\frac{\partial J_{\text{lr}}}{\partial \theta_0}$ will be a scalar since we have separated θ_0 from θ here.

Putting everything together, our gradient descent algorithm for logistic regression becomes:

LR-GRADIENT-DESCENT($\theta_{\text{init}}, \theta_{0\text{init}}, \eta, \epsilon$)

1. $\theta^{(0)} = \theta_{\text{init}}$

2. $\theta_0^{(0)} = \theta_{0\text{init}}$

3. $t = 0$

4. repeat

5. $t = t + 1$

6. $\theta^{(t)} = \theta^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^n \left[\sigma(\theta^{(t-1)T}x^{(i)} + \theta_0^{(t-1)}) - y^{(i)}\right]x^{(i)} + 2\lambda\theta^{(t-1)}\right)$

7. $\theta_0^{(t)} = \theta_0^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^n \left[\sigma(\theta^{(t-1)T}x^{(i)} + \theta_0^{(t-1)}) - y^{(i)}\right]\right)$

8. until $|J_{\text{lr}}(\theta^{(t)}, \theta_0^{(t)}) - J_{\text{lr}}(\theta^{(t-1)}, \theta_0^{(t-1)})| < \epsilon$

9. return $\theta^{(t)}, \theta_0^{(t)}$

4.4.1 Convexity of the NLL Loss Function

Much like the squared-error loss function that we saw for linear regression, the NLL loss function for linear logistic regression is a convex function. This means that running gradient descent with a reasonable set of hyperparameters will converge arbitrarily close to the minimum of the objective function.

Let $z = \theta^T x + \theta_0$; z is an affine function of θ and θ_0. It therefore suffices to show that the functions $f_1(z) = -\log(\sigma(z))$ and $f_2(z) = -\log(1-\sigma(z))$ are convex with respect to z.

First, we can see that since:

$\frac{d}{dz}f_1(z) = \frac{d}{dz}[-\log(1/(1 + \exp(-z)))]$

$= \frac{d}{dz}[\log(1 + \exp(-z))]$

$= -\exp(-z)/(1 + \exp(-z))$

$= -1 + \sigma(z)$

the derivative of the function $f_1(z)$ is a monotonically increasing function and therefore f_1 is a convex function.

Second, we can see that since:

$\frac{d}{dz}f_2(z) = \frac{d}{dz}[-\log(\exp(-z)/(1 + \exp(-z)))]$

$= \frac{d}{dz}[\log(1 + \exp(-z)) + z]$

$= \sigma(z)$

the derivative of the function $f_2(z)$ is also monotonically increasing and therefore f_2 is a convex function.

4.5 Handling multiple classes

So far, we have focused on the binary classification case, with only two possible classes. But what can we do if we have multiple possible classes (e.g., we want to predict the genre of a movie)? There are two basic strategies:

• Train multiple binary classifiers using different subsets of our data and combine their outputs to make a class prediction.

• Directly train a multi-class classifier using a hypothesis class that is a generalization of logistic regression, using a one-hot output encoding and NLL loss.

The method based on NLL is in wider use, especially in the context of neural networks, and is explored here. In the following, we will assume that we have a data set D in which the inputs $x^{(i)} \in \mathbb{R}^d$ but the outputs $y^{(i)}$ are drawn from a set of K classes $\{c_1, \ldots, c_K\}$. Next, we extend the idea of NLL directly to multi-class classification with K classes, where the training label is represented with what is called a one-hot vector $y = [y_1, \ldots, y_K]^T$, where $y_k = 1$ if the example is of class k and $y_k = 0$ otherwise.

Now, we have a problem of mapping an input $x^{(i)}$ that is in $\mathbb{R}^d$ into a K-dimensional output. Furthermore, we would like this output to be interpretable as a discrete probability distribution over the possible classes, which means the elements of the output vector have to be non-negative (greater than or equal to 0) and sum to 1.

We will do this in two steps. First, we will map our input $x^{(i)}$ into a vector value $z^{(i)} \in \mathbb{R}^K$ by letting θ be a whole $d \times K$ matrix of parameters, and θ_0 be a $K \times 1$ vector, so that:

$z = \theta^T x + \theta_0$

[Note: Let's check dimensions! θ^T is $K \times d$ and x is $d \times 1$, and θ_0 is $K \times 1$, so z is $K \times 1$ and we're good!]

Next, we have to extend our use of the sigmoid function to the multi-dimensional softmax function, that takes a whole vector $z \in \mathbb{R}^K$ and generates:

$g = \text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_K)/\sum_i \exp(z_i) \end{bmatrix}$

which can be interpreted as a probability distribution over K items. To make the final prediction of the class label, we can then look at g, find the most likely probability over these K entries in g, (i.e. find the largest entry in g,) and return the corresponding index as the "one-hot" element of 1 in our prediction.

Putting these steps together, our hypotheses will be:

$h(x; \theta, \theta_0) = \text{softmax}(\theta^T x + \theta_0)$

Now, we retain the goal of maximizing the probability that our hypothesis assigns to the correct output y_k for each input x. We can write this probability, letting g stand for our "guess", h(x), for a single example (x, y) as:

$\prod_{k=1}^K g_k^{y_k}$

The negative log of the probability that we are making a correct guess is, then, for one-hot vector y and probability distribution vector g:

$L_{\text{nllm}}(g, y) = -\sum_{k=1}^K y_k \cdot \log(g_k)$

We'll call this NLLM for negative log likelihood multiclass. It is also worth noting that the NLLM loss function is also convex; however, we will omit the proof.

4.6 Prediction accuracy and validation

The performance of a classifier is often characterized by its accuracy, which is the percentage of a data set that it predicts correctly in the case of 0-1 loss. We can see that accuracy of hypothesis h on data D is the fraction of the data set that does not incur any loss:

$A(h; D) = 1 - \frac{1}{n}\sum_{i=1}^n L_{01}(g^{(i)}, y^{(i)})$

where $g^{(i)}$ is the final guess for one class or the other that we make from $h(x^{(i)})$, e.g., after thresholding. It's noteworthy here that we use a different loss function for optimization than for evaluation. This is a compromise we make for computational ease and efficiency.

CHAPTER 5

Feature representation

Linear regression and classification are powerful tools, but in the real world, data often exhibit non-linear behavior that cannot immediately be captured by the linear models which we have built so far. For example, suppose the true behavior of a system (with d = 2) looks like this wavelet:

[Note: This plot is of the so-called jinc function $J_1(\rho)/\rho$ for $\rho^2 = x_1^2 + x_2^2$]

Such behavior is actually ubiquitous in physical systems, e.g., in the vibrations of the surface of a drum, or scattering of light through an aperture. However, no single hyperplane would be a very good fit to such peaked responses!

A richer class of hypotheses can be obtained by performing a non-linear feature transformation $\phi(x)$ before doing the regression. That is, $\theta^T x + \theta_0$ is a linear function of x, but $\theta^T\phi(x) + \theta_0$ is a non-linear function of x, if ϕ is a non-linear function of x.

5.1 Gaining intuition about feature transformations

Let's look at an example data set that starts in 1-D. These points are not linearly separable, but consider the transformation $\phi(x) = [x, x^2]^T$.

[Note: What's a linear separator for data in 1D? A point!]

A linear separator in ϕ space is a nonlinear separator in the original space! Let's see how this plays out in our simple example. Consider the separator $x^2 - 1 = 0$, which labels the half-plane $x^2 - 1 > 0$ as positive.

5.2 Systematic feature construction

5.2.1 Polynomial basis

If the features in your problem are already naturally numerical, one systematic strategy for constructing a new feature space is to use a polynomial basis. The idea is that, if you are using the kth-order basis (where k is a positive integer), you include a feature for every possible product of k different dimensions in your original input.

Here is a table illustrating the kth order polynomial basis for different values of k:

Order | d = 1 | in general (d > 1)

---|---|---

0 | $[1]$ | $[1]$

1 | $[1, x]^T$ | $[1, x_1, \ldots, x_d]^T$

2 | $[1, x, x^2]^T$ | $[1, x_1, \ldots, x_d, x_1^2, x_1x_2, \ldots]^T$

3 | $[1, x, x^2, x^3]^T$ | $[1, x_1, \ldots, x_1^2, x_1x_2, \ldots, x_1x_2x_3, \ldots]^T$

[Note: Specifically, this example uses $[1, x_1, x_2, x_1^2 + x_2^2, (x_1^2 + x_2^2)^2, (x_1^2 + x_2^2)^4]^T$]

5.2.2 Radial basis functions

For any particular point p in the input space X, we can construct a feature f_p which takes any element $x \in X$ and returns a scalar value that is related to how far x is from the p we started with.

Let's start with the basic case, in which $X = \mathbb{R}^d$. Then we can define:

$f_p(x) = e^{-\beta\|p-x\|^2}$

This function is maximized when p = x and decreases exponentially as x becomes more distant from p.

Given a dataset D containing n points, we can make a feature transformation ϕ that maps points in our original space, $\mathbb{R}^d$, into points in a new space, $\mathbb{R}^n$. It is defined as follows:

$\phi(x) = [f_{x^{(1)}}(x), f_{x^{(2)}}(x), \ldots, f_{x^{(n)}}(x)]^T$

5.3 Hand-constructing features for real domains

5.3.1 Discrete features

Getting a good encoding of discrete features is particularly important. The main encoding strategies are:

• Numeric: Assign each value a number, say $1.0/k, 2.0/k, \ldots, 1.0$

• Thermometer code: For ordered values 1,...,k, use a vector of length k binary variables

• Factored code: Decompose values into separate features

• One-hot code: Use a vector of length k, with 1.0 in the jth position

• Binary code: (Not recommended) Use binary representation

5.3.2 Text

The bag of words (BOW) model: Let d be the number of words in vocabulary, create a binary vector of length d, where element j has value 1.0 if word j occurs in the document.

5.3.3 Numeric values

For numeric features:

• Scale to range $[-1, +1]$

• Standardize: $\phi(x) = \frac{x - \bar{x}}{\sigma}$, where $\bar{x}$ is the average and σ is the standard deviation

• Consider polynomial basis transformations

[Note: Such standard variables are often known as "z-scores," for example, in the social sciences.]

CHAPTER 6

Neural Networks

You've probably been hearing a lot about "neural networks." Now that we have several useful machine-learning concepts (hypothesis classes, classification, regression, gradient descent, regularization, etc.) we are well equipped to understand neural networks in detail.

[Note: As with many good ideas in science, the basic idea for how to train non-linear neural networks with gradient descent was independently developed by more than one researcher.]

We can view neural networks from several different perspectives:

View 1: An application of stochastic gradient descent for classification and regression with a potentially very rich hypothesis class.

View 2: A brain-inspired network of neuron-like computing elements that learn distributed representations.

View 3: A method for building applications that make predictions based on huge amounts of data in very complex domains.

6.1 Basic element

The basic element of a neural network is a "neuron." We will also sometimes refer to a neuron as a "unit" or "node."

[Note: Sorry for changing our notation here. We were using d as the dimension of the input, but we are trying to be consistent here with many other accounts of neural networks.]

It is a non-linear function of an input vector $x \in \mathbb{R}^m$ to a single output value $a \in \mathbb{R}$. It is parameterized by a vector of weights $(w_1, \ldots, w_m) \in \mathbb{R}^m$ and an offset or threshold $w_0 \in \mathbb{R}$.

The function represented by the neuron is expressed as:

$a = f(z) = f\left(\left(\sum_{j=1}^m x_jw_j\right) + w_0\right) = f(w^T x + w_0)$

Given a loss function $L(\text{guess}, \text{actual})$ and a dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, we can do (stochastic) gradient descent, adjusting the weights w, w_0 to minimize:

$J(w, w_0) = \sum_i L(NN(x^{(i)}; w, w_0), y^{(i)})$

6.2 Networks

6.2.1 Single layer

A layer has input $x \in \mathbb{R}^m$ and output (also known as activation) $a \in \mathbb{R}^n$. For a layer with m inputs, n units, and n outputs:

• W is an $m \times n$ matrix

• W_0 is an $n \times 1$ column vector

• X, the input, is an $m \times 1$ column vector

• $Z = W^TX + W_0$, the pre-activation, is an $n \times 1$ column vector

• A, the activation, is an $n \times 1$ column vector

The output vector is:

$A = f(Z) = f(W^TX + W_0)$

6.2.2 Many layers

For layer l:

- m_l is number of inputs

- n_l is number of outputs

- W_l and W_0^l are of shape $m_l \times n_l$ and $n_l \times 1$

- $m_l = n_{l-1}$

- A_{l-1} is of shape $m_l \times 1$ or $n_{l-1} \times 1$

[Note: It is technically possible to have different activation functions within the same layer, but for convenience in specification and implementation, we generally have the same activation function within a layer.]

The pre-activation outputs are the $n_l \times 1$ vector:

$Z^l = W_l^T A^{l-1} + W_0^l$

and the activation outputs are the $n_l \times 1$ vector:

$A^l = f^l(Z^l)$

6.3 Choices of activation function

If we let f be the identity, then in a network with L layers:

$A^L = W_L^T A^{L-1} = W_L^T W_{L-1}^T \cdots W_1^T X$

So:

$A^L = W_{total}X$

which is a linear function of X! Having all those layers did not change the representational capacity of the network: the non-linearity of the activation function is crucial.

The step function is defined as:

$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$

Rectified linear unit (ReLU):

$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} = \max(0,z)$

Sigmoid function (Also known as a logistic function. This can sometimes be interpreted as probability, because for any value of z the output is in (0,1)):

$\sigma(z) = \frac{1}{1 + e^{-z}}$

Hyperbolic tangent (Always in the range (-1,1)):

$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Softmax function (Takes a whole vector $Z \in \mathbb{R}^n$ and generates as output a vector $A \in (0,1)^n$ with the property that $\sum_{i=1}^n A_i = 1$, which means we can interpret it as a probability distribution over n items):

$\text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_n)/\sum_i \exp(z_i) \end{bmatrix}$

6.4 Loss functions and activation functions

Different loss functions make different assumptions about the range of values they will get as input. When designing a neural network, matching loss functions with the activation function in the last layer, f^L, is important:

Loss | f^L | task

---|---|---

squared | linear | regression

NLL | sigmoid | binary classification

NLLM | softmax | multi-class classification

6.5 Error back-propagation

For SGD with a training example (x, y), we need to compute $\nabla_W L(NN(x; W), y)$, where W represents all weights W^l, W_0^l in all the layers l = (1, ..., L).

[Note: Remember the chain rule! If $a = f(b)$ and $b = g(c)$, so that $a = f(g(c))$, then $\frac{da}{dc} = \frac{da}{db} \cdot \frac{db}{dc} = f'(b)g'(c) = f'(g(c))g'(c)$.]

6.5.1 First, suppose everything is one-dimensional

For layer l:

$a^l = f^l(z^l), z^l = w^l a^{l-1} + w_0^l$

For SGD, we want to compute $\partial L(NN(x; W), y)/\partial w^l$ and $\partial L(NN(x; W), y)/\partial w_0^l$ for each layer l and each data point (x, y).

For l = L:

$\frac{\partial \text{loss}}{\partial w^L} = \frac{\partial \text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} = \frac{\partial \text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot a^{L-1}$

For general l:

$\frac{\partial \text{loss}}{\partial w^l} = \frac{\partial \text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdots \frac{\partial z^{l+1}}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial w^l}$

$= \frac{\partial \text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot w^L \cdot (f^{L-1})'(z^{L-1}) \cdots w^{l+1} \cdot (f^l)'(z^l) \cdot a^{l-1}$

$= \frac{\partial \text{loss}}{\partial z^l} \cdot a^{l-1}$

6.5.2 The general case

For the matrix case:

$\frac{\partial \text{loss}}{\partial W^l} = A^{l-1} \left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T$

$\frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot \frac{\partial Z^{l+1}}{\partial A^l} \cdots \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L}$

$= \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdots \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L}$

6.5.3 Derivations for the general case

The key trick is to break every equation down to its scalar meaning. The (i,j) element of $\frac{\partial \text{loss}}{\partial W^l}$ is $\frac{\partial \text{loss}}{\partial W_{i,j}^l}$. The loss is a function of the elements of Z^l, and the elements of Z^l are a function of the $W_{i,j}^l$. With n^l elements of Z^l, we can use the chain rule to write:

$\frac{\partial \text{loss}}{\partial W_{i,j}^l} = \sum_{k=1}^{n^l} \frac{\partial \text{loss}}{\partial Z_k^l}\frac{\partial Z_k^l}{\partial W_{i,j}^l}$ (6.7)

Remember that $Z^l = (W^l)^T A^{l-1} + W_0^l$. We can write one element of the Z^l vector as:

$Z_b^l = \sum_{a=1}^{m^l} W_{a,b}^l A_a^{l-1} + (W_0^l)_b$

It follows that $\frac{\partial Z_k^l}{\partial W_{i,j}^l}$ will be zero except when k = j, so we can rewrite Eq. 6.7 as:

$\frac{\partial \text{loss}}{\partial W_{i,j}^l} = \frac{\partial \text{loss}}{\partial Z_j^l}\frac{\partial Z_j^l}{\partial W_{i,j}^l} = \frac{\partial \text{loss}}{\partial Z_j^l}A_i^{l-1}$ (6.8)

6.5.4 Reflecting on backpropagation

[Note: We could call this "blame propagation". Think of loss as how mad we are about the prediction just made. Then $\frac{\partial \text{loss}}{\partial A^L}$ is how much we blame A^L for the loss. The last module has to take in $\frac{\partial \text{loss}}{\partial A^L}$ and compute $\frac{\partial \text{loss}}{\partial Z^L}$, which is how much we blame Z^L for the loss.]

Each module must provide these methods:

- forward: $u \rightarrow v$

- backward: $u, v, \frac{\partial L}{\partial v} \rightarrow \frac{\partial L}{\partial u}$

- weight grad: $u, \frac{\partial L}{\partial v} \rightarrow \frac{\partial L}{\partial W}$ (only needed for modules with weights W)

6.6 Training

Algorithm: SGD-NEURAL-NET($D_n, T, L,(m^1,...,m^L),(f^1,...,f^L), \text{Loss}$)

```
1  for l = 1 to L
2      W_{ij}^l \sim \text{Gaussian}(0, 1/m^l)
3      W_0^l_j \sim \text{Gaussian}(0, 1)
4  for t = 1 to T
5      i = \text{random sample from } \{1,...,n\}
6      A^0 = x^{(i)}
7      // forward pass to compute the output A^L
8      for l = 1 to L
9          Z^l = (W^l)^T A^{l-1} + W_0^l
10         A^l = f^l(Z^l)
```

```
11    loss = \text{Loss}(A^L, y^{(i)})

12    for l = L to 1:

13        // error back-propagation

14        \frac{\partial \text{loss}}{\partial A^l} = \begin{cases}

            \frac{\partial Z^{l+1}}{\partial A^l} \cdot \frac{\partial \text{loss}}{\partial Z^{l+1}} &
\text{if } l < L \\

            \frac{\partial \text{loss}}{\partial A^L} & \text{otherwise}

          \end{cases}

15        \frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot \frac{\partial
\text{loss}}{\partial A^l}

16        // compute gradient with respect to weights

17        \frac{\partial \text{loss}}{\partial W^l} = A^{l-1} \cdot (\frac{\partial \text{loss}}{\partial
Z^l})^T

18        \frac{\partial \text{loss}}{\partial W_0^l} = \frac{\partial \text{loss}}{\partial Z^l}

19        // stochastic gradient descent update

20        W^l = W^l - \eta(t) \cdot \frac{\partial \text{loss}}{\partial W^l}

21        W_0^l = W_0^l - \eta(t) \cdot \frac{\partial \text{loss}}{\partial W_0^l}
```

Initializing W is important. We need to:

1. Initialize weights to random values to break symmetry

2. Keep initial weights small to avoid activation function saturation

3. Use mean 0 and standard deviation $(1/m)$ where m is the number of inputs to the unit

As long as we pick points uniformly at random from the data set, and decrease η at an appropriate rate, we are guaranteed, with high probability, to converge to at least a local optimum.

For a mini-batch of size K, we select K distinct data points uniformly at random from the data set and do the update based on their contributions to the gradient:

$W_t = W_{t-1} - \eta\sum_{i=1}^K \nabla_W L(h(x^{(i)}; W_{t-1}), y^{(i)})$

Algorithm: MINI-BATCH-SGD(NN, data, K)

```
1  n = length(data)
2  while not done:
3      RANDOM-SHUFFLE(data)
4      for i = 1 to ⌈n/K⌉
5          BATCH-GRADIENT-UPDATE(NN, data[(i-1)K : iK])
```

[Note: In line 4, $\lceil \cdot \rceil$ is the ceiling function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.]

6.7.2 Adaptive step-size

Looking at Eq. 6.6, we can see that the output gradient is multiplied by all the weight matrices of the network and is "fed back" through all the derivatives of all the activation functions. This can lead to:

- Exploding gradients: back-propagated gradient is too big

- Vanishing gradients: back-propagated gradient is too small

6.8 Regularization

6.8.1 Methods related to ridge regression

Three strategies with similar effects:

1. Early stopping: Train on training set, evaluate loss on validation set at each epoch

2. Weight decay: Penalize the norm of weights, using objective:

   $J(W) = \sum_{i=1}^n L(NN(x^{(i)}), y^{(i)}; W) + \lambda\|W\|^2$

   Leading to update:

   $W_t = W_{t-1}(1 - 2\lambda\eta) - \eta\nabla_W L(NN(x^{(i)}), y^{(i)}; W_{t-1})$

3. Adding noise to training data

6.8.2 Dropout

During training:

- For each example, randomly set $a_j^l = 0$ with probability p

- In forward pass: $a^l = f(z^l) * d^l$

  where $*$ is component-wise product and d^l is a vector of 0's and 1's drawn randomly with probability p

- During prediction: multiply all weights by p

6.8.3 Batch normalization

Addresses covariate shift by standardizing input values for each mini-batch:

- Subtract mean

- Divide by standard deviation

- Applied to z^l, between product with W^l and input to f^l

[Note: Originally justified for covariate shift, but may improve performance through regularization effects similar to dropout and noise addition.]

The key methods have complementary strengths:

- Mini-batches balance computation efficiency and gradient accuracy

- Adaptive step-sizes handle varying gradient magnitudes across layers

- Regularization methods prevent overfitting through different mechanisms:

  - Weight decay/early stopping control model complexity

  - Dropout enforces distributed representations

  - Batch normalization stabilizes training and provides mild regularization

CHAPTER 7

Convolutional Neural Networks

So far, we have studied fully connected neural networks, where all units at one layer are connected to all units in the next layer. When we have prior knowledge about our problem domain, we can build it into the network structure. This can:

1. Save computation time

2. Reduce required training data

3. Improve generalization

One important application domain is signal processing. For image processing, we can leverage two key properties:

• Spatial locality: The set of pixels we need to consider are near one another in the image.

[Note: So we won't have to consider some combination of pixels in the four corners of the image to detect features.]

• Translation invariance: The pattern of pixels characterizing a feature is the same regardless of location.

[Note: Cats don't look different if they're on the left or right side of the image.]

7.1 Filters

An image filter is a function that takes in a local spatial neighborhood of pixel values and detects patterns in that data.

[Note: In AI/ML/CS/Math, the word "filter" has multiple meanings: it can describe a temporal process (like moving averages) and even somewhat esoteric algebraic structures.]

For a 1-dimensional binary "image" and a filter F of size two, with X as the original image of size d, pixel i of the output image Y is:

$Y_i = F \cdot (X_{i-1}, X_i)$

Example with filters:

- Let filter $F_1 = (-1, +1)$

- Let filter $F_2 = (-1, +1, -1)$

For 2D images:

- Input image: $n \times n$

- Color image: $n \times n \times 3$ tensor (RGB channels)

- After applying k filters: $n \times n \times k$ tensor

- Each filter creates a new "channel"

Filter operations:

1. Original filters applied to input image

2. Result is a tensor (3D data structure)

3. Next set of filters are 3-dimensional:

   - Applied to sub-range of row/column indices

   - Applied across all channels

[Note: There are now many useful neural-network software packages, such as TensorFlow and PyTorch that make operations on tensors easy.]

2D Filtering Example:

- Two 3×3 filters in first layer (f_1 and f_2)

- Input image: $n \times n$

- After filtering: $n \times n \times 2$ tensor

- Tensor filter combines horizontal and vertical features

- Results in single $n \times n$ output image

[Note: When we have a color image as input, we treat it as having three channels, hence as an $n \times n \times 3$ tensor.]

Key CNN Concepts:

1. Filter banks correspond to neural network layers

2. Filter values are the weights (plus bias) trained using gradient descent

3. Same weights are used multiple times (weight sharing)

4. This allows expression of translation-invariant transformations with fewer parameters

7.4 Backpropagation in a simple CNN

Consider a simple CNN architecture with:

- One-dimensional single-channel image X of size $n \times 1 \times 1$

- Single filter W^1 of size $k \times 1 \times 1$ (omitting bias)

- ReLU activation

- Fully-connected layer with weights W^2

- No final activation function

Forward pass equations (assuming k is odd):

$Z^1_i = (W^1)^T A^0_{[i-\lfloor k/2 \rfloor:i+\lfloor k/2 \rfloor]}$

$A^1 = \text{ReLU}(Z^1)$

$A^2 = Z^2 = (W^2)^T A^1$

$L_{\text{square}}(A^2, y) = (A^2 - y)^2$

7.4.1 Weight update

To update weights in filter W^1, we need:

$\frac{\partial \text{loss}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1}\frac{\partial A^1}{\partial Z^1}\frac{\partial \text{loss}}{\partial A^1}$

where:

1. $\frac{\partial Z^1}{\partial W^1}$ is the $k \times n$ matrix where $\frac{\partial Z^1_i}{\partial W^1_j} = X_{i-\lfloor k/2 \rfloor+j-1}$

   Example: for i = 10 and k = 5, column 10 contains $[X_8, X_9, X_{10}, X_{11}, X_{12}]$

2. $\frac{\partial A^1}{\partial Z^1}$ is the $n \times n$ diagonal matrix where:

   $\frac{\partial A^1_i}{\partial Z^1_i} = \begin{cases} 1 & \text{if } Z^1_i > 0 \\ 0 & \text{otherwise} \end{cases}$

3. $\frac{\partial \text{loss}}{\partial A^1} = \frac{\partial \text{loss}}{\partial A^2}\frac{\partial A^2}{\partial A^1} = 2(A^2 - y)W^2$, an $n \times 1$ vector

The final gradient has shape $k \times 1$.

7.4.2 Max pooling

For backpropagation through max-pooling, consider:

$y = \max(a_1, a_2)$

If $a_1 > a_2$:

- Error at y propagates entirely to network computing a_1

- Network computing a_2 receives no gradient

- Only weights computing the maximum value are updated

[Note: This means max pooling creates a kind of "competition" between inputs, where only the "winner" gets to influence the network's learning.]

Key points:

1. Gradients flow through the paths that contributed to the maximum

2. Zero gradient for non-maximum inputs

3. This creates sparsity in the gradient flow

4. Can lead to more selective feature detection

CHAPTER 8

Transformers

Transformers are a very recent family of architectures that have revolutionized fields like natural language processing (NLP), image processing, and multi-modal generative AI. Transformers were originally introduced in the field of NLP in 2017, as an approach to process and understand human language. Human language is inherently sequential in nature (e.g., characters form words, words form sentences, and sentences form paragraphs and documents). Prior to the advent of the transformers architecture, recurrent neural networks (RNNs) briefly dominated the field for their ability to process sequential information (RNNs are described in Appendix C for reference). However, RNNs, like many other architectures, processed sequential information in an iterative/sequential fashion, whereby each item of a sequence was individually processed one after another. Transformers offer many advantages over RNNs, including their ability to process all items in a sequence in a parallel fashion (as do CNNs).

Like CNNs, transformers factorize the signal processing problem into stages that involve independent and identically processed chunks. However, they also include layers that mix information across the chunks, called attention layers, so that the full pipeline can model dependencies between the chunks.

In this chapter, we describe transformers from the bottom up. We start with the idea of embeddings and tokens (Section 8.1). We then describe the attention mechanism (Section 8.2).

And finally we then assemble all these ideas together to arrive at the full transformer architecture in Section 8.3.

8.1 Vector embeddings and tokens

Before we can understand the attention mechanism in detail, we need to first introduce a new data structure and a new way of thinking about neural processing for language.

The field of NLP aims to represent words with vectors (aka word embeddings) such that they capture semantic meaning. More precisely, the degree to which any two words are related in the 'real-world' to us humans should be reflected by their corresponding vectors (in terms of their numeric values). So, words such as 'dog' and 'cat' should be represented by vectors that are more similar to one another than, say, 'cat' and 'table' are. Nowadays, it's also typical for every individual occurrence of a word to have its own distinct representation/vector. So, a story about a dog may mention the word 'dog' a dozen times, with each vector being slightly different based on its context in the sentence and story at large.

To measure how similar any two word embeddings are (in terms of their numeric values) it is common to use cosine similarity as the metric:

$\frac{u^T v}{|u| |v|} = \cos \langle u, v \rangle$

where $|u|$ and $|v|$ are the lengths of the vectors, and $\langle u, v \rangle$ is the angle between u and v. The cosine similarity is +1 when u = v, zero when the two vectors are perpendicular to each other, and −1 when the two vectors are diametrically opposed to each other. Thus, higher values correspond to vectors that are numerically more similar to each other.

While word embeddings – and various approaches to create them – have existed for decades, the first approach that produced astonishingly effective word embeddings was word2vec in 2012. This revolutionary approach was the first highly-successful approach of applying deep learning to NLP, and it enabled all subsequent progress in the field, including Transformers. The details of word2vec are beyond the scope of this course, but we note two facts: (1) it created a single word embedding for each distinct word in the training corpus (not on a per-occurrence basis); (2) it produced word embeddings that were so useful, many relationships between the vectors corresponded with real-world semantic relatedness. For example, when using Euclidean distance as a distance metric between two vectors, word2vec produced word embeddings with properties such as (where v_{word} is the vector for word):

$v_{paris} - v_{france} + v_{italy} \approx v_{rome}$

This corresponds with the real-world property that Paris is to France what Rome is to Italy. This incredible finding existed not only for geographic words but all sorts of real-world concepts in the vocabulary. Nevertheless, to some extent, the exact values in each embedding is arbitrary, and what matters most is the holistic relation between all embeddings, along with how performant/useful they are for the exact task that we care about.

For example, an embedding may be considered good if it accurately captures the conditional probability for a given word to appear next in a sequence of words. You probably have a good idea of what words might typically fill in the blank at the end of this sentence:

After the rain, the grass was ___

Or a model could be built that tries to correctly predict words in the middle of sentences:

The child fell ___ during the long car ride

The model can be built by minimizing a loss function that penalizes incorrect word guesses, and rewards correct ones. This is done by training a model on a very large corpus of written material, such as all of Wikipedia, or even all the accessible digitized written materials produced by humans.

While we will not dive into the full details of tokenization, the high-level idea is straightforward: the individual inputs of data that are represented and processed by a model are referred to as tokens. And, instead of processing each word as a whole, words are typically split into smaller, meaningful pieces (akin to syllables). Thus, when we refer to tokens, know that we're referring to each individual input, and that in practice, nowadays, they tend to be sub-words (e.g., the word 'talked' may be split into two tokens, 'talk' and 'ed'.

8.2 Query, key, value, and attention

Attention is a strategy for processing global information efficiently, focusing just on the parts of the signal that are most salient to the task at hand. What we present below is the so-called "dot-product attention" mechanism; there can be other variants that involve more complex attention functions.

It might help our understanding of the "attention" mechanism to think about a dictionary look-up scenario. Consider a dictionary with keys k mapping to some values v(k). For example, let k be the name of some foods, such as pizza, apple, sandwich, donut, chili, burrito, sushi, hamburger, ... The corresponding values may be information about the food, such as where it is available, how much it costs, or what its ingredients are.

Suppose that instead of looking up foods by a specific name, we wanted to query by cuisine, e.g., "mexican" foods. Clearly, we cannot simply look for the word "mexican" among the dictionary keys, since that word is not a food. What does work is to utilize again the idea of finding "similarity" between vector embeddings of the query and the keys. The end result we'd hope to get, is a probability distribution over the foods, p(k|q) indicating which are best matches for a given query q. With such a distribution, we can look for keys that are semantically close to the given query.

More concretely, to get such distribution, we follow these steps: First, embed the word we are interested in ("mexican" in our example) into a so-called query vector, denoted simply as $q \in \mathbb{R}^{d_k \times 1}$ where d_k is the embedding dimension.

Next, suppose our given dictionary has n number of entries/entries, we embed each one of these into a so-called key vector. In particular, for each of the j^{th} entry in the dictionary, we produce a $k_j \in \mathbb{R}^{d_k \times 1}$ key vector, where j = 1, 2, 3, ..., n.

We can then obtain the desired probability distribution using a softmax (see Chapter 6) applied to the inner-product between the key and query:

$p(k|q) = \text{softmax}[q^T k_1; q^T k_2; q^T k_3; \ldots; q^T k_n]$

This vector-based lookup mechanism has come to be known as "attention" in the sense that p(k|q) is a conditional probability distribution that says how much attention should be given to the key k_j for a given query q.

In other words, the conditional probability distribution p(k|q) gives the "attention weights," and the weighted average value

$\sum_j p(k_j|q) v_j$

is the "attention output."

The meaning of this weighted average value may be ambiguous when the values are just words. However, the attention output really becomes meaningful when the value are projected in some semantic embedding space (and such projection are typically done in transformers via learned embedding weights).

The same weighted-sum idea generalizes to multiple query, key, and values. In particular, suppose there are n_q number of queries, n_k number of keys (and therefore n_k number of values), one can compute an attention matrix

$A = \begin{bmatrix}

\text{softmax}[q_1^T k_1 \quad q_1^T k_2 \quad \cdots \quad q_1^T k_{n_k}]/\sqrt{d_k} \\

\text{softmax}[q_2^T k_1 \quad q_2^T k_2 \quad \cdots \quad q_2^T k_{n_k}]/\sqrt{d_k} \\

\vdots \\

\text{softmax}[q_{n_q}^T k_1 \quad q_{n_q}^T k_2 \quad \cdots \quad q_{n_q}^T k_{n_k}]/\sqrt{d_k}

\end{bmatrix}$

Here, softmax_j is a softmax over the n_k-dimensional vector indexed by j, so in Eq. 8.2 this means a softmax computed over keys. In this equation, the normalization by $\sqrt{d_k}$ is done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing d_k, making it difficult for (overall) training.

Let α_{ij} be the entry in i^{th} row and j^{th} column in the attention matrix A. Then α_{ij} helps answer the question "which tokens $x^{(j)}$ help the most with predicting the corresponding output token $y^{(i)}$?" The attention output is given by a weighted sum over the values:

$y^{(i)} = \sum_{j=1}^n \alpha_{ij}v_j$

8.2.1 Self Attention

Self-attention is an attention mechanism where the keys, values, and queries are all generated from the same input.

At a very high level, typical transformer with self-attention layers maps $\mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times d}$. In particular, the transformer takes in data (a sequence of tokens) $X \in \mathbb{R}^{n \times d}$ and for each token $x^{(i)} \in \mathbb{R}^{d \times 1}$, it computes (via learned projection, to be discussed in Section 8.3.1), a query $q_i \in \mathbb{R}^{d_q \times 1}$, key $k_i \in \mathbb{R}^{d_k \times 1}$, and value $v_i \in \mathbb{R}^{d_v \times 1}$. In practice, $d_q = d_k = d_v$ and we often denote all three embedding dimension via a unified d_k. Note that d_k differs from d: d is the dimension of raw input token $\in \mathbb{R}^{d_q \times 1}$

The self-attention layer then take in these query, key, and values, and compute a self-attention matrix:

$A = \begin{bmatrix} \text{softmax}[q_1^T k_1 \quad q_1^T k_2 \quad \cdots \quad q_1^T k_{n_k}]/\sqrt{d_k} \ \text{softmax}[q_2^T k_1 \quad q_2^T k_2 \quad \cdots \quad q_2^T k_{n_k}]/\sqrt{d_k} \ \text{softmax}[q_{n_q}^T k_1 \quad q_{n_q}^T k_2 \quad \cdots \quad q_{n_q}^T k_{n_k}]/\sqrt{d_k} \end{bmatrix}$

Here, softmaxj is a softmax over the nk-dimensional vector indexed by j, so in Eq. 8.2

this means a softmax computed over keys. In this equation, the normalization by √ dk is

done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing d_k, making it difficult for (overall) training.

Let α_{ij} be the entry in i^{th} row and j^{th} column in the attention matrix A. Then α_{ij} helps answer the question "which tokens $x^{(j)}$ help the most with predicting the corresponding output token $y^{(i)}$?" The attention output is given by a weighted sum over the values:

$y^{(i)} = \sum_{j=1}^n \alpha_{ij}v_j$

8.2.1 Self Attention

Self-attention is an attention mechanism where the keys, values, and queries are all generated from the same input.

At a very high level, typical transformer with self-attention layers maps $\mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times d}$. In particular, the transformer takes in data (a sequence of tokens) $X \in \mathbb{R}^{n \times d}$ and for each token $x^{(i)} \in \mathbb{R}^{d \times 1}$, it computes (via learned projection, to be discussed in Section 8.3.1), a query $q_i \in \mathbb{R}^{d_q \times 1}$, key $k_i \in \mathbb{R}^{d_k \times 1}$, and value $v_i \in \mathbb{R}^{d_v \times 1}$. In practice, $d_q = d_k = d_v$ and we often denote all three embedding dimension via a unified d_k.

Note that d_k differs from d as d is the dimension of raw input token.

The self-attention layer then take in these query, key, and values, and compute a self-attention matrix:

$A = \begin{bmatrix} \text{softmax}[q_1^T k_1 \quad q_1^T k_2 \quad \cdots \quad q_1^T k_{n_k}]/\sqrt{d_k} \ \text{softmax}[q_2^T k_1 \quad q_2^T k_2 \quad \cdots \quad q_2^T k_{n_k}]/\sqrt{d_k} \end{bmatrix}$

Study Question: We have five colored tokens in the diagram above (gray, blue, orange, green, yellow). Could you spell out the correspondences between the color and input, query, keys, values, output?

Note that the size of the output is the same as the size of the input. Also, query that there is no apparent notion of ordering of the input words in the depicted structure. Positional information can be added by encoding a number for each q_i (e.g., q_i = "token's position"). Further note that unlike a CNN network where a kernel is fixed and applied to each token, here each token has a given query and must pay attention to all other tokens in the sequence. The token used for the query is used for a key or used for a value.

More generally, a mask may be applied to limit which tokens are used in the attention computation. For example, one commonly mask limits the attention computation to future tokens (tokens not yet seen). This turns out to be useful in (language-model) applications where the transformer is being used to predict one token at a time.

Continuing with the end of Image 2 and Image 3:

A transformer is the composition of a number of transformer blocks, each of which has multiple attention heads. At a very high-level, the goal of a transformer block is to develop a really rich, useful representation for each input token, all for the sake of being highly-performant for whatever task it is trained to learn.

Rather than depicting the transformer graphically, it is worth returning to the beauty of the math equations!

8.3.1 Learned embedding

For simplicity, we assume the transformer internally uses self-attention. Full generalization favors work similarly.

Let us have a parameterized function f that maps $\mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times d}$, where input data $X \in \mathbb{R}^{n \times d}$ is represented as a sequence of n tokens, with each token $x^{(i)} \in \mathbb{R}^d$.

The projection matrices (weights) W_q, W_k, W_v are to be learned, such that for each token, we'll project x into query, key, and value vectors: $q^{(i)} = x^{(i)}W_q$, $k^{(i)} = x^{(i)}W_k$, and $v^{(i)} = x^{(i)}W_v$ respectively, where these learned projection weights W_q, W_k, $W_v \in \mathbb{R}^{d \times d_k}$.

If we stack these query, key, value vectors into matrix-form, such that $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{n \times d_k}$, and $V \in \mathbb{R}^{n \times d_k}$, then we can more compactly write out the learned transformation from the tokens X:

$Q = XW_q$

$K = XW_k$

$V = XW_v$

These Q, K, V triple can be used to produce one (self)attention-layer output. One such layer requires the matrices described above.

One can have more than one "attention head," such that the queries, keys, and values are embedded via encoding matrices:

$q^{(h)} = XW_q^h$

$k^{(h)} = XW_k^h$

$v^{(h)} = XW_v^h$

and W_q^h, W_k^h, $W_v^h \in \mathbb{R}^{d \times d_k}$ where d_k is the size of the key/query embedding space, and $h \in \{1, \ldots, H\}$ is an index over "attention heads."

This is then standardized and combined with $X^{(1)}$ using a LayerNorm function (defined below) to become:

$u^{(l)} = \text{LayerNorm}(X^{(l)} + u^{(l)}_1v_h)$

with parameters $\gamma_l \in \mathbb{R}^d$

[Note: There's a footnote that says "*Layer normalization here follows the equation by John Thickstun."]

[There are also two blue note boxes:

1. "For each attention head h, we learn one set of W_q^h, W_k^h, W_v^h"

2. "$u^{(l)}$ is the $d \times 1$ value embedding vector that corresponds to the input token l."]

Let me transcribe this content with proper LaTeX formatting:

To get the final output, we follow the "intermediate output then layer norm" recipe again. In particular, we first get the transformer block output $z^{(l)}$ given by

$z^{(l)} = W_1 \text{ReLU}(W_2u^{(l)})$ (8.11)

with weights $W_1 \in \mathbb{R}^{d \times m}$ and $W_2 \in \mathbb{R}^{m \times d}$. This is then standardized and combined with $u^{(l)}$ to give the final output $x^{(l)}$:

$x^{(l)} = \text{LayerNorm}(u^{(l)} + z^{(l)}, \gamma_2)$ (8.12)

with parameters $\gamma_2, \beta_2 \in \mathbb{R}^d$. These vectors are then assembled (e.g., through parallel computation) to produce $z \in \mathbb{R}^{n \times d}$.

The LayerNorm function transforms a d-dimensional input z with parameters $\gamma, \beta \in \mathbb{R}^d$:

$\text{LayerNorm}(z, \gamma, \beta) = \gamma \frac{z - \mu_z}{\sigma_z} + \beta,$ (8.13)

where μ_z is the mean and σ_z the standard deviation of z:

$\mu_z = \frac{1}{d}\sum_{i=1}^d z_i$ (8.14)

$\sigma_z = \sqrt{\frac{1}{d}\sum_{i=1}^d(z_i - \mu_z)^2}$ (8.15)

Layer normalization is done to improve convergence stability during training.

The model parameters comprise the weight matrices W_q, W_k, W_v, W_1, W_2 and the LayerNorm parameters γ_1, γ_2, β_1, β_2. A transformer is the composition of L transformer blocks, each with its own parameters:

$f_{\theta} = \circ_{l=1}^L f_{\theta_l} (x) \in \mathbb{R}^{n \times d}$ (8.16)

The hyperparameters of this model are d, d_k, m, H, and L.

8.3.2 Variations and training

Many variants on this transformer structure exist. For example, the LayerNorm may be moved to other steps of the neural pipeline. Or a more sophisticated attention function may be employed instead of the simple dot product used in §8.2. Transformers may also be used for "transfer," for example, to process the input and a separate one to process the output. This is useful for tasks like translation, summarization, and cross-attention, where some input data are used to generate queries and other input data generate keys and values. Positional encoding and masking are also common, though they are left out of the basic model described here.

How are transformers trained? The number of parameters θ it can have is very large, making transformer models like GPT3 have many billions of parameters to move. A good deal of data is thus necessary to train such models, else the models may simply overfit small datasets.

Large language transformer models are thus generally done in two stages. A first "pre-training" stage employs a very large dataset to train the model to mimic patterns. This is done with unsupervised (or self-supervised) learning and unlabeled data. For example, the well-known BERT model was pre-trained using sentences with words masked. The model was trained to predict the masked words. BERT was also trained on sequences of

sentences, where the model was trained to predict whether two sentences are likely to be

contextually close together or not. The pre-training stage is generally very expensive.

The second “fine-tuning” stage trains the model for a specific task, such as classification

or question answering. This training stage can be relatively inexpensive, but it generally

requires labeled data.

CHAPTER 10

Markov Decision Processes

So far, most of the learning problems we have looked at have been supervised, that is, for each training example (x,y) we are told which value $y^{(i)}$ should correspond to a given $x^{(i)}$. (The one exception we will look at in Chapter 11 and also in Chapter 12.1 is the semi-supervised learning problems, in which we are given data and not expected outputs, and we will look at later.) However, it's worth pointing out one major difference at a very high level: in supervised learning, our goal is to learn a one-time static mapping to make predictions, whereas in RL the setup requires us to sequentially take actions and in the presence of uncertainty.This setup change necessitates additional mathematical and algorithmic tools for us to understand RL. Markov decision process (MDP) is precisely such a classical and fundamental tool.

10.1 Definition and value functions

Formally, a Markov decision process is (S,A,T,R,γ) where S is the state space, A is the action space, and:

1. $T: S \times A \times S \rightarrow \mathbb{R}$ is a transition model, where

$T(s, a, s') = P(s_t = s' | s_t = s, A_{t-1} = a)$

specifying a conditional probability distribution;

2. $R: S \times A \rightarrow \mathbb{R}$ is a reward function, where $R(s,a)$ specifies an immediate reward for taking action a when in state s; and

3. $\gamma \in [0, 1]$ is a discount factor, which we'll discuss in Section 10.1.2.

In this class, we assume the rewards are deterministic functions. Further, in the MDP chapter, we assume the state space and action space are finite (or fact, typically small).

[Note box: "The notation $s_t = s'$ and $s_t = s$ means a capital S_t is a random variable, and small s_t means it takes on some value. So s' here has no special meaning other than that it can take on values out of S as well."]

The following description of a simple machines's Markov decision process provides a concrete example of an MDP. The machine has three possible operations (actions): "wash", "paint", and "eject" (each with a corresponding button). Objects are put into the machine. Each time you press a button, something is done to the object inside (wash it, paint it, or eject it). Note that the machine has a simple camera inside that can clearly detect what is going on with the object and tell us about the state of the object: "dirty", "clean", "painted" or "ejected". For each action, this is what is done to the object:

* If you perform the "wash" operation on any object, whether it's dirty, clean, or painted, it will become "clean" with probability 0.9 and "dirty" otherwise.

Paint:

* If you perform the "paint" operation on a clean object, it will become nicely "painted" with probability 0.9. With probability 0.1, the paint mess but the object stays clean, and thus with probability 0.1, the machine dumps only that all over the object and it becomes "dirty".

* If you perform the "paint" operation on a "painted" object, it stays "painted" with probability 1.0.

* If you perform the "paint" operation on a "dirty" part, it stays "dirty" with probability 1.0.

Eject:

* If you perform an "eject" operation on any part, the part comes out of the machine and this fun game is over. The part remains "ejected" ever after without any further action.

These descriptions specify the transition model T, and the transition function for each action can be depicted as a state machine diagram. For example, here is the diagram for "wash":

[The diagram shows states connected by arrows with probabilities:

- "dirty" connects to "clean" with 0.9 probability and back to "dirty" with 0.1 probability

- "clean" connects to "clean" with 0.9 probability and to "dirty" with 0.1 probability

- "painted" connects to "clean" with 0.4 probability

- A state labeled "ejected" is also shown]

You get reward +10 for ejecting a painted object, reward 0 for ejecting a non-painted object, reward 0 for any action on an ejected object, and reward -1 otherwise. The discount factor γ will be completely 0.9. The specified reward factor.

Let me transcribe this content, preserving the text and converting mathematical notations to LaTeX:

A policy is a function $\pi: S \rightarrow A$ that specifies what action to take in each state. The policy π tells you what button to press at each moment in [the context of] this entire game. Before we talk about the initial state towards this overall goal. We describe how to evaluate how good a policy is, first in the finite horizon case (Section 10.1.1) when the total number of transition steps is finite. Then we consider the infinite horizon case (Section 10.1.2), when you don't know where the game will be over.

10.1.1 Finite-horizon value functions

The set up again is to maximize the expected total reward. There are several formulations that do form to make. Let's first consider the case where there is a finite horizon H, indicating the total number of steps/interactions that the agent will have with the MDP.

We seek to measure the goodness of a policy. We do so by defining for a given start state s and horizon h, the "horizon-h value" of a state, $V_h^{\pi}(s)$. We do this by induction on the horizon length, from the last step on up to h steps left remaining.

The base case is when there are no steps remaining, in which case no matter what state we are in, the value is 0:

$V_0^{\pi}(s) = 0$ (10.1)

Then, the value of a policy in state s at horizon h is equal to the reward it gets in that state plus the next state's expected horizon-h value, discounted by a factor γ. So, starting from horizon 1 and working our way up, in the general case, we have:

$V_1^{\pi}(s) = R(s, \pi(s))$ (10.2)

$V_2^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')V_1^{\pi}(s')$ (10.3)

$\vdots$

$V_h^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')V_{h-1}^{\pi}(s')$ (10.4)

The sum over s' is important: it describes all the next states that could be reached with average of their (h−1)-horizon values, weighted by the probability that the transition function from state s with the action chosen by the policy $\pi(s)$ assigns to arriving in state s', and discounted by γ.

[Note box: "In the induction case, we usually set the discount factor γ to 1."]

Study Question: What is $\sum_{s'} T(s, a, s')$ for any particular s and a?

Study Question: Convince yourself that Eqs. 10.1 and 10.3 are special cases of Eq. 10.4.

Then we can say that a policy π_1 is better than policy π_2 for horizon h, i.e., $\pi_1 >_h \pi_2$, if $V_h^{\pi_1}(s) > V_h^{\pi_2}(s)$ for some s and $V_h^{\pi_1}(s) \geq V_h^{\pi_2}(s)$ for all other s.

10.1.2 Infinite-horizon value functions

More practically, the size finite horizon is not known, i.e., when you don't know where the game will be over? This is related to [the] policy horizon problem: How does one evaluate the goodness of a policy in the infinite horizon case?

If we tried to just add up rewards from infinite steps, as we did before for infinite time, we could get in trouble. Imagine we got a reward of +1 at each step under one policy and a reward of +2 at each step under a different policy. Then the reward as the number of steps grows to each case keeps growing to become infinite in the limit of over and more steps.

10.1.2 Infinite-horizon value functions

More practically, the size finite horizon is not known, i.e., when you don't know where the game will be over? This is related to [the] policy horizon problem: How does one evaluate the goodness of a policy in the infinite horizon case?

If we tried to just add up rewards from infinite steps, as we did before for infinite time, we could get in trouble. Imagine we got a reward of +1 at each step under one policy and a reward of +2 at each step under a different policy. Then the reward as the number of steps grows to each case keeps growing to become infinite in the limit of over and more steps.

Let me transcribe this content, preserving text and formatting mathematical equations in LaTeX:

Even though it seems intuitive that the second policy should be better, we can't justify that by summing infinite rewards.

One standard approach to deal with this problem is to consider the discounted infinite horizon. We can prevent far from the finite-horizon case by adding a discount factor.

In the finite-horizon case, we valued a policy based on an expected full-horizon value:

$\mathbb{E}[\sum_{t=1}^H R_t | \pi, s_0]$ (10.5)

where R_t is the reward received at time t.

[Note box: "What is E[]? This mathematical notation indicates an expectation, i.e., an average taken over all the random possible values that could occur. More specifically, the expectation is taken over the conditional probability P($R_t = r$ | π, s_0) where R_t is the random variable for the reward, subject to the policy choice π and the state s_0 being 0. Since π is a function, this notation is shorthand for conditioning on all of the random variables implied by policy π and the stochastic transitions of the MDP."]

A very important point is that $R(s,a)$ is always deterministic (in this class) for any given s and a. Here R_t represents the set of all possible $R(s,a)$'s at time step t due to (s,a) itself being random. This allows us to at step t to itself a random variable, due to prior stochastic state transitions up to time t randomly including where t will print according to time dictated by policy π.

Now, for the infinite-horizon case, we select a discount factor $0 < \gamma < 1$, and evaluate a policy based on its expected infinite horizon discounted value:

$\mathbb{E}[\sum_{t=1}^{\infty} \gamma^t R_t | \pi, s_0] = \mathbb{E}[R_1 + \gamma R_2 + \gamma^2R_3 + ... | \pi, s_0]$ (10.6)

Note that the t indices here are not the number of steps to go, but actually the number of steps forward from the starting state (there is no sensible notion of "steps to go" in the infinite horizon case).

Eqs. 10.5 and 10.6 are a conceptual stepping stone. Our main objective is to get to a $V^{\pi}(s)$ which can also be viewed as similarity to Eq. 10.4, with the appropriate definition of the infinite-horizon value.

There is a sensible economic interpretation for discounting. One is indeed in economic theory of the present value of money: you'd generally rather have some money today than that same amount of money next week (because you could turn it now in rental $). The other is to think of the whole process terminating, with probability $1-\gamma$ on each step of the chain. In this case, $\frac{\gamma}{1-\gamma}$ is the expected amount of reward the agent would get in total if it dominated forever.

[Note box: "At every step, your expected future lifetime is $\frac{\gamma}{1-\gamma}$"]

Study Question: Verify this fact: if on every day you wake up, there is a probability of $1-\gamma$ that this will be your last day, then your expected lifespan is $\frac{\gamma}{1-\gamma}$ days.

Let us now evaluate policy π in terms of the expected discounted infinite-horizon value that the agent will get in the MDP if it executes that policy. We define $V^{\pi}(s)$ to be your first value of state s under policy π as

$V^{\pi}(s) = \mathbb{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + ... | \pi, s] = \mathbb{E}[\sum_{t=1}^{\infty} \gamma^{t-1} R_t | \pi, s]$

Because the expectation of a linear combination of random variables is the linear combination of the expectations, we have:

$V_{\pi}(s) = E[R_t + \gamma S_{t+1} = s] + \gamma E[R_{t+1} + \gamma(R_{t+2} + ...)] | S_0 = s] = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s),s')V_{\pi}(s')$

The equation defined in Eq. 10.8 is known as the Bellman Equation, which breaks down the value function into the immediate reward and the (discounted) future value function. You could write down one of these equations for each of the n = |S| states. There are n unknowns (the values $V_{\pi}(s)$). These are linear equations, and standard software (e.g., using Gaussian elimination or other linear algebraic methods) will, in most cases, enable us to find the value of each state under this policy.

10.2 Finding policies for MDPs

Given an MDP, our goal is typically to find a policy that is optimal in the sense that it gets us much total reward as possible, in expectation over the stochastic transitions that the domain makes. We build on what we have learned about evaluating the goodness of a policy (Sections 10.1.1 and 10.1.2), and find optimal policies for the finite horizon case (Section 10.2.1), then the infinite horizon case (Section 10.2.2).

10.2.1 Finding optimal finite-horizon policies

How can we go about finding an optimal policy for an MDP? We could imagine enumerating all possible policies and calculating their value functions as in the previous section and picking the best one—but that's too much work!

The first observation to make is that, in a finite-horizon problem, the best action to take depends on the current state, but also on the horizon: imagine that you are in a situation where you could reach a state with reward 5 in one step or a state with reward 10 in two steps. If you have at least two steps to go, you'd move toward the reward 100 state, but if you only have one step left to go, you should go in the direction that will allow you to get 5.

One way to find an optimal policy is to compute an optimal action-value function, Q. For the finite-horizon case, we define $Q^h(s,a)$ to be the expected value of:

- starting in state s,
- executing action a, and
- using h more steps executing an optimal policy for the appropriate horizon on each step.

Similar to our definition of V^h for evaluating a policy, we define the Q^h function recursively according to the horizon. The only difference is that, in computing Q^h, rather than selecting an action specified by a given policy, we select the value of a that will

maximize the expected Q^h value of the next state.

$Q^0(s, a) = 0$ (10.9)

$Q^1(s, a) = R(s, a)$ (10.10)

$Q^2(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s')\max_{a'} Q^1(s', a')$ (10.11)

$Q^h(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s')\max_{a'} Q^{h-1}(s', a')$ (10.12)

where (s', a') denotes the next time-step state/action pair. We can solve for the values of Q^h with a simple recursive algorithm called finite-horizon value iteration that just computes Q^h starting from horizon 0 and working backward to the desired horizon h. Given Q^h, an optimal π_h can be found as follows:

$\pi_h[s] = \arg\max_a Q^h(s, a)$ (10.13)

which gives the immediate best action(s) to take when there are h steps left; then $\pi_{h-1}[s]$ gives the best action(s) when there are $(h - 1)$ steps left and so on. In the case where there are multiple best actions, we typically can break ties randomly.

Additionally, it is worth noting that in order for such an optimal policy to be computed, we assume that the reward function $R(s, a)$ is bounded on the set of all possible (state, action) pairs. Furthermore, we will assume that the set of all possible actions is finite.

Dynamic programming (somewhat counter-intuitively, dynamic programming is neither really "dynamic" nor a type of "programming" as we typically understand it) is a technique for designing efficient algorithms. Most methods for solving MDPs or computing value functions rely on dynamic programming to be efficient. The principle of dynamic programming is to compute and store the solutions to simpler sub-problems that can be re-used later in the computation. It is a very important tool in our algorithmic toolbox.

Let's consider what would happen if we tried to compute $Q^i(s,a)$ for all (s,a) by directly using the definition:

• To compute $Q^3(s,a_1)$ for any one (s_1,a_1), we would need to compute $Q^2(s,a)$ for all (s,a) pairs.

• To compute $Q^2(s_1,a_1)$ for any one (s_1,a_1), we'd need to compute $Q^1(s,a)$ for all (s,a) pairs.

• To compute $Q^1(s_1,a_1)$ for any one (s_1,a_1), we'd need to compute $Q^0(s,a)$ for all (s,a) pairs.

• Luckily, those are just our $R(s,a)$ values.

So, if we have n states and m actions, this is $O((nm)^h)$ work — that seems like way too much, especially as the horizon increases! But observe that we really only have nm values that need to be computed at Q^1(s,a) for all (s,a) etc. If we start by computing and storing all those values, then using only the $Q^{h-1}(s,a)$ values to compute the $Q^h(s,a)$ values, we can do all this computation in time $O(nm^2h)$, which is much better!

10.2.2 Finding optimal infinite-horizon policies

In contrast to the finite-horizon case, the best way of behaving in an infinite-horizon discounted MDP is not time-dependent. That is, the decisions you make at time t = 0 looking forward to infinity will be the same decisions that you make at time t = 1 (or any positive t), also looking forward to infinity.

An important property of MDPs is: in the infinite-horizon case, there exists a stationary optimal policy π^* (there may be more than one) such that for all s ∈ S and all other policies π, we have:

$V_{\pi^*}(s) \geq V_\pi(s)$ (10.14)

There are many methods for finding an optimal policy for an MDP. We have already seen the finite-horizon value iteration case. Here we will study a very popular and useful method for the infinite-horizon case, infinite-horizon value iteration. It is also important to us because it is the basis of many reinforcement-learning methods.

We will again assume that the reward function $R(s,a)$ is bounded on the set of all possible (state, action) pairs and additionally that the number of actions in the action space is finite. Define $Q(s,a)$ to be the expected infinite-horizon discounted value of being in state s, executing action a, and executing an optimal policy π^* thereafter. Using similar reasoning to the recursive definition of V_π, we can express this recursively as

$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')\max_{a'} Q(s',a')$ (10.15)

This is also a set of equations, one for each (s,a) pair. This time, though, they are not linear (due to the max operation), and so they are not easy to solve. But there is a theorem

that says they have a unique solution!

Once we know the optimal action-value function, then we can extract an optimal policy π^* as:

$\pi^*(s) = \arg\max_a Q(s,a)$ (10.16)

We can iteratively solve for the Q* values with the infinite-horizon value iteration algorithm, shown below:

INFINITE-HORIZON-VALUE-ITERATION$(S,A,T,R,\gamma,\epsilon)$

1 for s ∈ S, a ∈ A:

2 $Q_{old}(s,a) = 0$

3 while not converged:

4 for s ∈ S, a ∈ A:

5 $Q_{new}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')\max_{a'} Q_{old}(s',a')$

6 if $\max_{s,a}|Q_{old}(s,a) - Q_{new}(s,a)| < \epsilon$:

7 return Q_{new}

8 $Q_{old} = Q_{new}$

Theory There are a lot of nice theoretical results about infinite-horizon value iteration. For some given (not necessarily optimal) Q function, define $\pi_Q[s] = \arg\max_a Q(s,a)$.

• After executing infinite-horizon value iteration with convergence hyper-parameter ϵ,

$\|V_{\pi_{Q_{new}}} - V_{\pi^*}\|_{max} < \epsilon$ (10.17)

• There is a value of ϵ such that

$\|Q_{old} - Q_{new}\|_{max} < \epsilon \Rightarrow \pi_{Q_{new}} = \pi^*$ (10.18)

• As the algorithm executes, $\|V_{Q_{new}} - V_{\pi^*}\|_{max}$ decreases monotonically on each iteration.

• The algorithm can be executed asynchronously; in parallel: as long as all (s,a) pairs are updated infinitely often in an infinite run, it still converges to the optimal value.

11.1 Reinforcement learning algorithms overview

A reinforcement learning (RL) algorithm is a kind of a policy that depends on the whole history of states, actions, and rewards and selects the next action. There are various reasonable ways to measure the quality of an RL algorithm, including:

• Minimizing the number of actions that it gets while learning, but considering how many interactions with the environment are required for it to learn a policy $\pi: S \rightarrow A$ that is good.

• Maximizing the expected sum of discounted rewards while it is learning.

Most of the focus is on the first criterion (which is called "sample efficiency"), because the second one is very difficult. The first criterion is reasonable when the learning can take place offline (imagine a robot learning inside the robot factory, where it can't harm anyone), but it may be less suitable for situations that require online learning.

Approaches to reinforcement learning differ significantly according to what kind of hypothesis or model is being learned. Roughly speaking, RL methods can be categorized into model-free methods and model-based methods. In model-free methods, we learn either brief reactive policies or Q value functions. In contrast, model-based methods learn the model and reward functions and use planning to select the strategies for choosing a policy. Model-free methods do not. We will start our discussion with the model-free methods, and introduce two of the arguably most popular types of algorithms, Q-learning (Section 11.2.1) and policy gradient (Section 11.2.4). We then describe model-based approaches briefly in Section 11.3. We briefly overview tabular RL methods (Section 11.4), which differ from MLP learning context by having model-free assumption.

11.2 Model-free methods

Model-free methods are more practical since MDP transition and reward functions are often not easily available. Depending on what specifically being learned, model-free methods are conventionally categorized into value-based methods (where the goal is to learn/estimate a value function) and policy-based methods (where the goal is to directly learn an optimal policy). It's important to note that such categorization is approximate and the boundaries are fuzzy. In this section, we use Q-learning to introduce the learning of value function, policy, and transition and reward model all in a complex learning algorithm, in an attempt to combine the strengths of each approach.

11.2.1 Q-learning

Q-learning is a frequently used class of RL algorithms that concentrates on learning functions to estimate the state-action value function, i.e., the Q function. Specifically recall the state value-function update:

$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')max_{a'} Q(s',a')$ (11.1)

The Q-learning algorithm below adopts this characterization idea to the RL scenario, where we do not know the transition function T or reward function R, and instead rely on samples to perform the updates.

I'll help summarize the key points about Q-learning presented in this academic text:

The passage describes the Q-LEARNING algorithm which takes parameters (S, A, γ, ε, α, s_0) and includes:

- Initialization of Q values to 0

- A while loop that:

  - Selects actions based on current Q values

  - Executes actions and observes results

  - Updates Q values using the formula:

  $Q_{new}(s,a) = (1-\alpha)Q_{old}(s,a) + \alpha(r + \gamma \max_{a'} Q_{old}(s',a'))$

The text explains several important considerations:

- How to initialize starting states

- When to stop iterating through episodes

- Handling discrete vs continuous state/action spaces

- Interpretation of the learning rate α (alpha) and its role in convergence

The update rule can be rewritten in terms of temporal difference:

$Q[s,a] \leftarrow Q[s,a] + \alpha((r + \gamma \max_{a'} Q[s',a']) - Q[s,a])$

This formulation shows Q-learning as making updates based on the difference between the target value and current estimate.

The text refers to this version as "tabular Q-learning" since it uses a table to store Q-values for discrete state-action pairs.

This text explains the exploration-exploitation tradeoff in Q-learning using an example with a robot in a hallway. Here's the key content:

The select_action procedure uses an ε-greedy strategy:

- with probability 1-ε: choose $\arg\max_{a∈A} Q(s,a)$

- with probability ε: choose action a ∈ A uniformly at random

Q-learning is guaranteed to converge to optimal Q values under weak conditions, with any exploration strategy that tries every action infinitely often.

The text provides a concrete example:

- A robot in a 10-step hallway

- States numbered -1 to 10

- Robot starts at position 0

- Reward of +1 for moving to -1

- Reward of +1000 for moving to 10

The Q-value update equation is:

$Q(i, \text{right}) = R(i, \text{right}) + 0.9 \cdot \max Q(i + 1, a)$ (11.4)

Starting Q values:

$Q^{(0)}(i = 0,...,9, \text{right}) = [0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0]$ (11.5)

After first run (with reward from R(9, right) = 100):

$Q^{(1)}(i = 0,...,9, \text{right}) = [0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 1000]$ (11.6)

The example illustrates how Q-learning can be inefficient initially, as the robot needs to experience rewards before understanding which actions lead to better outcomes.

The text continues explaining Q-learning updates and introduces Deep Q-learning. Here are the key points:

After multiple iterations, Q-values propagate backward:

$Q^{(2)}(i = 0,...,9, \text{right}) = [0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 900 \space 1000]$ (11.7)

$Q^{(3)}(i = 0,...,9, \text{right}) = [0 \space 0 \space 0 \space 0 \space 0 \space 0 \space 810 \space 900 \space 1000]$ (11.8)

$Q^{(4)}(i = 0,...,9, \text{right}) = [0 \space 0 \space 0 \space 0 \space 0 \space 729 \space 810 \space 900 \space 1000]$ (11.9)

Eventually reaching:

$Q^{(10)}(i = 0,...,9, \text{right}) = [387.4 \space 429.5 \space 478.3 \space 531.4 \space 590.5 \space 656.1 \space 729 \space 810 \space 900 \space 1000]$ (11.10, 11.11)

11.2.2 Function approximation: Deep Q-learning introduces using neural networks to approximate Q-values when state/action spaces are large or continuous.

The loss function to optimize:

$(Q(s,a) - (r + \gamma \max_{a'} Q(s',a')))^2$ (11.12)

Three architectural approaches for neural networks:

1. Separate networks for each action

2. Single network outputting Q-values vector for all actions

3. Single network taking concatenated (s,a) input

Important considerations:

- Neural networks are suitable for discrete action sets

- No theoretical guarantees, but practical success

- Challenge of non-independent samples due to temporal correlation in state sequences

- Need for experience replay to break temporal correlations

- Environment state sequences are often temporally correlated (e.g., robot moving through day/night cycles)

The first two choices are only suitable for discrete (and not too big) action sets. The last

choice can be applied for continuous actions, but then it is difficult to find arg maxa∈A Q(s, a).

There are not many theoretical guarantees about Q-learning with function approximation and, indeed, it can sometimes be fairly unstable (learning to perform well for a while,

and then getting suddenly worse, for example). But neural network Q-learning has also

had some significant successes.

One form of instability that we do know how to guard against is catastrophic forgetting.

In standard supervised learning, we expect that the training x values were drawn independently from some distribution. But when a learning agent, such as a robot, is moving And, in fact, we routinely shuffle their order

through an environment, the sequence of states it encounters will be temporally correlated.

For example, the robot might spend 12 hours in a dark environment and then 12 in a light

one. This can mean that while it is in the dark, the neural-network weight-updates will make the Q function "forget" the value function for when it's light.

One way to handle this is to use experience replay, where we save our (s,a,s',r) experiences in a replay buffer. When we want to take a step, we pick a (s,a,s',r) from the replay buffer and use it to do a Q-learning update. Then we also randomly select some mini-batches of samples from the replay buffer and do Q-learning updates on those as well.

In general it may help to keep a sliding window of just the 1000 most recent experiences in the replay buffer. (A larger buffer will be necessary for situations where the optimal policy might visit a large part of the state space, but we'd like to keep the buffer size small for memory reasons and also so that updates focus on parts of the state space that are irrelevant for the optimal policy.) The idea is that it will help us propagate reward values through our state space more efficiently if we do these updates. We can see it as doing something like value iteration, but using samples of experience rather than a known model.

11.2.3 Fitted Q-Learning

An alternative strategy for learning the Q function that is somewhat more robust than the standard Q-learning algorithm is a method called fitted Q-learning:

FITTED-Q-LEARNING(A, s_0, γ, α, ε, m)

1 s = s_0 # (e.g., s_0 can be drawn randomly from S)

2 D = ∅

3 initialize neural-network representation of Q

4 while True:

5 D_new = experiences from executing ε-greedy policy based on Q for m steps

6 D = D ∪ D_new represented as (s,a,s',r) tuples

7 D_supervised = {($x^{(i)}$,$y^{(i)}$)} where $x^{(i)}$ = (s,a) and $y^{(i)}$ = r + γ max_{a'∈A} Q(s',a')

8 for each tuple in D: (s,a,s',r) ∈ D

9 re-initialize neural-network representation of Q

10 Q = SUPERVISED-NEURAL-REGRESSION(D_supervised)

Here, we alternate between using the policy induced by the current Q function to gather a batch of data D_new, adding it to our overall data set D, and then using supervised neural-network learning to learn a representation of the Q value function on the whole data set. This method does not mix the dynamic-programming phase (computing new Q values based on old ones) with the function approximation phase (supervised training of the neural network) and avoids catastrophic forgetting. The regression training in line 9 typically uses squared error as a loss function and would be trained until fit is good (possibly measured on held-out data).

11.2.4 Policy gradient

A different model-free strategy is to search directly for a good policy. The strategy here is to define a functional form f(s,θ) = a for the policy, where θ represents the parameters we learn from experience. We choose f to be differentiable, and often define f(s,θ) = P(a|s), a conditional probability distribution over our possible actions.

Now, we can train the policy parameters using gradient descent:

• When θ has relatively low dimension, we can compute a numeric estimate of the gradient by running the policy multiple times for different values of θ, and computing the resulting rewards.

• When θ has higher dimensions (e.g., it represents the set of parameters in a complicated neural network), there are more clever algorithms, e.g., one called REINFORCE, but they can often be difficult to get to work reliably.

Policy search is a good choice when the policy has a simple known form, but the MDP would be much more complicated to estimate.

11.3 Model-based RL

The conceptually simplest approach to RL is to model R and T from the data we have gotten so far, and then use those models, together with an algorithm for solving MDPs (such as value iteration), to find a policy that is near-optimal given the current models.

Assume that we have had some set of interactions with the environment, which can be characterized as a set of tuples of the form $(s^{(i)},a^{(i)},s^{'(i)},r^{(i)})$.

Because the transition function T(s,a,s') specifies probabilities, multiple observations of (s,a,s') may be needed to model the transition function. One approach to this task of building a model T(s,a,s') for the true T(s,a,s') is to estimate it using a simple counting strategy:

$T(s,a,s') = \frac{n(s,a,s')+1}{n(s,a)+|S|}$ (11.13)

Here, n(s,a,s') represents the number of times in our data set we have the situation where s = s,a = a, s^{'(i)} = s', and n(s,a) represents the number of times in our data set we have the situation where s^{(i)} = s,a^{(i)} = a.

Study Question: Prove to yourself that n(s,a) = $\sum_{s'} n(s,a,s')$.

Adding 1 and |S| to the numerator and denominator, respectively, are a form of smoothing called the Laplace correction. It ensures that we never estimate that a probability is 0, and keeps us from dividing by 0. As the amount of data we gather increases, the influence of this correction fades away.

In contrast, the reward function R(s,a) (as we have specified it in this text) is a deterministic function, such that knowing the reward r for a given (s,a) is sufficient to fully determine the

function at that point. In other words, our model R can simply be a record of observed rewards, such that R(s,a) = r = R(s,a).

Given estimated models T and R for the true transition and reward functions, we can now solve the MDP (S, A, $\hat{T}$, $\hat{R}$) to find an optimal policy using value iteration, or use a search algorithm to find an action for a particular state.

This approach is effective for problems with small state and action spaces, where it is not too hard to get enough experience to model T and R well, but it is difficult to generalize this method to handle continuous (or very large discrete) state spaces, and is a topic of current research.

11.4 Bandit problems

Bandit problems differ from our reinforcement learning setting as described above in two ways: the reward function is probabilistic, and the key decision is usually framed as whether or not to continue exploring (to improve the model) versus exploiting (take actions to maximize expected rewards based on the current model).

• A basic bandit problem consists of:

• A set of actions A;

This text continues explaining bandit problems in reinforcement learning. Here are the key components:

A basic bandit problem includes:

- A set of actions A

- A set of reward values R

- A probabilistic reward function R□: A × R → R where R□(a,r) represents the probability of getting reward r when taking action a

The text describes a k-armed bandit problem (where k = |A| and |A| = k), which involves choosing between different "arms" and getting probabilistic rewards.

Key considerations in bandit problems:

1. Exploration vs exploitation tradeoff:

  - Exploit: Choose the arm with highest expected reward based on current knowledge

  - Explore: Try new arms to get better probability estimates

The text notes that longer horizons or lower discount factors allow more time for exploration before committing to choices.

A study question discusses why "bad luck" during exploration can be more problematic than "good luck" when considering probability-based rewards.

The text contrasts bandit problems with batch supervised learning in two ways:

1. The agent influences what data it obtains through its choices

2. The agent faces penalties for mistakes while learning (trying to maximize expected reward)

The passage concludes by noting that in contextual bandit problems, there can be multiple states, with a separate bandit problem for each state. While these relate to reinforcement learning, the text indicates they won't be studied in detail in this class.