CHAPTER 5

Logistic regression

1 Machine learning as optimization

The perceptron algorithm was originally written down directly via cleverness and intuition, and later analyzed theoretically. Another approach to designing machine learning algorithms is to frame them as optimization problems, and then use standard optimization algorithms and implementations to actually find the hypothesis. Taking this approach will allow us to take advantage of a wealth of mathematical and algorithmic technique for understanding and solving optimization problems, which will allow us to move to hypothesis classes that are substantially more complex than linear separators.

We begin by writing down an *objective function* $J(\Theta)$, where Θ stands for *all* the parameters in our model. Note that we will sometimes write $J(\theta, \theta_0)$ because when studying linear classifiers, we have used these two names for parts of our whole collection of parameters, so $\Theta = (\theta, \theta_0)$. We also often write $J(\Theta; \mathcal{D})$ to make clear the dependence on the data $\mathcal{D}$. 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:

You can think about Θ^* here as "the theta that minimizes J".

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

A very common form for an ML objective is

$$J(\Theta) = \left(\frac{1}{n}\sum_{i=1}^{n} \underbrace{\mathcal{L}(h(x^{(i)};\Theta), y^{(i)})}_{\text{loss}}\right) + \underbrace{\lambda}_{\text{constant}} \underbrace{R(\Theta)}_{\text{regularizer}} \quad . \tag{5.1}$$

The *loss* tells us how unhappy we are about the prediction $h(x^{(i)};\Theta)$ that Θ makes for $(x^{(i)}, y^{(i)})$. A common example is the 0-1 loss, introduced in chapter 1:

$$L_{01}(h(x;\Theta), y) = \begin{cases} 0 & \text{if } y = h(x;\Theta) \\ 1 & \text{otherwise} \end{cases} \quad ,$$

which gives a value of 0 for a correct prediction, and a 1 for an incorrect prediction. In the case of linear separators, this becomes:

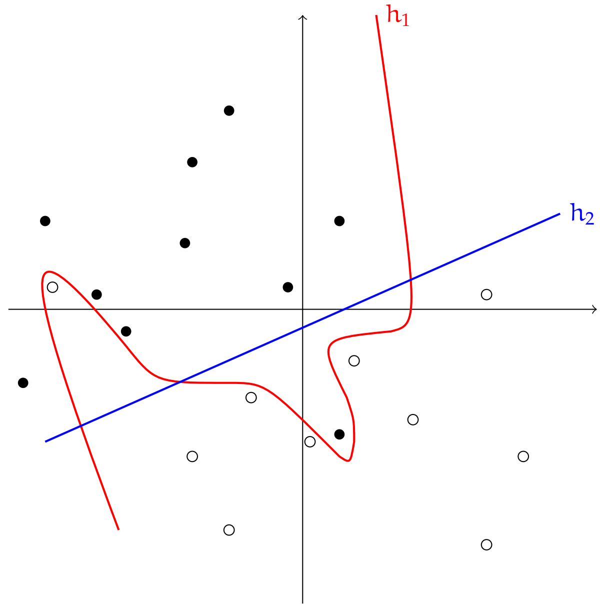
$$L_{01}(h(x;\theta,\theta_0), y) = \begin{cases} 0 & \text{if } y(\theta^T x + \theta_0) > 0 \\ 1 & \text{otherwise} \end{cases} \quad .$$

2 Regularization

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. We have already discussed one way to describe an assumption about such a regularity, which 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.

This trade-off is illustrated in the figure below. Hypothesis h_1 has 0 training loss, but is very complicated. Hypothesis h_2 mis-classifies two points, but is very simple. In absence of other beliefs about the solution, it is often better to prefer that the solution be "simpler," and so we might prefer h_2 over h_1, expecting it to perform better on future examples drawn from this same distribution. Another nice way of thinking about regularization is that we would like to prevent our hypothesis from being too dependent on the particular training data that we were given: we would like for it to be the case that if the training data were changed slightly, the hypothesis would not change by much.

To establish some vocabulary, we say that h_1 is *overfit* to the training data.



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

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

when we have some idea in advance that θ ought to be near some value Θ_{prior}. In the absence of such knowledge a default is to *regularize toward zero*:

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

Learn about Bayesian methods in machine learning to see the theory behind this and cool results!

3 A new hypothesis class: linear logistic classifiers

For classification, it is natural to make predictions in $\{+1, -1\}$ and use the $0-1$ loss function. However, even for simple linear classifiers, it is very difficult to find values for θ, θ_0 that minimize simple training error

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

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.

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 through the space of hypotheses for a good one.

For these reasons, we are going to investigate a new hypothesis class: *linear logistic classifiers*. These hypotheses are still parameterized by a d-dimensional vector θ and a scalar θ_0, but instead of making predictions in $\{+1, -1\}$, they generate real-valued outputs in the interval $(0, 1)$. A linear logistic classifier has the form

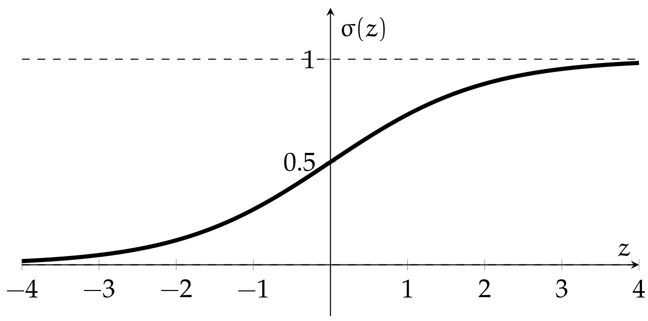
$$h(x; \theta, \theta_0) = \sigma(\theta^T x + \theta_0) \ .$$

This looks familiar! What's new?

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

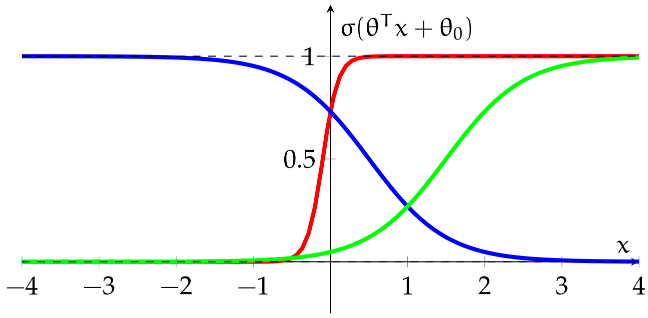
$$\sigma(z) = \frac{1}{1 + e^{-z}} \ ,$$

and 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 $(0, 1)$.



Study Question: Convince yourself the output of σ is always in the interval $(0, 1)$. Why can't it equal 0 or equal 1? For what value of z does $\sigma(z) = 0.5$?

What does a linear logistic classifier (LLC) look like? Let's consider the simple case where $d = 1$, so our input points simply lie along the x axis. The plot below shows LLCs for three different parameter settings: $\sigma(10x + 1)$, $\sigma(-2x + 1)$, and $\sigma(2x - 3)$.



Study Question: Which plot is which? What governs the steepness of the curve? What governs the x value where the output is equal to 0.5?

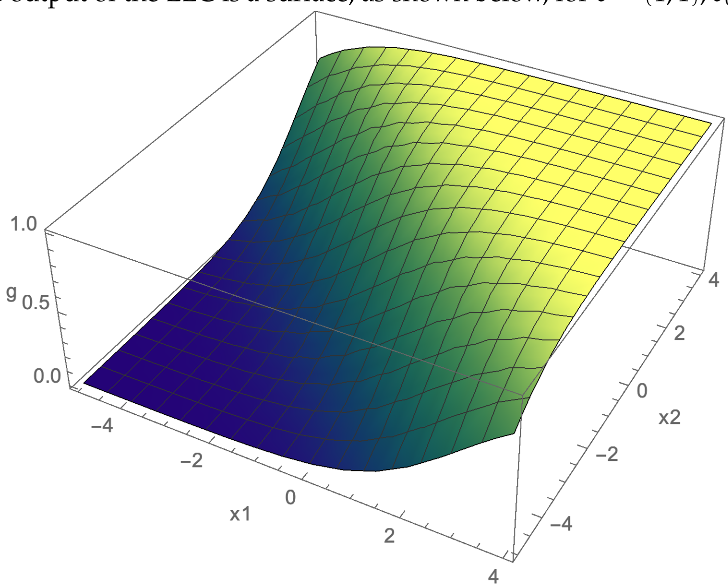
But wait! Remember that the definition of a classifier from chapter 2 is that it's a mapping from $\mathbb{R}^d \rightarrow \{-1, +1\}$ or to some other discrete set. So, then, it seems like an LLC is actually not a classifier!

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*.

In fact, for different problem settings, we might prefer to pick a different prediction threshold. The field of *decision theory* considers how to make this choice from the perspective of Bayesian reasoning. For example, if the consequences of predicting $+1$ when the answer should be -1 are much worse than the consequences of predicting -1 when the answer should be $+1$, then we might set the prediction threshold to be greater than 0.5.

Study Question: Using a prediction threshold of 0.5, for what values of x do each of the LLCs shown in the figure above predict $+1$?

When $d = 2$, then our inputs x lie in a two-dimensional space with axes x_1 and x_2, and the output of the LLC is a surface, as shown below, for $\theta = (1, 1), \theta_0 = 2$.



Study Question: Convince yourself that the set of points for which $\sigma(\theta^T x + \theta_0) = 0.5$, that is, the separator between positive and negative predictions with prediction threshold 0.5 is a line in (x_1, x_2) space. What particular line is it for the case in the figure above? How would the plot change for $\theta = (1, 1)$, but now with $\theta_0 = -2$? For $\theta = (-1, -1), \theta_0 = 2$?

4 Loss function for logistic classifiers

We have defined a class, LLC, of hypotheses whose outputs are in $(0, 1)$, but we have training data with y values in $\{+1, -1\}$. How can we define a loss function? Intuitively, we would like to have *low loss if we assign a low probability to the incorrect 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 will assume that (or transform so that) the labels in the training data are $y \in \{0, 1\}$, enabling them to be interpreted as probabilities of being a member of the class of interest. We would like to pick the parameters of our classifier to maximize the probability assigned by the LCC 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

Remember to be sure your y values have this form if you try to learn an LLC using NLL!

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

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

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

Study Question: Be sure you can see why these two expressions are the same.

Now, because products are kind of hard to deal with, and because the log function is monotonic, the θ, θ_0 that maximize the log of this quantity will be the same as the θ, θ_0 that maximize the original, so we can try to maximize

$$\sum_{i=1}^{n} \left(y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log(1 - g^{(i)}) \right) .$$

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

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

where $\mathcal{L}_{\text{nll}}$ is the *negative log-likelihood* loss function:

$$\mathcal{L}_{\text{nll}}(\text{guess}, \text{actual}) = -\left(\text{actual} \cdot \log(\text{guess}) + (1 - \text{actual}) \cdot \log(1 - \text{guess})\right) .$$

This loss function is also sometimes referred to as the *log loss* or *cross entropy*.

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.

5 Logistic classification as optimization

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

That's a lot of fancy words!

$$J_{lr}(\theta, \theta_0; \mathcal{D}) = \left(\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{nll}(\sigma(\theta^T x^{(i)} + \theta_0), y^{(i)}) \right) + \lambda \left\| \theta \right\|^2 \quad .$$

Study Question: Consider the case of linearly separable data. What will the θ values that optimize this objective be like if $\lambda = 0$? What will they be like if λ is very big? Try to work out an example in one dimension with two data points.

CHAPTER 6

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 $\Theta^* = \arg\min_\Theta J(\Theta)$. There is an enormous, 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*.

Which you should consider studying some day!

Intuitively, in one or two dimensions, we can easily think of $J(\Theta)$ as defining a surface over Θ; that same idea extends to higher dimensions. Now, our objective is to find the Θ 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.

Here's a very old-school humorous description of gradient descent and other optimization algorithms using analogies involving kangaroos: `ftp://ftp.sas.com/pub/neural/kangaroos.txt`

1 One dimension

We will 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 pseudocode for gradient descent on an arbitrary function f. Along with f and its gradient f', we have to specify the initial value for parameter Θ, a *step-size* parameter η, and an *accuracy* parameter ϵ:

1D-GRADIENT-DESCENT($\Theta_{init}, \eta, f, f', \epsilon$)

$\Theta^{(0)} = \Theta_{init}$
$t = 0$
repeat
  $t = t + 1$
  $\Theta^{(t)} = \Theta^{(t-1)} - \eta\, f'(\Theta^{(t-1)})$
until $|f(\Theta^{(t)}) - f(\Theta^{(t-1)})| < \epsilon$
return $\Theta^{(t)}$

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. These include the following.

- 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 $\left|\Theta^{(t)} - \Theta^{(t-1)}\right| < \epsilon$.
- Stop when the derivative f' at the latest value of Θ is sufficiently small, i.e. when $\left|f'(\Theta^{(t)})\right| < \epsilon$.

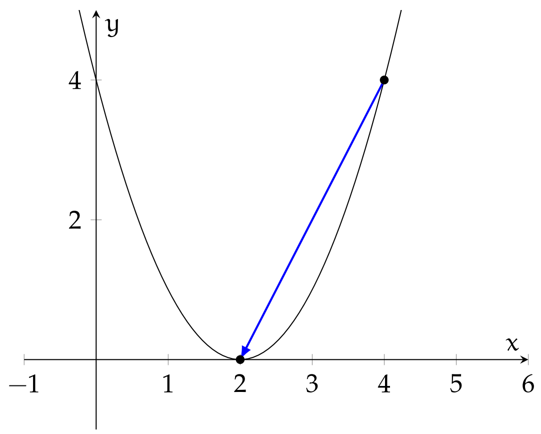
Study Question: In the list of possible stopping criteria for 1D-GRADIENT-DESCENT above, how do the final two potential criteria relate to each other?

Theorem 1.1. *If J is convex, for any desired accuracy ϵ, there is some step size η such that gradient descent will converge to within ϵ of the optimal Θ.*

A function is convex if the line segment between any two points on the graph of the function lies above or on the graph.

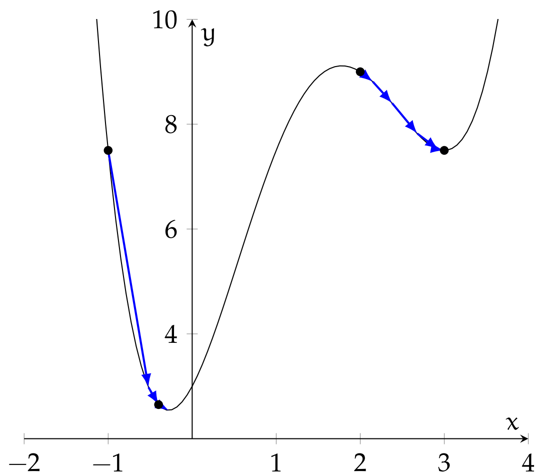
However, we must be careful when choosing the step size to prevent slow convergence, oscillation around the minimum, or divergence.

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



Study Question: What happens in this example with very small η? With very big η?

If J is non-convex, where gradient descent converges to depends on θ_{init}. When it reaches a value of θ where $f'(\theta) = 0$ and $f''(\theta) > 0$, but it is not a minimum of the function, it is called a *local minimum* or *local optimum*. The plot below shows two different θ_{init}, and two different resulting local optima.



2 Multiple dimensions

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

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

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

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

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

3 Application to logistic regression objective

We can now solve the optimization problem for our linear logistic classifier as formulated in chapter 5. We begin by stating the objective and the gradient necessary for doing gradient descent. In our problem where we are considering linear separators, the entire parameter vector is described by parameter vector θ and scalar θ_0 and so we will have to adjust them both and compute gradients of J with respect to each of them. The objective and gradient (note that we have replaced the constant λ with $\frac{\lambda}{2}$ for convenience), are, letting $g^{(i)} = \sigma(\theta^T x^{(i)} + \theta_0)$

The following step requires passing familiarity with matrix derivatives. A foolproof way of computing them is to compute partial derivative of J with respect to each component θ_i of θ.

$$\begin{aligned} J_{lr}(\theta, \theta_0) &= \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{nll}(g^{(i)}, y^{(i)}) + \frac{\lambda}{2} \|\theta\|^2 \\ \nabla_\theta J_{lr}(\theta, \theta_0) &= \frac{1}{n} \sum_{i=1}^{n} \left(g^{(i)} - y^{(i)} \right) x^{(i)} + \lambda\theta \\ \frac{\partial J_{lr}(\theta, \theta_0)}{\partial \theta_0} &= \frac{1}{n} \sum_{i=1}^{n} \left(g^{(i)} - y^{(i)} \right) \quad . \end{aligned}$$

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

Study Question: Convince yourself that the dimensions of all these quantities are correct, under the assumption that θ is $d \times 1$. How does d relate to m as discussed for Θ in the previous section?

Study Question: Compute $\nabla_\theta \|\theta\|^2$ by finding the vector of partial derivatives $(\partial \|\theta\|^2 / \partial \theta_1, \ldots, \partial \|\theta\|^2 / \partial \theta_d)$. What is the shape of $\nabla_\theta \|\theta\|^2$?

Study Question: Compute $\nabla_\theta \mathcal{L}_{nll}(\sigma(\theta^T x + \theta_0), y)$ by finding the vector of partial derivatives $(\partial \mathcal{L}_{nll}(\sigma(\theta^T x + \theta_0), y) / \partial \theta_1, \ldots, \partial \mathcal{L}_{nll}(\sigma(\theta^T x + \theta_0), y) / \partial \theta_d)$.

Study Question: Use these last two results to verify our derivation above.

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

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

1 $\theta^{(0)} = \theta_{init}$
2 $\theta_0^{(0)} = \theta_{0init}$
3 $t = 0$
4 **repeat**
5 $\quad t = t + 1$
6 $\quad \theta^{(t)} = \theta^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left({\theta^{(t-1)}}^T x^{(i)} + \theta_0^{(t-1)}\right) - y^{(i)}\right)x^{(i)} + \lambda\theta^{(t-1)}\right)$
7 $\quad \theta_0^{(t)} = \theta_0^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left({\theta^{(t-1)}}^T x^{(i)} + \theta_0^{(t-1)}\right) - y^{(i)}\right)\right)$
8 **until** $\left|J_{lr}(\theta^{(t)}, \theta_0^{(t)}) - J_{lr}(\theta^{(t-1)}, \theta_0^{(t-1)})\right| < \epsilon$
9 **return** $\theta^{(t)}, \theta_0^{(t)}$

Study Question: Is it okay that λ doesn't appear in line 7?

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.

The word "stochastic" means probabilistic, or random; so does "aleatoric," which is a very cool word. Look up aleatoric music, sometime.

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

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

Here is pseudocode for applying SGD to an objective f; it assumes we know the form of $\nabla_\Theta f_i$ for all i in $1 \ldots n$:

STOCHASTIC-GRADIENT-DESCENT$(\Theta_{init}, \eta, f, \nabla_\Theta f_1, \ldots, \nabla_\Theta f_n, T)$

$\Theta^{(0)} = \Theta_{init}$
for $t = 1$ **to** T
$\quad$ randomly select $i \in \{1, 2, \ldots, n\}$
$\quad \Theta^{(t)} = \Theta^{(t-1)} - \eta(t)\,\nabla_\Theta f_i(\Theta^{(t-1)})$
return $\Theta^{(t)}$

Note that now instead of a fixed value of η, η 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 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 4.1. *If* J *is convex, and* $\eta(t)$ *is a sequence satisfying*

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

then SGD converges with probability one *to the optimal* Θ.

We have left out some gnarly conditions in this theorem. Also, you can learn more about the subtle difference between "with probability one" and "always" by taking an advanced probability course.

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.

Study Question: If you start a long way from the optimum, would making $\eta(t)$ decrease more slowly tend to make you move more quickly or more slowly to the optimum?

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

- If your f is actually non-convex, but has many shallow local optima that might trap BGD, then taking *samples* from the gradient at some point Θ might "bounce" you around the landscape and out of the local optima.

- 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.

- 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.

CHAPTER 7

Regression

Now we will turn to a slightly different form of machine-learning problem, called *regression*. It is still supervised learning, so our data will still have the form

"Regression," in common parlance, means moving backwards. But this is forward progress!

$$S_n = \left\{ \left(x^{(1)}, y^{(1)}\right), \dots, \left(x^{(n)}, y^{(n)}\right) \right\} \quad .$$

But now, instead of the y values being discrete, they will be real-valued, and so our hypotheses will have the form

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

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 categories.

The first step is to pick 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 error* (SE):

$$\text{Loss}(\text{guess}, \text{actual}) = (\text{guess} - \text{actual})^2 \quad .$$

It 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, but with Gaussian-distributed noise added to the y values.

We will consider the case of a linear hypothesis class,

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

remembering that we can get a rich class of hypotheses by performing a non-linear feature transformation before doing the regression. So, $\theta^T x + \theta_0$ is a linear function of x, but $\theta^T \varphi(x) + \theta_0$ is a non-linear function of x if φ is a non-linear function of x.

We will treat regression as an optimization problem, in which, given a data set $\mathcal{D}$, we wish to find a linear hypothesis that minimizes mean squared error. Our objective, often called *mean squared error*, is to find values for $\Theta = (\theta, \theta_0)$ that minimize

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

resulting in the solution:

$$\theta^*, \theta_0^* = \arg\min_{\theta, \theta_0} J(\theta, \theta_0) \ . \tag{7.1}$$

1 Analytical solution: ordinary least squares

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

What does "closed form" mean? Generally, that it involves direct evaluation of a mathematical expression using a fixed number of "typical" operations (like arithmetic operations, trig functions, powers, etc.). So equation 7.1 is not in closed form, because it's not at all clear what operations one needs to perform to find the solution.

Everything is easier to deal with if we assume that the $x^{(i)}$ have been augmented with an extra input dimension (feature) that always has value 1, so we may ignore θ_0. (See chapter 3, section 2 for a reminder about this strategy).

We will 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 is an additional step 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 $1, \dots, d$,
- Constructing a set of k equations of the form $\partial J/\partial\theta_k = 0$, and
- Solving the system for values of θ_k.

We will use d here for the total number of features in each $x^{(i)}$, including the added 1.

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.

Study Question: Work through this and check your answer against ours below.

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:

$$X = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} & \dots & y^{(n)} \end{bmatrix} \ .$$

Study Question: What are the dimensions of X and Y?

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, W and T, which are just transposes of our X and Y, and then work with them:

$$W = X^T = \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \dots & x_d^{(n)} \end{bmatrix} \quad T = Y^T = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} \ .$$

Study Question: What are the dimensions of W and T?

Now we can write

$$J(\theta) = \frac{1}{n} \underbrace{(W\theta - T)^T}_{1\times n} \underbrace{(W\theta - T)}_{n\times 1} = \frac{1}{n} \sum_{i=1}^{n} \left(\left(\sum_{j=1}^{d} W_{ij}\theta_j \right) - T_i \right)^2$$

and using facts about matrix/vector calculus, we get

$$\nabla_\theta J = \frac{2}{n}\underbrace{W^T}_{d\times n}\underbrace{(W\theta - T)}_{n\times 1} \quad .$$

Setting to 0 and solving, we get:

$$\begin{aligned}
\frac{2}{n}W^T(W\theta - T) &= 0\\
W^TW\theta - W^TT &= 0\\
W^TW\theta &= W^TT\\
\theta &= (W^TW)^{-1}W^TT
\end{aligned}$$

And the dimensions work out!

$$\theta = \underbrace{\left(W^TW\right)^{-1}}_{d\times d}\underbrace{W^T}_{d\times n}\underbrace{T}_{n\times 1}$$

So, given our data, we can directly compute the linear regression that minimizes mean squared error. That's pretty awesome!

2 Regularizing linear regression

Well, actually, there are some kinds of trouble we can get into. What if $\left(W^TW\right)$ is not invertible?

> **Study Question:** Consider, for example, a situation where the data-set is just the same point repeated twice: $x^{(1)} = x^{(2)} = (1, 2)^T$. What is W in this case? What is W^TW? What is $(W^TW)^{-1}$?

Another kind of problem is *overfitting*: we have formulated an objective that is just about fitting the data as well as possible, but as we discussed in the context of margin maximization, 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 $(W^TW)^{-1}$ and the problem of overfitting using a mechanism called *ridge regression*. We add a regularization term $\|\theta\|^2$ to the OLS objective, with trade-off parameter λ.

> **Study Question:** When we add a regularizer of the form $\|\theta\|^2$, what is our most "preferred" value of θ, in the absence of any data?

Here is the ridge regression objective function:

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

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, and so you shouldn't make it harder to fit a data set where the y values tend to be around one million than one where they tend to be around one. The other parameters control the orientation of the regression surface, and we prefer it to have a not-too-crazy orientation.

There is an analytical expression for the θ, θ_0 values that minimize J_{ridge}, but it's a little bit more complicated to derive than the solution for OLS because θ_0 needs special treatment.

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

$$\nabla_\theta J_{\text{ridge}} = \frac{2}{n} W^T (W\theta - T) + 2\lambda\theta \quad .$$

Setting to 0 and solving, we get:

$$\begin{aligned}
\frac{2}{n} W^T (W\theta - T) + 2\lambda\theta &= 0 \\
\frac{1}{n} W^T W\theta - \frac{1}{n} W^T T + \lambda\theta &= 0 \\
\frac{1}{n} W^T W\theta + \lambda\theta &= \frac{1}{n} W^T T \\
W^T W\theta + n\lambda\theta &= W^T T \\
(W^T W + n\lambda I)\theta &= W^T T \\
\theta &= (W^T W + n\lambda I)^{-1} W^T T
\end{aligned}$$

Whew! So,

$$\theta_{\text{ridge}} = \left(W^T W + n\lambda I\right)^{-1} W^T T$$

which becomes invertible when $\lambda > 0$.

This is called "ridge" regression because we are adding a "ridge" of λ values along the diagonal of the matrix before inverting it.

Study Question: Derive this version of the ridge regression solution.

Talking about regularization 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 errors on test data. We have

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 sin 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 $h \in \mathcal{H}$.

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

There are technical definitions of these concepts that are studied in more advanced treatments of machine learning. Structural error is referred to as *bias* and estimation error is referred to as *variance*.

Study Question: Consider using a polynomial basis of order k as a feature transformation ϕ on your data. Would increasing k tend to increase or decrease structural error? What about estimation error?

3 Optimization via gradient descent

Inverting the $d \times d$ matrix $\left(W^T W\right)$ takes $O(d^3)$ time, which makes the analytic solution impractical for large d. If we have high-dimensional data, we can fall back on gradient descent.

Well, actually, Gauss-Jordan elimination, a popular algorithm, takes $O(d^3)$ arithmetic operations, but the bit complexity of the intermediate results can grow exponentially! There are other algorithms with polynomial bit complexity. (If this just made no sense to you, don't worry.)

Study Question: Why is having large n not as much of a computational problem as having large d?

Recall the ridge objective

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

and its gradient with respect to θ

$$\nabla_\theta J = \frac{2}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)x^{(i)} + 2\lambda\theta$$

and partial derivative with respect to θ_0

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right) \quad .$$

Armed with these derivatives, we can do gradient descent, using the regular or stochastic gradient methods from chapter 6.

Even better, the objective functions for OLS and ridge regression are *convex*, which means they have only one minimum, which means, with a small enough step size, gradient descent is *guaranteed* to find the optimum.

CHAPTER 8

Neural Networks

Unless you live under a rock with no internet access, you've 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 completely well equipped to understand neural networks in detail.

This, in some sense, the "third wave" of neural nets. The basic idea is founded on the 1943 model of neurons of McCulloch and Pitts and learning ideas of Hebb. There was a great deal of excitement, but not a lot of practical success: there were good training methods (e.g., perceptron) for linear functions, and interesting examples of non-linear functions, but no good way to train non-linear functions from data. Interest died out for a while, but was re-kindled in the 1980s when several people came up with a way to train neural networks with "back-propagation," which is a particular style of implementing gradient descent, which we will study here. By the mid-90s, the enthusiasm waned again, because although we could train non-linear networks, the training tended to be slow and was plagued by a problem of getting stuck in local optima. Support vector machines (SVMs) (regularization of high-dimensional hypotheses by seeking to maximize the margin) and kernel methods (an efficient and beautiful way of using feature transformations to non-linearly transform data into a higher-dimensional space) provided reliable learning methods with guaranteed convergence and no local optima.

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.

However, during the SVM enthusiasm, several groups kept working on neural networks, and their work, in combination with an increase in available data and computation, has made them rise again. They have become much more reliable and capable, and are now the method of choice in many applications. There are many, many variations of neural networks, which we can't even begin to survey. We will study the core "feed-forward" networks with "back-propagation" training, and then, in later chapters, address some of the major advances beyond this core.

The number increases daily, as may be seen on `arxiv.org`.

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.

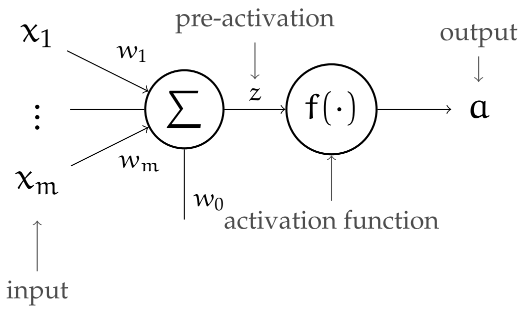
View 3: A method for building applications that make predictions based on huge amounts of data in very complex domains.

We will mostly take view 1, with the understanding that the techniques we develop will enable the applications in view 3. View 2 was a major motivation for the early development of neural networks, but the techniques we will study do not seem to actually account for the biological learning processes in brains.

Some prominent researchers are, in fact, working hard to find analogues of these methods in the brain

1 Basic element

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



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* $w_0 \in \mathbb{R}$. In order for the neuron to be non-linear, we also specify an *activation function* $f : \mathbb{R} \to \mathbb{R}$, which can be the identity ($f(x) = x$), but can also be any other function, though we will only be able to work with it if it is differentiable.

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 impossible to be consistent with all of them though—there are many different ways of telling this story.

The function represented by the neuron is expressed as:

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

Before thinking about a whole network, we can consider how to train a single unit. Given a loss function $L(guess, actual)$ 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

This should remind you of our θ and θ_0 for linear models.

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

where NN is the output of our neural net for a given input.

We have already studied two special cases of the neuron: linear logistic classifiers (LLC) with NLL loss and regressors with quadratic loss! The activation function for the LLC is $f(x) = \sigma(x)$ and for linear regression it is simply $f(x) = x$.

Study Question: Just for a single neuron, imagine for some reason, that we decide to use activation function $f(z) = e^z$ and loss function $L(g, a) = (g - a)^2$. Derive a gradient descent update for w and w_0.

2 Networks

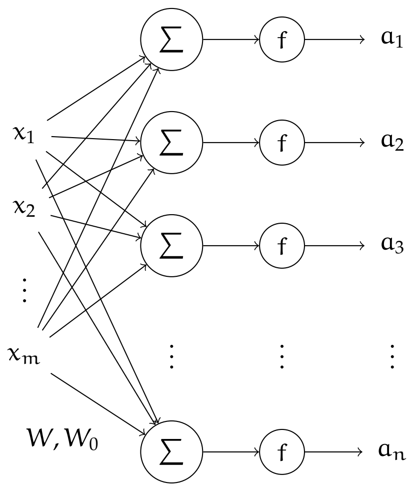
Now, we'll put multiple neurons together into a *network*. A neural network in general takes in an input $x \in \mathbb{R}^m$ and generates an output $a \in \mathbb{R}^n$. It is constructed out of multiple neurons; the inputs of each neuron might be elements of x and/or outputs of other neurons. The outputs are generated by n *output units*.

In this chapter, we will only consider *feed-forward* networks. In a feed-forward network, you can think of the network as defining a function-call graph that is *acyclic*: that is, the input to a neuron can never depend on that neuron's output. Data flows, one way, from the inputs to the outputs, and the function computed by the network is just a composition of the functions computed by the individual neurons.

Although the graph structure of a neural network can really be anything (as long as it satisfies the feed-forward constraint), for simplicity in software and analysis, we usually organize them into *layers*. A layer is a group of neurons that are essentially "in parallel": their inputs are outputs of neurons in the previous layer, and their outputs are the input to the neurons in the next layer. We'll start by describing a single layer, and then go on to the case of multiple layers.

2.1 Single layer

A *layer* is a set of units that, as we have just described, are not connected to each other. The layer is called *fully connected* if, as in the diagram below, the inputs to each unit in the layer are the same (i.e. $x_1, x_2, \ldots x_m$ in this case). A layer has input $x \in \mathbb{R}^m$ and output (also known as *activation*) $a \in \mathbb{R}^n$.



Since each unit has a vector of weights and a single offset, we can think of the weights of the whole layer as a matrix, W, and the collection of all the offsets as a vector W_0. If we have m inputs, n units, and n outputs, then

- 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,
- $Z = W^T X + W_0$, the *pre-activation*, is an $n \times 1$ column vector,
- A, the *activation*, is an $n \times 1$ column vector,

and the output vector is

$$A = f(Z) = f(W^T X + W_0) \ .$$

The activation function f is applied element-wise to the pre-activation values Z.

What can we do with a single layer? We have already seen single-layer networks, in the form of linear separators and linear regressors. All we can do with a single layer is make a linear hypothesis. The whole reason for moving to neural networks is to move in the direction of *non-linear* hypotheses. To do this, we will have to consider multiple layers, where we can view the last layer as still being a linear classifier or regressor, but where we interpret the previous layers as learning a non-linear feature transformation $\phi(x)$, rather than having us hand-specify it.

We have used a step or sigmoid function to transform the linear output value for classification, but it's important to be clear that the resulting *separator* is still linear.

2.2 Many layers

A single neural network generally combines multiple layers, most typically by feeding the outputs of one layer into the inputs of another layer.

We have to start by establishing some nomenclature. We will use l to name a layer, and let m^l be the number of inputs to the layer and n^l be the number of outputs from the layer. Then, W^l and W_0^l are of shape $m^l \times n^l$ and $n^l \times 1$, respectively. Let f^l be the activation function of layer l. Then, the pre-activation outputs are the $n^l \times 1$ vector

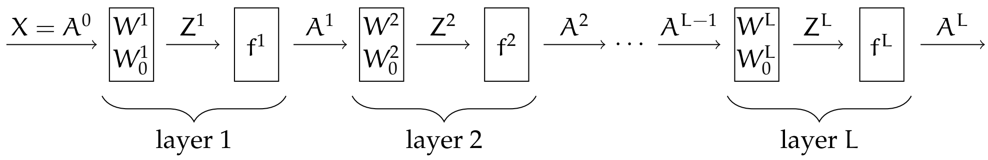
It is technically possible to have different activation functions within the same layer, but, again, for convenience in specification and implementation, we generally have the same activation function within a layer.

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

and the activation outputs are simply the $n^l \times 1$ vector

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

Here's a diagram of a many-layered network, with two blocks for each layer, one representing the linear part of the operation and one representing the non-linear activation function. We will use this structural decomposition to organize our algorithmic thinking and implementation.



3 Choices of activation function

There are many possible choices for the activation function. We will start by thinking about whether it's really necessary to have an f at all.

What happens if we let f be the identity? Then, in a network with L layers (we'll leave out W_0 for simplicity, but keeping it wouldn't change the form of this argument),

$$A^L = {W^L}^T A^{L-1} = {W^L}^T {W^{L-1}}^T \cdots {W^1}^T X \ .$$

So, multiplying out the weight matrices, we find that

$$A^L = W^{\text{total}} X \ ,$$

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.

Study Question: Convince yourself that any function representable by any number of linear layers (where f is the identity function) can be represented by a single layer.

Now that we are convinced we need a non-linear activation, let's examine a few common choices.

Step function

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

Rectified linear unit

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

Sigmoid function Also known as a *logistic* function, can be interpreted as probability, because for any value of z the output is in $[0, 1]$

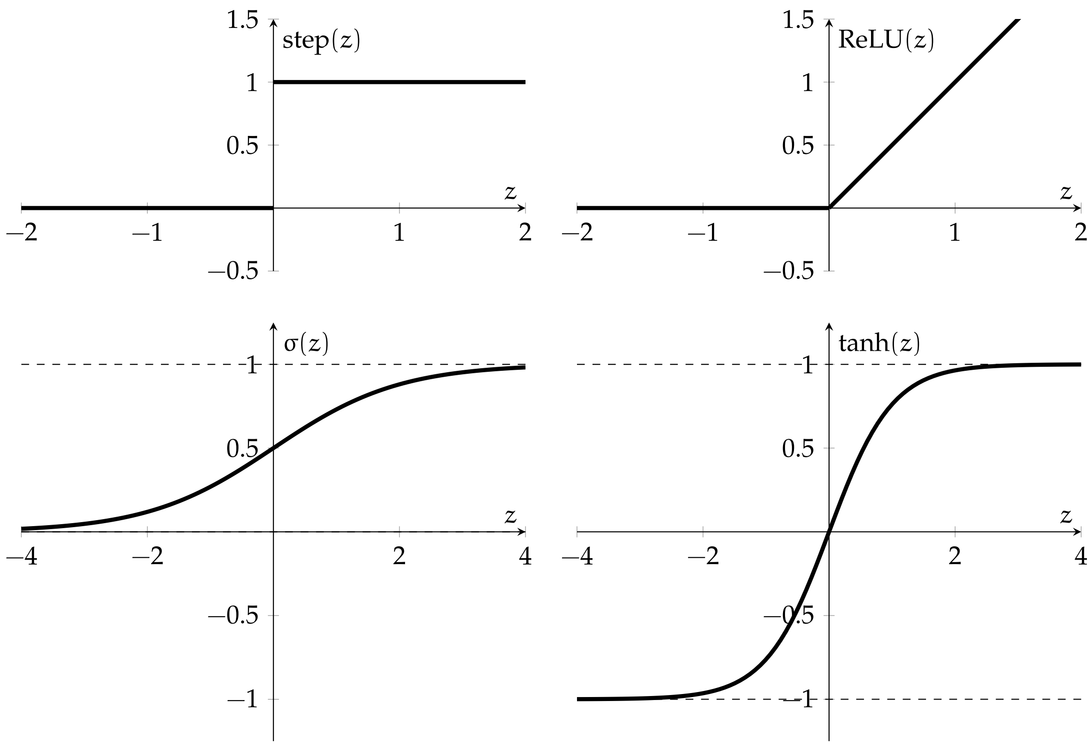
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

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:

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



The original idea for neural networks involved using the **step** function as an activation, but because the derivative is discontinuous, we won't be able to use gradient-descent methods to tune the weights in a network with step functions, so we won't consider them further. They have been replaced, in a sense, by the sigmoid, relu, and tanh activation functions.

Study Question: Consider sigmoid, relu, and tanh activations. Which one is most like a step function? Is there an additional parameter you could add to a sigmoid that would make it be more like a step function?

Study Question: What is the derivative of the relu function? Are there some values of the input for which the derivative vanishes?

ReLUs are especially common in internal ("hidden") layers, and sigmoid activations are common for the output for binary classification and softmax for multi-class classification (see section 4 for an explanation).

4 Error back-propagation

We will train neural networks using gradient descent methods. It's possible to use *batch* gradient descent, in which we sum up the gradient over all the points (as in section 2 of chapter 6) or stochastic gradient descent (SGD), in which we take a small step with respect to the gradient considering a single point at a time (as in section 4 of chapter 6).

Our notation is going to get pretty hairy pretty quickly. To keep it as simple as we can, we'll focus on computing the contribution of one data point $x^{(i)}$ to the gradient of the loss with respect to the weights, for SGD; you can simply sum up these gradients over all the data points if you wish to do batch descent.

So, to do SGD for a training example (x, y), we need to compute $\nabla_W \text{Loss}(\text{NN}(x; W), y)$, where W represents all weights W^l, W_0^l in all the layers $l = (1, \ldots, L)$. This seems terrifying, but is actually quite easy to do using the chain rule.

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

Remember that we are always computing the gradient of the loss function *with respect to the weights* for a particular value of (x, y). That tells us how much we want to change the weights, in order to reduce the loss incurred on this particular training example.

First, let's see how the loss depends on the weights in the final layer, W^L. Remembering that our output is A^L, and using the shorthand loss to stand for $\text{Loss}((\text{NN}(x; W), y)$ which is equal to $\text{Loss}(A^L, y)$, and finally that $A^L = f^L(Z^L)$ and $Z^L = {W^L}^T A^{L-1}$, we can use the chain rule:

$$\frac{\partial \text{loss}}{\partial W^L} = \underbrace{\frac{\partial \text{loss}}{\partial A^L}}_{\text{depends on loss function}} \cdot \underbrace{\frac{\partial A^L}{\partial Z^L}}_{f^{L\prime}} \cdot \underbrace{\frac{\partial Z^L}{\partial W^L}}_{A^{L-1}} .$$

It might reasonably bother you that $\partial Z^L/\partial W^L = A^{L-1}$. We're somehow thinking about the derivative of a vector with respect to a matrix, which seems like it might need to be a three-dimensional thing. But note that $\partial Z^L/\partial W^L$ is really $(\partial {W^L}^T A^{L-1})/\partial W^L$ and it seems okay in at least an informal sense that it's A^{L-1}.

To actually get the dimensions to match, we need to write this a bit more carefully, and note that it is true for any l, including $l = L$:

$$\underbrace{\frac{\partial \text{loss}}{\partial W^l}}_{m^l \times n^l} = \underbrace{A^{l-1}}_{m^l \times 1} \underbrace{\left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T}_{1 \times n^l} \tag{8.1}$$

Yay! So, in order to find the gradient of the loss with respect to the weights in the other layers of the network, we just need to be able to find $\partial \text{loss}/\partial Z^l$.

If we repeatedly apply the chain rule, we get this expression for the gradient of the loss with respect to the pre-activation in the first layer:

$$\frac{\partial \text{loss}}{\partial Z^1} = \underbrace{\underbrace{\frac{\partial \text{loss}}{\partial A^L} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \cdots \cdot \frac{\partial A^2}{\partial Z^2}}_{\partial \text{loss}/\partial Z^2} \cdot \frac{\partial Z^2}{\partial A^1}}_{\partial \text{loss}/\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \ . \tag{8.2}$$

This derivation was informal, to show you the general structure of the computation. In fact, to get the dimensions to all work out, we just have to write it backwards! Let's first understand more about these quantities:

- $\partial \text{loss}/\partial A^L$ is $n^L \times 1$ and depends on the particular loss function you are using.
- $\partial Z^l/\partial A^{l-1}$ is $m^l \times n^l$ and is just W^l (you can verify this by computing a single entry $\partial Z^l_i/\partial A^{l-1}_j$).
- $\partial A^l/\partial Z^l$ is $n^l \times n^l$. It's a little tricky to think about. Each element $a^l_i = f^l(z^l_i)$. This means that $\partial a^l_i/\partial z^l_j = 0$ whenever $i \neq j$. So, the off-diagonal elements of $\partial A^l/\partial Z^l$ are all 0, and the diagonal elements are $\partial a^l_i/\partial z^l_j = {f^l}'(z^l_j)$.

Now, we can rewrite equation 8.2 so that the quantities match up as

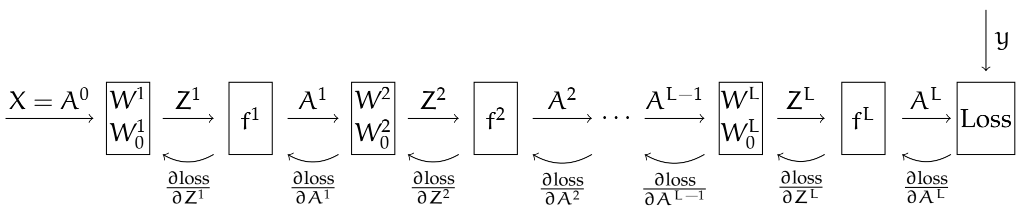
$$\frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \frac{\partial A^{l+1}}{\partial Z^{l+1}} \cdot \ldots W^{L-1} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{A^L} \ . \tag{8.3}$$

Using equation 8.3 to compute $\partial \text{loss}/\partial Z^l$ combined with equation 8.1, lets us find the gradient of the loss with respect to any of the weight matrices.

Study Question: Apply the same reasoning to find the gradients of loss with respect to W^l_0.

This general process is called *error back-propagation*. The idea is that we first do a *forward pass* to compute all the a and z values at all the layers, and finally the actual loss on this example. Then, we can work backward and compute the gradient of the loss with respect to the weights in each layer, starting at layer L and going back to layer 1.

I like to think of this as "blame propagation". You can think of loss as how mad we are about the prediction that the network just made. Then $\partial \text{loss}/\partial A^L$ is how much we blame A^L for the loss. The last module has to take in $\partial \text{loss}/\partial A^L$ and compute $\partial \text{loss}/\partial Z^L$, which is how much we blame Z^L for the loss. The next module (working backwards) takes in $\partial \text{loss}/\partial Z^L$ and computes $\partial \text{loss}/\partial A^{L-1}$. So every module is accepting its blame for the loss, computing how much of it to allocate to each of its inputs, and passing the blame back to them.



If we view our neural network as a sequential composition of modules (in our work so far, it has been an alternation between a linear transformation with a weight matrix, and a component-wise application of a non-linear activation function), then we can define a simple API for a module that will let us compute the forward and backward passes, as well as do the necessary weight updates for gradient descent. Each module has to provide the following "methods." We are already using letters a, x, y, z with particular meanings, so here we will use u as the vector input to the module and v as the vector output:

- forward: $u \to v$
- backward: $u, v, \partial L/\partial v \to \partial L/\partial u$
- weight grad: $u, \partial L/\partial v \to \partial L/\partial W$ only needed for modules that have weights W

In homework we will ask you to implement these modules for neural network components, and then use them to construct a network and train it as described in the next section.

5 Training

Here we go! Here's how to do stochastic gradient descent training on a feed-forward neural network. After this pseudo-code, we motivate the choice of initialization in lines 2 and 3. The actual computation of the gradient values (e.g. $\partial\text{loss}/\partial A^L$) is not directly defined in this code, because we want to make the structure of the computation clear.

Study Question: What is $\partial Z^l/\partial W^l$?

Study Question: Which terms in the code below depend on f^L?

SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \ldots, m^L), (f^1, \ldots, f^L)$)

1 **for** $l = 1$ **to** L
2     $W^l_{ij} \sim \text{Gaussian}(0, 1/m^l)$
3     $W^l_{0j} \sim \text{Gaussian}(0, 1)$
4 **for** $t = 1$ **to** T
5   i = random sample from $\{1, \ldots, n\}$
6     $A^0 = x^{(i)}$
7   // forward pass to compute the output A^L
8   **for** $l = 1$ **to** L
9         $Z^l = W^{lT}A^{l-1} + W^l_0$
10         $A^l = f^l(Z^l)$
11     $\text{loss} = \text{Loss}(A^L, y^{(i)})$
12   **for** $l = L$ **to** 1:
13   // error back-propagation
14         $\partial\text{loss}/\partial A^l =$ **if** $l < L$ **then** $\partial\text{loss}/\partial Z^{l+1} \cdot \partial Z^{l+1}/\partial A^l$ **else** $\partial\text{loss}/\partial A^L$
15         $\partial\text{loss}/\partial Z^l = \partial\text{loss}/\partial A^l \cdot \partial A^l/\partial Z^l$
16   // compute gradient with respect to weights
17         $\partial\text{loss}/\partial W^l = \partial\text{loss}/\partial Z^l \cdot \partial Z^l/\partial W^l$
18         $\partial\text{loss}/\partial W^l_0 = \partial\text{loss}/\partial Z^l \cdot \partial Z^l/\partial W^l_0$
19   // stochastic gradient descent update
20         $W^l = W^l - \eta(t) \cdot \partial\text{loss}/\partial W^l$
21         $W^l_0 = W^l_0 - \eta(t) \cdot \partial\text{loss}/\partial W^l_0$

Initializing W is important; if you do it badly there is a good chance the neural network training won't work well. First, it is important to initialize the weights to random values. We want different parts of the network to tend to "address" different aspects of the problem; if they all start at the same weights, the symmetry will often keep the values from moving in useful directions. Second, many of our activation functions have (near) zero slope when the pre-activation z values have large magnitude, so we generally want to keep the initial weights small so we will be in a situation where the gradients are non-zero, so that gradient descent will have some useful signal about which way to go.

One good general-purpose strategy is to choose each weight at random from a Gaussian (normal) distribution with mean 0 and standard deviation $(1/m)$ where m is the number of inputs to the unit.

Study Question: If the input x to this unit is a vector of 1's, what would the expected pre-activation z value be with these initial weights?

We write this choice (where $\sim$ means "is drawn randomly from the distribution")

$$W^l_{ij} \sim \text{Gaussian}\left(0, \frac{1}{m^l}\right) \quad .$$

It will often turn out (especially for fancier activations and loss functions) that computing

$$\frac{\partial \text{loss}}{\partial Z^L}$$

is easier than computing

$$\frac{\partial \text{loss}}{\partial A^L} \text{ and } \frac{\partial A^L}{\partial Z^L} .$$

So, we may instead ask for an implementation of a loss function to provide a backward method that computes $\partial \text{loss}/\partial Z^L$ directly.

6 Loss functions and activation functions

Different loss functions make different assumptions about the range of inputs they will get as input and, as we have seen, different activation functions will produce output values in different ranges. When you are designing a neural network, it's important to make these things fit together well. In particular, we will think about matching loss functions with the activation function in the last layer, f^L. Here is a table of loss functions and activations that make sense for them:

Loss	f^L
squared	linear
hinge	linear
NLL	sigmoid
NLLM	softmax

6.1 Two-class classification and log likelihood

For classification, the natural loss function is 0-1 loss, but we have already discussed the fact that it's very inconvenient for gradient-based learning because its derivative is discontinuous.

We have also explored *negative log likelihood* (NLL) in chapter 5. It is nice and smooth, and extends nicely to multiple classes as we will see below.

Hinge loss gives us another way, for binary classification problems, to make a smoother objective, penalizing the *margin*s of the labeled points relative to the separator. The hinge loss is defined to be

$$\mathcal{L}_h(\text{guess}, \text{actual}) = \max(1 - \text{guess} \cdot \text{actual}, 0) \;\; ,$$

when actual $\in \{+1, -1\}$. It has the property that, if the sign of guess is the same as the sign of actual and the magnitude of guess is greater than 1, then the loss is 0.

It is trying to enforce not only that the guess have the correct sign, but also that it should be some distance away from the separator. Using hinge loss, together with a squared-norm regularizer, actually forces the learning process to try to find a separator that has the maximum *margin* relative to the data set. This optimization set-up is called a *support vector machine*, and was popular before the renaissance of neural networks and gradient descent, because it has a quadratic form that makes it particularly easy to optimize.

6.2 Multi-class classification and log likelihood

We can extend the idea of NLL directly to multi-class classification with K classes, where the training label is represented with the one-hot vector $y = \begin{bmatrix} y_1, \ldots, y_K \end{bmatrix}^T$, where $y_k = 1$ if the example is of class k. Assume that our network uses *softmax* as the activation function in the last layer, so that the output is $a = \begin{bmatrix} a_1, \ldots, a_K \end{bmatrix}^T$, which represents a probability distribution over the K possible classes. Then, the probability that our network predicts the correct class for this example is $\prod_{k=1}^{K} a_k^{y_k}$ and the log of the probability that it is correct is $\sum_{k=1}^{K} y_k \log a_k$, so

$$\mathcal{L}_{\text{nllm}}(\text{guess}, \text{actual}) = -\sum_{k=1}^{K} \text{actual}_k \cdot \log(\text{guess}_k) \enspace .$$

We'll call this NLLM for *negative log likelihood multiclass.*

Study Question: Show that L_{nllm} for $K = 2$ is the same as L_{nll}.

7 Optimizing neural network parameters

Because neural networks are just parametric functions, we can optimize loss with respect to the parameters using standard gradient-descent software, but we can take advantage of the structure of the loss function and the hypothesis class to improve optimization. As we have seen, the modular function-composition structure of a neural network hypothesis makes it easy to organize the computation of the gradient. As we have also seen earlier, the structure of the loss function as a sum over terms, one per training data point, allows us to consider stochastic gradient methods. In this section we'll consider some alternative strategies for organizing training, and also for making it easier to handle the step-size parameter.

7.1 Batches

Assume that we have an objective of the form

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ ,$$

where h is the function computed by a neural network, and W stands for all the weight matrices and vectors in the network.

When we perform *batch* gradient descent, we use the update rule

$$W := W - \eta \nabla_W J(W) \ ,$$

which is equivalent to

$$W := W - \eta \sum_{i=1}^{n} \nabla_W \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ .$$

So, we sum up the gradient of loss at each training point, with respect to W, and then take a step in the negative direction of the gradient.

In *stochastic* gradient descent, we repeatedly pick a point $(x^{(i)}, y^{(i)})$ at random from the data set, and execute a weight update on that point alone:

$$W := W - \eta \nabla_W \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ .$$

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.

These two methods have offsetting virtues. The batch method takes steps in the exact gradient direction but requires a lot of computation before even a single step can be taken, especially if the data set is large. The stochastic method begins moving right away, and can sometimes make very good progress before looking at even a substantial fraction of the whole data set, but if there is a lot of variability in the data, it might require a very small η to effectively average over the individual steps moving in "competing" directions.

An effective strategy is to "average" between batch and stochastic gradient descent by using *mini-batches*. 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 just on their contributions to the gradient

$$W := W - \eta \sum_{i=1}^{k} \nabla_W \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ .$$

Most neural network software packages are set up to do mini-batches.

Study Question: For what value of k is mini-batch gradient descent equivalent to stochastic gradient descent? To batch gradient descent?

Picking k unique data points at random from a large data-set is potentially computationally difficult. An alternative strategy, if you have an efficient procedure for randomly shuffling the data set (or randomly shufffling a list of indices into the data set) is to operate in a loop, roughly as follows:

```
MINI-BATCH-SGD(NN, data, k)
n = length(data)
while not done:
    RANDOM-SHUFFLE(data)
    for i = 1 to n/k
        BATCH-GRADIENT-UPDATE(NN, data[(i − 1)k : ik])
```

7.2 Adaptive step-size

Picking a value for η is difficult and time-consuming. If it's too small, then convergence is slow and if it's too large, then we risk divergence or slow convergence due to oscillation. This problem is even more pronounced in stochastic or mini-batch mode, because we know we need to decrease the step size for the formal guarantees to hold.

It's also true that, within a single neural network, we may well want to have different step sizes. As our networks become *deep* (with increasing numbers of layers) we can find that magnitude of the gradient of the loss with respect the weights in the last layer, $\partial \text{loss}/\partial W_L$, may be substantially different from the gradient of the loss with respect to the weights in the first layer $\partial \text{loss}/\partial W_1$. If you look carefully at equation 8.3, you 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 a problem of *exploding* or *vanishing* gradients, in which the back-propagated gradient is much too big or small to be used in an update rule with the same step size.

So, we'll consider having an independent step-size parameter *for each weight*, and updating it based on a local view of how the gradient updates have been going.

This section is very strongly influenced by Sebastian Ruder's excellent blog posts on the topic: `ruder.io/optimizing-gradient-descent`

7.2.1 Running averages

We'll start by looking at the notion of a *running average*. It's a computational strategy for estimating a possibly weighted average of a sequence of data. Let our data sequence be $a_1, a_2, \ldots$; then we define a sequence of running average values, $A_0, A_1, A_2, \ldots$ using the equations

$$\begin{aligned} A_0 &= 0 \\ A_t &= \gamma_t A_{t-1} + (1-\gamma_t) a_t \end{aligned}$$

where $\gamma_t \in (0, 1)$. If γ_t is a constant, then this is a *moving* average, in which

$$\begin{aligned} A_T &= \gamma A_{T-1} + (1-\gamma) a_T \\ &= \gamma(\gamma A_{T-2} + (1-\gamma) a_{T-1}) + (1-\gamma) a_T \\ &= \sum_{t=0}^{T} \gamma^{T-t} (1-\gamma) a_t \end{aligned}$$

So, you can see that inputs a_t closer to the end of the sequence have more effect on A_t than early inputs.

If, instead, we set $\gamma_t = (t-1)/t$, then we get the actual average.

Study Question: Prove to yourself that the previous assertion holds.

7.2.2 Momentum

Now, we can use methods that are a bit like running averages to describe strategies for computing η. The simplest method is *momentum*, in which we try to "average" recent gradient updates, so that if they have been bouncing back and forth in some direction, we take out that component of the motion. For momentum, we have

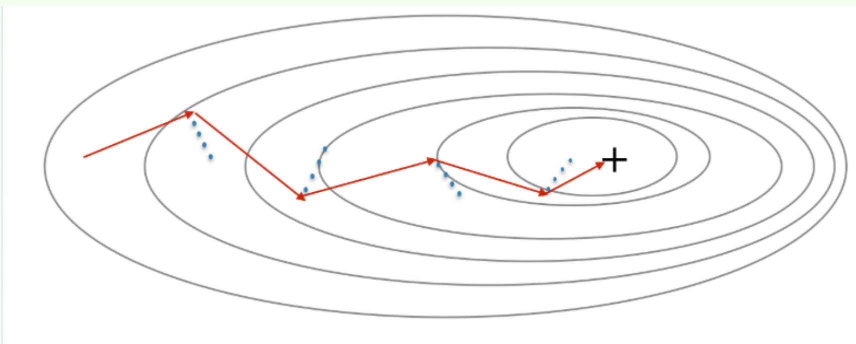
$$\begin{aligned} V_0 &= 0 \\ V_t &= \gamma V_{t-1} + \eta \nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - V_t \end{aligned}$$

This doesn't quite look like an adaptive step size. But what we can see is that, if we let $\eta = \eta'(1-\gamma)$, then the rule looks exactly like doing an update with step size η' on a moving average of the gradients with parameter γ:

$$\begin{aligned} M_0 &= 0 \\ M_t &= \gamma M_{t-1} + (1-\gamma)\nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - \eta' M_t \end{aligned}$$

Study Question: Prove to yourself that these formulations are equivalent.

We will find that V_t will be bigger in dimensions that consistently have the same sign for ∇_W and smaller for those that don't. Of course we now have *two* parameters to set (η and γ), but the hope is that the algorithm will perform better overall, so it will be worth trying to find good values for them. Often γ is set to be something like 0.9.



The red arrows show the update after one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient with respect to the mini-batch at each step. Momentum smooths the path taken towards the local minimum and leads to faster convergence.

Study Question: If you set $\gamma = 0.1$, would momentum have more of an effect or less of an effect than if you set it to 0.9?

7.2.3 Adadelta

Another useful idea is this: we would like to take larger steps in parts of the space where $J(W)$ is nearly flat (because there's no risk of taking too big a step due to the gradient being large) and smaller steps when it is steep. We'll apply this idea to each weight independently, and end up with a method called *adadelta*, which is a variant on *adagrad* (for adaptive gradient). Even though our weights are indexed by layer, input unit and output unit, for simplicity here, just let W_j be any weight in the network (we will do the same thing for all of them).

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ G_{t,j} &= \gamma G_{t-1,j} + (1-\gamma) g_{t,j}^2 \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j}+\epsilon}} g_{t,j} \end{aligned}$$

The sequence $G_{t,j}$ is a moving average of the square of the jth component of the gradient. We square it in order to be insensitive to the sign—we want to know whether the magnitude is big or small. Then, we perform a gradient update to weight j, but divide the step size by $\sqrt{G_{t,j}+\epsilon}$, which is larger when the surface is steeper in direction j at point W_{t-1} in weight space; this means that the step size will be smaller when it's steep and larger when it's flat.

7.2.4 Adam

Adam has become the default method of managing step sizes neural networks. It combines the ideas of momentum and adadelta. We start by writing moving averages of the gradient and squared gradient, which reflect estimates of the mean and variance of the gradient for weight j:

Although, interestingly, it may actually violate the convergence conditions of SGD: arxiv.org/abs/1705.08292

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ m_{t,j} &= B_1 m_{t-1,j} + (1-B_1) g_{t,j} \\ v_{t,j} &= B_2 v_{t-1,j} + (1-B_2) g_{t,j}^2 \ . \end{aligned}$$

A problem with these estimates is that, if we initialize $m_0 = v_0 = 0$, they will always be biased (slightly too small). So we will correct for that bias by defining

$$\begin{aligned} \hat{m}_{t,j} &= \frac{m_{t,j}}{1-B_1^t} \\ \hat{v}_{t,j} &= \frac{v_{t,j}}{1-B_2^t} \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{\hat{v}_{t,j}+\epsilon}} \hat{m}_{t,j} \ . \end{aligned}$$

Note that B_1^t is B_1 raised to the power t, and likewise for B_2^t. To justify these corrections, note that if we were to expand $m_{t,j}$ in terms of $m_{0,j}$ and $g_{0,j}, g_{1,j}, \ldots, g_{t,j}$ the coefficients would sum to 1. However, the coefficient behind $m_{0,j}$ is B_1^t and since $m_{0,j} = 0$, the sum of coefficients of non-zero terms is $1 - B_1^t$, hence the correction. The same justification holds for $v_{t,j}$.

Now, our update for weight j has a step size that takes the steepness into account, as in adadelta, but also tends to move in the same direction, as in momentum. The authors of this method propose setting $B_1 = 0.9, B_2 = 0.999, \epsilon = 10^{-8}$. Although we now have even more parameters, Adam is not highly sensitive to their values (small changes do not have a huge effect on the result).

Study Question: Define $\hat{m}_j$ directly as a moving average of $g_{t,j}$. What is the decay (γ parameter)?

Even though we now have a step-size for each weight, and we have to update various quantities on each iteration of gradient descent, it's relatively easy to implement by maintaining a matrix for each quantity ($m_t^\ell, v_t^\ell, g_t^\ell, {g_t^2}^\ell$) in each layer of the network.

8 Regularization

So far, we have only considered optimizing loss on the training data as our objective for neural network training. But, as we have discussed before, there is a risk of overfitting if we do this. The pragmatic fact is that, in current deep neural networks, which tend to be very large and to be trained with a large amount of data, overfitting is not a huge problem. This runs counter to our current theoretical understanding and the study of this question is a hot area of research. Nonetheless, there are several strategies for regularizing a neural network, and they can sometimes be important.

8.1 Methods related to ridge regression

One group of strategies can, interestingly, be shown to have similar effects: early stopping, weight decay, and adding noise to the training data.

Result is due to Bishop, described in his textbook and here `doi.org/10.1162/neco.1995.7.1.108`.

Early stopping is the easiest to implement and is in fairly common use. The idea is to train on your training set, but at every *epoch* (pass through the whole training set, or possibly more frequently), evaluate the loss of the current W on a *validation set*. It will generally be the case that the loss on the training set goes down fairly consistently with each iteration, the loss on the validation set will initially decrease, but then begin to increase again. Once you see that the validation loss is systematically increasing, you can stop training and return the weights that had the lowest validation error.

Another common strategy is to simply penalize the norm of all the weights, as we did in ridge regression. This method is known as *weight decay*, because when we take the gradient of the objective

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

we end up with an update of the form

$$\begin{aligned} W_t &= W_{t-1} - \eta \left(\left(\nabla_W \text{Loss}(\text{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) + \lambda W_{t-1} \right) \\ &= W_{t-1}(1 - \lambda\eta) - \eta \left(\nabla_W \text{Loss}(\text{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) \quad . \end{aligned}$$

This rule has the form of first "decaying" W_{t-1} by a factor of $(1 - \lambda\eta)$ and then taking a gradient step.

Finally, the same effect can be achieved by perturbing the $x^{(i)}$ values of the training data by adding a small amount of zero-mean normally distributed noise before each gradient computation. It makes intuitive sense that it would be more difficult for the network to overfit to particular training data if they are changed slightly on each training step.

8.2 Dropout

Dropout is a regularization method that was designed to work with deep neural networks. The idea behind it is, rather than perturbing the data every time we train, we'll perturb the network! We'll do this by randomly, on each training step, selecting a set of units in each layer and prohibiting them from participating. Thus, all of the units will have to take a kind of "collective" responsibility for getting the answer right, and will not be able to rely on any small subset of the weights to do all the necessary computation. This tends also to make the network more robust to data perturbations.

During the training phase, for each training example, for each unit, randomly with probability p temporarily set $a_j^\ell := 0$. There will be no contribution to the output and no gradient update for the associated unit.

Study Question: Be sure you understand why, when using SGD, setting an activation value to 0 will cause that unit's weights not to be updated on that iteration.

When we are done training and want to use the network to make predictions, we multiply all weights by p to achieve the same average activation levels.

Implementing dropout is easy! In the forward pass during training, we let

$$a^\ell = f(z^\ell) * d^\ell$$

where $*$ denotes component-wise product and d^ℓ is a vector of 0's and 1's drawn randomly with probability p. The backwards pass depends on a^ℓ, so we do not need to make any further changes to the algorithm.

It is common to set p to 0.5, but this is something one might experiment with to get good results on your problem and data.

8.3 Batch Normalization

A more modern alternative to dropout, which tends to achieve better performance, is *batch normalization*. It was originally developed to address a problem of *covariate shift*: that is, if you consider the second layer of a two-layer neural network, the distribution of its input values is changing over time as the first layer's weights change. Learning when the input distribution is changing is extra difficult: you have to change your weights to improve your predictions, but also just to compensate for a change in your inputs (imagine, for instance, that the magnitude of the inputs to your layer is increasing over time—then your weights will have to decrease, just to keep your predictions the same).

For more details see `arxiv.org/abs/1502.03167`.

So, when training with mini-batches, the idea is to *standardize* the input values for each mini-batch, just in the way that we did it in section 2.3 of chapter 4, subtracting off the mean and dividing by the standard deviation of each input dimension. This means that the scale of the inputs to each layer remains the same, no matter how the weights in previous layers change. However, this somewhat complicates matters, because the computation of the weight updates will need to take into account that we are performing this transformation. In the modular view, batch normalization can be seen as a module that is applied to z^l, interposed after the product with W^l and before input to f^l.

Batch normalization ends up having a regularizing effect for similar reasons that adding noise and dropout do: each mini-batch of data ends up being mildly perturbed, which prevents the network from exploiting very particular values of the data points.

Let's think of the batch-norm layer as taking z^l as input and producing an output $\widehat{Z}^l$ as output. But now, instead of thinking of Z^l as an $n^l \times 1$ vector, we have to explicitly think about handling a mini-batch of data of size K, all at once, so Z^l will be $n^l \times K$, and so will the output $\widehat{Z}^l$.

Our first step will be to compute the *batchwise* mean and standard deviation. Let μ^l be the $n^l \times 1$ vector where

$$\mu_i^l = \frac{1}{K} \sum_{j=1}^{K} Z_{ij}^l \ ,$$

and let σ^l be the $n^l \times 1$ vector where

$$\sigma^l_i = \sqrt{\frac{1}{K}\sum_{j=1}^{K}(Z^l_{ij} - \mu_i)^2} \ .$$

The basic normalized version of our data would be a matrix, element (i, j) of which is

$$\overline{Z}^l_{ij} = \frac{Z^l_{ij} - \mu^l_i}{\sigma^l_i + \epsilon} \ ,$$

where ϵ is a very small constant to guard against division by zero. However, if we let these be our $\widehat{Z}^l$ values, we really are forcing something too strong on our data—our goal was to normalize across the data batch, but not necessarily force the output values to have exactly mean 0 and standard deviation 1. So, we will give the layer the "opportunity" to shift and scale the outputs by adding new weights to the layer. These weights are G^l and B^l, each of which is an $n^l \times 1$ vector. Using the weights, we define the final output to be

$$\widehat{Z}^l_{ij} = G^l_i \overline{Z}^l_{ij} + B^l_i \ .$$

That's the forward pass. Whew!

Now, for the backward pass, we have to do two things: given $\partial L/\partial \widehat{Z}^l$,

- Compute $\partial L/\partial Z^l$ for back-propagation, and
- Compute $\partial L/\partial G^l$ and $\partial L/\partial B^l$ for gradient updates of the weights in this layer.

Schematically

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial B} \ .$$

It's hard to think about these derivatives in matrix terms, so we'll see how it works for the components. B_i contributes to $\widehat{Z}_{ij}$ for all data points j in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial B_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial B_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}} \ ,\end{aligned}$$

Similarly, G_i contributes to $\widehat{Z}_{ij}$ for all data points j in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial G_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial G_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\overline{Z}_{ij} \ .\end{aligned}$$

Now, let's figure out how to do backprop. We can start schematically:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial Z} \ .$$

And because dependencies only exist across the batch, but not across the unit outputs,

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K}\frac{\partial L}{\partial \widehat{Z}_{ik}}\frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} \ .$$

The next step is to note that

$$\begin{aligned}\frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} &= \frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} \\ &= G_i \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}}\end{aligned}$$

And now that

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \left(\delta_{jk} - \frac{\partial \mu_i}{\partial Z_{ij}}\right) \frac{1}{\sigma_i} - \frac{Z_{ik} - \mu_i}{\sigma_i^2} \frac{\partial \sigma_i}{\partial Z_{ij}} \ ,$$

where $\delta_{jk} = 1$ if $j = k$ and $\delta_{jk} = 0$ otherwise. Getting close! We need two more small parts:

$$\begin{aligned}\frac{\partial \mu_i}{\partial Z_{ij}} &= \frac{1}{K} \\ \frac{\partial \sigma_i}{\partial Z_{ij}} &= \frac{Z_{ij} - \mu_i}{K\sigma_i}\end{aligned}$$

Putting the whole crazy thing together, we get

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} G_i \frac{1}{K\sigma_i} \left(\delta_{jk} K - 1 - \frac{(Z_{ik} - \mu_i)(Z_{ij} - \mu_i)}{\sigma_i^2}\right)$$

CHAPTER 9

Convolutional Neural Networks

So far, we have studied what are called *fully connected* neural networks, in which all of the units at one layer are connected to all of the units in the next layer. This is a good arrangement when we don't know anything about what kind of mapping from inputs to outputs we will be asking the network to learn to approximate. But if we *do* know something about our problem, it is better to build it into the structure of our neural network. Doing so can save computation time and significantly diminish the amount of training data required to arrive at a solution that generalizes robustly.

One very important application domain of neural networks, where the methods have achieved an enormous amount of success in recent years, is signal processing. Signals might be spatial (in two-dimensional camera images or three-dimensional depth or CAT scans) or temporal (speech or music). If we know that we are addressing a signal-processing problem, we can take advantage of *invariant* properties of that problem. In this chapter, we will focus on two-dimensional spatial problems (images) but use one-dimensional ones as a simple example. Later, we will address temporal problems.

Imagine that you are given the problem of designing and training a neural network that takes an image as input, and outputs a classification, which is positive if the image contains a cat and negative if it does not. An image is described as a two-dimensional array of *pixels*, each of which may be represented by three integer values, encoding intensity levels in red, green, and blue color channels.

A *pixel* is a "picture element."

There are two important pieces of prior structural knowledge we can bring to bear on this problem:

- **Spatial locality:** The set of pixels we will have to take into consideration to find a cat will be near one another in the image.
- **Translation invariance:** The pattern of pixels that characterizes a cat is the same no matter where in the image the cat occurs.

So, for example, we won't have to consider some combination of pixels in the four corners of the image, in order to see if they encode cat-ness.

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

We will design neural network structures that take advantage of these properties.

1 Filters

We begin by discussing *image filters*. An image filter is a function that takes in a local spatial neighborhood of pixel values and detects the presence of some pattern in that data.

Unfortunately in AI/ML/CS/Math, the word "filter" gets used in many ways: in addition to the one we describe here, it can describe a temporal process (in fact, our moving averages are a kind of filter) and even a somewhat esoteric algebraic structure.

Let's consider a very simple case to start, in which we have a 1-dimensional binary "image" and a filter F of size two. The filter is a vector of two numbers, which we will move along the image, taking the dot product between the filter values and the image values at each step, and aggregating the outputs to produce a new image.

Let X be the original image, of size d; then pixel i of the the output image is specified by

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

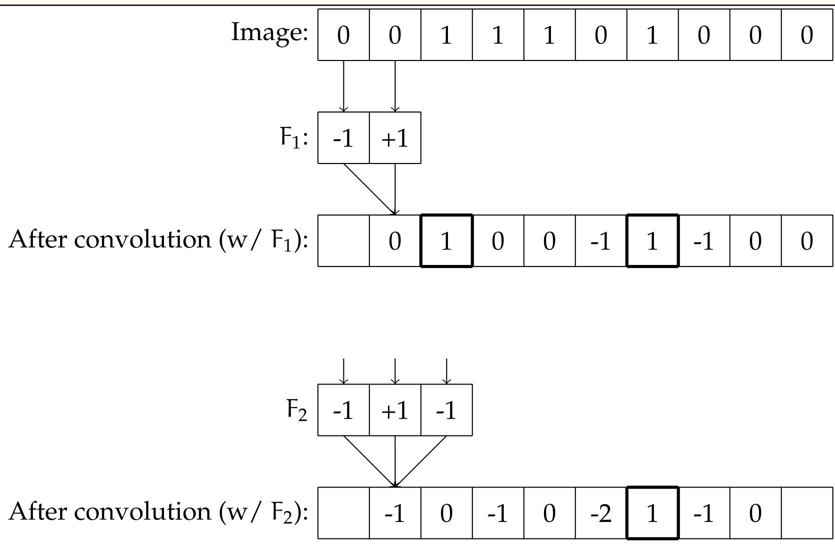
To ensure that the output image is also of dimension d, we will generally "pad" the input image with 0 values if we need to access pixels that are beyond the bounds of the input image. This process of applying the filter to the image to create a new image is called "convolution."

And filters are also sometimes called *convolutional kernels*.

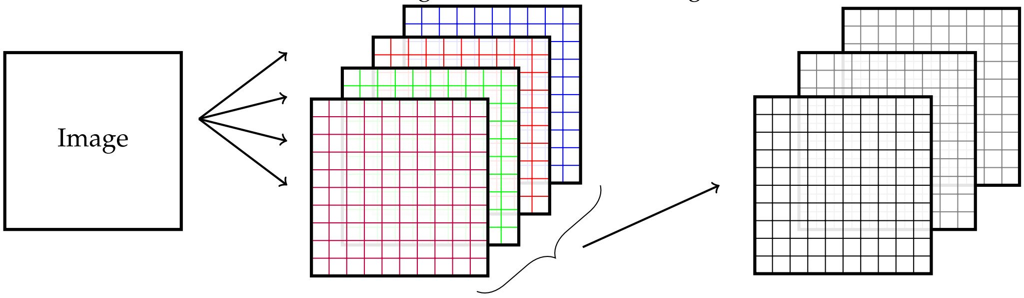
If you are already familiar with what a convolution is, you might notice that this definition corresponds to what is often called a correlation and not to a convolution. Indeed, correlation and convolution refer to different operations in signal processing. However, in the neural networks literature, most libraries implement the correlation (as described in this chapter) but call it convolution. The distinction is not significant; in principle, if convolution is required to solve the problem, the network could learn the necessary weights. For a discussion of the difference between convolution and correlation and the conventions used in the literature you can read section 9.1 in this excellent book: `https://www.deeplearningbook.org`.

Here is a concrete example. Let the filter $F_1 = (-1, +1)$. Then given the first image below, we can convolve it with filter F_1 to obtain the second image. You can think of this filter as a detector for "left edges" in the original image—to see this, look at the places where there is a 1 in the output image, and see what pattern exists at that position in the input image. Another interesting filter is $F_2 = (-1, +1, -1)$. The third image below shows the result of convolving the first image with F_2.

Study Question: Convince yourself that filter F_2 can be understood as a detector for isolated positive pixels in the binary image.



Two-dimensional versions of filters like these are thought to be found in the visual cortex of all mammalian brains. Similar patterns arise from statistical analysis of natural images. Computer vision people used to spend a lot of time hand-designing *filter banks*. A filter bank is a set of sets of filters, arranged as shown in the diagram below.



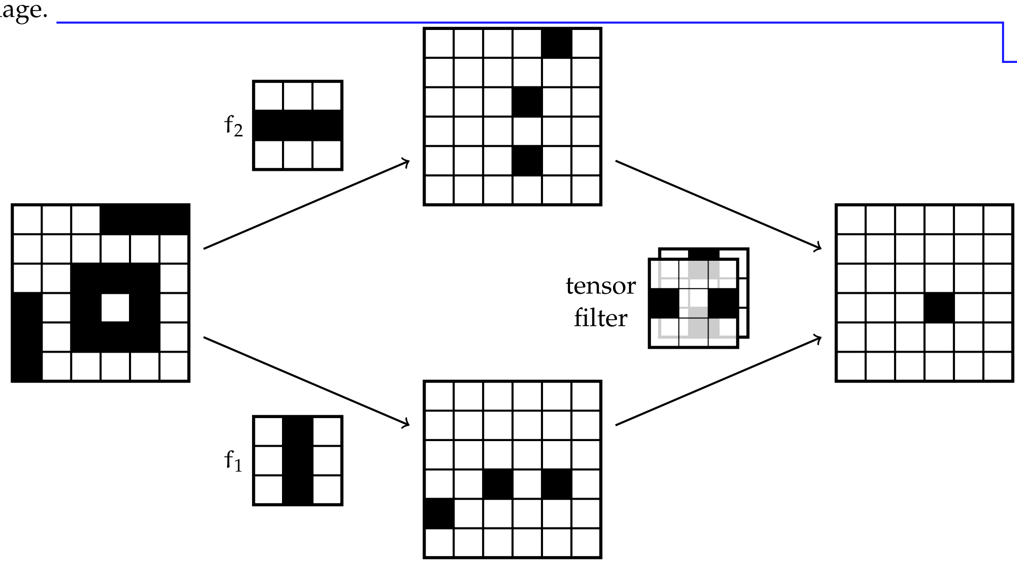
All of the filters in the first group are applied to the original image; if there are k such filters, then the result is k new images, which are called *channels*. Now imagine stacking all these new images up so that we have a cube of data, indexed by the original row and column indices of the image, as well as by the channel. The next set of filters in the filter bank will generally be *three-dimensional*: each one will be applied to a sub-range of the row and column indices of the image and to all of the channels.

These 3D chunks of data are called *tensors*. The algebra of tensors is fun, and a lot like matrix algebra, but we won't go into it in any detail.

We will use a popular piece of neural-network software called *Tensorflow* because it makes operations on tensors easy.

Here is a more complex example of two-dimensional filtering. We have two 3×3 filters in the first layer, f_1 and f_2. You can think of each one as "looking" for three pixels in a row, f_1 vertically and f_2 horizontally. Assuming our input image is $n \times n$, then the result of filtering with these two filters is an $n \times n \times 2$ tensor. Now we apply a tensor filter (hard to draw!) that "looks for" a combination of two horizontal and two vertical bars (now represented by individual pixels in the two channels), resulting in a single final $n \times n$ image.

When we have a color image as input, we treat it as having 3 channels, and hence as an $n \times n \times 3$ tensor.



We are going to design neural networks that have this structure. Each "bank" of the filter bank will correspond to a neural-network layer. The numbers in the individual filters will be the "weights" (plus a single additive bias or offset value for each filter) of the network, which we will train using gradient descent. What makes this interesting and powerful (and somewhat confusing at first) is that the same weights are used many many times in the computation of each layer. This *weight sharing* means that we can express a transformation on a large image with relatively few parameters; it also means we'll have to take care in figuring out exactly how to train it!

We will define a filter layer l formally with:

For simplicity, we are assuming that all images and filters are square (having the same number of rows and columns). That is in no way necessary, but is usually fine and definitely simplifies our notation.

- *number* of filters m^l;
- *size* of one filter is $k^l \times k^l \times m^{l-1}$ plus 1 bias value (for this one filter);
- *stride* s^l is the spacing at which we apply the filter to the image; in all of our examples so far, we have used a stride of 1, but if we were to "skip" and apply the filter only at odd-numbered indices of the image, then it would have a stride of two (and produce a resulting image of half the size);
- *input tensor size* $n^{l-1} \times n^{l-1} \times m^{l-1}$
- *padding*: p^l is how many extra pixels – typically with value 0 – we add around the edges of the input. For an input of size $n^{l-1} \times n^{l-1} \times m^{l-1}$, our new effective input size with padding becomes $(n^{l-1} + 2 * p^l) \times (n^{l-1} + 2 * p^l) \times m^{l-1}$.

This layer will produce an output tensor of size $n^l \times n^l \times m^l$, where $n^l = \lceil (n^{l-1} + 2 * p^l - (k^l - 1))/s^l \rceil$.[1] The weights are the values defining the filter: there will be m^l different $k^l \times k^l \times m^{l-1}$ tensors of weight values; plus each filter may have a bias term, which means there is one more weight value per filter. A filter with a bias operates just like the filter examples above, except we add the bias to the output. For instance, if we incorporated a bias term of 0.5 into the filter F_2 above, the output would be $(-0.5, 0.5, -0.5, 0.5, -1.5, 1.5, -0.5, 0.5)$ instead of $(-1, 0, -1, 0, -2, 1, -1, 0)$.

This may seem complicated, but we get a rich class of mappings that exploit image structure and have many fewer weights than a fully connected layer would.

Study Question: How many weights are in a convolutional layer specified as above?

Study Question: If we used a fully-connected layer with the same size inputs and outputs, how many weights would it have?

2 Max Pooling

It is typical to structure filter banks into a *pyramid*, in which the image sizes get smaller in successive layers of processing. The idea is that we find local patterns, like bits of edges in the early layers, and then look for patterns in those patterns, etc. This means that, effectively, we are looking for patterns in larger pieces of the image as we apply successive filters. Having a stride greater than one makes the images smaller, but does not necessarily aggregate information over that spatial range.

Both in engineering and in nature

Another common layer type, which accomplishes this aggregation, is *max pooling*. A max pooling layer operates like a filter, but has no weights. *You can think of it as a pure functional layer, like a ReLU layer in a fully connected network.* It has a filter size, as in a filter layer, but simply returns the maximum value in its field. Usually, we apply max pooling with the following traits:

We sometimes use the term *receptive field* or just *field* to mean the area of an input image that a filter is being applied to.

- stride > 1, so that the resulting image is smaller than the input image; and
- $k \geqslant$ stride, so that the whole image is covered.

[1]Here, $\lceil \cdot \rceil$ is known as the *ceiling* function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.

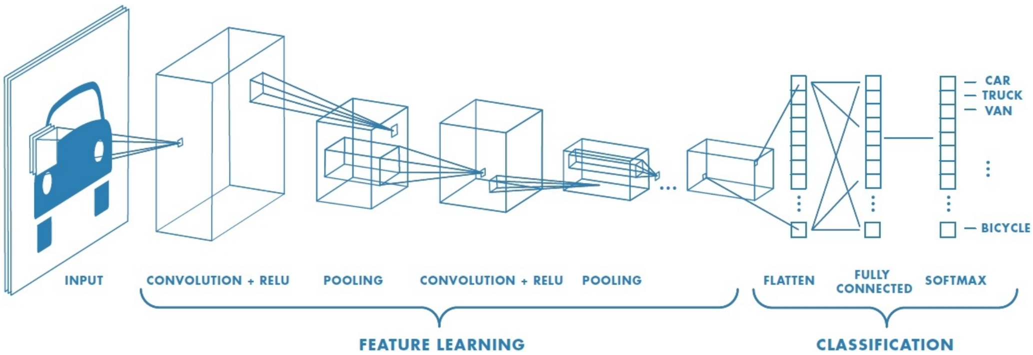
As a result of applying a max pooling layer, we don't keep track of the precise location of a pattern. This helps our filters to learn to recognize patterns independent of their location.

Consider a max pooling layer of stride $= k = 2$. This would map a $64 \times 64 \times 3$ image to a $32 \times 32 \times 3$ image. Note that max pooling layers do not have additional bias or offset values.

Study Question: Maximilian Poole thinks it would be a good idea to add two max pooling layers of size k, one right after the other, to their network. What single layer would be equivalent?

3 Typical architecture

Here is the form of a typical convolutional network:



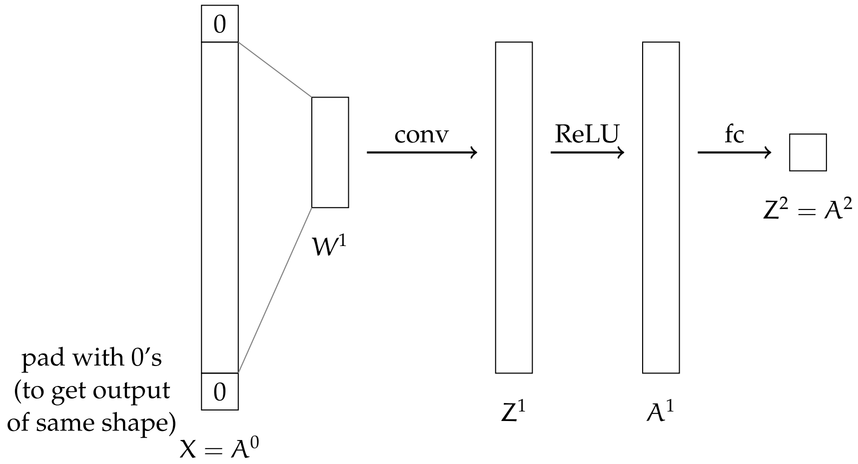
Source: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html

After each filter layer there is generally a ReLU layer; there maybe be multiple filter/ReLU layers, then a max pooling layer, then some more filter/ReLU layers, then max pooling. Once the output is down to a relatively small size, there is typically a last fully-connected layer, leading into an activation function such as softmax that produces the final output. The exact design of these structures is an art—there is not currently any clear theoretical (or even systematic empirical) understanding of how these various design choices affect overall performance of the network.

The critical point for us is that this is all just a big neural network, which takes an input and computes an output. The mapping is a differentiable function of the weights, which means we can adjust the weights to decrease the loss by performing gradient descent, and we can compute the relevant gradients using back-propagation!

Well, the derivative is not continuous, both because of the ReLU and the max pooling operations, but we ignore that fact.

Let's work through a *very* simple example of how back-propagation can work on a convolutional network. The architecture is shown below. Assume we have a one-dimensional single-channel image, of size $n \times 1 \times 1$ and a single $k \times 1 \times 1$ filter (where we omit the filter bias) in the first convolutional layer. Then we pass it through a ReLU layer and a fully-connected layer with no additional activation function on the output.



For simplicity assume k is odd, let the input image $X = A^0$, and assume we are using squared loss. Then we can describe the forward pass as follows:

$$\begin{aligned} Z_i^1 &= {W^1}^T \cdot A^0_{[i-\lfloor k/2 \rfloor : i+\lfloor k/2 \rfloor]} \\ A^1 &= \text{ReLU}(Z^1) \\ A^2 &= {W^2}^T A^1 \\ L(A^2, y) &= (A^2 - y)^2 \end{aligned}$$

Study Question: For a filter of size k, how much padding do we need to add to the top and bottom of the image?

How do we update the weights in filter W^1?

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

- $\partial Z^1/\partial W^1$ is the $k \times n$ matrix such that $\partial Z_i^1/\partial W_j^1 = X_{i-\lfloor k/2 \rfloor + j - 1}$. So, for example, if $i = 10$, which corresponds to column 10 in this matrix, which illustrates the dependence of pixel 10 of the output image on the weights, and if $k = 5$, then the elements in column 10 will be $X_8, X_9, X_{10}, X_{11}, X_{12}$.

- $\partial A^1/\partial Z^1$ is the $n \times n$ diagonal matrix such that

$$\partial A_i^1/\partial Z_i^1 = \begin{cases} 1 & \text{if } Z_i^1 > 0 \\ 0 & \text{otherwise} \end{cases}$$

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

Multiplying these components yields the desired gradient, of shape $k \times 1$.

CHAPTER 10

Sequential models

So far, we have limited our attention to domains in which each output y is assumed to have been generated as a function of an associated input x, and our hypotheses have been "pure" functions, in which the output depends only on the input (and the parameters we have learned that govern the function's behavior). In the next few weeks, we are going to consider cases in which our models need to go beyond functions.

- In *recurrent neural networks*, the hypothesis that we learn is not a function of a single input, but of the whole sequence of inputs that the predictor has received.
- In *reinforcement learning*, the hypothesis is either a *model* of a domain (such as a game) as a recurrent system or a *policy* which is a pure function, but whose loss is determined by the ways in which the policy interacts with the domain over time.

Before we engage with those forms of learning, we will study models of sequential or recurrent systems that underlie the learning methods.

1 State machines

A *state machine* is a description of a process (computational, physical, economic) in terms of its potential sequences of *states*.

This is such a pervasive idea that it has been given many names in many subareas of computer science, control theory, physics, etc., including: *automaton*, *transducer*, *dynamical system*, *system*, etc.

The *state* of a system is defined to be all you would need to know about the system to predict its future trajectories as well as possible. It could be the position and velocity of an object or the locations of your pieces on a game board, or the current traffic densities on a highway network.

Formally, we define a *state machine* as $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f, g)$ where

There are a huge number of major and minor variations on the idea of a state machine. We'll just work with one specific one in this section and another one in the next, but don't worry if you see other variations out in the world!

- $\mathcal{S}$ is a finite or infinite set of possible states;
- $\mathcal{X}$ is a finite or infinite set of possible inputs;
- $\mathcal{Y}$ is a finite or infinite set of possible outputs;
- $s_0 \in \mathcal{S}$ is the initial state of the machine;
- $f : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ is a *transition function*, which takes an input and a previous state and produces a next state;

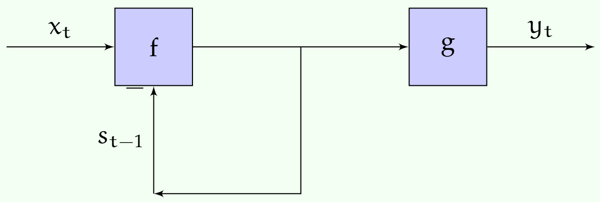
- $g : \mathcal{S} \to \mathcal{Y}$ is an *output function*, which takes a state and produces an output.

The basic operation of the state machine is to start with state s_0, then iteratively compute for $t \geqslant 1$:

In some cases, we will pick a starting state from a set or distribution.

$$s_t = f(s_{t-1}, x_t)$$
$$y_t = g(s_t)$$

The diagram below illustrates this process. Note that the "feedback" connection of s_t back into f has to be buffered or delayed by one time step—-otherwise what it is computing would not generally be well defined.



So, given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs

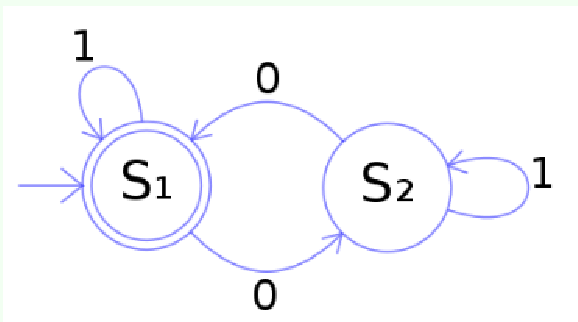
$$\underbrace{g(f(s_0, x_1))}_{y_1}, \underbrace{g(f(f(s_0, x_1), x_2))}_{y_2}, \ldots \quad .$$

We sometimes say that the machine *transduces* sequence x into sequence y. The output at time t can have dependence on inputs from steps 1 to t.

One common form is *finite state machines*, in which $\mathcal{S}$, $\mathcal{X}$, and $\mathcal{Y}$ are all finite sets. They are often described using *state transition diagrams* such as the one below, in which nodes stand for states and arcs indicate transitions. Nodes are labeled by which output they generate and arcs are labeled by which input causes the transition.

All computers can be described, at the digital level, as finite state machines. Big, but finite!

One can verify that the state machine below reads binary strings and determines the parity of the number of zeros in the given string. Check for yourself that all inputted binary strings end in state S_1 if and only if they contain an even number of zeros.



Another common structure that is simple but powerful and used in signal processing and control is *linear time-invariant (LTI) systems*. In this case, $\mathcal{S} = \mathbb{R}^m$, $\mathcal{X} = \mathbb{R}^l$ and $\mathcal{Y} = \mathbb{R}^n$, and f and g are linear functions of their inputs. In discrete time, they can be defined by a linear difference equation, like

$$y[t] = 3y[t-1] + 6y[t-2] + 5x[t] + 3x[t-2] \quad ,$$

(where $y[t]$ is y at time t) and can be implemented using state to store relevant previous input and output information.

We will study *recurrent neural networks* which are a lot like a non-linear version of an LTI system, with transition and output functions

$$f(s, x) = f_1(W^{sx}x + W^{ss}s + W_0^{ss})$$
$$g(s) = f_2(W^0 s + W_0^0)$$

defined by weight matrices

$$W^{sx} : m \times \ell$$
$$W^{ss} : m \times m$$
$$W_0^{ss} : m \times 1$$
$$W^0 : n \times m$$
$$W_0^0 : n \times 1$$

and activation functions f_1 and f_2. We will see that it's actually possible to learn weight values for a recurrent neural network using gradient descent.

2 Markov decision processes

A *Markov decision process* (MDP) is a variation on a state machine in which:

- The transition function is *stochastic*, meaning that it defines a probability distribution over the next state given the previous state and input, but each time it is evaluated it draws a new state from that distribution.
- The output is equal to the state (that is g is the identity function).
- Some states (or state-action pairs) are more desirable than others.

An MDP can be used to model interaction with an outside "world," such as a single-player game.

We will focus on the case in which $\mathcal{S}$ and $\mathcal{X}$ are finite, and will call the input set $\mathcal{A}$ for *actions* (rather than $\mathcal{X}$). The idea is that an agent (a robot or a game-player) can model its environment as an MDP and try to choose actions that will drive the process into states that have high scores.

Formally, an MDP is $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ where:

- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a *transition model*, where

$$T(s, a, s') = P(S_t = s' | S_{t-1} = s, A_{t-1} = a) \quad ,$$

  specifying a conditional probability distribution;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, where $R(s, a)$ specifies how desirable it is to be in state s and take action a; and
- $\gamma \in [0, 1]$ is a *discount factor*, which we'll discuss in section 2.2.

A *policy* is a function $\pi : \mathcal{S} \to \mathcal{A}$ that specifies what action to take in each state.

Recall that stochastic is another word for *probabilistic*; we don't say "random" because that can be interpreted in two ways, both of which are incorrect. We don't pick the transition function itself at random from a distribution. The transition function doesn't pick its output *uniformly* at random.

There is an interesting variation on MDPs, called a *partially observable* MDP, in which the output is also drawn from a distribution depending on the state.

And there is an interesting, direct extension to two-player zero-sum games, such as Chess and Go.

The notation here uses capital letters, like S, to stand for random variables and small letters to stand for concrete values. So S_t here is a random variable that can take on elements of $\mathcal{S}$ as values.

2.1 Finite-horizon solutions

Given an MDP, our goal is typically to find a policy that is optimal in the sense that it gets as much total reward as possible, in expectation over the stochastic transitions that the domain makes. In this section, we will consider the case where there is a finite *horizon* H, indicating the total number of steps of interaction that the agent will have with the MDP.

2.1.1 Evaluating a given policy

Before we can talk about how to find a good policy, we have to specify a measure of the goodness of a policy. We will do so by defining for a given MDP policy π and horizon h, the "horizon h *value*" of a state, $V_\pi^h(s)$. We do this by induction on the horizon, which is the *number of steps left to go*.

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

$$V_\pi^0(s) = 0 \ .$$

Then, the value of a policy in state s at horizon $h+1$ is equal to the reward it will get in state s plus the next state's expected horizon h value. So, starting with horizons 1 and 2, and then moving to the general case, we have:

$$\begin{aligned}
V_\pi^1(s) &= R(s, \pi(s)) + 0 \\
V_\pi^2(s) &= R(s, \pi(s)) + \sum_{s'} T(s, \pi(s), s') \cdot R(s', \pi(s')) \\
&\vdots \\
V_\pi^h(s) &= R(s, \pi(s)) + \sum_{s'} T(s, \pi(s), s') \cdot V_\pi^{h-1}(s')
\end{aligned}$$

The sum over s' is an *expected value*: it considers all possible next states s', and computes an 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'.

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

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 and only if for all $s \in \mathcal{S}$, $V_{\pi_1}^h(s) \geqslant V_{\pi_2}^h(s)$ and there exists at least one $s \in \mathcal{S}$ such that $V_{\pi_1}^h(s) > V_{\pi_2}^h(s)$.

2.1.2 Finding an optimal policy

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, then you'd move toward the reward 10 state, but if you only have step left to go, you should go in the direction that will allow you to gain 5!

One way to find an optimal policy is to compute an *optimal action-value function*, Q. We define $Q^h(s, a)$ to be the expected value of

- starting in state s,

- executing action a, and
- continuing for $h-1$ more steps executing an optimal policy for the appropriate horizon on each step.

Similar to our definition of V for evaluating a policy, we define the Q function recursively according to the horizon. The only difference is that, on each step with horizon 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.

$$
\begin{aligned}
Q^0(s,a) &= 0 \\
Q^1(s,a) &= R(s,a) + 0 \\
Q^2(s,a) &= R(s,a) + \sum_{s'} T(s,a,s') \max_{a'} R(s',a') \\
&\vdots \\
Q^h(s,a) &= R(s,a) + \sum_{s'} T(s,a,s') \max_{a'} Q^{h-1}(s',a')
\end{aligned}
$$

We can solve for the values of Q with a simple recursive algorithm called *value iteration* which just computes Q^h starting from horizon 0 and working backward to the desired horizon H. Given Q, an optimal policy is easy to find:

$$\pi_h^*(s) = \arg\max_a Q^h(s,a) \quad .$$

There may be multiple possible optimal policies.

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 simple 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 $Q^4(s,a)$ for all (s,a) by directly using the definition:

- To compute $Q^4(s_i,a_j)$ for any one (s_i,a_j), we would need to compute $Q^3(s,a)$ for all (s,a) pairs.
- To compute $Q^3(s_i,a_j)$ for any one (s_i,a_j), we'd need to compute $Q^2(s,a)$ for all (s,a) pairs.
- To compute $Q^2(s_i,a_j)$ for any one (s_i,a_j), we'd need to compute $Q^1(s,a)$ 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 $O((mn)^3)$ work—that seems like way too much, especially as the horizon increases! But observe that we really only have mnh values that need to be computed, $Q^h(s,a)$ for all h, s, a. If we start with $h=1$, compute and store those values, then using and reusing the $Q^{h-1}(s,a)$ values to compute the $Q^h(s,a)$ values, we can do all this computation in time $O(mnh)$, which is much better!

2.2 Infinite-horizon solutions

It is actually more typical to work in a regime where the actual finite horizon is not known. This is called the *infinite horizon* version of the problem, when you don't know when the game will be over! However, if we tried to simply take our definition of Q^h above and set $h = \infty$, we would be in trouble, because it could well be that the Q^∞ values for all actions would be infinite, and there would be no way to select one over the other.

There are two standard ways to deal with this problem. One is to take a kind of *average* over all time steps, but this can be a little bit tricky to think about. We'll take a different approach, which is to consider the *discounted* infinite horizon. We select a discount factor $0 < \gamma < 1$. Instead of trying to find a policy that maximizes expected finite-horizon undiscounted value,

$$\mathbb{E}\left[\sum_{t=0}^{h} R_t \mid \pi, s_0\right] \quad ,$$

we will try to find one that maximizes the expected *infinite horizon discounted value*, which is

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid \pi, S_0\right] = \mathbb{E}\left[R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots \mid \pi, s_0\right] \quad .$$

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).

There are two good intuitive motivations for discounting. One is related to economic theory and the present value of money: you'd generally rather have some money today than that same amount of money next week (because you could use it now or invest it). The other is to think of the whole process terminating, with probability $1 - \gamma$ on each step of the interaction. This value is the expected amount of reward the agent would gain under this terminating model.

2.2.1 Evaluating a policy

We will start, again, by evaluating a policy, but now in terms of the expected discounted infinite-horizon value that the agent will get in the MDP if it executes that policy. We define the value of a state s under policy π as

$$V_\pi(s) = \mathbb{E}[R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots \mid \pi, S_0 = s] = \mathbb{E}[R_0 + \gamma(R_1 + \gamma(R_2 + \gamma \ldots))) \mid \pi, S_0 = s] \quad .$$

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

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[R_0 \mid \pi, S_0 = s] + \gamma \mathbb{E}[R_1 + \gamma(R_2 + \gamma \ldots))) \mid \pi, S_0 = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_\pi(s') \end{aligned}$$

You could write down one of these equations for each of the $n = |\mathcal{S}|$ states. There are n unknowns $V_\pi(s)$. These are linear equations, and so it's easy to solve them using Gaussian elimination to find the value of each state under this policy.

This is *so* cool! In a discounted model, if you find that you survived this round and landed in some state s', then you have the same expected future lifetime as you did before. So the value function that is relevant in that state is exactly the same one as in state s.

2.2.2 Finding an optimal policy

The best way of behaving in an infinite-horizon discounted MDP is not time-dependent: at every step, your expected future lifetime, given that you have survived until now, is $1/(1 - \gamma)$.

Study Question: Verify this fact: if, on every day you wake up, there is a probability of $1-\gamma$ that today will be your last day, then your expected lifetime is $1/(1-\gamma)$ days.

An important theorem about MDPs is: there exists a stationary optimal policy π^* (there may be more than one) such that for all $s \in \mathcal{S}$ and all other policies π, we have

$$V_{\pi^*}(s) \geqslant V_\pi(s) \ .$$

Stationary means that it doesn't change over time; the optimal policy in a finite-horizon MDP is *non-stationary.*

There are many methods for finding an optimal policy for an MDP. We will study a very popular and useful method called *value iteration*. It is also important to us, because it is the basis of many *reinforcement-learning* methods.

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 value recursively as

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

This is also a set of equations, one for each (s,a) pair. This time, though, they are not linear, and so they are not easy to solve. But there is a theorem that says they have a unique solution!

If we knew the optimal action-value function, then we could derive an optimal policy π^* as

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

Study Question: The optimal value function is unique, but the optimal policy is not. Think of a situation in which there is more than one optimal policy.

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

VALUE-ITERATION$(\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon)$

```
for s ∈ S, a ∈ A :
    Q_old(s, a) = 0
while True:
    for s ∈ S, a ∈ A :
        Q_new(s, a) = R(s, a) + γ Σ_s' T(s, a, s') max_a' Q_old(s', a')
    if max_{s,a}|Q_old(s, a) − Q_new(s, a)| < ε :
        return Q_new
    Q_old := Q_new
```

2.2.3 Theory

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

- After executing value iteration with parameter ϵ, $\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max} < \epsilon$.
- There is a value of ϵ such that
$$\|Q_{\text{old}} - Q_{\text{new}}\|_{\max} < \epsilon \Longrightarrow \pi_{Q_{\text{new}}} = \pi^*$$
- As the algorithm executes, $\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max}$ 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 optimal value.

This is new notation! Given two functions f and f', we write $\|f - f'\|_{\max}$ to mean $\max_x |f(x) - f'(x)|$. It measures the maximum absolute disagreement between the two functions at any input x.

This is very important for reinforcement learning.

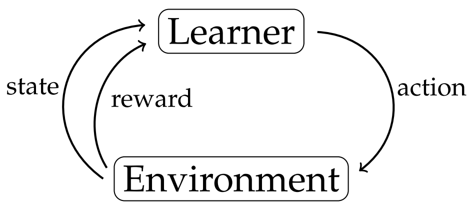
CHAPTER 11

Reinforcement learning

So far, all the learning problems we have looked at have been *supervised*: that is, for each training input $x^{(i)}$, we are told which value $y^{(i)}$ should be the output. A very different problem setting is *reinforcement learning*, in which the learning system is not directly told which outputs go with which inputs. Instead, there is an interaction of the form:

- Learner observes *input* $s^{(i)}$
- Learner generates *output* $a^{(i)}$
- Learner observes *reward* $r^{(i)}$
- Learner observes *input* $s^{(i+1)}$
- Learner generates *output* $a^{(i+1)}$
- Learner observes *reward* $r^{(i+1)}$
- ...

The learner is supposed to find a *policy*, mapping s to a, that maximizes expected reward over time.



This problem setting is equivalent to an *online* supervised learning under the following assumptions:

1. The space of possible outputs is binary (e.g. $\{+1, -1\}$) and the space of possible rewards is binary (e.g. $\{+1, -1\}$);
2. $s^{(i)}$ is independent of all previous $s^{(j)}$ and $a^{(j)}$; and
3. $r^{(i)}$ depends only on $s^{(i)}$ and $a^{(i)}$.

In this case, for any experience tuple $(s^{(i)}, a^{(i)}, r^{(i)})$, we can generate a supervised training example, which is equal to $(s^{(i)}, a^{(i)})$ if $r^{(i)} = +1$ and $(s^{(i)}, -a^{(i)})$ otherwise.

Study Question: What supervised-learning loss function would this objective correspond to?

Reinforcement learning is more interesting when these properties do not hold. When we relax assumption 1 above, we have the class of *bandit problems*, which we will discuss in section 1. If we relax assumption 2, but assume that the environment that the agent is interacting with is an MDP, so that $s^{(i)}$ depends only on $s^{(i-1)}$ and $a^{(i-1)}$ then we are in the classical *reinforcement-learning* setting, which we discuss in section 2. Weakening the assumptions further, for instance, not allowing the learner to observe the current state completely and correctly, makes the problem into a *partially observed MDP* (POMDP), which is substantially more difficult, and beyond the scope of this class.

1 Bandit problems

A basic bandit problem is given by

- A set of actions $\mathcal{A}$;
- A set of reward values $\mathcal{R}$; and
- A probabilistic reward function $R : A \to \text{Dist}(\mathbb{R})$ where $R(a)$ is drawn from a probability distribution over possible reward values in $\mathcal{R}$ conditioned on which action is selected. Each time the agent takes an action, a new value is drawn from this distribution.

The most typical bandit problem has $\mathcal{R} = \{0, 1\}$ and $|\mathcal{A}| = k$. This is called a k*-armed bandit problem*. There is a lot of mathematical literature on optimal strategies for k-armed bandit problems under various assumptions. The important question is usually one of *exploration versus exploitation*. Imagine that you have tried each action 10 times, and now you have an estimate $\hat{p}_j$ for the expected value of $R(a_j)$. Which arm should you pick next? You could

Why? Because in English slang, "one-armed bandit" is a name for a slot machine (an old-style gambling machine where you put a coin into a slot and then pull its arm to see if you get a payoff.) because it has one arm and takes your money! What we have here is a similar sort of machine, but with k arms.

exploit your knowledge, and choose the arm with the highest value of $\hat{p}_j$ on all future trials; or

explore further, by trying some or all actions more times, hoping to get better estimates of the p_j values.

The theory ultimately tells us that, the longer our horizon H (or, similarly, closer to 1 our discount factor), the more time we should spend exploring, so that we don't converge prematurely on a bad choice of action.

Study Question: Why is it that "bad" luck during exploration is more dangerous than "good" luck? Imagine that there is an action that generates reward value 1 with probability 0.9, but the first three times you try it, it generates value 0. How might that cause difficulty? Why is this more dangerous than the situation when an action that generates reward value 1 with probability 0.1 actually generates reward 1 on the first three tries?

Note that what makes this a very different kind of problem from the batch supervised learning setting is that:

- The agent gets to influence what data it gets (selecting a_j gives it another sample from r_j), and
- The agent is penalized for mistakes it makes while it is learning (if it is trying to maximize the expected sum of r_t it gets while behaving).

There is a setting of supervised learning, called *active learning*, where instead of being given a training set, the learner gets to select values of x and the environment gives back a label y; the problem of picking good x values to query is interesting, but the problem of deriving a hypothesis from (x, y) pairs is the same as the supervised problem we have been studying.

In a *contextual* bandit problem, you have multiple possible states, drawn from some set $\mathcal{S}$, and a separate bandit problem associated with each one.

Bandit problems will be an essential sub-component of reinforcement learning.

2 Sequential problems

In the more typical (and difficult!) case, we can think of our learning agent interacting with an MDP, where it knows $\mathcal{S}$ and $\mathcal{A}$, but not $T(s, a, s')$ or $R(s, a)$. The learner can interact with the environment by selecting actions. So, this is somewhat like a contextual bandit problem, but more complicated, because selecting an action influences not only what the immediate reward will be, but also what state the system ends up in at the next time step and, therefore, what additional rewards might be available in the future.

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 to take. There are several different ways to measure the quality of an RL algorithm, including:

- Ignoring the r_t values that it gets *while* learning, but consider how many interactions with the environment are required for it to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that is nearly optimal.
- Maximizing the expected discounted sum of total rewards while it is learning.

Most of the focus is on the first criterion, because the second one is very difficult. The first criterion is reasonable when the learning can take place somewhere safe (imagine a robot learning, inside the robot factory, where it can't hurt itself too badly) or in a simulated environment.

Approaches to reinforcement-learning differ significantly according to what kind of hypothesis or model they learn. In the following sections, we will consider several different approaches.

2.1 Model-based RL

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

Assume that we have had some set of interactions with the environment, which can be characterized as a set of tuples of the form $(s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)})$).

We can estimate $T(s, a, s')$ using a simple counting strategy,

$$\hat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |\mathcal{S}|}.$$

Here, $\#(s, a, s')$ represents the number of times in our data set we have the situation where $s_t = s, a_t = a, s_{t+1} = s'$ and $\#(s, a)$ represents the number of times in our data set we have the situation where $s_t = s, a_t = a$.

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

Adding 1 and $|\mathcal{S}|$ 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.

We also estimate the reward function $R(s, a)$:

$$\hat{R}(s, a) = \frac{\sum r \mid s, a}{\#(s, a)}$$

where

$$\sum r \mid s, a = \sum_{\{t \mid s_t = s, a_t = a\}} r^{(t)} \quad .$$

This is just the average of the observed rewards for each s, a pair.

We can now solve the MDP $(\mathcal{S}, \mathcal{A}, \hat{T}, \hat{R})$ to find an optimal policy using value iteration, or use a finite-depth expecti-max search to find an action to take for a particular state.

This technique is effective for problems with small state and action spaces, where it is not too hard to get enough experience to estimate 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.

2.2 Policy search

A very different strategy is to search directly for a good policy, without first (or ever!) estimating the transition and reward models. The strategy here is to define a functional form $f(s;\theta) = a$ for the policy, where θ represents the parameters we learn from experience. We choose f to be differentiable, and often let $f(s;\theta) = P(a)$, a 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 $\theta \pm \epsilon$, and computing the resulting rewards.
- When θ has higher dimensions (e.g., it is 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 model would be much more complicated to estimate.

2.3 Value function learning

The most popular class of algorithms learns neither explicit transition and reward models nor a direct policy, but instead concentrates on learning a value function. It is a topic of current research to describe exactly under what circumstances value-function-based approaches are best, and there are a growing number of methods that combine value functions, transition and reward models and policies into a complex learning algorithm in an attempt to combine the strengths of each approach.

We will study two variations on value-function learning, both of which estimate the Q function.

2.3.1 Q-learning

This is the most typical way of performing reinforcement learning. Recall the value-iteration update:

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

We will adapt this update to the RL scenario, where we do not know the transition function T or reward function R.

The thing that most students seem to get confused about is when we do value iteration and when we do Q learning. Value iteration assumes you know T and R and just need to *compute* Q. In Q learning, we don't know or even directly estimate T and R: we estimate Q directly from experience!

Q-LEARNING($\mathcal{S}, \mathcal{A}, s_0, \gamma, \alpha$)

```
for s ∈ S, a ∈ A :
    Q[s, a] = 0
s = s_0 // Or draw an s randomly from S
while True:
    a = select_action(s, Q)
    r, s' = execute(a)
    Q[s, a] = (1 − α)Q[s, a] + α(r + γ max_a' Q[s', a'])
    s = s'
```

Here, α represents the "learning rate," which needs to decay for convergence purposes, but in practice is often set to a constant.

Note that the update can be rewritten as

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

which looks something like a gradient update! This is often called *temporal difference* learning method, because we make an update based on the difference between the current estimated value of taking action a in state s, which is $Q[s, a]$, and the "one-step" sampled value of taking a in s, which is $r + \gamma \max_{a'} Q[s', a']$.

It is actually not a gradient update, but later, when we consider function approximation, we will treat it as if it were.

You can see this method as a combination of two different iterative processes that we have already seen: the combination of an old estimate with a new sample using a running average with a learning rate α, and the dynamic-programming update of a Q value from value iteration.

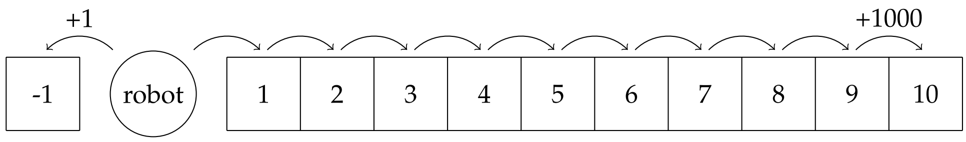
Our algorithm above includes a procedure called *select_action*, which, given the current state s, has to decide which action to take. If the Q value is estimated very accurately and the agent is behaving in the world, then generally we would want to choose the apparently optimal action $\arg\max_{a \in \mathcal{A}} Q(s, a)$. But, during learning, the Q value estimates won't be very good and exploration is important. However, exploring completely at random is also usually not the best strategy while learning, because it is good to focus your attention on the parts of the state space that are likely to be visited when executing a good policy (not a stupid one).

A typical action-selection strategy is the ϵ-greedy strategy:

- with probability $1 - \epsilon$, choose $\arg\max_{a \in \mathcal{A}} Q(s, a)$
- with probability ϵ, choose the action $a \in \mathcal{A}$ uniformly at random

Q-learning has the surprising property that it is *guaranteed* to converge to the actual optimal Q function under fairly weak conditions! Any exploration strategy is okay as long as it tries every action infinitely often on an infinite run (so that it doesn't converge prematurely to a bad action choice).

Q-learning can be very sample-inefficient: imagine a robot that has a choice between moving to the left and getting a reward of 1, then returning to its initial state, or moving to the right and walking down a 10-step hallway in order to get a reward of 1000, then returning to its initial state.



The first time the robot moves to the right and goes down the hallway, it will update the Q value for the last state on the hallway to have a high value, but it won't yet understand that moving to the right was a good choice. The next time it moves down the hallway it updates the value of the state before the last one, and so on. After 10 trips down the hallway, it now can see that it is better to move to the right than to the left.

More concretely, consider the vector of Q values $Q(0 : 10, \text{right})$, representing the Q values for moving right at each of the positions $0, \ldots, 9$. Then, for $\alpha = 1$ and $\gamma = 0.9$,

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

Starting with Q values of 0,

$$Q^{(0)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since the only nonzero reward from moving right is $R(9, \text{ right}) = 1000$, after our robot makes it down the hallway once, our new Q vector is

We are violating our usual notational conventions here, and writing $Q^{(i)}$ to mean the Q value function that results after the robot runs all the way to the end of the hallway, when executing the policy that always moves to the right.

$$Q^{(1)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 \end{bmatrix}$$

After making its way down the hallway again, $Q(8, \text{ right}) = 0 + 0.9 \cdot Q(9, \text{ right}) = 900$ updates:

$$Q^{(2)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 900 & 1000 & 0 \end{bmatrix}$$

Similarly,

$$Q^{(3)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 810 & 900 & 1000 & 0 \end{bmatrix}$$
$$Q^{(4)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 729 & 810 & 900 & 1000 & 0 \end{bmatrix}$$
$$\vdots$$
$$Q^{(10)}(0:10, \text{ right}) = \begin{bmatrix} 387.4 & 420.5 & 478.3 & 531.4 & 590.5 & 656.1 & 729 & 810 & 900 & 1000 & 0 \end{bmatrix},$$

and the robot finally sees the value of moving right from position 0.

We can see how this interacts with the exploration/exploitation dilemma: from the perspective of s_0, it will seem, for a long time, that getting the immediate reward of 1 is a better idea, and it would be easy to converge on that as a strategy without exploring the long hallway sufficiently.

Study Question: Determine the Q value functions that will result from updates due to the robot always executing the "move left" policy.

2.3.2 Function approximation

In our Q-learning algorithm above, we essentially keep track of each Q value in a table, indexed by s and a. What do we do if $\mathcal{S}$ and/or $\mathcal{A}$ are large (or continuous)?

We can use a function approximator like a neural network to store Q values. For example, we could design a neural network that takes in inputs s and a, and outputs $Q(s, a)$. We can treat this as a regression problem, optimizing the squared Bellman error, with loss:

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

where $Q(s, a)$ is now the output of the neural network.

There are actually several different architectural choices for using a neural network to approximate Q values:

- One network for each action a_j, that takes s as input and produces $Q(s, a_j)$ as output;
- One single network that takes s as input and produces a vector $Q(s, \cdot)$, consisting of the Q values for each action; or

- One single network that takes s, a concatenated into a vector (if a is discrete, we would probably use a one-hot encoding, unless it had some useful internal structure) and produces $Q(s, a)$ as output.

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 $\arg\max_{\mathcal{A}} Q(s, a)$.

For continuous action spaces, it is increasingly popular to use a class of methods called *actor-critic* methods, which combine policy and value-function learning. We won't get into them in detail here, though.

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

And, in fact, we routinely shuffle their order in the data file, anyway.

For example, it might spend 12 hours in a dark environment and then 12 in a light one.

One way to handle this is to use *experience replay*, where we save our (s, a, r, s') experiences in a *replay buffer*. Whenever we take a step in the world, we add the (s, a, r, s') to the replay buffer and use it to do a Q-learning update. Then we also randomly select some number of tuples from the replay buffer, and do Q-learning updates based on them, 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 when the optimal policy might visit a large part of the state space, but we like to keep the buffer size small for memory reasons and also so that we don't focus on parts of the state space that are irrelevant for the optimal policy.) The idea is that it will help you propagate reward values through your state space more efficiently if you do these updates. You can see it as doing something like value iteration, but using samples of experience rather than a known model.

2.3.3 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*.

FITTED-Q-LEARNING($\mathcal{A}, s_0, \gamma, \alpha, \epsilon, m$)

1 $s = s_0$ // Or draw an s randomly from $\mathcal{S}$
2 $\mathcal{D} = \{\}$
3 initialize neural-network representation of Q
4 **while** True:
5     $\mathcal{D}_{new}$ = experience from executing ϵ-greedy policy based on Q for m steps
6     $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_{new}$ represented as (s, a, r, s') tuples
7     $D_{sup} = \{(x^{(i)}, y^{(i)})\}$ where $x^{(i)} = (s, a)$ and $y^{(i)} = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$
8   for each tuple $(s, a, r, s')^{(i)} \in \mathcal{D}$
9   re-initialize neural-network representation of Q
10     $Q = \text{supervised\_NN\_regression}(D_{sup})$

Here, we alternate between using the policy induced by the current Q function to gather a batch of data $\mathcal{D}_{new}$, adding it to our overall data set $\mathcal{D}$, and then using supervised neural-network training 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 (training 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 the fit is good (possibly measured on held-out data).

CHAPTER 12

Recurrent Neural Networks

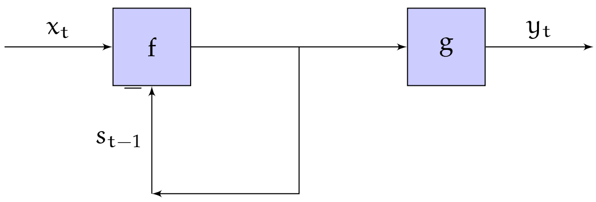
In chapter 8 we studied neural networks and how we can train the weights of a network, based on data, so that it will adapt into a function that approximates the relationship between the (x, y) pairs in a supervised-learning training set. In section 1 of chapter 10, we studied state-machine models and defined *recurrent neural networks* (RNNs) as a particular type of state machine, with a multidimensional vector of real values as the state. In this chapter, we'll see how to use gradient-descent methods to train the weights of an RNN so that it performs a *transduction* that matches as closely as possible a training set of input-output *sequences*.

1 RNN model

Recall that the basic operation of the state machine is to start with some state s_0, then iteratively compute for $t \geqslant 1$::

$$\begin{aligned} s_t &= f(s_{t-1}, x_t) \\ y_t &= g(s_t) \end{aligned}$$

as illustrated in the diagram below (remembering that there needs to be a delay on the feedback loop):



So, given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs

$$\underbrace{g(f(s_0, x_1))}_{y_1}, \underbrace{g(f(f(s_0, x_1), x_2,))}_{y_2}, \ldots \quad .$$

A *recurrent neural network* is a state machine with neural networks constituting functions f and g:

$$\begin{aligned} f(s, x) &= f_1(W^{sx}x + W^{ss}s + W_0^{ss}) \\ g(s) &= f_2(W^O s + W_0^O) \ . \end{aligned}$$

The inputs, outputs, and states are all vector-valued:

We are very sorry! This course material has evolved from different sources, which used $W^T x$ in the forward pass for regular feed-forward NNs and Wx for the forward pass in RNNs. This inconsistency doesn't make any technical difference, but is a potential source of confusion.

$$\begin{aligned} x_t &: \ell \times 1 \\ s_t &: m \times 1 \\ y_t &: v \times 1 \ . \end{aligned}$$

The weights in the network, then, are

$$\begin{aligned} W^{sx} &: m \times \ell \\ W^{ss} &: m \times m \\ W_0^{ss} &: m \times 1 \\ W^O &: v \times m \\ W_0^O &: v \times 1 \end{aligned}$$

with activation functions f_1 and f_2. Finally, the operation of the RNN is described by

$$\begin{aligned} s_t &= f_1\left(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}\right) \\ y_t &= f_2\left(W^O s_t + W_0^O\right) \ . \end{aligned}$$

Study Question: Check dimensions here to be sure it all works out. Remember that we apply f_1 and f_2 elementwise.

2 Sequence-to-sequence RNN

Now, how can we train an RNN to model a transduction on sequences? This problem is sometimes called *sequence-to-sequence* mapping. You can think of it as a kind of regression problem: given an input sequence, learn to generate the corresponding output sequence.

One way to think of training a sequence **classifier** is to reduce it to a transduction problem, where $y_t = 1$ if the sequence $x_1, \ldots, x_t$ is a *positive* example of the class of sequences and -1 otherwise.

A training set has the form $\left[\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(q)}, y^{(q)}\right)\right]$, where

- $x^{(i)}$ and $y^{(i)}$ are length $n^{(i)}$ sequences;
- sequences in the *same pair* are the same length; and sequences in different pairs may have different lengths.

Next, we need a loss function. We start by defining a loss function on sequences. There are many possible choices, but usually it makes sense just to sum up a per-element loss function on each of the output values, where p is the predicted sequence and y is the actual one:

$$\text{Loss}_{\text{seq}}\left(p^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \text{Loss}_{\text{elt}}\left(p_t^{(i)}, y_t^{(i)}\right) \ .$$

The per-element loss function Loss_{elt} will depend on the type of y_t and what information it is encoding, in the same way as for a supervised network.. Then, letting $\theta =$

So it could be NLL, squared loss, etc.

$(W^{sx}, W^{ss}, W^O, W_0^{ss}, W_0^O)$, our overall objective is to minimize

$$J(\theta) = \sum_{i=1}^{q} \text{Loss}_{\text{seq}}\left(\text{RNN}(x^{(i)}; \theta), y^{(i)}\right) \quad ,$$

where $\text{RNN}(x; \theta)$ is the output sequence generated, given input sequence x.

It is typical to choose f_1 to be *tanh* but any non-linear activation function is usable. We choose f_2 to align with the types of our outputs and the loss function, just as we would do in regular supervised learning.

Remember that it looks like a sigmoid but ranges from -1 to +1.

3 Back-propagation through time

Now the fun begins! We can find θ to minimize J using gradient descent. We will work through the simplest method, *back-propagation through time* (BPTT), in detail. This is generally not the best method to use, but it's relatively easy to understand. In section 5 we will sketch alternative methods that are in much more common use.

Calculus reminder: total derivative Most of us are not very careful about the difference between the *partial derivative* and the *total derivative*. We are going to use a nice example from the Wikipedia article on partial derivatives to illustrate the difference. The volume of a circular cone depends on its height and radius:

$$V(r, h) = \frac{\pi r^2 h}{3} \quad .$$

The partial derivatives of volume with respect to height and radius are

$$\frac{\partial V}{\partial r} = \frac{2\pi r h}{3} \quad \text{and} \quad \frac{\partial V}{\partial h} = \frac{\pi r^2}{3} \quad .$$

They measure the change in V assuming everything is held constant except the single variable we are changing. Now assume that we want to preserve the cone's proportions in the sense that the ratio of radius to height stay constant, then we can't really change one without changing the other. In this case, we really have to think about the *total derivative*, which sums the "paths" along which r might influence V:

$$\begin{aligned}
\frac{dV}{dr} &= \frac{\partial V}{\partial r} + \frac{\partial V}{\partial h}\frac{dh}{dr} \\
&= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{dh}{dr} \\
\frac{dV}{dh} &= \frac{\partial V}{\partial h} + \frac{\partial V}{\partial r}\frac{dr}{dh} \\
&= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}\frac{dr}{dh}
\end{aligned}$$

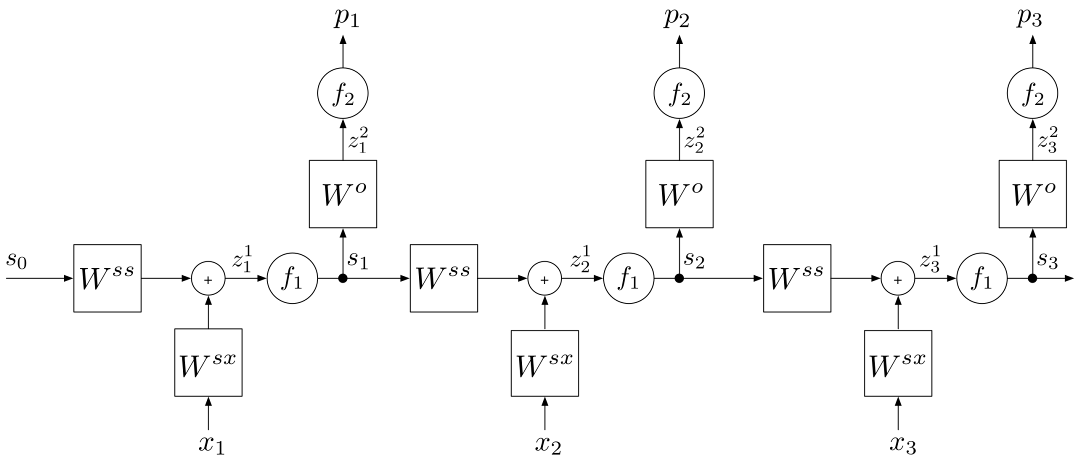
Just to be completely concrete, let's think of a right circular cone with a fixed angle $\alpha = \tan r/h$, so that if we change r or h then α remains constant. So we have $r = h \tan{}^{-}1\alpha$; let constant $c = \tan^{-1}\alpha$, so now $r = ch$. Now, we know that

$$\begin{aligned}
\frac{dV}{dr} &= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{1}{c} \\
\frac{dV}{dh} &= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}c
\end{aligned}$$

The BPTT process goes like this:

(1) Sample a training pair of sequences (x, y); let their length be n.

(2) "Unroll" the RNN to be length n (picture for $n = 3$ below), and initialize s_0:



Now, we can see our problem as one of performing what is almost an ordinary back-propagation training procedure in a feed-forward neural network, but with the difference that the weight matrices are shared among the layers. In many ways, this is similar to what ends up happening in a convolutional network, except in the conv-net, the weights are re-used spatially, and here, they are re-used temporally.

(3) Do the *forward pass*, to compute the predicted output sequence p:

$$
\begin{aligned}
z_t^1 &= W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss} \\
s_t &= f_1(z_t^1) \\
z_t^2 &= W^O s_t + W_0^O \\
p_t &= f_2(z_t^2)
\end{aligned}
$$

(4) Do *backward pass* to compute the gradients. For both W^{ss} and W^{sx} we need to find

$$\frac{dL_{\text{seq}}}{dW} = \sum_{u=1}^{n} \frac{dL_u}{dW} \tag{12.1}$$

Letting $L_u = L_{\text{elt}}(p_u, y_u)$ and using the *total derivative*, which is a sum over all the ways in which W affects L_u, we have

$$= \sum_{u=1}^{n}\sum_{t=1}^{n} \frac{\partial L_u}{\partial s_t} \cdot \frac{\partial s_t}{\partial W} \tag{12.2}$$

Re-organizing, we have

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \sum_{u=1}^{n} \frac{\partial L_u}{\partial s_t} \tag{12.3}$$

Because s_t only affects $L_t, L_{t+1}, \ldots, L_n$,

$$
\begin{aligned}
&= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \sum_{u=t}^{n} \frac{\partial L_u}{\partial s_t} \\
&= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \left(\frac{\partial L_t}{\partial s_t} + \underbrace{\sum_{u=t+1}^{n} \frac{\partial L_u}{\partial s_t}}_{\delta^{s_t}} \right) \qquad (12.4)
\end{aligned}
$$

δ^{s_t} is the dependence of the loss on steps after t on the state at time t.

That is, δ^{s_t} is how much we can blame state s_t for all the future element losses.

We can compute this backwards, with t going from n down to 1. The trickiest part is figuring out how early states contribute to later losses. We define *future loss*

$$F_t = \sum_{u=t+1}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \ ,$$

so

$$\delta^{s_t} = \frac{\partial F_t}{\partial s_t} \ .$$

At the last stage, $F_n = 0$ so $\delta^{s_n} = 0$.

Now, working backwards,

$$
\begin{aligned}
\delta^{s_{t-1}} &= \frac{\partial}{\partial s_{t-1}} \sum_{u=t}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \frac{\partial}{\partial s_t} \sum_{u=t}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \frac{\partial}{\partial s_t} \left[\text{Loss}_{\text{elt}}(p_t, y_t) + \sum_{u=t+1}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \right] \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \left[\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t} + \delta^{s_t} \right]
\end{aligned}
$$

Now, we can use the chain rule again to find the dependence of the element loss at time t on the state at that same time,

$$\underbrace{\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t}}_{(m \times 1)} = \underbrace{\frac{\partial z_t^2}{\partial s_t}}_{(m \times v)} \cdot \underbrace{\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial z_t^2}}_{(v \times 1)} \ ,$$

and the dependence of the state at time t on the state at the previous time, noting that we are performing an *elementwise* multiplication between W_{ss}^T and the vector of $f^{1\prime}$ values, $\partial s_t / \partial z_t^1$:

There are two ways to think about $\partial s_t / \partial z_t$: here, we take the view that it is an $m \times 1$ vector and we multiply each column of W^T by it. Another, equally good, view, is that it is an $m \times m$ diagonal matrix, with the values along the diagonal, and then this operation is a matrix multiply. Our software implementation will take the first view.

$$\underbrace{\frac{\partial s_t}{\partial s_{t-1}}}_{(m \times m)} = \underbrace{\frac{\partial z_t^1}{\partial s_{t-1}}}_{(m \times m)} \cdot \underbrace{\frac{\partial s_t}{\partial z_t^1}}_{(m \times 1)} = \underbrace{W^{ss\,T} * f^{1\prime}(z_t^1)}_{\text{not dot!}} \ .$$

Putting this all together, we end up with

$$\delta^{s_{t-1}} = \underbrace{W^{ss\,T} * f^{1\prime}(z_t^1)}_{\frac{\partial s_t}{\partial s_{t-1}}} \cdot \underbrace{\left(W^{O\,T} \frac{\partial L_t}{\partial z_t^2} + \delta^{s_t} \right)}_{\frac{\partial F_{t-1}}{\partial s_t}}$$

We're almost there! Now, we can describe the actual weight updates. Using equation 12.4 and recalling the definition of $\delta^{s_t} = \partial F_t / \partial s_t$, as we iterate backwards, we can accumulate the terms in equation 12.4 to get the gradient for the whole loss:

$$\frac{dL_{seq}}{dW^{ss}} += \frac{\partial F_{t-1}}{\partial W^{ss}} = \frac{\partial z_t^1}{\partial W^{ss}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t}$$
$$\frac{dL_{seq}}{dW^{sx}} += \frac{\partial F_{t-1}}{\partial W^{sx}} = \frac{\partial z_t^1}{\partial W^{sx}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t}$$

We can handle W^O separately; it's easier because it does not effect future losses in the way that the other weight matrices do:

$$\frac{\partial L_{seq}}{\partial W^O} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial W^O} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial z_t^2} \cdot \frac{\partial z_t^2}{\partial W^O}$$

Assuming we have $\frac{\partial L_t}{\partial z_t^2} = (p_t - y_t)$, (which ends up being true for squared loss, softmax-NLL, etc.), then on each iteration

$$\underbrace{\frac{\partial L_{seq}}{\partial W^O}}_{v \times m} += \underbrace{(p_t - y_t)}_{v \times 1} \cdot \underbrace{s_t^T}_{1 \times m}$$

Whew!

Study Question: Derive the updates for the offsets W_0^{ss} and W_0^O.

4 Training a language model

A *language model* is just trained on a set of input sequences, $(c_1^{(i)}, c_2^{(i)}, \ldots, c_{n^i}^{(i)})$, and is used to predict the next character, given a sequence of previous tokens:

A "token" is generally a character or a word.

$$c_t = \text{RNN}(c_1, c_2, \ldots, c_{t-1})$$

We can convert this to a sequence-to-sequence training problem by constructing a data set of (x, y) sequence pairs, where we make up new special tokens, start and end, to signal the beginning and end of the sequence:

$$x = (\langle \text{start} \rangle, c_1, c_2, ;c_n)$$
$$y = (c_1, c_2, \ldots, \langle \text{end} \rangle)$$

5 Vanishing gradients and gating mechanisms

Let's take a careful look at the backward propagation of the gradient along the sequence:

$$\delta^{s_{t-1}} = \frac{\partial s_t}{\partial s_{t-1}} \cdot \left[\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t} + \delta^{s_t} \right] .$$

Consider a case where only the output at the end of the sequence is incorrect, but it depends critically, via the weights, on the input at time 1. In this case, we will multiply the loss at step n by

$$\frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_3}{\partial s_2} \cdots \frac{\partial s_n}{\partial s_{n-1}} .$$

In general, this quantity will either grow or shrink exponentially with the length of the sequence, and make it very difficult to train.

Study Question: The last time we talked about exploding and vanishing gradients, it was to justify per-weight adaptive step sizes. Why is that not a solution to the problem this time?

An important insight that really made recurrent networks work well on long sequences is the idea of *gating*.

5.1 Simple gated recurrent networks

A computer only ever updates some parts of its memory on each computation cycle. We can take this idea and use it to make our networks more able to retain state values over time and to make the gradients better-behaved. We will add a new component to our network, called a *gating network*. Let g_t be a $m \times 1$ vector of values and let W^{gx} and W^{gs} be $m \times l$ and $m \times m$ weight matrices, respectively. We will compute g_t as

It can have an offset, too, but we are omitting it for simplicity.

$$g_t = \text{sigmoid}(W^{gx}x_t + W^{gs}s_{t-1})$$

and then change the computation of s_t to be

$$s_t = (1 - g_t) * s_{t-1} + g_t * f_1(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}) \enspace ,$$

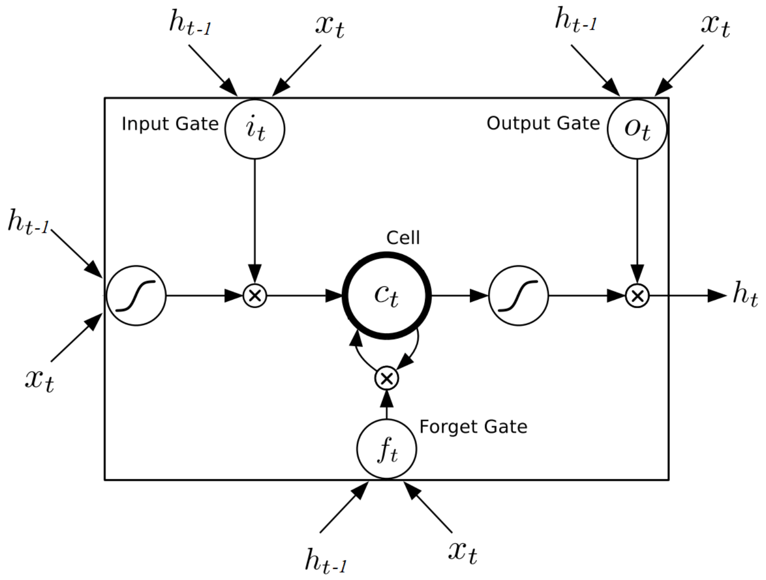
where $*$ is component-wise multiplication. We can see, here, that the output of the gating network is deciding, for each dimension of the state, how much it should be updated now. This mechanism makes it much easier for the network to learn to, for example, "store" some information in some dimension of the state, and then not change it during future state updates, or change it only under certain conditions on the input or other aspects of the state.

Study Question: Why is it important that the activation function for g be a sigmoid?

5.2 Long short-term memory

The idea of gating networks can be applied to make a state-machine that is even more like a computer memory, resulting in a type of network called an LSTM for "long short-term memory." We won't go into the details here, but the basic idea is that there is a memory cell (really, our state vector) and three (!) gating networks. The *input* gate selects (using a "soft" selection as in the gated network above) which dimensions of the state will be updated with new values; the *forget* gate decides which dimensions of the state will have its old values moved toward 0, and the *output* gate decides which dimensions of the state will be used to compute the output value. These networks have been used in applications like language translation with really amazing results. A diagram of the architecture is shown below:

Yet another awesome name for a neural network!


h_{t-1}
x_t
h_{t-1}
x_t
Input Gate
i_t
Output Gate
o_t
h_{t-1}
Cell
c_t
h_t
x_t
Forget Gate
f_t
h_{t-1}
x_t

CHAPTER 13

Recommender systems

The problem of choosing items from a large set to recommend to a user comes up in many contexts, including music services, shopping, and online advertisements. As well as being an important application, it is interesting because it has several formulations, some of which take advantage of a particular interesting structure in the problem.

Concretely, we can think about a company like Netflix, which recommends movies to its users. Netflix knows the ratings given by many different people to many different movies, and knows your ratings on a small subset of all possible movies. How should it use this data to recommend a movie for you to watch tonight?

There are two prevailing approaches to this problem. The first, *content-based recommendation*, is formulated as a supervised learning problem. The second, *collaborative filtering*, introduces a new learning problem formulation.

1 Content-based recommendations

In content-based recommendation, we try to learn a predictor, f, that uses the movies that you have rated so far as training data, find a hypothesis that maps a movie into a prediction of what rating you would give it, and then return some movies with high predicted ratings.

The first step is designing representations for the input and output.

It's actually pretty difficult to design a good feature representation for movies. Reasonable approaches might construct features based on the movie's genre, length, main actors, director, location, or even ratings given by some standard critics or aggregation sources. This design process would yield

$$\phi : \text{movie} \rightarrow \text{vector} \quad .$$

Movie ratings are generally given in terms of some number of stars, so the output domain might be $\{1, 2, 3, 4, 5\}$. It's not appropriate for one-hot encoding on the output, and pretending that these are real values is also not entirely sensible. Nevertheless, we will treat the output as if it's in $\mathbb{R}$.

Thermometer coding might be reasonable, but it's hard to say without trying it. Some more advanced techniques try to predict rankings (would I prefer movie A over movie B) rather than raw ratings.

Study Question: What is the disadvantage of using one-hot? What is the disadvantage of using $\mathbb{R}$?

Now that we have an encoding, we can make a training set based on *your* previous ratings of movies. Here, $x^{(i)}$ represents the ith movie, $\phi(x^{(i)})$ gives our feature representation of the ith movie, and $y^{(i)} = \text{rating}(x^{(i)})$ is your rating for the ith movie. If you rated j movies so far, our resulting training set looks like

$$D_a = \left\{ \left(\phi(x^{(1)}), \text{rating}(x^{(1)})\right), \left(\phi(x^{(2)}), \text{rating}(x^{(2)})\right), \dots, \left(\phi(x^{(j)}), \text{rating}(x^{(j)})\right) \right\}$$

The next step is to pick a loss function. This is closely related to the choice of output encoding. Since we decided to treat the output as a real, we can formulate the problem as a regression from $\phi \to \mathbb{R}$, with $\text{Loss}(p, y) = \frac{1}{2}(y - p)^2$ We will generally need to regularize because we typically have a very small amount of data (unless you really watch a lot of movies!).

Finally, we need to pick a hypothesis space. The simplest thing would be to make it linear, but you could definitely use something fancier, like a neural network.

If we put all this together, with a linear hypothesis space, we end up with the objective

$$J(\theta) = \frac{1}{2} \sum_{i \in D_a} (y^{(i)} - \theta^T \phi(x^{(i)}) - \theta_0)^2 + \frac{\lambda}{2} \|\theta\|^2 \quad .$$

This is our old friend, ridge regression, and can be solved analytically or with gradient descent.

2 Collaborative filtering

There are two difficulties with content-based recommendation systems:

- It's hard to design a good feature set to represent movies.
- They only use your previous movie ratings, but don't have a way to use the vast majority of their data, which is ratings from other people.

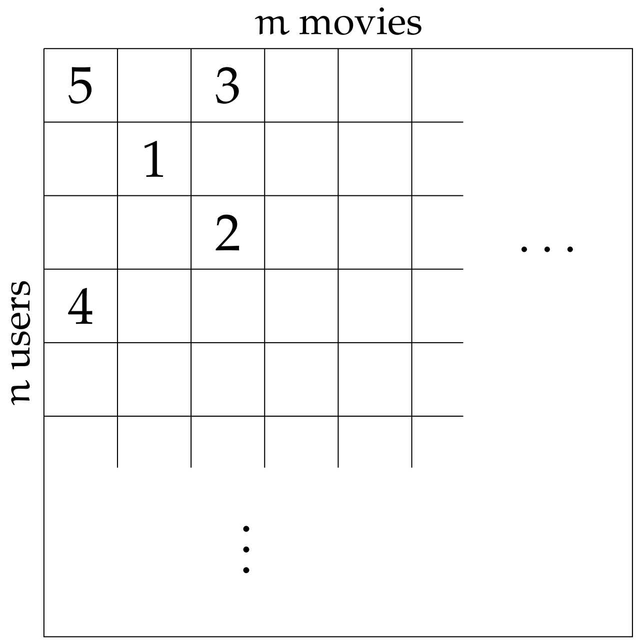
In collaborative filtering, we'll try to use *all* the ratings that other people have made of movies to help make better predictions for you.

Intuitively, we can see this process as finding the kinds of people who like the kinds of movies I like, and then predicting that I will like other movies that they like.

In fact, there's a third strategy that is really directly based on this idea, in which we concretely try to find other users who are our "nearest neighbors" in movie preferences, and then predict movies they like. The approach we discuss here has similar motivations but is more robust.

Formally, we will start by constructing a *data matrix* Y, where Y_{ai} represents the score given by user a to movie i. So, if we have n users and m movies, Y has shape $n \times m$.

We will in fact not *actually* represent the whole data matrix explicitly—it would be too big. But it's useful to think about.



Y is very sparse (most entries are empty). So, we will think of our training data-set as a set of tuples $\{(a, i, r)\}$, where a is the index assigned to a particular user, i is the index assigned to a particular movie, and r is user a's rating of movie i. We will use $D = \{(a, i) : Y_{ai} \text{ is non-empty}\}$ as the set of indices for which we have a rating.

In the Netflix challenge data set, there are 400,000 users and 17,000 movies. Only 1% of the data matrix is filled.

We are going to try to find a way to use D to predict values for missing entries. Let X be our predicted matrix of ratings. Now, we need to find a loss function that relates X and Y, so that we can try to optimize it to find a good predictive model.

Idea #1 Following along with our previous approaches to designing loss functions, we might want to say that our predictions X_{ai} should agree with our data Y_{ai}, and then add some regularization, yielding loss function

$$\text{Loss}(X, Y) = \frac{1}{2} \sum_{(a,i) \in D} (X_{ai} - Y_{ai})^2 + \sum_{\text{all } (a,i)} X_{ai}^2 \quad .$$

This is a **bad** *idea!* It will set $X_{ai} = 0$ for all $(a, i) \notin D$.

Study Question: Convince yourself of that!

We need to find a different kind of regularization that will force some generalization to unseen entries.

Linear algebra idea: The *rank* of a matrix is the maximum number of linearly independent rows in the matrix (which is equal to the maximum number of linearly independent columns in the matrix).

If an $n \times m$ matrix X is rank 1, then there exist U and V of shapes $n \times 1$ and $m \times 1$, respectively, such that

$$X = UV^T \quad .$$

If X is rank k, then there exist U and V of shape $n \times k$ and $m \times k$, respectively, such that

$$X = UV^T \quad .$$

Idea #2 Find the rank 1 matrix X that fits the entries in Y as well as possible. This is a much lower-dimensional representation (it has $m + n$ parameters rather than $m \cdot n$ parameters) and the same parameter is shared among many predictions, so it seems like it might have better generalization properties than our previous idea.

So, we would need to find vectors U and V such that

$$UV^T = \begin{bmatrix} U^{(1)} \\ \vdots \\ U^{(n)} \end{bmatrix} \begin{bmatrix} V^{(1)} & \cdots & V^{(m)} \end{bmatrix} = \begin{bmatrix} U^{(1)}V^{(1)} & \cdots & U^{(1)}V^{(m)} \\ \vdots & \ddots & \vdots \\ U^{(n)}V^{(1)} & \cdots & U^{(n)}V^{(m)} \end{bmatrix} = X \ .$$

And, since we're using squared loss, our objective function would be

$$J(U, V) = \frac{1}{2} \sum_{(a,i) \in D} (U^{(a)}V^{(i)} - Y_{ai})^2 \ .$$

Now, how can we find the optimal values of U and V? We could take inspiration from our work on linear regression and see what the gradients of J are with respect to the parameters in U and V. For example,

$$\frac{\partial J}{\partial U^{(a)}} = \sum_{\{i \mid (a,i) \in D\}} (U^{(a)}V^{(i)} - Y_{ai})V^{(i)} \ .$$

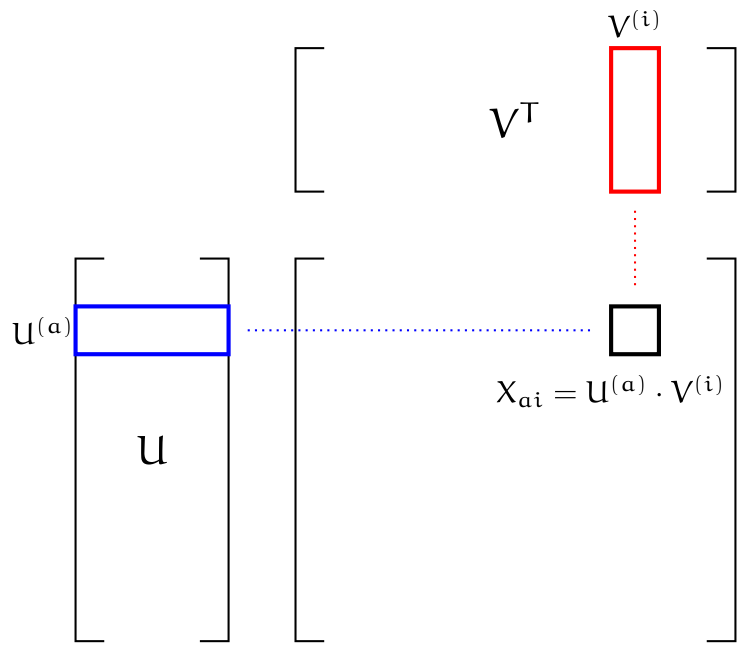
We could get an equation like this for each parameter $U^{(a)}$ or $V^{(i)}$. We don't know how to get an immediate analytic solution to this set of equations because the parameters U and V are multiplied by one another in the predictions, so the model does not have a linear dependence on the parameters. We could approach this problem using gradient descent, though, and we'll do that with a related model in the next section.

But, before we talk about optimization, let's think about the expressiveness of this model. It has one parameter per user (the elements of U) and one parameter per movie (the elements of V), and the predicted rating is the product of these two. It can really represent only each user's general enthusiasm and each movie's general popularity, and predict the user's rating of the movie to be the product of these values.

Study Question: What if we had two users, 1 and 2, and two movies, A and B. Can you find U, V that represents the data set $(1, A, 1), (1, B, 5), (2, A, 5), (2, B, 1)$ well?

Idea #3 If using a rank 1 decomposition of the matrix is not expressive enough, maybe we can try a *rank* k decomposition! In this case, we would try to find an $n \times k$ matrix U and an $m \times k$ matrix V that minimize

$$J(U, V) = \frac{1}{2} \sum_{(a,i) \in D} (U^{(a)} \cdot V^{(i)} - Y_{ai})^2 \ .$$



Here, the length k vector $U^{(a)}$ is the a^{th} row of U, and represents the k "features" of person a. Likewise, the length k vector $V^{(i)}$ is the i^{th} row of V, and represents the k "features" of movie i. Performing the matrix multiplication $X = UV^T$, we see what the prediction for person a and movie i is $X_{ai} = U^{(a)} \cdot V^{(i)}$.

The total number of parameters that we have is $nk + mk$. But, it is a redundant representation. We have 1 extra scaling parameter when $k = 1$, and k^2 extra parameters in general. So, we really effectively have $nk + mk - k^2$ "degrees of freedom."

Study Question: Imagine $k = 3$. If we were to take the matrix U and multiply the first column by 2, the second column by 3 and the third column by 4, to make a new matrix U', what would we have to do to V to get a V' so that $U'V'^T = UV^T$? How does this question relate to the comments above about redundancy?

It is still useful to add offsets to our predictions, so we will include an $n \times 1$ vector b_U and an $m \times 1$ vector b_V of offset parameters, and perform regularization on the parameters in U and V. So our final objective becomes

$$J(U, V) = \frac{1}{2} \sum_{(a,i) \in D} (U^{(a)} \cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai})^2 + \frac{\lambda}{2} \sum_{a=1}^{n} \left\| U^{(a)} \right\|^2 + \frac{\lambda}{2} \sum_{i=1}^{m} \left\| V^{(i)} \right\|^2 \ .$$

Study Question: What would be an informal interpretation of $b_U^{(a)}$? Of $b_V^{(i)}$?

2.1 Optimization

Now that we have an objective, it's time to optimize! There are two reasonable approaches to finding U, V, b_U, and b_V that optimize this objective: alternating least squares (ALS), which builds on our analytical solution approach for linear regression, and stochastic gradient descent (SGD), which we have used in the context of neural networks and other models.

2.1.1 Alternating least squares

One interesting thing to notice is that, if we were to fix U and b_U, then finding the minimizing V and b_V is a linear regression problem that we already know how to solve. The same is true if we were to fix V and b_V, and seek U and b_U. So, we will consider an algorithm that takes alternating steps of this form: we fix U, b_U, initially randomly, find the best V, b_V; then fix those and find the best U, b_U, etc.

This is a kind of optimization sometimes called "coordinate descent," because we only improve the model in one (or, in this case, a set of) coordinates of the parameter space at a time. Generally, coordinate descent has similar kinds of convergence properties as gradient descent, and it cannot guarantee that we find a global optimum. It is an appealing choice in this problem because we know how to directly move to the optimal values of one set of coordinates given that the other is fixed.

More concretely, we:

1. Initialize V and b_V at random

2. For each a in $1, 2, \ldots, n$:

   - Construct a linear regression problem to find $U^{(a)}$ to minimize

   $$\frac{1}{2} \sum_{\{i|(a,i)\in D\}} \left(U^{(a)} \cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai}\right)^2 + \frac{\lambda}{2} \left\|U^{(a)}\right\|^2 \quad .$$

   - Recall minimizing the least squares objective (we are ignoring the offset and regularizer in the following so you can see the basic idea):

   $$(W\theta - T)^T (W\theta - T) \quad .$$

   In this scenario,

     – $\theta = U^{(a)}$ is the $k \times 1$ parameter vector that we are trying to find,
     – T is a $m_a \times 1$ vector of target values (for the m_a movies a has rated), and
     – W is the $m_a \times k$ matrix whose rows are the $V^{(i)}$ where a has rated movie i.

   The solution to the least squares problem using ridge regression is our new $U^{(a)}$ and $b_U^{(a)}$.

3. For each i in $1, 2, \ldots, m$

   - Construct a linear regression problem to find $V^{(i)}$ and $b_V^{(i)}$ to minimize

   $$\frac{1}{2} \sum_{\{i|(a,i)\in D\}} \left(U^{(a)} \cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai}\right)^2 + \frac{\lambda}{2} \left\|V^{(i)}\right\|^2$$

   - Now, $\theta = V^{(i)}$ is a $k \times 1$ parameter vector, T is a $n_i \times 1$ target vector (for the n_i users that have rated movie i), and W is the $n_i \times k$ matrix whose rows are the $U^{(a)}$ where i has been rated by user a.

   Again, we solve using ridge regression for a new value of $V^{(i)}$ and $b_V^{(i)}$.

4. Alternate between steps 2 and 3, optimizing U and V, and stop after a fixed number of iterations or when the difference between successive parameter estimates is small.

2.1.2 Stochastic gradient descent

Finally, we can approach this problem using stochastic gradient descent. It's easier to think about if we reorganize the objective function to be

$$J(U,V) = \frac{1}{2}\sum_{(a,i)\in D}\left(\left(U^{(a)}\cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai}\right)^2 + \lambda_U^{(a)}\left\|U^{(a)}\right\|^2 + \lambda_V^{(i)}\left\|V^{(i)}\right\|^2\right)$$

where

$$\lambda_U^{(a)} = \frac{\lambda}{\text{\# times } (a,\_) \in D} = \frac{\lambda}{\sum_{\{i|(a,i)\in D\}} 1}$$
$$\lambda_V^{(i)} = \frac{\lambda}{\text{\# times } (\_,i) \in D} = \frac{\lambda}{\sum_{\{a|(a,i)\in D\}} 1}$$

Then,

$$\frac{\partial J(U,V)}{\partial U^{(a)}} = \sum_{\{i|(a,i)\in D\}}\left[\left(U^{(a)}\cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai}\right)V^{(i)} + \lambda_U^{(a)}U^{(a)}\right]$$
$$\frac{\partial J(U,V)}{\partial b_U^{(a)}} = \sum_{\{i|(a,i)\in D\}}\left(U^{(a)}\cdot V^{(i)} + b_U^{(a)} + b_V^{(i)} - Y_{ai}\right)$$

We can similarly obtain gradients with respect to $V^{(i)}$ and $b_V^{(i)}$.

Then, to do gradient descent, we draw an example (a, i, Y_{ai}) from D at random, and do gradient updates on $U^{(a)}$, $b_U^{(a)}$, $V^{(i)}$, and $b_V^{(i)}$.

Study Question: Why don't we update the other parameters, such as $U^{(a')}$ for some other user a' or $V^{(i')}$ for some other movie i'?

CHAPTER 14

Non-parametric methods

1 Intro

We will continue to broaden the class of models that we can fit to our data. Neural networks have adaptable complexity, in the sense that we can try different structural models and use cross validation to find one that works well on our data.

We now turn to models that automatically adapt their complexity to the training data. The name *non-parametric methods* is misleading: it is really a class of methods that does not have a fixed parameterization in advance. Some non-parametric models, such as decision trees, which we might call *semi-parametric methods*, can be seen as dynamically constructing something that ends up looking like a more traditional parametric model, but where the actual training data affects exactly what the form of the model will be. Other non-parametric methods, such as nearest-neighbor, rely directly on the data to make predictions and do not compute a model that summarizes the data.

The semi-parametric methods tend to have the form of a composition of simple models. We'll look at:

- *Tree models*: partition the input space and use different simple predictions on different regions of the space; this increases the hypothesis space.
- *Additive models*: train several different classifiers on the whole space and average the answers; this decreases the estimation error.

Boosting is a way to construct an additive model that decreases both estimation and structural error, but we won't address it in this class.

Why are we studying these methods, in the heyday of neural networks?

- They are fast to implement and have few or no hyper-parameters to tune.
- They often work as well or better than more complicated methods.
- Both can be easier to explain to a human user: decision-trees are fairly directly human-interpretable and nearest neighbor methods can justify their decision to some extent by showing a few training examples that the prediction was based on.

2 Trees

The idea here is that we would like to find a partition of the input space and then fit very simple models to predict the output in each piece. The partition is described using a (typically binary) "decision tree," which recursively splits the space.

These methods differ by:

- The class of possible ways to split the space at each node; these are generally linear splits, either aligned with the axes of the space, or sometimes more general classifiers.
- The class of predictors within the partitions; these are often simply constants, but may be more general classification or regression models.
- The way in which we control the complexity of the hypothesis: it would be within the capacity of these methods to have a separate partition for each individual training example.
- The algorithm for making the partitions and fitting the models.

The primary advantage of tree models is that they are easily interpretable by humans. This is important in application domains, such as medicine, where there are human experts who often ultimately make critical decisions and who need to feel confident in their understanding of recommendations made by an algorithm.

These methods are most appropriate for domains where the input space is not very high-dimensional and where the individual input features have some substantially useful information individually or in small groups. They would not be good for image input, but might be good in cases with, for example, a set of meaningful measurements of the condition of a patient in the hospital.

We'll concentrate on the CART/ID3 family of algorithms, which were invented independently in the statistics and the artificial intelligence communities. They work by greedily constructing a partition, where the splits are *axis aligned* and by fitting a *constant* model in the leaves. The interesting questions are how to select the splits and how to control complexity. The regression and classification versions are very similar.

2.1 Regression

The predictor is made up of

- a partition function, π, mapping elements of the input space into exactly one of M regions, $R_1, \dots, R_M$, and
- a collection of M output values, O_m, one for each region.

If we already knew a division of the space into regions, we would set O_m, the constant output for region R_m, to be the average of the training output values in that region; that is:

$$O_m = \text{average}_{\{i \mid x^{(i)} \in R_m\}} y^{(i)} \quad .$$

Define the error in a region as

$$E_m = \sum_{\{i \mid x^{(i)} \in R_m\}} (y^{(i)} - O_m)^2 \quad .$$

Ideally, we would select the partition to minimize

$$\lambda M + \sum_{m=1}^{M} E_m \quad ,$$

for some regularization constant λ. It is enough to search over all partitions of the training data (not all partitions of the input space!) to optimize this, but the problem is NP-complete.

Study Question: Be sure you understand why it's enough to consider all partitions of the training data, if this is your objective.

2.1.1 Building a tree

So, we'll be greedy. We establish a criterion, given a set of data, for finding the best single split of that data, and then apply it recursively to partition the space. We will select the partition of the data that *minimizes the sum of the mean squared errors of each partition.*

Given a data set D, let

- $R^{+}_{j,s}(D) = \{x \in D \mid x_j \geqslant s\}$ be the set of examples in data set D whose value in dimension j is greater than or equal to s;
- $R^{-}_{j,s}(D) = \{x \in D \mid x_j < s\}$ be the set of examples in D whose value in dimension j is less than s;
- $\hat{y}^{+}_{j,s} = \text{average}_{\{i \mid x^{(i)} \in R^{+}_{j,s}(D)\}} y^{(i)}$ be the average y value of the data points in set $R^{+}_{j,s}(D)$; and
- $\hat{y}^{-}_{j,s} = \text{average}_{\{i \mid x^{(i)} \in R^{-}_{j,s}(D)\}} y^{(i)}$ be the average y value of the data points in set $R^{-}_{j,s}(D)$.

Now, here is the pseudocode.

BuildTree(D, k):

- If $|D| \leqslant k$: return Leaf(D)
- Find the dimension j and split point s that minimizes: $E_{R^{-}_{j,s}(D)} + E_{R^{+}_{j,s}(D)}$.
- Return **Node**$(j, s,$ **BuildTree**$(R^{-}_{j,s}(D, k)),$ **BuildTree**$(R^{+}_{j,s}(D, k)))$

Each call to **BuildTree** considers $O(dn)$ splits (for d dimensions, since we only need to split between each data point in each dimension); each requires $O(n)$ work where n is the number of data points considered ($n = |D|$ if all data points in D are used).

Study Question: Concretely, what would be a good set of split-points to consider for dimension j of a dataset D?

2.1.2 Pruning

It might be tempting to regularize by using a somewhat large value of k, or by stopping when splitting a node does not significantly decrease the error. One problem with short-sighted stopping criteria is that they might not see the value of a split that will require one more split before it seems useful.

Study Question: Apply the decision-tree algorithm to the XOR problem in two dimensions. What is the training-set error of all possible hypothesis based on a single split?

So, we will tend to build a tree that is too large, and then prune it back.

Define *cost complexity* of a tree T, where m ranges over its leaves as

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m(T) + \alpha|T| \ .$$

For a fixed α, we can find a T that (approximately) minimizes $C_\alpha(T)$ by "weakest-link" pruning:

- Create a sequence of trees by successively removing the bottom-level split that minimizes the increase in overall error, until the root is reached.
- Return the T in the sequence that minimizes the criterion.

We can choose an appropriate α using cross validation.

2.2 Classification

The strategy for building and pruning classification trees is very similar to the strategy for regression trees.

Given a region R_m corresponding to a leaf of the tree, we would pick the output class y to be the value that exists most frequently (the *majority value*) in the data points whose x values are in that region:

$$O_m = \text{majority}_{\{i | x^{(i)} \in R_m\}} y^{(i)} \ .$$

Define the error in a region as the number of data points that do not have the value O_m:

$$E_m = \left|\{i \mid x^{(i)} \in R_m \text{ and } y^{(i)} \neq O_m\}\right| \ .$$

Define the *empirical probability* of an item from class k occurring in region m as:

$$\hat{P}_{mk} = \hat{P}(R_m)(k) = \frac{\left|\{i \mid x^{(i)} \in R_m \text{ and } y^{(i)} = k\}\right|}{N_m} \ ,$$

where N_m is the number of training points in region m. We'll define the empirical probabilities of split values, as well, for later use.

$$\hat{P}_{mjv} = \hat{P}(R_{mj})(v) = \frac{\left|\{i \mid x^{(i)} \in R_m \text{ and } x_j^{(i)} \geqslant v\}\right|}{N_m}$$

Splitting criteria In our greedy algorithm, we need a way to decide which split to make next. There are many criteria that express some measure of the "impurity" in child nodes. Some measures include:

- *Misclassification error*:
$$Q_m(T) = \frac{E_m}{N_m} = 1 - \hat{P}_{mO_m}$$
- *Gini index*:
$$Q_m(T) = \sum_k \hat{P}_{mk}(1 - \hat{P}_{mk})$$
- *Entropy*:
$$Q_m(T) = H(R_m) = -\sum_k \hat{P}_{mk} \log_2 \hat{P}_{mk}$$

  So that this is well-defined when $\hat{P} = 0$, we will stipulate that $0 \log_2 0 = 0$.

They are very similar, and it's not entirely obvious which one is better. We will focus on entropy, just to be concrete.

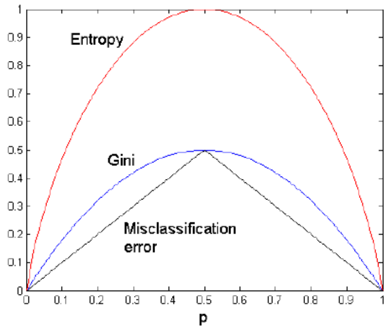
Choosing the split that minimizes the entropy of the children is equivalent to maximize the *information gain* of the test $X_j = v$, defined by

$$\text{infoGain}(X_j = v, R_m) \quad = \quad H(R_m) - \left(\hat{P}_{mjv} H(R^{+}_{j,v}) + (1 - \hat{P}_{mjv}) H(R^{-}_{j,v})\right)$$

In the two-class case, all the criteria have the values

$$\begin{cases} 0.0 & \text{when } \hat{P}_{m0} = 0.0 \\ 0.0 & \text{when } \hat{P}_{m0} = 1.0 \end{cases}$$

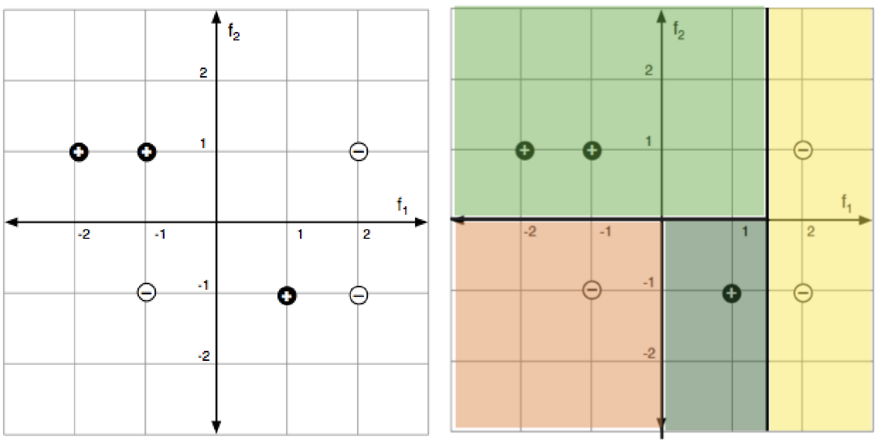
The respective impurity curves are shown below, where $p = \hat{p}_{m0}$:



There used to be endless haggling about which one to use. It seems to be traditional to use:

- Entropy to select which node to split while growing the tree
- Misclassification error in the pruning criterion

As a concrete example, consider the following images:



The left image depicts a set of labeled data points and the right shows a partition into regions by a decision tree.

Points about trees There are many variations on this theme:

- Linear regression or other regression or classification method in each leaf

- Non-axis-parallel splits: e.g., run a perceptron for a while to get a split.

What's good about trees:

- Easily interpretable
- Fast to train!
- Easy to handle multi-class classification
- Easy to handle different loss functions (just change predictor in the leaves)

What's bad about trees:

- High estimation error: small changes in the data can result in very big changes in the hypothesis.
- Often not the best predictions

Hierarchical mixture of experts Make a "soft" version of trees, in which the splits are probabilistic (so every point has some degree of membership in every leaf). Can be trained with a form of gradient descent.

3 Bagging

Bootstrap aggregation is a technique for reducing the estimation error of a non-linear predictor, or one that is adaptive to the data.

- Construct B new data sets of size n by sampling them with replacement from $\mathcal{D}$
- Train a predictor on each one: $\hat{f}^b$
- *Regression case*: bagged predictor is

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)$$

- *Classification case*: majority bagged predictor: let $\hat{f}^b(x)$ be a "one-hot" vector with a single 1 and $K-1$ zeros, so that $\hat{y}^b(x) = \arg\max_k \hat{f}^b(x)_k$. Then

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x),$$

  which is a vector containing the proportion of classifiers that predicted each class k for input x; and the predicted output is

$$\hat{y}_{\text{bag}}(x) = \arg\max_k \hat{f}_{\text{bag}}(x)_k \quad .$$

There are theoretical arguments showing that bagging does, in fact, reduce estimation error. However, when we bag a model, any simple intrepetability is lost.

3.1 Random Forests

Random forests are collections of trees that are constructed to be de-correlated, so that using them to vote gives maximal advantage. In competitions, they often have the best classification performance among large collections of much fancier methods.

For $b = 1..B$

- Draw a bootstrap sample $\mathcal{D}_b$ of size n from $\mathcal{D}$
- Grow a tree on data $\mathcal{D}_b$ by recursively repeating these steps:
  - Select m variables at random from the d variables
  - Pick the best variable and split point among them
  - Split the node
- return tree T_b

Given the ensemble of trees, vote to make a prediction on a new x.

4 Nearest Neighbor

In nearest-neighbor models, we don't do any processing of the data at training time – we just remember it! All the work is done at prediction time.

Input values x can be from any domain $\mathcal{X}$ ($\mathbb{R}^d$, documents, tree-structured objects, etc.). We just need a distance metric, $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$, which satisfies the following, for all $x, x', x'' \in \mathcal{X}$:

$$\begin{aligned} d(x, x) &= 0 \\ d(x, x') &= d(x', x) \\ d(x, x'') &\leqslant d(x, x') + d(x', x'') \end{aligned}$$

Given a data-set $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, our predictor for a new $x \in \mathcal{X}$ is

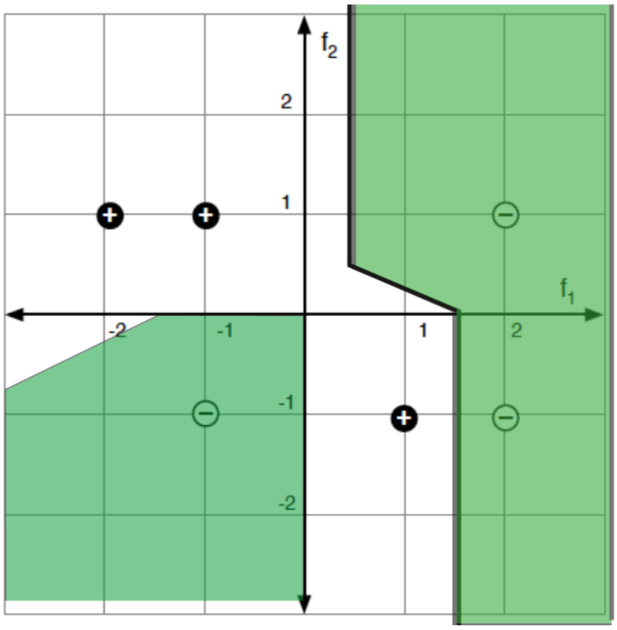
$$h(x) = y^{(i)} \quad \text{where} \quad i = \arg\min_i d(x, x^{(i)}) \ ,$$

that is, the predicted output associated with the training point that is closest to the query point x.

This same algorithm works for regression *and* classification!

It's a floor wax *and* a dessert topping!

The nearest neighbor prediction function can be described by a Voronoi partition (dividing the space up into regions whose closest point is each individual training point) as shown below:



In each region, we predict the associated y value.

> **Study Question:** Convince yourself that these boundaries do represent the nearest-neighbor classifier derived from these 6 data points.

There are several useful variations on this method. In k-*nearest-neighbors*, we find the k training points nearest to the query point x and output the majority y value for classification or the average for regression. We can also do *locally weighted regression* in which we fit locally linear regression models to the k nearest points, possibly giving less weight to those that are farther away. In large data-sets, it is important to use good data structures (e.g., ball trees) to perform the nearest-neighbor look-ups efficiently (without looking at all the data points each time).

CHAPTER 15

Appendices

1 Collaborative filtering and the SVD

This appendix presents the relationship between collaborative filtering and the singular value decomposition, a fundamental matrix decomposition method in linear algebra.

Recall that the goal of collaborative filtering is to approximate an $n \times m$ matrix Y into a rank k matrix X, where X is decomposed as the product of an $n \times k$ matrix U, and a $k \times m$ matrix V^T, i.e., $X = UV^T$. Let us denote the k *columns* of U as $\{u_i\}$ (for i in $1 \cdots k$), and the k *rows* of V^T as $\{v_i\}$. This decomposition may be visualized as the following matrix product:

$$X = UV^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_k \\ | & | & & | \end{bmatrix} \begin{bmatrix} — & v_1 & — \\ — & v_2 & — \\ & \vdots & \\ — & v_k & — \end{bmatrix} \tag{15.1}$$

Recall that k is the rank of X, and thus each u_i is linearly independent of all other column vectors of U, and similarly for v_i. (This makes the k features independent of each other.) Because of this linear independence, we may choose u_i such that $(u_i)^T u_j = ||u_i||^2 \delta_{ij}$ is zero for $i \neq j$, meaning that each u_i is *orthogonal* to other column vectors of U. This orthogonalization can be done using a Gram-Schmidt process, for example. The row vectors v_i can similarly be constructed to be orthogonal, and in the following we assume the $\{u_i\}$ and $\{v_i\}$ are each sets of orthogonal vectors.

Consider, now, what happens when the $n \times m$ matrix X is left-multiplied by one of the $1 \times n$ vectors $(u_i)^T$. Evidently

$$(u_i)^T X = \sum_j (u_i)^T u_j v_j = ||u_i||^2 v_i \,, \tag{15.2}$$

where $||u_i||^2$ is the square of the norm of the vector u_i. Similarly, when X is right-multiplied by one of the $m \times 1$ vectors $(v_i)^T$, we get:

$$X(v_i)^T = \sum_j u_j v_j (v_i)^T = ||v_i||^2 u_i \,. \tag{15.3}$$

Combining these to compute the right-multiplication of $X^T X$ by $(v_i)^T$, we observe something very interesting:

$$X^T X (v_i)^T = ||v_i||^2 X^T u_i = ||v_i||^2 ((u_i)^T X)^T = ||v_i||^2 ||u_i||^2 (v_i)^T \,, \tag{15.4}$$

which means that $(v_i)^T$ is a "right" *eigenvector* of $X^T X$, with eigenvalue $||v_i||^2 ||u_i||^2$! Similarly, the left-multiplication of XX^T by $(u_i)^T$ gives:

$$(u_i)^T X X^T = ||u_i||^2 v_i X^T = ||u_i||^2 (X (v_i)^T)^T = ||u_i||^2 ||v_i||^2 (u_i)^T \,, \tag{15.5}$$

which means that $(u_i)^T$ is a "left" eigenvector of XX^T, with eigenvalue $||u_i||^2 ||v_i||^2$.

How many eigenvectors do we have? Well, $X^T X$ is an $m \times m$ positive-definite matrix, so it must have m eigenvectors; let us thus extend our definition of $\{v_i\}$ to be all these m eigenvectors. And XX^T is an $n \times n$ positive-definite matrix, which has n eigenvectors, so let us similarly extend our definition of $\{u_i\}$.

Eigenvectors provide orthogonal bases for linear vector spaces, and thus it is convenient to normalize them. Let us define

$$\tilde{u}_i = \frac{u_i}{||u_i||} \tag{15.6}$$

$$\tilde{v}_i = \frac{v_i}{||v_i||} \tag{15.7}$$

as the columns of matrix $\tilde{U}$ and the rows of matrix $\tilde{V}^T$. Let us also define the diagonal matrix $\Lambda_{ii} = ||u_i|| ||v_i||$. Using these newly defined matrices, we may now rewrite X as

$$X = UV^T = \tilde{U} \Lambda \tilde{V}^T \,, \tag{15.8}$$

where, to summarize, the $n \times n$ matrix $\tilde{U}$ has as its columns the normalized left-eigenvectors of XX^T, the $m \times m$ matrix $\tilde{V}$ has as its rows the normalized right-eigenvectors of $X^T X$, and Λ is a diagonal matrix of the products of the square roots of the eigenvalues.

This is known as the *singular value decomposition* of X! The SVD is a standard matrix decomposition, and the diagonal elements of Λ are known as the *singular values* of X. These singular values are non-negative, real values, and thus Λ may be viewed as a scaling matrix. Meanwhile, $\tilde{U}$ and $\tilde{V}^T$ geometrically act as rotation matrices, because they are *unitary* matrices: $\tilde{U}^T \tilde{U} = I$ and similarly for $\tilde{V}^T$.

How does this relate to the collaborative filtering decomposition, where U and V are not square matrices? Well, the largest singular values of Y contribute "most" to Y, and thus an important method of approximating Y to some degree k is to compute $Y' = \tilde{U} \Lambda' \tilde{V}^T$, where Λ' only keeps the k largest singular values, and drops the rest to zero. We may also drop rows of $\tilde{V}^T$ and columns of $\tilde{U}$ corresponding to the dropped singular values, producing rank-k matrices U and V. This is known as taking the rank k *principal component* of Y, providing Y' which has the smallest possible Frobenius norm with Y, i.e., minimizing the square root of the sum of the absolute square of the elements of $Y - Y'$.

This shows how singular value decomposition is mathematically related to collaborative filtering, but how do they compare algorithmically? In practice, Y may have missing (or hidden) entries, and standard techniques for computing the SVD may not be robust against such missing data. Computing full sets of eigenvectors and eigenvalues can also be computationally expensive, especially if you only want those corresponding to the largest k singular values. Thus, there can be advantages to collaborative filtering algorithms, e.g., those based on gradient descent, although modifications can also be made to improve the robustness of the SVD approach.