Introduction to Deep Learning

Eugene Charniak

Contents

1 Feed-Forward Neural Nets **1**
- 1.1 Perceptrons . . . 3
- 1.2 Cross-entropy Loss functions for Neural Net . . . 8
- 1.3 Derivatives and Stochastic Gradient Descent . . . 12
- 1.4 Writing our Program . . . 16
- 1.5 Matrix Representation of Neural Nets . . . 19
- 1.6 Data Independence . . . 22
- 1.7 Written Exercises . . . 23

2 Tensorflow **25**
- 2.1 Tensorflow Preliminaries . . . 25
- 2.2 A TF Program . . . 28
- 2.3 Multi-layered NNs . . . 33
- 2.4 Other pieces . . . 35
  - 2.4.1 Checkpointing . . . 35
  - 2.4.2 Simplifying TF Graph Creation . . . 36
- 2.5 Written Exercises . . . 37

3 Convolutional Neural Networks **39**
- 3.1 Filters, Strides, and Padding . . . 40
- 3.2 A Simple TF Convolution Example . . . 46
- 3.3 Multi-level Convolution . . . 48
- 3.4 Written Execises . . . 51

4 Word Embeddings and Recurrent NNs **53**
- 4.1 Word Embeddings for Language Models . . . 53
- 4.2 Building Feed-Forward Language Models . . . 58
- 4.3 Improving Feed-Forward Language Models . . . 60
- 4.4 Overfitting . . . 61

4.5 Recurrent Networks 64
4.6 Long Short-Term Memory 70
4.7 Written Exercises 72

5 Sequence to Sequence Learning 73
5.1 The Seq2Seq Paradigm 74
5.2 Writing a Seq2Seq MT program 77
5.3 Attention in Seq2seq 80
5.4 More Advanced Models 85
5.4.1 Multi-Length Seq2Seq 85
5.4.2 Complicated Attention 86
5.4.3 Multiple-Language Machine Translation 86
5.5 Programming Exercise 86

6 Deep Reinforcement Learning 91
6.1 Value Iteration 92
6.2 Q-learning 95
6.3 Basic Deep Q-Learning 97
6.4 Policy Gradient Methods 100
6.5 Actor-Critic Methods 107
6.6 Experience Replay 109
6.7 Written Exercises 110

7 Unsupervised Neural-Network Models 113
7.1 Basic Autoencoding 113
7.2 Written Exercises 116

Chapter 1

Feed-Forward Neural Nets

It is standard to start one's exploration of *deep learning* (or *neural nets*, we use the terms interchangeably) with their use in computer vision. This area of artificial intelligence has been revolutionized by the technique and its basic starting point — *light intensity* — is naturally represented by real numbers, which is what neural nets manipulate.

To make this more concrete, consider the problem of identifying hand written digits — the numbers from zero to nine. If we were to start from scratch we would first need to build a camera to focus light rays in order to build up an image of what we see. We would then need light-sensors to turn the light-rays into electrical impulses that a computer can "sense." And finally, since we are dealing with digital computers, we need to *discretize* the image — that is, represent the colors and intensities of the light as numbers in a two-dimensional array. Fortunately we have a dataset on line in which all this has been done for us — the *Mnist* dataset (pronounced "em-nist"). (The "nist" here comes from the U.S. *National Institute of Standards*, (or *NIST*) who was responsible for gathering the data.) In this data each image is a 28 by 28 array of integers as in Figure 1.1. (We have removed the left and right border regions to make it fit better on the page.)

In Figure 1.1, 0 indicates white, 255 is black, and numbers in between are shades of grey. We call these numbers *pixel values* where a *pixel* is the smallest portion of an image that our computer can resolve. The actual "size" of the area in the world represented by a pixel depends on our camera, how far away it is from the object surface etc. But for our simple digit problem we need not worry about this.

Looking at this image closely can suggest some simpleminded ways we might go about our task. For example, notice that the pixel in position

	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	185	159	151	60	36	0	0	0	0	0	0	0	0	0
8	254	254	254	254	241	198	198	198	198	198	198	198	198	170
9	114	72	114	163	227	254	225	254	254	254	250	229	254	254
10	0	0	0	0	17	66	14	67	67	67	59	21	236	254
11	0	0	0	0	0	0	0	0	0	0	0	83	253	209
12	0	0	0	0	0	0	0	0	0	0	22	233	255	83
13	0	0	0	0	0	0	0	0	0	0	129	254	238	44
14	0	0	0	0	0	0	0	0	0	59	249	254	62	0
15	0	0	0	0	0	0	0	0	0	133	254	187	5	0
16	0	0	0	0	0	0	0	0	9	205	248	58	0	0
17	0	0	0	0	0	0	0	0	126	254	182	0	0	0
18	0	0	0	0	0	0	0	75	251	240	57	0	0	0
19	0	0	0	0	0	0	19	221	254	166	0	0	0	0
20	0	0	0	0	0	3	203	254	219	35	0	0	0	0
21	0	0	0	0	0	38	254	254	77	0	0	0	0	0
22	0	0	0	0	31	224	254	115	1	0	0	0	0	0
23	0	0	0	0	133	254	254	52	0	0	0	0	0	0
24	0	0	0	61	242	254	254	52	0	0	0	0	0	0
25	0	0	0	121	254	254	219	40	0	0	0	0	0	0
26	0	0	0	121	254	207	18	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1.1: An Mnist discretized version of an image

$[8, 8]$ is dark. Given that this is an image of a '7' this is quite reasonable. Similarly sevens often have a light patch in the middle – i.e. pixel $[13, 13]$ has a zero for its intensity value. Contrast this with the number '1', which often has the opposite values for these two positions since a standard drawing of the number does not occupy the upper left-hand corner, but does fill the exact middle. With a little though we could think of a lot of *heuristics* (rules that often work, but may not always) such as those, and then write a classification program using them.

However, this is not what we are going to do since in this book we are concentrating on *machine learning*. That is, we approach tasks by asking how we can enable a computer to learn by giving it examples along with the correct answer. In this case we want our program to learn how to identify 28x28 images of digits by giving examples of them along with the answers (also called *labels*).

Once we have abstracted away the details of dealing with the world of light rays and surfaces we are left with a *classification problem* — given a set of inputs (often called *features*) identify (or *classify*) the entity which gave rise to those inputs (or has those features) as one of a finite number of alternatives. In our case the inputs are pixels, and the classification is into ten possibilities. We denote the vector of l inputs (pixels) as $\mathbf{x} = [x_1, x_2 \ldots x_l]$ and the answer is a. In general the inputs are real numbers, and may be both positive and negative, though in our case they are all

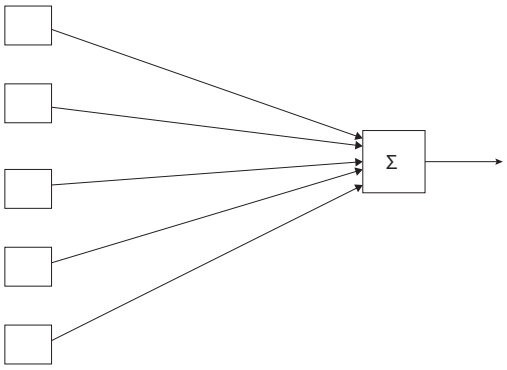

Figure 1.2: Schematic diagram of a perceptron

Figure 1.3: A typical neuron

positive integers.

1.1 Perceptrons

We start, however, with a simpler problem. We create a program to decide if an image is a zero, or not a zero. This is a *binary classification problem*. One of the earliest machine learning schemes for binary classification is the *perceptron*, shown in Figure 1.2.

Perceptrons were invented as simple computational models of neurons. A single neuron (see Figure 1.3) typically has many inputs (*dendrites*), a *cell body*, and a single output (the *axon*). Echoing this, the perceptron takes many inputs, and has one output. A simple perceptron for deciding if our 28x28 image is of a zero would have 784 inputs, one for each pixel, and one output. For ease of drawing, the perceptron in Figure 1.2 has five inputs.

A perceptron consists of a vector of *weights* $\mathbf{w} = [w_1 \dots w_m]$, one for each input, plus a distinguished weight, b, called the *bias*. We call $\mathbf{w}$ and b the *parameters* of the perceptron. More generally we use Φ to denote parameters with $\phi_i \in \Phi$ the i'th parameter. For a perceptron $\Phi = \{\mathbf{w} \cup b\}$

With these parameters the perceptron computes the following function:

$$f_\Phi(\mathbf{x}) = \begin{cases} 1 & \text{if } b + \sum_{i=1}^{l} x_i w_i > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1.1}$$

Or in words, we multiply each perceptron input by the weight for that input and add the bias. If this value is greater than zero we return 1, otherwise 0. Perceptrons, remember, are binary classifiers, so 1 indicates that $\mathbf{x}$ is a member of the class and 0, not a member.

It is standard to define the *dot product* of two vectors of length l as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{l} x_i w_i \tag{1.2}$$

so we can simplify the notation for the perceptron computation as follows:

$$f_\Phi(\mathbf{x}) = \begin{cases} 1 & \text{if } b + \mathbf{w} \cdot \mathbf{x} > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1.3}$$

Elements that compute $b + \mathbf{w} \cdot \mathbf{x}$ are called *linear units* and as in Figure 1.2 we identify them with a Σ. Also, when we discuss adjusting the parameters it is useful to recast the bias as another weight in $\mathbf{w}$, one who's feature value is always 1. (This way we only need to talk about adjusting the $\mathbf{w}$'s.)

We care about perceptrons because there is a remarkably simple and robust algorithm — the *perceptron algorithm* — for finding these Φ given *training examples*. We indicate which example we are discussing with a superscript. So the input for the k'th example is $\mathbf{x}^k = [x_1^k \dots x_l^k]$ and its answer as a^k. For a binary classifier such as a perceptron the answer is a one or zero indicating membership in the class, or not, When classifying into m classes the answer would be an integer from 0 to $m - 1$.

As in all machine-learning research we assume we have at least two, and preferably three sets of problem examples. The first is the *training set*. It is used to adjust the parameters of the model. The second is called the *development set* and is used to test the model as we try to improve it. (It is also referred to as the *held-out set* or the *validation set*.) The third is the *test set*. Once the model is fixed and (if we are lucky) producing good results, we then evaluate on the test set examples. This prevents us from accidentally developing a program that works on the development set, but not on yet unseen problems. These sets are sometimes called *corpora*, as in the "test corpus". The Mnist data we use is available on the web. The training data

1. set b and all of the $\mathbf{w}$'s to 0.
2. for N iterations, or until the weights do not change
   (a) for each training example $\mathbf{x^k}$ with answer a^k
      i. if $a^k - f(\mathbf{x}^k) = 0$ continue
      ii. else for all weights w_i, $\Delta w_i = (a^k - f(\mathbf{x}^k))x_i$

Figure 1.4: The perceptron algorithm

consists of 60,000 images and their correct labels, and the development/test set has 10,000 images and labels.

The great property of the perceptron algorithm is that if there is a set of parameter values that enables the perceptron to classify all of the training set correctly, the algorithm is guaranteed to find it. Unfortunately for most real world examples there is no such set. On the other hand, even then perceptrons often work remarkably well in the sense that there are parameter settings that label a very high percentage of the examples correctly.

The algorithm works by iterating over the training set several times, adjusting the parameters to increase the number of correctly identified examples. If we get though the training set without any of the parameters needing to change, we know we have a correct set and we can stop. However, if there is no such set then they continue to change forever. To prevent this we cut off training after N iterations, where N is a system parameter set by the programmer. Typically N grows with the total number of parameters to be learned. Henceforth we will be careful to distinguish between the system parameters Φ, and other numbers associated with our program that we might otherwise call "parameters", but are not part of Φ, such as N, the number of iterations though the training set. We call the latter *hyper parameters*. Figure 1.4 gives psuedo-code for this algorithm. Note the use of Δx in its standard use as change in x.

The critical lines here are 2(a)i and 2(a)ii. Here a_k is either one or zero indicating if the image is a member of the class ($a_k = 1$) or not. Thus the first of the two lines says, in effect, if the output of the perceptron is the correct label, do nothing. The second specifies how to change the weight w_i so that if we were to immediately try this example again the perceptron would either get it right, or at least get it less wrong, namely add $(a_k - f(\mathbf{x^k}))x_i^k$ to each parameter w_i.

The best way to see that line 2(a)ii does what we want is to go through

the possible things that can happen. Suppose the training example x_k is a member of the class, This means that its label $a_k = 1$. Since we got this wrong, $f(\mathbf{x^k})$ (the output of the perceptron on the k'th training example) must have been 0, So $(a^k - f(\mathbf{x^k})) = 1$ and for all i $\Delta w_i = x_i$. Since all are pixel values are ≥ 0 the algorithm increases the weights, and next time $f(x^k)$ returns a larger value —- it is "less wrong". (We leave it as an exercise for the reader to show that the formula does what we want in the opposite situation — when the example is not in class, but the perceptron says that it is.)

With regard to the bias b, we are treating it as a a weight for an imaginary feature x_0 who's value is always 1 and the above discussion goes through without modification.

Let us do a small example where we only look at (and adust) the weights for four pixes, those for pixels $[7, 7]$ (center of top left corner) $[7, 14]$(top center), $[14, 7]$ and $[4, 14]$. It is usually convenient to divide the pixel values to make them come out between zero and one. Assume that our image is a zero , so $(a = 1)$, and the pixel values for these four locations are .8, .9, .6, and 0 respectively. Since initially all of our parameters are zero, when we evaluate $f(x)$ on the first image $\mathbf{w} \cdot \mathbf{x} + b = 0$, so $f(\mathbf{x}) = 0$, so our image was classified incorrectly and $a(1) - f(\mathbf{x_1}) = 1$. Thus the weight $w_{7,7}$ becomes $(0 + 0.8 * 1) = 0.8$. In the same fashion, the next two w_js become 0.9 and 0.6. The center pixel weight stays zero (because the image value there is zero). The bias becomes 1.0. Note in particular that if we feed this same image into the perceptron a second time, with the new weights it would be correctly classified.

Suppose the next image is not a zero, but rather a one, and the two center pixels have value one, and the others zero. First $b + \mathbf{w} \cdot \mathbf{x} = 1 + .8 * 0 + .9 * 1 + .6 * 0 + 0 * 1 = 1.9$ so $f(x) > 0$ and the perceptron misclassifies the example as a zero. Thus $f(x) - l_x = 0 - 1 = -1$ and we adjust each weight according to Line 2(a)ii. $w_{0,0}$ and $w_{14,7}$ are unchanged because the pixel values are zero, while $w_{7,14}$ now becomes $.9 - .9 * 1 = 0$ (the previous value minus the weight times the current pixel value). We leave the new values for b and $w_{14,14}$ to the reader.

Note that we go through the training data multiple times. Each pass through the data is called an *epoch*. Also, note that the if the training data is presented to the program in a different order the weights we learn are different. Good practice is to randomize the order in which the training data is presented each epoch. We come back to this point in Section 1.6. However, for students just coming to this material for the first time, we can give ourselves some latitude here and omit this niceity.

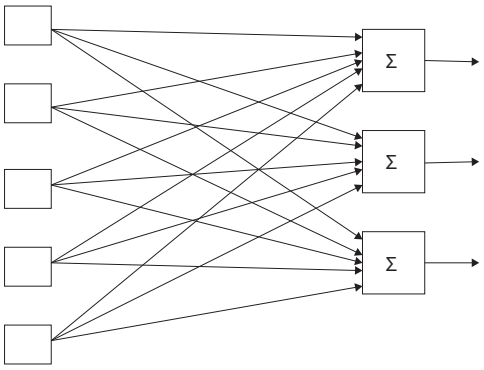

Figure 1.5: Multiple perceptrons for identification of multiple classes

We can extend perceptrons to *multi-class decision problems* by creating not one perception, but one for each class we want to recognize. For our original ten digit problem we would have ten, one for each digit, and then return the class who's perceptron value is the highest. Graphically this is shown in Figure 1.5. where we show 3 perceptrons for identifying an image as being of one of three classes of objects.

While Figure 1.5 looks very interconnected, in actuality this is simply three separate perceptrons which share the same inputs. Except for the fact that the answer returned by the multi-class perceptron is the number of the linear unit that returns the highest value, all of the perceptrons are trained independently from the others, using exactly the same algorithm shown earlier. So given an image and label we run the perceptron algorithm step (a) ten times for the ten perceptrons. If the label is, say, five, but the perceptron with the highest value is six, then the perceptrons for zero to four do not change their parameters (since they correctly said, I am not a zero, or one, etc. The same is true for six to nine. On the other hand, perceptrons five and six do modify their parameters since they reported incorrect decisions.

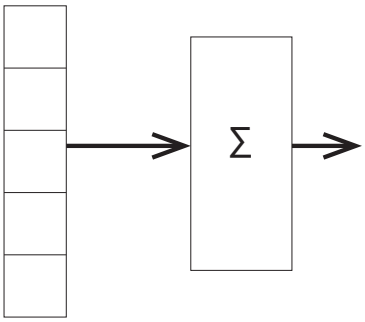

Figure 1.6: NN showing layers

1.2 Cross-entropy Loss functions for Neural Net

In their infancy, a discussion of neural nets (we henceforth abbreviate as NN) would be accompanied by diagrams much like that in Figure 1.5 with the stress on individual computing elements (the linear units). These days we expect the number of such elements to be large so we talk of the computation in terms of *layers* — a group of storage or computational units which can be thought of as working in parallel and then passing values on to another layer. Figure 1.6 is a revised version of Figure 1.5 that emphasizes this view. It shows an input layer feeding into a computational layer.

Implicit in the "layer" language is the idea that there may be many of them, each feeding into the next. This is so, and this piling of layers is the "deep" in "deep learning".

Multiple layers, however, do not work well with perceptrons, so we need another method of learning how to change weights. In this section we consider how to do this in the next simplest network configuration, *feed forward neural networks* and a relatively simple learning technique, *gradient descent*

Before we can talk about gradient descent, however, we first need to discuss *loss functions*. A loss function is a function from an outcome to how "bad" the outcome is for us. When learning model parameters our goal is to minimize loss. The loss function for perceptrons has the value zero if we got a training example correct, one if was incorrect. This is known as a *zero-one*

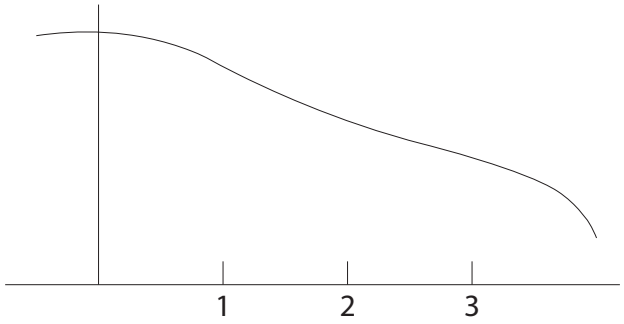

Figure 1.7: Loss as a function of ϕ_1

loss. Zero-one loss has the advantage of being pretty obvious, so obvious that we never bothered to justify its use. However, it has disadvantages. In particular it does not work well with gradient descent learning where the basic idea is to modify a parameter according to the rule

$$\Delta\phi_i = -\mathcal{L}\frac{\partial L}{\partial \phi_i} \tag{1.4}$$

Here $\mathcal{L}$ is the *learning rate*, a real number that scales how much we change a parameter at a given time. The important part is the partial derivative of the loss L with respect to the parameter we are adjusting. Or to put it another way, if we can find how the loss is affected by the parameter in question, we should change the parameter to decrease the loss (thus the minus sign preceding $\mathcal{L}$) In our perceptron, or more generally in NNs, the outcome is determined by Φ, the model parameters, so in such models the loss is a function $L(\Phi)$.

To make this easy to visualize, suppose our perceptron has only two parameters. Then we can think of a Euclidian plane, with two axes, ϕ_1 and ϕ_2 and for every point in the plane the value of the loss function hanging over (or under) the point. Say our current values for the parameters are 1.0 and 2.2 respectively. Look at the plane at position (1,2.2) and observe how L behaves at that point. Figure 1.7 shows a slice along the plane $\phi_2 = 2.2$ showing how an imaginary loss behaves as a function of ϕ_1. Look at the loss

when $\phi_1 = 1$. We see that the tangent line has a slope of about $-\frac{1}{2}$ If the learning rate $\mathcal{L} = .5$ then Equation 1.4 tells us to add $(-.5) * (-\frac{1}{2}) = .25$ That is, move about .25 units to the right, which indeed decreases the loss.

For Equation 1.4 to work the loss has to be a differentiable function of the parameters, which the zero-one loss is not. To see this, imagine a graph of the number of mistakes we make as a function of some parameter, ϕ. Say we just evaluated our perceptron on an example, and got it wrong. Well, if, say, we keep increasing ϕ (or perhaps decrease it) and we do it enough, eventually $f(x)$ changes its value, and we get the example correct. So when we look at the graph we see a step function. But step functions are not differentiable.

There are, however, other loss functions. The most popular, the closest thing to a "standard" loss function, is the *cross-entropy loss* function. In this section we explain what this is, and how our network is going to compute it. The subsequent section uses it for parameter learning.

Currently our network of Figure 1.5 outputs a vector of values, one for each linear unit, and we choose the class with the highest output value. We are now going to change our network so that the numbers output are (an estimate of) the probability distribution over classes. In our case the probability that the correct class random variable $C = c$ for $c \in [0, 1, 2, \ldots, 9]$. A *probability distribution* is a set of non-negative numbers that sum to one. Currently our network outputs numbers, but they are generally both positive and negative. Fortunately there is a convenient function for turning sets of numbers into probability distributions, *softmax*.

$$\sigma(\mathbf{x})_j = \frac{e^{x_j}}{\sum_i e^{x_i}} \tag{1.5}$$

Sofmax is guaranteed to return a probability distribution because even if x is negative e^x is positive, and the values sum to one because the denominator sums over all possible values of the numerator. For example $\sigma([-1, 0, 1]) \approx [0.09, 0.244, 0.665]$ A special case that we refer to in our further discussion is when all of the NN outputs into softmax are zero. $e^0 = 1$, so if there are ten option all of them receive probability $\frac{1}{10}$ which naturally generalizes to $\frac{1}{n}$ if there are n options.

By the way, the "softmax" gets its name from the fact that it is a "soft" version of the "max" function. The output of the max function is completely determined by the maximum input value. Softmax's output is mostly, by not completely determined by the maximum. Many machine learning functions have the name softX, for different X, but were the outputs are "softened".

Figure 1.8 shows a network with a softmax layer added in. As before the

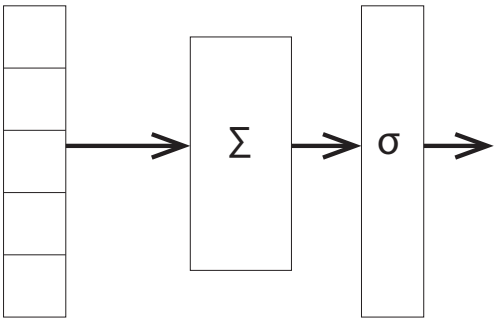

Figure 1.8: A simple network with a softmax layer

numbers coming in on the left are the image pixel values, however now the numbers going out on the right are class probabilities. It is also useful to have a name for the numbers leaving the linear units and going into the softmax function. These are typically called *logits* — a term for un-normalized numbers that we are about to turn into probabilities using softmax. (There seem to be several different pronunciations of "logit". The most common seems to be LOW-jit.) We use $\mathbf{l}$ to denote the vector of logits (one for each class).

Now we are in a position to define our cross-entropy loss function (X)

$$X(\Phi, x) = -\ln p_\Phi(a_x) \tag{1.6}$$

The cross entropy loss for an example x is the negative log probability assigned to x's label . Or to put it another way, we compute the probabilities of all the alternatives using softmax, then pluck out the one for the correct answer. The loss is the negative log probability of that number.

Let's see why this is reasonable. First, it goes in the right direction. If X is a *loss* function, it should increase as our model gets worse. Well, a model that is improving should assign higher and higher probability to the correct answer. So we put a minus sign in front so that the number gets smaller as the probability gets higher. Next, the log of a number increases/decreases as the number does. So indeed, $X(\Phi, x)$ is larger for bad parameters than for good ones.

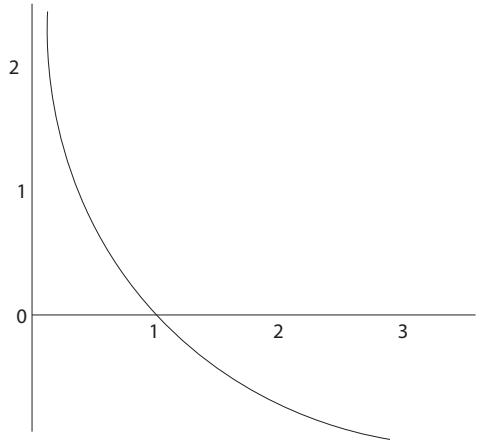

Figure 1.9: Graph of -ln(x)

But why put in the log? We are use to thinking of logarithms as shrinking distances between numbers. The difference between log(10,000) and log(1,000) is 1. One would think that would be a bad property for a loss function. It would make bad situations look less bad. But this characterization of logarithms is misleading. It is true as x gets larger log x does not increase to the same degree, But consider the graph of -ln(x) in Figure 1.9. As x goes to zero, changes in the logarithm are much larger than the changes to x. And since we are dealing with probabilities, this is the region we care about.

As for why this function is called *cross-entropy loss*, in information theory when a probability distribution is intended to approximate some true distribution, the *cross-entropy* of the two distributions is a measure of how different they are. The cross-entropy loss is an approximation of the negative of the cross-entropy As we do not have need to go deeper into information theory in this book, we leave it with this shallow explanation.

1.3 Derivatives and Stochastic Gradient Descent

We now have our loss function and we can compute it using the following equations:

$$X(\Phi, x) = -\ln p(a) \tag{1.7}$$

$$p(a) = \sigma_a(\mathbf{l}) = \frac{e^{l_a}}{\sum_i e^{l_i}} \tag{1.8}$$

$$l_j = b_j + \mathbf{x} \cdot \mathbf{w_j} \tag{1.9}$$

We first compute the logits $\mathbf{l}$ from Equation 1.9. These are then used by the softmax layer to compute the probabilities (Equation 1.8) and then we computer the loss, the negative natural-logarithm of the probability of the correct answer (Equation 1.7). Note that previously the weights for a linear unit were denoted as $\mathbf{w}$. Now we have many such units and so $\mathbf{w_j}$ are the weights for the j'th unit, and b_j is its bias.

This process, going from input to the loss, is called the *forward pass* of the learning algorithm, and it computes the values that are going to be used in the *backward pass* — the weight adjustment pass. There are several method that are standardly used here. Here we use *stochastic gradient descent*. There term *gradient descent* gets its name from looking at the slope of the loss function (its *gradient*), and then having the system lower its loss (descend) by following the gradient.

We start by looking at the simplest case of gradient estimation, that for one of the biases, b_j. We can see from Equations 1.7-1.9 that b_j changes loss by first changing the value of the logit l_j, which then changes the probability and hence the loss. Let's take this in steps. (In this we are only considering the error induced by a single training example, so we write $X(\Phi, x)$ as $X(\Phi)$).) First:

$$\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j}\frac{\partial X(\Phi)}{\partial l_j} \tag{1.10}$$

This uses the chain rule to say what we said earlier in words— changes in b_j cause changes in X in virtue of the changes they induce in the logit l_j.

Look now at the first partial derivative on the right in Equation 1.10. It's value, is, in fact, just 1

$$\frac{\partial l_j}{\partial b_j} = \frac{\partial}{\partial b_j}(b_j + \sum_i x_i w_{i,j}) = 1 \tag{1.11}$$

where $w_{i,j}$ is the i'th weight of the j'th linear unit. Since the only thing in $b_j + \sum_i x_i w_{i,j}$ that changes as a function of b_j is b_j itself, the derivative is 1.

We next consider how X changes as a function of l_j:

$$\frac{\partial X(\Phi)}{\partial l_j} = \frac{\partial p_a}{\partial l_j}\frac{\partial X(\phi)}{\partial p_a} \tag{1.12}$$

where p_i is the probability assigned to class i by the network. So this says that since X is only dependent on the probability of the correct answer, l_j only affects X by changing this probability. In turn,

$$\frac{\partial X(\phi)}{\partial p_a} = \frac{\partial}{\partial p_a}(-\ln p_a) = -\frac{1}{p_a} \tag{1.13}$$

(From basic calculus.)

This leaves one term yet to evaluate.

$$\frac{\partial p_a}{\partial l_j} = \frac{\partial \sigma_a(\mathbf{l})}{\partial l_j} = \begin{cases} (1-p_j)p_a & a = j \\ -p_j p_a & a \neq j \end{cases} \tag{1.14}$$

The first equality of Equation 1.14 comes from the fact that we get our probabilities by computing softmax on the logits. The second equality comes from Wikipedia. The derivation requires careful manipulation of terms and we do not carry it out. However we can make it seems reasonable. We are asking how changes in the logit l_j is going to effect the probability that comes out of softmax. Reminding ourselves that

$$\sigma_a(\mathbf{l}) = \frac{e^{l_a}}{\sum_i e^{l_i}}$$

it makes sense that there are two cases, Suppose the logit we are varying (j) is not equal to a. That is, suppose this is a picture of a 6, but we are asking about the bias that determines logit 8. In this case l_j only appears in the denominator, and the derivative should be negative (or zero) since the larger l_j, the smaller p_a. This is the second case in Equation 1.14, and sure enough, this case produces a number less than or equal to zero since the two probabilities we multiply cannot be negative.

On the other hand, if $j = a$, then l_j appears in both the numerator and denominator. Its appearance in the denominator tend to decrease the output, but in this case it is more than offset by the increase in the numerator. Thus for this case we expect a positive (or zero) derivative and this is what the first case of Equation 1.14 delivers.

With this result in hand we can now derive the equation for modifying the bias parameters b_j. Substituting Equations 1.13 and 1.14 into Equation 1.12 gives us:

$$\frac{\partial X(\Phi)}{\partial l_j} = -\frac{1}{p_a}\begin{cases} (1-p_j)p_a & a = j \\ -p_j p_a & a \neq j \end{cases} \tag{1.15}$$

$$= \begin{cases} -(1-p_j) & a = j \\ p_j & a \neq j \end{cases} \tag{1.16}$$

The rest is pretty simple. We noted in Equation 1.10 that

$$\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j}\frac{\partial X(\Phi)}{\partial l_j}$$

and then that the first of the derivatives on the right has value one, So the derivative of the loss with respect to b_j is given by Equation 1.12. Lastly, using the rule for changing weights (Equation 1.10), we get the rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L}\begin{cases}(1-p_j) & a = j\\ -p_j & a \neq j\end{cases} \tag{1.17}$$

The equation for changing weight parameters (as opposed to bias) is a minor variation of Equation 1.17. The corresponding equation to Equation 1.10 for weights is:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}}\frac{\partial X(\Phi)}{\partial l_j} \tag{1.18}$$

First note that the right-most derivative is the same as in 1.10. This means that during the weight adjustment phase we should save this result when we are doing the bias changes to reuse here. The first of the two derivatives on the right evaluates to

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}(b_j + (w_{1,j}x_1 + \ldots + w_{i,j}x_i + \ldots)) = x_i \tag{1.19}$$

(If we had taken to heart the idea that a bias is simply a weight who's corresponding feature value is always one we could have just derived this equation, and then Equation 1.11 would have followed immediately from 1.19 when applied to this new pseudo weight.)

Using this result we get our equation for weight updates

$$\Delta w_{i,j} = -\mathcal{L}x_i\frac{\partial X(\Phi)}{\partial l_j} \tag{1.20}$$

We have now derived how the parameters of our model should be adjusted in light of a single training example. The *gradient descent* algorithm would then have us go thought all of the training examples recording how each would recommend moving the parameter values, but not actually changing them until we have made a complete pass through all of them. At this point we modify each parameter by the sum of the changes from the individual examples.

1. for j from 0 to 9 set b_j randomly (but close to zero)
2. for j from 0 to 9 and for i from 0 to 783 set $w_{i,j}$ similarly
3. until development accuracy stops increasing
   (a) for each training example k in batches of m examples
      i. do the forward pass using Equations 1.7 1.8, and 1.9
      ii. do the backward pass using Equations 1.20, 1.17, and 1.12
      iii. every m examples, modify all Φ's with the summed updates
   (b) compute the accuracy of the model by running the forward pass on all examples in the development corpus
4. output the Φ from the iteration *before* the decrease in development accuracy.

Figure 1.10: Pseudo code for simple feed-forward digit recognition

The problem here is that this algorithm can be very slow, particularly if training set is large. We typically need to adjust the parameters often since they are going to interact in different ways as each increase and decreases as the result of particular test examples. Thus in practice we almost never use gradient descent, but rather *stochastic gradient descent* in which updates the parameters every m examples, for m much less that the size of the training set. A typical m might be twenty. This is called the *batch size*.

In general the smaller the batch size, the smaller the learning rate $\mathcal{L}$ should be set. The idea is that any one example is going to push the weights toward classifying that example correctly at the expense of the others. If the learning rate is low, this does not matter that much, since the changes made to the parameters are correspondingly small. Conversely, with larger batch-size we are implicitly averaging over m different examples so the dangers of tilting parameters to the idiosyncrasies of one example are lessened and changes made to the parameters can be larger.

1.4 Writing our Program

We now have the broad sweep of our first NN program. The pseudo code is in Figure 1.10. Starting from the top, the first thing we do is initialize the model parameters. Sometimes it is fine to initialize all to zero as we

did in the perceptron algorithm. While this is the case for our current problem as well, it is not always the case. Thus general good practice is to set weights randomly but close zero. You might also want to give the Python random number generator a key so when you are debugging you aways set the parameters to the same initial values, and thus should get exactly the same output. (If you do not, Python uses the some numbers from the environment like the last few digits from the clock as the seed.)

Note that at every iteration of the training we first modify the parameters, and then use the model on the development set to see how well the model performs with its current set of parameters. When we run development examples we do *not* run the backward training pass. If we were actually going to be using our program for some real purpose (e.g., reading zip codes on mail) the examples we see are not ones on which we have been able to train, and thus we want to know how well our program works "in the wild." Our development data is an approximation to this situation.

A few pieces of empirical knowledge come in handy here. First, it is common practice to have pixel values, not to stray too far from minus one to plus one. In our case since the original pixel values were 0 to 255, we simply divided them by 255 before using them in our network. This is an instance of a process called *data normalization*. There are no hard and fast rules, but often keeping inputs from -1, to 1, or 0 to 1 makes sense. One place we can see why this is true here is earlier in Equation 1.20 where we saw that the difference between the equation for adjusting the bias term, and that for a weight coming from one of the NN inputs, was the later had multiplicative term x_i, the value of the input term. At the time we said that if we had taken our comment that the bias term was simply a weight term who's input value was always one, the equation for updating bias parameters would have fallen out of Equation 1.20. Thus, if we leave the input values unmodified, and one of the pixels has the value 255, we modified its weight value 255 times more than we modify a bias. Given we have no a-priori reason to think one needs more correction than the other, this seems strange.

Next there is the question of setting $\mathcal{L}$, the learning rate. This can be tricky. In our implementation we used 0.0001. The first thing to note is that setting it to large is much worse than too small. If you do this you get a math overflow error from softmax. Referring again to Equation 1.5 one of the first things that should strike you are the exponentials in both the numerator and denominator. Raising e, (≈ 2.7) to a large value is a fool proof way to get an overflow, which is what we will be doing if any of the logits get large, which in turn can happen if we have a learning rate that

is too big. Even if an error message does not give you the striking message that something is amiss, a too high learning rate can cause your program to wander around in an unprofitable area of the learning curve.

For this reason it is standard practice to observe what happens to the loss on individual examples as our computation proceeds. Let us start with what to expect on the very first training image. The numbers go through the NN and get fed out to the logits layer.. All our weights and biases are zero plus or minus a small bit (say .1). This means all of the logit values are very close to zero, so all of the probabilities are very close to $\frac{1}{10}$. (See the discussion on page 10) The loss is minus the natural log of the probability assigned to the correct answer, $-\ln(\frac{1}{10}) \approx 2.3$ As a general trend we expect individual losses to decline as we train on more examples. But naturally, some images are further from the norm than others, and thus are classified by the NN with less certainty. Thus we see individual losses that go higher or lower, and the trend may be difficult to discern. Thus, rather than print out one loss at a time, we sum all of them as we go along and print the average every, say 100 batches. This average should, decrease in an easily observable fashion, though even here, you may see jitter.

Returning to our discussion of learning rate and the perils of setting it too high, a learning rate that is too low can really slow down the rate at which your program converges to a good set of parameters. So staring small and experimenting with larger values is usually the best course of action.

Because so many parameters are all changing at the same time, NN algorithms can be hard to debug. As with all debugging the trick is to change as few things as possible before the bug manifests itself. First remember the point that when we modify weights, if you were to immediate run the same training example a second time, the loss is less. If this is not true then either there is a bug, or you set the learning rate too high. Second remember that it is not necessary to change all of the weights to see the loss decrease. You can change just one of them, or one group of them. For example, when you first run the algorithm only change the biases. (However, if you think about it, a bias in a one layer network is mostly going to capture the fact that different classes occur with different frequencies. This does not happen much in the Mnist data, so we do not get much improvement by just leaning biases in this case.)

If your program is working correctly you should get an accuracy on the development data of about 91% or 92%. This is not very good for this task. In later chapters we see how to achieve about 99%. But it is a start.

One nice thing about really simple NNs that that sometimes we can directly interpret the values of individual parameters and decide if they are

reasonable or not. You may remember in our discussion of Figure 1.1, we noted that the pixel (8,8) was dark — it had a pixel value of 254. We commented that this was somewhat diagnostic of images of the digit 7, as opposed to, for example, the digit 1, which would not normally have markings in the upper-left-hand corner. We can turn this observation into a prediction about values in our weight matrix $w_{i,j}$, where i is the pixel number and j is the answer value. If the pixel values go from 0 to 784, then the position (8,8) would be pixel $8 \cdot 28 + 8 = 232$, and the weight connecting it to the answer 7 (the correct answer) would be $w_{232,7}$ while that connecting it to 1 would be $w_{232,1}$. You should make sure you see that this now suggests that $w_{232,7}$ should be larger than $w_{232,1}$. We ran our program several times with low variance random initialization of our weights. In each case the former number was positive (e.g., .25) while the second was negative (e.g., -.17).

1.5 Matrix Representation of Neural Nets

Linear Algebra gives us another way to represent what is going on in a NN — using matrices. A *matrix* is a two dimensional array of elements. In our case these elements are real numbers. The dimensions of a matrix are the number of rows and columns respectively. So a l by m matrix looks like this:

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ & & \dots & \\ x_{l,1} & x_{l,2} & \dots & x_{l,m} \end{pmatrix} \tag{1.21}$$

The primary operations on matrices are addition and multiplication. Addition of two matrices (which must be of the same dimensions) is element-wise. That is if we add two matrices, $\mathbf{X} = \mathbf{Y} + \mathbf{Z}$ then $x_{i,j} = y_{i,j} + z_{i,j}$

Multiplication of two matrices $\mathbf{X} = \mathbf{YZ}$ is defined when $\mathbf{Y}$ has dimensions l and m and those of $\mathbf{Z}$ are m and n. The result is a matrix of size l by n, where:

$$x_{i,j} = \sum_{k=1}^{k=m} y_{i,k} z_{k,j} \tag{1.22}$$

As a quick example,

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 9 & 12 & 15 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \end{pmatrix}$$
$$= \begin{pmatrix} 16 & 20 & 24 \end{pmatrix}$$

We can use this combination of matrix multiplication and addition to define the operation of our linear units. In particular the input features are a 1xl matrix $\mathbf{X}$. In the digit problem $l = 784$. The weights on for the units are $\mathbf{W}$ where $w_{i,j}$ is the i'th weight for unit j. So the dimension of $\mathbf{W}$ are the number of pixels by the number of digits, 784x10. $\mathbf{B}$ is a 1x10 matrix of biases, and

$$\mathbf{L} = \mathbf{XW} + \mathbf{B} \tag{1.23}$$

where $\mathbf{L}$ is a 1x10 matrix of logits. It is a good habit when first seeing an equation like this to make sure the dimensions work. In this case we have (1x10). = (1x784)(784x10) + (1x10).

We can now express the loss (L) for our feed forward Mnist model as following

$$\Pr(A(x)) = \sigma(\mathbf{xW} + \mathbf{b}) \tag{1.24}$$

$$L(x) = -\log(\Pr(A(x) = a)) \tag{1.25}$$

where the first equation gives the probability distribution over the possible classes ($A(x)$), and the section specifies the cross entropy loss.

We can also express the backward pass more compactly. First, we introduce the *gradient operator*

$$\nabla_{\mathbf{l}} X(\Phi) = \left(\frac{\partial X(\Phi)}{\partial l_1} \dots \frac{\partial X(\Phi)}{\partial l_m} \right) \tag{1.26}$$

The inverted triange, $\nabla_{\mathbf{x}} f(\mathbf{x})$ denotes a vector created by taking the partial derivative of f with respect to all of the values in $\mathbf{x}$. Previously we just talked about the partial derivative with respect to individual l_j. Here we define the derivative with respect to all of $\mathbf{l}$ as the vector of individual derivatives. We also remind the reader of the transpose of a matrix — making the rows of the matrix into columns, and vice versa.

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ & & \dots & \\ x_{l,1} & x_{l,2} & \dots & x_{l,m} \end{pmatrix}^T = \begin{pmatrix} x_{1,1} & x_{2,1} & \dots & x_{l,1} \\ x_{1,2} & x_{2,2} & \dots & x_{l,2} \\ & & \dots & \\ x_{1,m} & x_{2,m} & \dots & x_{l,m} \end{pmatrix} \tag{1.27}$$

With these we can rewrite Equation 1.20 as

$$\mathbf{\Delta W} = -\mathcal{L}\mathbf{X}^T \nabla_{\mathbf{l}} X(\Phi) \tag{1.28}$$

On the right we are multiplying a 784 by 1 times a 1 by 10 matrix to get a 784 by 10 matrix of changes to the 784 by 10 matrix of weights $\mathbf{W}$.

This is an elegant summary of what is going on when the input layer feeds into the layer of linear units to produce the logits, and then following the loss derivatives back to the changes in the parameters. But there is also a practical reason for preferring this new notation. When run with a large number of linear units, linear algebra in general, and deep learning training in particular can be very time consuming. However, a great many problems can be expressed in matrix notation, and many programming languages have special packages that allow you to program using linear algebra constructs. Furthermore, these packages are optimized to make them more efficient that if you had coded them by hand. In particular, if you program in Python it is well worth using the *Numpy* package and its matrix operations. Typically you get an order of magnitude speedup.

Furthermore, one particular application of linear algebra is computer graphics and its use in game-playing programs. This has resulted in specialized hardware call *graphics processing units* or *GPUs*. GPUs have slow processors compared to CPUs, but it has a lot of them, along with the software to use them efficiently in parallel for linear algebraic computations. Some specialized languages for NNs (e.g., *Tensorflow* have built in software that senses the availability of GPUs and uses them without any change in code. This typically gives another order of magnitude increase in speed.

There is a yet a third reason for adopting matrix notation in this case. Both the special purpose software packages (e.g., Numpy) and hardware (GPUs) are more efficient if we process several training examples in parallel. Furthermore, this fits with the idea that we want to process some number m of training examples (the batch size) before we update the model parameters. To this end, it is common practice to input all m of them to our matrix processing to run together. In Equation 1.23 we envisioned the image $\mathbf{x}$ as a matrix of size 1x784. This was one training example, with 784 pixels. We now change this so the matrix has dimensions m by 784. Interestingly, this almost works without any changes to our processing (and the necessary changes are already built into, e.g., Numpy and Tensorflow). Let's see why.

First consider the matrix multiplication XW where now X has m rows rather than 1. Of course, with one row we get an output of size 1x784. With m rows the output is m by 784. Furthermore as you might remember from linear algebra, but in any case can confirm by consulting the definition of matrix multiplication, the output rows are as if in each case we did multiplication of a single row and then stacked them together to get the m by 784 matrix.

Adding on the bias term in the equation does not work out as well. We said that matrix addition requires both matrices to have the same dimen-

sions. This is no-longer true for Equation 1.23 as $\mathbf{XW}$ now has size m by 10, whereas $\mathbf{B}$, the bias terms, has size 1 by 10. This is where the modest changes come in.

Numpy and Tensorflow have *broadcasting*. When some arithmetic operation requires arrays to have particular sizes other than the ones they have, arrays dimensions can sometimes be adjusted. In particular, when one of the arrays has dimension, 1 x n and we require m x n, the first gets $m-1$ (virtual) copies made of its one row or column so that it is the correct size. This is exactly what we want here. This makes $\mathbf{B}$, effectively m by 10. So we add the bias to all of the terms in the m by 10 output from the multiplication. Remember what we did when this was 1 by 10. Each of the ten were one possible decision for what the correct answer might be, and we added the bias to the number for that decision. Now we are doing the same, but for each possible decision, and all of the m examples we are running in parallel.

1.6 Data Independence

All of the theorems to the effect that if the following assumptions hold, then our NN models, in fact, converge to the correct solution depend on the *iid assumption* — that our data are independent and identically distributed. A canonical example is cosmic ray measurements — the rays stream in and the processes involved are random and unchanging.

Our data seldom (almost never) look like this —imagine NIST providing a constant stream of new examples. Processing the data in the first epoch the iid assumption looks pretty good, but as soon as we start on the second, our data are identical to the first time. In some cases our data can fail to be iid starting with training example two. This is often the case in deep reinforcement learning, (Chapter 6), and for this reason networks in that branch of deep RL often suffer from *instability* — the failure of the net to converge on the correct solution, or sometimes *any* solution. Here we consider a relatively small example where just entering the data in a non-random order can lead to disastrous results

Suppose for each Mnist image we added a second that is identical, but with black and white reversed — i.e., if the original has a pixel value of v, the reversed image has -v. We now train our Mnist perceptron on this new corpus, but using different training example orders. (And we assume the batch size is some even integer.) In the first ordering each original Mnist digit image is immediately followed by its reversed version. The claim is (and

we verified this empirically) that our simple Mnist NN fails to perform better than chance. A few moments thought should make this seem reasonable. We see image one, and the backward pass modifies the weights. We now process the second, reversed image. Because the input is minus the previous input and everything else is the same, the changes to all of the weights exactly cancel out the previous ones, and at the end of the training set there are no changes to any of the weights. So no learning, and the same random choices we started with.

On the other hand, there really should not be anything too difficult about learning to handle each data set, regular and reversed, separately and it should only be modestly more difficult for a single set of weights to handle both. Indeed, simply randomizing the order of input is sufficient to get performance back to nearly the level of the original problem. If we see the reverse image, say, 10,000 samples later, the weights have changed sufficiently so the reverse image does not exactly cancel out the original learning. If we had an unending source of images, and flipped a coin to decide to feed the NN the original or reversed, then even this small cancelation would go away.

1.7 Written Exercises

Exercise 1.1: In our feed-forward Mnist program we set batch size is one, and we look at the bias variables before and after training on the first example. If they are being set correctly, describe the changes you should see in their values.

Exercise 1.2: We simplify our Mnist computation by assuming our "image" has two binary valued pixels, zero and one, there are no bias parameters, and we are performing a binary classification problem. (a) Compute the forward pass probabilities when the pixel values are [0,1], and the weights are:

$$\begin{matrix} .2 & -.3 \\ -.1 & .4 \end{matrix}$$

(b) If the correct answer is 1 (not 0),m compute $\Delta w_{0,0}$ on the backward pass.

Exercise 1.3: A fellow student asks you "In elementary calculus we found minima of a function by differentiating it, setting the resulting expression to zero, and then solving the equation. Since our loss function is differentiable,

why don't we do that rather than bothering with gradient descent?" Explain why this is not, in fact, possible.

Exercise 1.4: In situations where we know the exact numeric value we want our n linear units to compute for a given input we often use the *sum of squares loss*

$$L(\mathbf{l}) = \sum_{i=0}^{n} (l_i - t_t)^2 \tag{1.29}$$

where t_i is the i'th target value, and l_i is the output of the i'th unit. Derive the equation for the derivative of the loss with respect to b_i.

Chapter 2

Tensorflow

2.1 Tensorflow Preliminaries

Tensorflow is an open-source programming language developed by Google that is specifically designed to make progrmming deep learning programs easy, or at least easier. We start with the traditional first program.

```
import tensorflow as tf
x = tf.constant("Hello World")
ses = tf.Session()
print(ses.run(x)) #will print out "Hello World"
```

If this looks like Python code, that is because it is. In fact Tensorflow (hence forth *TF*) is a collection of functions that can be called from inside different programming languages. The most complete interface is from inside Python, and that is what we use here.

The next thing to note is that TF functions do not so much execute a program but rather define a computation that is only executed when we call the `run` command, as in the last line of the above program. More precisely, the TF function `Session` in the third line creates a session, and associated with this session is a graph defining a computation. Commands like `constant` add elements to this computation. In this case the element is just constant data item who's value is the Python string "Hello World". The third line tells TF to evaluate the TF variable pointed to by `x` inside the graph associated with the session `ses`. As you might expect, this results in the printout "Hello World".

It is instructive to contrast this behavior with what happens if we replace the last line with `print(x)` . This prints out

```
Tensor("Const:0", shape=(), dtype=string)
```

The point is that the Python variable 'x' is not bound to a string, but rather to a piece of the Tensorflow computation graph. It is only when we evaluate this portion of the graph by executing `ses.run(x)` that we access the value of the TF constant.

So to perhaps belabor the obvious, in the above code 'x', and 'ses' are Python variables, and as such could have been named whatever we wanted. `import` and `print` are Python functions, and must be spelled this way for Python to understand which function we want executed. Lastly `constant`, `Session` and `run` are TF commands and again the spelling must be exact (including the capital "S" in `Session`). Also we always first need to `import tensorflow`. Since this is fixed we henceforth omit it.

In the following code, as before, `x` is a python variable, who's value is a TF constant, in this case the floating point number 2.0. Next, `z` is a python variable who's value is a TF*placeholder*.

```
x = tf.constant(2.0)
z = tf.placeholder(tf.float32)
ses= tf.Session()
comp=tf.add(x,z)
print(ses.run(comp,feed_dict={z:3.0}))  # Prints out 5.0
print(ses.run(comp,feed_dict={z:16.0})) # Prints out 18.0
print(ses.run(x)) # Prints out 2.0
print(ses.run(comp)) # Prints out a very long error message
```

A placeholder in TF is like the formal variable in a programming language function. Suppose we had the following python code:

```
x = 2.0
def sillyAdd(z):
   return z+x
print(sillyAdd(3))  # Prints out 5.0
print(sillyAdd(16)) # Prints out 18.0
```

Here 'z' is the name of `sillyAdd`'s argument, and when we call the function as in `sillyAdd(3)` it is replaced by its value, 3. The TF version works similarly, except the way to give TF placeholders a value is different, as seen in:

```
print(ses.run(comp,feed_dict={z:3.0}))
```

Here `feed_dict` is a named argument of `run` (so it's name must be spelled correctly). It takes as possible values Python dictionaries. In the dictionary each placeholder required by the computation must be given a value. So the first `ses.run` prints out the sum of 2.0 and 3.0, and the second 18.0. The third is there to note that if the computation does not require the placeholder's value, then there is no need to supply it. On the other hand, as the comment on the fourth print statement indicates, if the computation requires a value and it is *not* supplied you get an error.

Tensorflow is called **Tensor**flow because it's fundamental data-structures are *tensors* — typed multi-dimensional arrays. There are fifteen or so tensor types. Above when we defined the placeholder `z` we gave its type as a `float32`. Along with its type, a tensor has a *shape*. So consider a two by three matrix. It has shape [2,3]. A vector of length 4 has shape [4]. (This is different from a 1 by 4 matrix, which has shape [1,4], or a 4 by 1 matrix who's shape is [4,1].) A 3 by 17 by 6 array has shape [3,17,6]. They are all tensors. Scalers (i.e., numbers) have the null shape, and are tensors as well. The next section starts with a simple Tensorflow program for Mnist digit recognition. The primary TF code will take an image and run the forward NN pass to get the network's guess as to what digit we are looking at. Also, during the training phase it runs the backward pass and modifies the program's parameters. To hand the program the image the image we define a placeholder. It will be of type `float32`, and shape [28,28], or possibly [784], depending if we handed it a two or one dimensional python list. E.g.,

```
img=tf.placeholder(tf.float32,shape=[28,28])
```

Note that `shape` is a named argument of the `placeholder` function.

One more TF data structure before we dive into the real program. As noted before, NN models are defined by their parameters and how they are combined with the input values to produce its answer (the architecture). The parameters (e.g., the weights **w** that connect the input image to the answer logits) are (typically) initialized randomly, and the NN modifies them to minimize the loss on the training data. There are three stages to creating TF parameters. First, create a tensor with initial values. Then turn the tensor into a `variable` (which is what TF calls parameters) and then initialize the variables/parameter. For example, let's create the parameters we need for the feed-forward Mnist pseudo code in Figure 1.10, First the bias terms `b`, then the weights `W`

```
bt = tf.random_normal([10], stddev=.1)
b = tf.Variable(bt)
```

```
W = tf.Variable(tf.random_normal([784,10],stddev=.1))
ses=tf.Session()
ses.run(tf.global_variables_initializer())
print(ses.run(b))
```

The first line adds an instruction to create a tensor of shape [10] who's ten values are random numbers generated from a normal distribution with standard deviation 0.1. (It has mean 0.0, as this is the default). The second line takes `bt` and adds a piece of the TF graph that creates a variable with the same shape and values. Because we seldom need the original tensor once we have created the variable, normally we combine the two events without saving a pointer to the tensor, as in the third line which creates the parameters `W`. Before we can use either `b` or `W` we need to initialize them in the session we have created. This is done in the fifth line. The sixth line prints out (when we just ran it, it will be different every time):

```
[-0.05206999  0.08943175 -0.09178174 -0.13757218  0.15039739
 0.05112269  -0.02723283 -0.02022207  0.12535755 -0.12932496]
```

If we had reversed the order of the last two lines we would have received an error message when we attempted to evaluate the variable pointed to by `b` in the print command.

So in TF programs we create variables in which we store the model parameters. Initially their values are uninformative, typically random with small standard deviation. In line with the previous discussion, the backward pass of gradient descent modifies them. Once modified, the session (above pointed to by `ses` retains the new values, and uses them the next time we do a run of the session.

2.2 A TF Program

In Figure 2.1 we give an (almost) complete TF program for a feed-forward NN Mnist program. It should work as written. The key element that you do not see here is the code `mnist.train.next_batch`, which handles the details of reading in the Mnist data. Just to orient yourself, everything before the dashed line is concerned with setting up the TF computation graph, everything after is using the graph to first training the parameters, and then run the program to see how accrate it is on the test data. We now go through this line by line.

After importing Tensorflow and the code for reading in Mnist data we define our two sets of parameters in lines 5 and 6. This is a minor varia-

```
 0 import tensorflow as tf
 1 from tensorflow.examples.tutorials.mnist import input_data
 2 mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
 3
 4 batchSz=100
 5 W = tf.Variable(tf.random_normal([784, 10],stddev=.1))
 6 b = tf.Variable(tf.random_normal([10],stddev=.1))
 7
 8 img=tf.placeholder(tf.float32, [batchSz,784])
 9 ans = tf.placeholder(tf.float32, [batchSz, 10])
10
11 prbs = tf.nn.softmax(tf.matmul(img, W) + b)
12 xEnt = tf.reduce_mean(-tf.reduce_sum(ans * tf.log(prbs),
13                                       reduction_indices=[1]))
14 train = tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
15 numCorrect= tf.equal(tf.argmax(prbs,1), tf.argmax(ans,1))
16 accuracy = tf.reduce_mean(tf.cast(numCorrect, tf.float32))
17
18 sess = tf.Session()
19 sess.run(tf.global_variables_initializer())
20 #-----------------------------------------------
21 for i in range(1000):
22   imgs, anss = mnist.train.next_batch(batchSz)
23   sess.run(train, feed_dict={img: imgs, ans: anss})

25 sumAcc=0
26 for i in range(1000):
27    imgs, anss= mnist.test.next_batch(batchSz)
28    sumAcc+=sess.run(accuracy, feed_dict={img: imgs, ans: ans})
29 print "Test Accuracy: %r" % (sumAcc/1000)
```

Figure 2.1: Tensorflow code for a feed forward Mnist NN

tion of what we just saw in our discussion of TF variables. Next, we make placeholders for the data we feed into the NN. First in line 8 we have the placeholder for the image data. It is a tensor of shape [batchSz, 784]. In our discussion of why linear algebra was a good way to represent NN compuations (page 21) we noted that our computation is sped up when we process several examples at the same time, and furthermore, this fit nicely with the notion of a batch-size in stochastic gradient descent. Here we see how this plays out in TF. Namely, our placeholder for the image takes not 1 row of 784 pixels, but 100 of them (since this is the value of `batchSz`). Similarly, in line 9 we see that we give the program 100 of the image answers at a time.

One other point about line 9. We represent an answer by a vector of length 10 with all values zero except the ath, there a is the correct digit for that image. For example, we opened the first chapter with an image of a seven (Figure 1.1). The corresponding representation of the correct answer is `(0,0,0,0,0,0,0,1,0,0)`. Vectors of this form are called *one-hot vectors* because they have the property of selecting only one value to be active

Line 9 finishes with the parameters and inputs of our program and our code moves on to placing the actual computations in the graph. Line 11 in particular begins to show the power of TF for NN computations. It defines most of the forward NN pass of our model. In particular it specifies that we want to feed (a batch size of) images into our linear units (as defined by `W` and `b`) and then apply softmax on all of the results to get a vector of probabilities.

We recommend that when looking at code like this, first look at the shapes of the tensors involved to check that they make sense. Looking at the innermost computation, is a matrix multiplication `matmul` of the input images [100,784] times `W` [784, 10] to give us a matrix of shape [100,10], to which we add the biases, ending up with a matrix of shape [100,10]. These are the ten logits for the 100 image in our batch. We then pass this through the softmax function and end up with a [100,10] matrix of label probability assignments for our images.

I am going to speed over showing that lines 12 and 13 compute the average cross entropy loss over the 100 examples we process in parallel. Looking at the innermost computation '*' does element by element multiplication of two tensors with the same shape. This gives us rows in which everything is zero'd out except for the log of the probability of the correct answer. Then `reduce_sum` sums either columns (the default, with `reduction_index=[0]`, or, in this case, it sums over rows, `reduction_index=[1]`. This results in a [100,1] array with the log of the correct probability as the only entry in each row. Finally `reduce_mean` here sums all of the columns (again the default)

and returns the average.

I went thought this quickly because I really want to get to line 14. It is there that TF really shows its merits as line 14 is the entire backward pass of our computation.

```
tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
```

says to compute the weight changes using gradient descent and the cross entropy loss function we defined in lines 12, and 13, and a learning rate of .5. We do not have to worry about computing derivatives, or anything. If you express the forward computation in TF, and the loss in TF then the TF compiler knows how to compute the necessary derivatives and string them together in the right order to make the changes. We can modify this by choosing a different learning rate, or, if we had a different loss function, replace `xEnt` with something that pointed to a different TF computation. Naturally there are limits to TF's ability to put the backward pass on the basis of the forward pass. In particular, it is only able to do this if all of the forward pass computations are done with TF functions. For beginners such as us, this is not too great a limitations as TF has a wide variety of built in operations for which it knows how to differentiate and connect.

Lines 15 and 16 compute the `accuracy` of the model. That is, it counts the number of correct answers and divides by the number of images processed. First, focus on the standard mathematical function *argmax*, as in $\arg\max_x f(x)$ which returns the value of x that maximizes $f(x)$. In our use here `tf.argmax(prbs,1)` takes two arguments. The first is the tensor over which we are taking the argmax. The second is the *axis* of the tensor to use in the argmax. The axis is an integer from zero to one minus the length of the tensor shape. For example, if the tensor is ((0,2,4),(4,0,3)) and we use axis zero (the default) we get back (1,0,0), We first compared 0 to 4, and returned 1 since 4 was larger. We then compared 2, to 0 and returned 0 since 2 was larger. If we had used axis 1 we would have returned (2,0). In line 15 we have a batch size array of logits. The argmax function returns a batch-size array of maximum logit position. We next apply `tf.equal` to compare the max logits to the correct answer. It returns a batch-size vector of boolean values (True if they are equal), which tt tf.cast(tensor, tf.float32) turns into floating point numbers so that `tf.reduce_mean` can add them up and get the percentage correct. Do not cast the boolean values into integers as when you take the mean it will return an integer as well which in this case will always be zero.

Next, once we have defined our session (line 18) and initialized the parameter values (line 19) , we can train the model (lines 21 to 23). There we

use the code we got from the TF Mnist library to extract 100 images and their answers at a time and then run them by calling `ses.run` on the piece of the computation graph pointed to by `train` When this loop is finished we have trained on 1000 iterations with 100 images per iteration, or 100,000 test images all together. On my 4 processor Mac Pro this takes about 5 seconds. (More the first time to get the right things into the cache). I mention 4 processor because TF looks at the available computational power and generally does a good job of making using it without being told what to do.

Note one slightly odd thing about lines 21 to 23 — we never explicitly mention doing the forward pass! TF figures this out as well, based on the computation graph. From the `GradientDescentOptimizer` it knows that it needs to have performed the computation pointed to by `xEnt` (line 12), which requires the `probs` computation, which in turn specifies the forward pass computation on line 11.

Lastly, lines 25 through 29 shows how well we do on the test data in terms of percentage correct (91% or 92%). First just glancing at the organization of the graph, observe that the `accuracy` computation ultimately requires the forward pass computation `probs` but not the backward pass `train`. Thus, as we should expect, the weights are not modified to better handing the testing data.

As mentioned in Chapter 1, printing out the error rate as we train the model is a good debugging practice. As a general rule it decreases. To do this we change line 23 to

```
acc,ignore= sess.run([accuracy,train],
                     feed_dict={img: imgs, ans: anss})
```

The syntax here is normal Python for combining computations. The value of the first computation (that for `accuracy` is assigned to the variable `acc`, the second to `ignore`. A common Python idiom would be to replace `ignore` with the underscore symbol (_), the universal Python symbol used when the syntax requires a variable to accept a value, but we have no need to remember it.) Naturally we would also need to add a command to print out the value of `acc`.

We have mentioned this to encourage the reader to avoid a common mistake (at least your author and some of his beginning students have made it). The mistake is to leave line 23 alone, and add a new line, 23.5

```
acc= sess.run(accuracy, feed_dict={img: imgs, ans: anss}).
```

This, however, is less efficient as TF now does the forward pass twice, once we tell it to train and once when we ask for the accuracy. But there is

a more important reason to avoid this situation. Note that the first call modifies the weights to make the correct label for this image more likely. By placing the request for the accuracy after that the programmer gets an exaggerated idea of how well the program is performing. When we have one call to `sess.un` but ask for both values, this does not happen.

2.3 Multi-layered NNs

The program we have designed, in pseudo-code in Chapter 1 and just now in TF is single layered. There is one layer of linear units. The natural question is can we do better with multiple layers of such units. Early on NN researchers realized that the answer is "No". This follows almost immediately after we see that linear units can be recast as linear algebra matrices. That is, once we see that a one layer feed-forward NN is simply computing: $\mathbf{y} = \mathbf{XW}$. In our Mnist model $\mathbf{W}$ has shape [784,10] in order to transform the 784 pixel values into 10 logit values and and add an extra weight to replace the bias term. Suppose we add an extra layer of linear units $\mathbf{U}$ with shape [784,784] which in turn feeds into a layer $\mathbf{V}$ with the same shape as $\mathbf{W}$, [784,10]

$$\begin{aligned} \mathbf{y} &= (\mathbf{xU})\mathbf{V} && (2.1) \\ &= \mathbf{x}(\mathbf{UV}) && (2.2) \end{aligned}$$

The second line follows from the associative property of matrix multiplication. The point here is that whatever capabilities are captured in the two layer situation by the combination of $\mathbf{U}$ followed by the multiplication with $\mathbf{V}$ could be captured in by a one layer NN with $\mathbf{W} = \mathbf{UV}$

It turns out there is a simple solution — add some non-linear computation between the layers. The most commonly used option is *relu* (or ρ) which stands for *rectified linear unit* and is defined as

$$\rho(x) = \max(x, 0) \tag{2.3}$$

and is shown in Figure 2.2.

Non-linear functions put between layers in deep learning are called *activation functions*. While relu is (currently) the most popular, there are others that are in use — e.g., the *sigmoid* function, defined as:

$$S(x) = \frac{e^{-x}}{1 + e^{-x}} \tag{2.4}$$

and shown in Figure 2.3 In all cases activation are applied piecewise to the

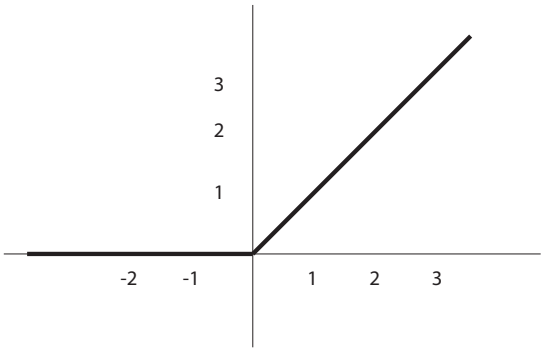

Figure 2.2: Behavior of relu

Figure 2.3: The sigmoid function

individual real numbers in the tensor argument. For example $\rho([1, 17, -3]) = [1, 17, 0]$

Before it was discovered that a non-linearity as simple as relu would work, sigmoid was very popular. But the range of values that sigmoid can take is quite limited, zero to one, whereas relu goes from zero to infinity. This is critical when we do the backward pass to find the gradient of how parameters affect the loss. Backpropagation through functions with a limited range of values can make the gradient effectively zero — a process known as the *vanishing gradient* problem. The simpler activation functions have been a great help here.

Putting the pieces of our multi-layer NN together our new model is:

$$\Pr(A(x)) = \sigma(\rho(\mathbf{x}\mathbf{U} + \mathbf{b}_u)\mathbf{V} + \mathbf{b}_v) \tag{2.5}$$

where $\mathbf{U}$ and $\mathbf{V}$ and the weights of the first and second layer of linear units, and $\mathbf{b}_u$ and $\mathbf{b}_v$ are their biases.

Let's now do this in TF. In Figure 2.4 we replace the definitions of $\mathbf{W}$ and $\mathbf{b}$ in lines 5 an 6 from Figure 2.1 with the two layers $\mathbf{U}$ and $\mathbf{V}$, lines 1 through 4 above. We also replace the computation of `prbs` in line 11 of Figure 2.1 with lines 5 though 7 above. This turns our code into

```
1 U = tf.Variable(tf.random_normal([784,784], stddev=.1))
2 bU = tf.Variable(tf.random_normal([784], stddev=.1))
3 V = tf.Variable(tf.random_normal([784,10], stddev=.1))
4 bV = tf.Variable(tf.random_normal([10], stddev=.1))
5 L1Output = tf.matmul(img,U)+bU
6 L1Output=tf.nn.relu(L1Output)
7 prbs=tf.nn.softmax(tf.matmul(L1Output,V)+bV)
```

Figure 2.4: TF graph construction code for multi-level digit recognition

a multi-layered NN. (Also, reflecting the larger number of parameters we need to lower the learning rate by a factor of 10.)While the old program plateaued at about 92% accuracy after training on 100,000 image, the new program achieves about 94% accuracy on 100,000 images. Furthermore, if we increase the number of training images performance on the test set keeps increasing to about 97%. Note that they only difference between this code and that without the non-linear function is line 6. If we delete it, performance indeed goes back down to about 92%. It is enough to make you believe in mathematics!

One other point. In a single layer network with array parameters **W**, the shape of **W** is fixed by number of inputs on one hand (784), and the number of possible outputs on the other (10) With two layers there is one more choice we are free to make, the *hidden-size*. So **U** is input-size by hidden-size, and **V** is hidden-size by output-size. In Figiure 2.4 we just set hidden-size to 784, the same as the input size, but nothing required this choice. Typically making it larger improves performance, but it plateaus.

2.4 Other pieces

2.4.1 Checkpointing

It is often useful to checkpoint a TF computation — save the tensors in a computation so the computation can be resumed at another time, or reused in a different program. In TF we do so by creating and using *saver objects*:

```
saveOb= tf.train.Saver()
```

As before, `saveOb` is Python variable, and the choice of name is yours. The object can be created at at any time prior to its use, but for reasons explained below, doing this just before initializing variables (calling

`global_variable_initialize`) is a logical place. Then after every n epochs of training, save the current values of all your variable:

```
saveOb.save(sess, "mylatest.ckpt")
```

The `save` method takes two arguments, the session to be saved, and the file name and location. In the above case the information goes in the same directory as the python program. if the argument had been `tmp/model.checkpt` it would have gone in the `tmp` subdirectory.

The call to `save` for a results in the creation of four files. The smallest, named `checkpoint` an ascii file specifying a few high level details of the checkpointing that has been done to that directory. The name `checkpoint` is fixed. If you name one of your files "checkpoint" it will be over written. The other three files names use the string you gave to `save`. In the case at hand they are named `mylatest.ckpt.data-00000-of-00001`, `mylatest.ckpt.index` and `mylatest.chpt.meta`. The first of these has the parameter values you saved. The other two contain meta information that TF uses when you want to import these values (as described shortly). If your program, calls save repeatedly, these files are overwritten each time.

Next we want to, say, do further training epochs on the same NN model we have already started training. The simplest thing to do is to modify the original training program. You retain the creation of the saver object, but now we want to initialize all TF variables with the saved values. Thus, one typically removes `global_variable_initialize` and replaces it with a call to to `restore` method of our saver object

```
saveOb.restore(sess, "mylatest.ckpt").
```

The next time you call the training program it resumes training with the TF variables set to the values they had when you last saved the variables in your previous training. Nothing else changes, however. So, if you training code printed out, say, epoch number following by the loss, this time around it will print out epoch numbers starting with one unless you rewrite your code to do otherwise. Naturally if you to fix this, or generally make things more elegant, you can, but writing better python code is not our first concern here.

2.4.2 Simplifying TF Graph Creation

Looking back at Figuire 2.4 we see that we needed seven lines of code to spell out our two layer feed-forward network. In the grand scheme of things

```
def myLayer(inpt, shape):
    W=tf.Variable(tf.random_normal(shape,stddev=.1))
    b=tf.Variable(tf.random_normal([shape[1]),,stddev=.1)
    return tf.add(tf.matmul(inpt, W), b))

L1Output = tf.nn.relu( myLayer(img, [784,784]) )
prbs=tf.nn.softmax( myLayer(L1Output, [784,10]) )
```

Figure 2.5: Use of subroutine for simplification of TF graph creation

that is not much — consider what would be required were we using Python directly. However, if we were creating, say, an eight-layer network — and by the end of this book you will be doing just that — in an eight layer network is going to require twenty four lines or so.

Fortunately, as students of computer science we know how to handle such situations, write a subroutine to handle the busy work. Figure creates the same TF graph as Figure 2.4. Note the use of the `myLayer` subroutine. We tell it the shape of the weight vector, and the input to the layer, and it adds the necessary pieces to the TF graph to implement the layer. We then call the /Volumes/iClicker Cloudsubroutine as part of of TF graph creation.

We could still further abbreviate the process by writing Python code that takes an input tensor plus a list of n TF shapes, and returns a pointer to the output of the n level feed-forward network. The point is that pieces of TF graphs can be passed around in Python like, lists, or dictionaries,

2.5 Written Exercises

Exercise 2.1: What is wrong with with the following slight modification of the code in Figure 2.5?

```
L1Output = tf.nn.relu( myLayer(img, [784,784]) )
L2Output = tf.nn.relu( myLayer(L1, [784,10]) )
prbs=tf.nn.softmax(L2Output)
```

You may assume the correct addition of the code for `myLayer`.

Chapter 3

Convolutional Neural Networks

The NNs considered so far have all been *fully connected*. That is, they have the property that all of the linear units in a layer are connected to all of the linear units in the next layer. However there is no requirement that NNs have this particular form. We can certainly imagine doing a forward pass in where a linear unit feeds its output to only some of the next layer's units. Slightly harder, but not all that hard, is seeing that, say, Tensorflow, if it knows which units are connected to which, can correctly compute the weight derivatives on the backward pass.

One special case of partially connected NNs are *convolutional neural networks*. Convolutional NNs are particularly useful in computer vision, and so we continue with our discussion of the Mnist data set.

The one level fully-connected Mnist NN learns to associate particular light intensities at certain positions in the image with certain digits — e.g. high values at position (8,14) with the number 1. But this is clearly not how people work. If we had photographed the digits in a brighter room this might add 10 to each pixel value, but would have little if any influence on our categorization — what matters in scene recognition is differences in pixel values, not their absolute value. Furthermore the the differences are only meaningful between near-by values. Suppose you are in a small room lit by a single lightbulb in one corner of the room. What we perceive as a light patch on, say some wall paper at the opposite end of the room could, in fact, be reflecting back fewer photons than a "dark" patch near the bulb. What matters in sorting out what is going on in a scene is local light intensity differences, with the emphasis on "local" and "differences". Naturally com-

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0

Figure 3.1: A simple filter for horizontal line detection

puter vision researchers are quite aware of this and the standard response to these facts has been the near universal adoption of convolutional methods.[1]

3.1 Filters, Strides, and Padding

For our purposes a *convolutional filter* (also called a *convolutional kernel*) is a (typically small) array of numbers. If we are dealing with a black and white image it is a two dimensional array. Mnist is black and white, so that is all we need here. If we had color we would need a three dimensional array — or equivalently three two dimensional arrays — one each for red, blue, and green (RBG) wavelengths of light, from which it is possible to reconstruct all colors. For the moment we ignore the complications of color. We come back to it later.

Consider the convolution filter shown in Figure 3.1 To *convolve* a filter with a patch of an image we take the dot product of the filter and an equal size piece of the image. You should remember that the dot product of two vectors does pair-wise multiplication of the corresponding elements of the vectors and sums the products to get a single number. Here we are generalizing this notion to two or more dimensional arrays, so we multiply the corresponding elements of the arrays and then sum all of the products.

Let us convolve this filter with the lower right-hand piece of the simple image of a square shown in Figure 3.2. The bottom two rows of the filter all overlap with zeros in the image. But the filter's top left four elements all overlap with the 2.0's of the square, so the value of the filter on this patch is 8. Naturally if all the pixel values were zero the filter application value would be zero. But if the entire image patch were all 10's, it would still be zero. In fact, it is not to hard to see that this filter has highest values for patches with a horizontal line running through the middle of the patch going

[1]The discussion here uses the term "convolution" as it is used in deep learning. This is close to, but not exactly the same as it is used in mathematics, where deep-learning convolution would be called *cross-correlation*.

0.0	0.0	0.0	0.0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Figure 3.2: Image of a small square

from high values on the top and lower values below. The point, of course, is that filters can be made to be sensitive to changes in light intensities rather than their absolute values, and, because filters are typically much smaller than complete images they concentrate on local changes. We can, of course, designed a filter kernel that has high values for image patches with straight lines going from upper left to lower right, or whatever.

In the above discussion we have presented the filter as if it were designed by the programmer to pick out a particular kind of feature in the image, and indeed, this is what was done before the advent deep convolutional filtering. However, what makes deep learning approaches special is that *the filters values are NN parameters — they are learned during the backward pass.* In our discussion of how convolution works it is easier to ignore this and we continue to present our filters "pre designed" until the next section.

Besides convolving a filter with an image *patch*, we also speak of convolving a filter with an *image*. This involves applying the filter to many of the patches in the image. Usually we have a lot of different filters. where the goal is for each filter to pick up a specific feature in the image. Having done this we can then feed all of the feature values to one or more fully connected layers and then into softmax and hence to the loss function. This architecture is shown in Figure 3.3. There we have represented the convolution filter layer with a three dimensional box (because a bank of filters is (at least) a three dimensional tensor, height by width by number of different filters).

Notice the deliberate vagueness above when we said we convolve a filter with "many" of the patches in an image. To begin to make this more precise this we first define *stride* — the distance between two applications of a filter. A stride of, say, two, would mean that we apply a filter at every other pixel. To make this still more specific we talk of both horizontal stride, s_h, and vertical stride, s_v. The idea is that as we go across the image we apply the filter after every s_h pixels. When we reach the end of a line we vertically

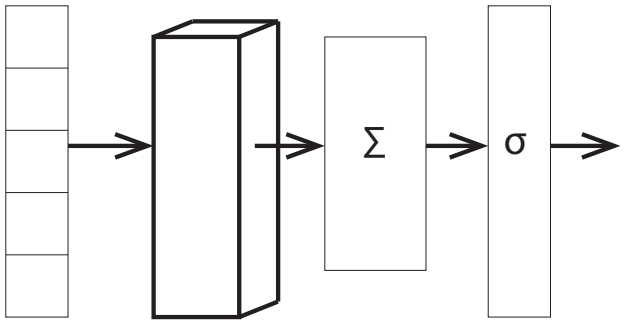

Figure 3.3: An image recognition architecture with convolution filters

descend s_v lines and repeat the process. When we apply a filter using a stride of two we still apply the filter to all of the pixels in the region (not, e.g., only every other).the only thing affected by the stride is where the filter is next applied.

Next we define what we mean by the "end of a line" in the application of a filter. This is done by specifying the *padding* for the convolution. TF allows for two possible paddings, called `Valid` and `Same` paddings. After convolving the filters with a particular patch of the image we move s_h to the right. There are three possibilities: (a) we are not anywhere near the image boundary, so we continue working on this line, (b) the left-most pixel for the next convolution patch is beyond the image edge., and (c) the left most pixel the filters look at is in the image, but the right-most is beyond the end of the image. Same padding stops in case (b), Valid stops in situation (c). For example, Figure 3.4 shows the situation where our image is 28 pixels wide, our filter is 4 pixels wide, 2 pixels high, and our stride is 1. With valid padding, we stop after pixel 24 with zero based counting.This is because our stride would take is to pixel 25, and to fit in a filter of width 4 would require a 29th pixel which does not exist. Same padding would continue to convolve until after pixel 27. Naturally, we make the same choice in the vertical direction when we reach the bottom of the image.

Since we are going to need this later, the number of patch convolutions

0	1		23	24	25	26	27
.	.	3.2	3.1	2.5	2..0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2..0	0	0
.	.	3.2	3.1	2.5	2.0	0	0

Figure 3.4: End of line with Valid and Same padding

we apply horizontally for Same padding is

$$\lfloor i_h/s_h \rfloor \tag{3.1}$$

where $\lfloor x \rfloor$ is the *floor function*. It returns the largest integer $\leq x$. The number of convolutions applied vertically is $\lfloor i_v/s_v \rfloor$. For Valid padding the number horizontally is

$$\lfloor (i_h - f_h + 1)/s_h \rfloor \tag{3.2}$$

If you don't see this last equation immediately first make sure you see that $i_h - f_h$ is how often you can shift (before running out of space) if the stride is one. But the number of applications is one plus the number of shifts.

The decision on where to stop is called *padding* because when going horizontally with same padding, by the time we stop we have to be using "imaginary" pixels. The left-hand side of the filter is within the image boundary, but the right-hand side is not. In TF the imaginary pixels have value zero. So with same padding we need to pad the boundary of the image with imaginary pixels. With valid padding it is almost never the case that we need actual padding since we stop convolving before any part of the filter moves beyond the image edge. When padding is required (with same padding) the padding is applied to all edges as equally as possible.

Despite its use of imaginary pixels same padding is quite popular because when combined with stride equal to one, it has the property that the size of the output is the same as that of the original image. Frequently we combine many layers of convolution, each output becoming the input for the next layer. No matter what the stride size, valid padding always has an output smaller than the input. With repeated layers of convolution it is guaranteed ? the result gets eaten away from the outside in.

(1, -1, -1)	(1, -1, -1)	(1, -1, -1)	(1, -1, -1)
(-1 , 0, 0)	(-1, 0, 0)	(-1, 0, 0)	(-1, 0, 0)
(-1 , 0, 0)	(-1, 0, 0)	(-1 0, 0)	(-1, 0, 0)

Figure 3.5: A simple filter for horizontal ketchup line detection

Before moving on to actual code we need to discuss how convolution affects how we represent images. The heart of a convolution NN in TF is the two dimensional convolution function

`tf.nn.conv2d`(*input, filters, strides, padding*)

plus optional named arguments that we ignore here. The `2d` in the name specifies that we are convolving an image. (There are also `1d` and `3d` versions that convolve one-dimensional objects, such as an audio signal, or for 3d, perhaps a video clip.) As you might expect the first argument is a batch size of individual images. So far we have though of an individual images as a 2D array of numbers —- each number a light intensity. If you include the fact that we have batch size of them, the input is a three dimensional tensor.

However, `tt.nn.conv2d` requires individual images to be three dimensional objects where the last dimension is a vector of *channels*. As mentioned earlier, normal color images have three channels — one each for red, blue, and green (RBG).. From now on when we discuss images, we are still talking about a 2D array of pixels, but each pixel is a list of intensities. That list has one value in it for black and white pictures, three values for colored ones.

The same is true for convolution filters. A m by n filter matches up with m by n pixels, but now both the pixels and the filter may have multiple channels. In a somewhat fanciful case, we create a filter to find horizontal edges of ketchup bottles in Figure 3.1. The topmost row of the filter is activated most highly when the input light is intense only for red. The next two rows want less red, (so there is some contrast) but otherwise do not care about the wave lengths.

Figure 3.6 shows a simple TF example of applying a small convolution feature to a small invented image. As noted above the first input to `conv2D` is a 4D tensor, here the constant `I`. In the comment just before declaring `I` we show what it would look like as a simple 2D array, without the extra dimensions added by batch-size (here one), and channel size (again one). The second argument is 4D tensor of filters, here `W`, again with a comment showing a 2D version, this time without the extra dimensions of number of

```
ii = [[ [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]] ]]
''' ((0 0 2 2)
     (0 0 2 2)
     (0 0 2 2)
     (0 0 2 2))'''
I = tf.constant(ii, tf.float32)

ww = [ [[[-1]],[[-1]],[[1]]],
       [[[-1]],[[-1]],[[1]]],
       [[[-1]],[[-1]],[[1]]] ]
'''((-1 -1 1)
    (-1 -1 1)
    (-1 -1 1))'''
W = tf.constant(ww, tf.float32)

C =  tf.nn.conv2d( I, W, strides=[1, 1, 1, 1], padding='VALID')
sess = tf.Session()
print sess.run(C)
'''[ [[ 6.]  [ 0.]]
     [[ 6.]  [ 0.]]]]
   ( (5 0)
     (6 ))'''
```

Figure 3.6: A simple exercise using `conv2D`

```
1  image = tf.reshape(img, [100, 28, 28, 1])
2     #Turns img into 4d Tensor
3  flts=tf.Variable(tf.truncated_normal([4,4,1,4],stddev=0.1))
4     #Create parameters for the filters
5  convOut = tf.nn.conv2d(image, flts, [1,  2,  2,  1], "SAME")
6     #Create graph to do convolution
7  convOut= tf.nn.relu(convOut)
8     #Don't forget to add nonlinearity
9  convOut=tf.reshape(convOut,[100, 784])
10    #Back to  100 1d image vectors
11 prbs = tf.nn.softmax(tf.matmul(convOut, W) + b)
```

Figure 3.7: Primary code needed to turn Figure 2.1 into a convolutional NN

channels and number of filters (one each). We lhen show the call to `conv2D` with horizontal and vertical strides both one, and valid padding. Looking at the result, first the 4D version that comes out of `conv2D`, and then the 2D version that ignores batch size and the number of channels, we see that the result is much educed from the image size, and we should expect when we use valid padding, and secondly the filter is quite active (with a value of 6) as we would expect since it is designed to pick up vertical lines, which is exactly what appears in the image.

3.2 A Simple TF Convolution Example

We now go through the exercise of turning the feed-forward TF Mnist program of Chapter 2 into a convolutional NN model. The code we create is given in Figure 3.7.

As already noted, the key TF function call is `tf.nn.conv2d`, and In Figure 3.7 we see in line 5

```
convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "SAME").
```

We look at each argument in turn. As just discussed, *input* is a four dimensional tensor —- in this case a vector of three dimensional imagers. We choose batch size to be 100, so `tf.nn.conv2d` wants 100 3D images. The functions that read the data in Chapter 2 read in vectors of one dimensional images (of length 784) so line one of Figure 3.7

```
image = tf.reshape(img,[100, 28, 28,1 ]),
```

converts the input to shape [100,28,28,1], where the final "1" indicates we have only one input channel. `tf.reshape` works pretty much like numpy reshape.

The next argument to `tf.nn.conv2d` in line 5 is a pointer to the filters to be used. This too is a 4d tensor, this time of shape

(*height, width, channels, number*)

The filter parameters are created in line 3. We have chosen 4 by 4 filters ([4,4]), each pixel has one channel ([4,4,1]), and we have chosen to make 4 filters ([4,4,1,4]). Note that the filter height and width, and how many we create, are all hyper-parameters. The number of channels (in this case 1) is determined by the number of channels in the image, so is fixed. Very importantly, we have finally done what we promised at the beginning. In line 3 we have created the filter values as parameters of the NN model (with initial values mean zero a standard deviation 0.1) so they are learned by the model.

The *strides* argument to `tf.nn.conv2d` is a list of four integers indicting the stride size in each of the four dimensions of *input*. In line 5 we see we have chosen strides of 1,2,2 and 1. In practice the first and last are almost always 1. At any rate, it is hard to imagine a case where they would not be one. After all, the first dimension is the separate 3d images in the batch. If the stride along this dimension were two, we would be skipping every-other image! Equally odd, if the last stride were greater than one, let's say two, and we had three color channels, then we would only look at the red and blue light, skipping green. So typical values for *stride* would be (1, 1, 1, 1), or if we only want to convolve every other image patch in both the horizontal and vertical directions, (1, 2, 2, 1). This is why you often see in discussions of `tf.nn.conv2d` instructions to the effect that the first and last strides must be one.

The final argument, *padding*, is a string equal to one of the padding types TF recognizes, e.g. `'SAME'`.

The output of `conv2d` is a lot like the input. It too is a 4d tensor, and like the input the first dimension of the output is the batch size. Or in other words, the output is a vector of convolution outputs, one for each input image. The next two dimensions are the number of filter applications, vertically followed by horizontally. These can be determined as in Equations 3.1 and 3.2. The last dimension of the output tensor is the number of filters were being convolved with the image. Above we said we would use four. That is, the output shape is

(*batch size, number horizontal, number vertical, number filters*)

In our case this is going to be (100, 14, 14, 4). If we think of the output as a sort of "image", then the input is 28 by 28 with one channel, but the output is (14 by 14) but with 4 channels. This means that in both cases an input image is represented by 784 numbers. We chose this deliberately to keep things similar to Chapter 2, but this need not have been the case. We could have, say, chosen to have 16, rather than four, different filters, in which case we would have an image represented by (14*14*16= 3136) numbers.

In line 11 we feed these 784 values into a fully connected layer, which produce logits for each image, which in turn are fed into softmax and then compute the cross-entropy loss (not shown in Figure 3.7) and we have a very simple convolutional NN for Mnist. The code has the general shape of that we showed in Figure 2.1. Also, line 7 above puts a non-linearity between the output of the convolution and the input of the fully connected layer. This is important. As seen before, without non-linear activation functions between linear units one does not get any improvement.

The performance of this program is significantly better than that of Chapter 2's. — 96% or a bit more, depending on the random initialization (compared to 92% for the feed forward version). The number of model parameters is virtually the same for the two versions. The feed-forward layer in both uses 7840 weights in $\mathbf{W}$ 100 biases in $\mathbf{b}$. (784 +10 weights in each unit in the fully connected layer, times 10 units). Convolution adds 4 convolution filters, each with 4*4 weights, or 64 more parameters. (This is why we choose to have the convolution output to be 784 values. To a zeroth approximation the quality of an NN goes up as we give it more parameters to use. Here, however, the number of parameters has essential remained constant.)

3.3 Multi-level Convolution

As stated earlier, we can improve the accuracy still further by going from one layer of convolution to several. In this section we construct model with two such layers. We also look a bit at what the convolution units are actually doing.

The key point in multi-level convolution is one we made in passing in the discussion of the output from `tf.conv2d`, it has the same format as the image input. Both are batch size vectors of 3d images, and the images are 2d plus one extra dimension for the number of channels. Thus the output from one layer of convolution can be the input to a second layer, and that

```
1 image = tf.reshape(img, [100, 28, 28, 1])
2 flts=tf.Variable(tf.normal([4, 4, 1, 16], stddev=0.1))
3 convOut = tf.nn.conv2d(image, flts, [1,  2,  2,  1], "SAME")
4 convOut= tf.nn.relu(convOut)
5 flts2=tf.Variable(tf.normal([2, 2, 16, 32], stddev=0.1))
6 convOut2 = tf.nn.conv2d(convOut, flts2, [1,  2,  2,  1], "SAME")
7 convOut2 = tf.reshape(convOut2, [100, 1568])
8 W = tf.Variable(tf.normal([1568,10],stddev=0.1))
9 prbs = tf.nn.softmax(tf.matmul(convOut2, W) + b)
```

Figure 3.8: Primary code needed to turn Figure 2.1 into a two layer convolutional NN

is exactly what one does. When we talked of the place holder image coming from the data the last dimension was the number of color channels. When we talk of the `conv2d` output we say it was the number of different filters we had in the convolution layer. The word "filter" here is a good one. After all, to only let blue light through to a lense we literally put a colored filters in front. So three filters give us images in the RBG spectrums. Now we get "images" in pseudo "spectra' like the "horizontal line-boundary spectra". This would be the imaginary image produced by the filter of, e.g., Figure 3.1. Furthermore just as filters for images with RBG have weights associated with all three of the spectra, in the second convolution layer there are weights for each channel output from the first.

We give the code for turning the feed-forward Minst NN into a two layer convolution model in Figure 3.8. Lines 1-4 are repeats of the first lines of Figure 3.7 except in line 2 we increased the number of filters in the first convolution layer to 16 (from 4 in the earlier version). Line 2 is responsible for creating the second convolution layer filters `flts2`. Note we created 32 of them. This is reflected in Line 5 where the values of the 32 filters become the 32 input channel values to the second convolution layer. When we linearize these output values in line 7 there are (784*4) of them. Remember we started with 784 pixels, and each convolution layer we used stride 2, both horizontally and vertically. So the resulting 3d image dimensions after the first convolution were (14, 14, 16). The second convolution also had stride two on the 14 by 14 image, and had 32 channels, so the output is [100,7,7,32] and the linearized version of a single image in line 7 has 7*7*32=1568 scalar values, which is then also the height of `W` that turns these image values into 10 logits.

```
-0.152168 -0.366335 -0.464648 -0.531652
 0.0182653 -0.00621072 -0.306908 -0.377731
 0.482902  0.581139  0.284986  0.0330535
 0.193956  0.407183  0.325831  0.284819

 0.0407645  0.279199  0.515349  0.494845
 0.140978  0.65135  0.877393  0.762161
 0.131708  0.638992  0.413673  0.375259
 0.142061  0.293672  0.166572 -0.113099

 0.0243751  0.206352  0.0310258 -0.339092
 0.633558  0.756878  0.681229  0.243193
 0.894955  0.91901  0.745439  0.452919
 0.543136  0.519047  0.203468  0.0879601

 0.334673  0.252503 -0.339239 -0.646544
 0.360862  0.405571 -0.117221 -0.498999
 0.520955  0.532992  0.220457  0.000427301
 0.464468  0.486983  0.233783  0.101901
```

Figure 3.9: Filters 0, 1, 2, and 7 of the eight filters created in one run of the two-layer convolution NN

In Chapter 2 we saw that to at least to a limited degree the numbers in $\mathbf{W}$ made sense. You may remember that since sevens tend to have high intensity values for pixel [8,8] as that often is situated in the upper left portion of a seven. But the number 1 rarely has anything written in that position. When images are converted into a linear vector of values pixel [8,8] is renumbered 8*28+8 = 232. We then observed that $\mathbf{W}[232, 1]$was always smaller that $\mathbf{W}[232, 7]$. Things make at least some sense with our convolution filters as well. Figure 3.9 shows the four by four weights for four of the eight first-level convolution filters in that were learned in one run of the code given in Figure 3.7. You might spend a few seconds on them to see if you can make out what they are looking for. For some you might. Others do not make much sense to me. However, cross correlating with the next Figure, 3.10, should help. Figure 3.10 was created by printing out for our now standard image of a seven the filter with the highest value for all 14 by 14 points in the image. Fairly quickly the impression of a seven emerges from a fog of zeros, so filter 0 must correspond to a region of all zeros. We

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 5 2 2 2 2 2 2 2 2 2 0 0
0 0 1 1 4 4 4 4 4 2 2 2 0 0
0 0 1 1 1 1 1 1 1 1 2 7 0 0
0 0 0 0 0 0 0 5 1 4 2 7 0 0
0 0 0 0 0 0 0 5 1 2 7 0 0 0
0 0 0 0 0 0 5 1 4 2 7 0 0 0
0 0 0 0 0 5 2 1 2 7 0 0 0 0
0 0 0 0 0 5 1 4 2 0 0 0 0 0
0 0 0 0 5 1 4 2 7 0 0 0 0 0
0 0 0 0 2 1 2 2 0 0 0 0 0 0
0 0 0 0 1 1 1 7 0 0 0 0 0 0
```

Figure 3.10: Most active feature for all 14 by 14 points in layer one after processing Figure 1.1

can then note that the right-hand edge of the seven's diagonal is pretty much all 7's, the bottom of the horizon pieces in the image corresponds to 1's. To me, the 1's, 2's ;and 7's all make sense. Note that there is nothing in filter 0 to suggest blank. However this too makes sense. We used numpy's arg-max function which returns the number of the largest number a list of numbers.) All the pixel values for the blank regions are zero, so all of the filters return 0. If the arg-max function returns the first value in the case when all values are equal, this is what we would expect.

Figure 3.11 is similar to Figure 3.10 except it shows the most active filters in layer two of the model. It is less interpretable than layer one. There are various arguments for why this might be the case. We have included it mostly because the first convolutional layer of illustrations are much more interpretable than most, and one should not assume that what we saw for layer one is typical.

3.4 Written Execises

Exercise 3.1: Suppose the input to a convolution NN is a 32 x 32 color image. We want to apply 8 convolution filters to it, all with shape 5x5. We are using Valid padding, and a stride of two both vertically and horizontally.

```
 0  0  0  0  0  0  0
17 11 31 17 17 16 16
 6 16 12  6  6  5  5
17 17 17  5 24  5 10
 0  0 11 26  3  5  0
 0 17 11 24  5 10  0
 0  6 24  8  5  0  0
```

Figure 3.11: Most active features for all 7 by 7 points in layer two after processing Figure 1.1

(a) What is the shape of variable in which we store the filters values? (b) What is the shape of the output of `tf.nn.conv2d`?

Chapter 4

Word Embeddings and Recurrent NNs

4.1 Word Embeddings for Language Models

A *language model* is a probability distribution over all strings in a language. At first blush this is a hard notion to get your head around. For example, consider the last sentence "At first blush ..." There is a good chance you have never seen this particular sentence, and unless you read this book again you will never see it a second time. Whatever its probability, it must be very small. Yet, contrast that sentence with the same words, but in reverse order. That is still less likely by a huge factor. So strings of words can be more or less reasonable. Furthermore programs that want to, say, translate Polish into English need to have some ability distinguish between sentence that sound like English and those that do not. A language model is a formalization of this idea.

We can get some further purchase on the idea by breaking the strings into individual words and then asking, what is the probability of the next word given the previous ones. So let $(E_{1,n}) = (E_1 \dots E_n)$ be a sequence of n random variables denoting a string of n words, and $\mathbf{e_{1,n}}$ is one candidate value. E.g. if n were 6 then perhaps $\mathbf{e_{1,6}}$ =(We live in a small world). and we could use the chain rule in probability to give usU

$$P(\text{We live in a small world}) = P(\text{We})P(\text{live}|\text{We})P(\text{in}|\text{We live})\dots \quad (4.1)$$

More generally

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{1,j-1} = e_{1,j-1}) \tag{4.2}$$

Before we go on, we should go back a bit to where we mentioned "breaking the strings into a sequence of words." This is called *tokenization* and if this were a book on text understanding we might spend as much as a chapter on this by itself. However we have different fish to fry, so we will simply say that a "word" for our purposes is any sequence of characters between two white spaces (where we consider a line feed as a white space). Note that this means that, e.g., "1066" is a word in the sentence "The Norman invasion happened in 1066." Actually, this is false, according to our definition of "word", the word that appears in the above sentence is "1066.", that is "1066" with a period after it. So we also assume that punctuation (e.g., periods, commas, colons) is split off from words, so that the final period becomes a word in its own right, separate from the 1066 word that preceded it. (You may now be beginning to see how we might spend an entire chapter on this.)

Also, we are going to cap our English vocabulary at some fixed size, say 10,000 different words. We use V to denote out vocabulary, and $|V|$ is its size. This is necessary because by the above definition of "word" we should expect to see words in our development and test sets that do not appear in the training set — e.g., "132,423" in the sentence "The population of Providence is 132,423." We do this by replacing all words not in V (so called *unknown words*) by a special word "*UNK*". So this sentence would now appear in our corpus as "The population of Providence is *UNK* ."

The data we are using in this chapter is known as the *Penn Tree-bank Corpus* or the *PTB* for short. The PTB consists of about 1,000,000 words of news articles from the Wall Street Journal. It has been tokenized, but not "unked"', so the vocabulary size is close to 50,000 words. It is called a "tree bank" because all of the sentences have been turned into trees that show their grammatical structure.. Here we ignore the trees as we are only interested in the words. We have also replaced all words that occur 10 times or less by *UNK*.

With that out of the way let us return to Equation 4.2. If we had a very large amount of English text we might be able to estimate the first two or three probabilities on its right-hand side simply by counting how often we see, e.g., "We live" and how often "in"' appears next, and then divide the second by the first (i.e,. use the maximum likelihood estimate) to give us an

estimate of, e.g., $P(\text{in}|\text{We live})$ But as n gets large this is impossible for the lack of any examples in the training corpus of a particular, say, fifty word sequence.

One standard response to this problem is to make an assumption that the probability of the next word only depends on the previous one or two words, and we can ignore all the words before that when estimating the probability of the next. The version where we assume words only depend on the previous word looks like this:

$$P(E_{1,n} = e_{1,n}) = P(E_1 = e_1) \prod_{j=2}^{j=n} P(E_j = e_j | E_{j-1} = e_{j-1}) \tag{4.3}$$

This is called a *bigram model* — where "bigram" means "two word" It is called this because each probability only depends on a sequence of two words. We can simplify this equation if we put a imaginary word, "STOP" at the beginning of the corpus, and then after every sentence. This is called *sentence padding*. So if the first "STOP" is e_0 Equation 4.3 becomes

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{j-1} = e_{j-1}) \tag{4.4}$$

Henceforth we assume that all our language corpora are sentence padded. Thus, except for the first STOP, our language model predicts all of the STOP's as well as all the real words.

With the simplifications we have put it place it should be clear that creating a bad language model is trivial. If, say , $|V| = 10,000$ we can assign the probability of any word coming after any other as $\frac{1}{10000}$. What we want, of course, is a good one — one where if the last word is "the" the distribution assigns very low probability to "a" and a much higher one to, say "cat" We do this using deep learning. That is, we give the deep network a word, w_i and expect as output a reasonable probability distribution over possible next words.

To start we need to somehow turn words into the sorts of things that deep networks can manipulate, i.e., floating-point numbers. The now standard solution is to associate each word with a vector of floats. These vectors are called *word embeddings*. For each word we initialize its embedding as a vector of e floats, where e is a system hyper-parameter. An e of 20 is small, 100 common, and 1000 not unknown. Actually we do this in two steps. First every word in the vocabulary V has a unique index (an integer) from 0 to $|V| - 1$. We then have an array $\mathbf{E}$ of dimensions $|V|$ by e. $\mathbf{E}$ holds all of the

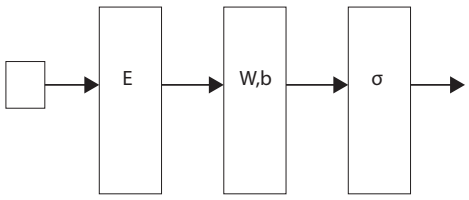

Figure 4.1: A feed-forward net for language modeling

word embeddings so that if, say, "the" has index 5, the 5'th row of **E** is the embedding of "the".

With this in mind, a very simple feed-forward network for estimating the probability of the next word is shown in Figure 4.1. The small square on the left is the input to the network — the integer index of the current word, e_i. On the right are the probabilities assigned to possible next words e_{i+1}, and the cross-entropy loss function is $-\ln P(e_c)$ the negative natural log of the probability assigned to the correct next word. Returning to the left again, the current word is immediately translated into its embedding by the *embedding layer* which looks up the e_i'th row in **E**. From that point on all NN operations are on the word embedding.

A critical point is that **E** is a parameter of the model. That is, initially the numbers in **E** are random with mean zero and small standard deviation, and their values are modified according to stochastic gradient decent. More generally, in the backward pass TF starts with the loss function and works backward looking for all parameters that affect the loss. **E** is one such parameter, so TF modifies it. What is amazing about this, aside from the fact that the process converges to a stable solution, is that the solution has the property that words which behave in similar ways end up with embeddings that are "close together". So if e (the size of the embedding vector) is, say, 30 then the prepositions "near" and "about" point in roughly

Word Num.	Word	Largest Cosine Similarity	Most Similar
0	under		
1	above	0.362	0
2	the	-0.160	0
3	a	0.127	2
4	recalls	0.479	1
5	says	0.553	4
6	rules	-0.066	4
7	laws	0.523	6
8	computer	0.249	2
9	machine	0.333	8

Figure 4.2: Ten words, the highest cosine similarity to the previous words, and the index of the word with highest similarity

the same direction. in 30-dimensional space, and neither is very close to, say, "computer' (which is closer to "machine').

With a bit more thought, however, perhaps this is not so amazing. Let us think more closely about what happens to embeddings as we try to minimize loss. As already stated the loss function is the cross entropy loss. Initially all the logit values are about equal since all of the model parameters are about equal (and close to zero).

Now, suppose we had already trained on the pair of words "says that". This would cause the model parameters to move such that the embedding for "says" leads to a higher probability for "that"' coming next. Now consider the first time the model sees the word "recalls", and furthermore it too is followed by "that". One way to modify the parameters to make "recalls" predict "that" with higher probability is to have the its embedding become more similar to that for "says" since it too wants to predict "that" as the next word. This is indeed what happens. More generally, two words which are followed by similar words get similar embeddings.

Figure 4.2 shows what happens when we run our model on about a million words of text, a vocabulary size of about 7,500 words and an embedding size of 30. The *cosine similarity* of two vectors is a standard measure of how close two vectors are to one another. In the case of two dimensional vectors it is the standard cosine function and is 1.0 if the vectors point in the same direction, 0 if they are orthogonal and -1.0 if in opposite directions. The

computation for arbitrary dimension cosine similarity is

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{(\sqrt{(\sum_{i=1}^{i=n} x_i^2)}(\sqrt{(\sum_{i=1}^{i=n} y_i^2)}} \tag{4.5}$$

In Figure 4.2 We have five pairs of similar words, numbered from zero to nine. For each word we compute its cosine similarity with all of the words that precede it. Thus we would expect all odd numbered words to be most similar to the word that immediately precedes it, and that is indeed the case. We would also expect that even numbered words (the first of each similar word pairs) not to be very similar to any of the previous words. For the most part this is true as well.

Because embedding similarity to a great extent mirrors meaning similarity, there has been a lot of study of them as a way to quantify "meaning" and we now know how to improve this result by quite a bit. The main factor is simply how many words we use for training, though there are other architectures that help as well. However, most methods suffer from similar limitations, For example, they are often blind when trying to distinguish between synonyms and antonyms. (Arguably "under" and "above" are antonyms.) Remember that a language model is trying to guess the next word, so words that have similar next words will get similar embedding, and very often antonyms do exact that. Also getting good models for embeddings of phrases rather than single words is much harder.

4.2 Building Feed-Forward Language Models

Now let us build a TF program for computing bigram probabilities. It is very similar to the digit recognition model in Figure 2.1 as in both cases we have a single fully connected layer, feed forward NN ending in a softmax to produce the probabilities needed for a cross-entropy loss. There are only a few differences.

First, rather than an image, the NN takes a word index i where $0 \leq i < |V|$ as input and the first thing is to replace it by $\mathbf{E}[i]$, the words embedding.

```
inpt=tf.placeholder(tf.int32, shape=[batchSz])
answr=tf.placeholder(tf.int32, shape=[batchSz])
E = tf.Variable(tf.random_normal([vocabSz, embedSz],
                                          stddev = 0.1))
embed = tf.nn.embedding_lookup(E, inpt)
```

We assume that there is unshown code that reads in the words and replacing the characters by unique word indices. Furthermore this code packages up `batchSz` of them in a vector. `inpt` points to this vector. The correct answer for each word (the next word of the text) is a similar vector, `answr`. Next we created the embedding lookup array `E`. The function `tf.nn.embedding_lookup` creates the necessary TF code and puts it into the computation graph. Future manipulations (e.g., `tf.matmul` will then operate on `embed`). Naturally, TF can determine how to update `E` to lower the loss, just like the other model parameters.

Turning to the other end of the feed-forward network, we use a built-in TF function to compute the cross-entropy loss:

```
xEnt =  tf.nn.sparse_softmax_cross_entropy_with_logits
              (logits=logits,labels=answr)
loss = tf.reduce_sum(xEnt)
```

The TF function `tf.nnsparse_softmax_cross_entropy_with_logits` takes two named arguments. Here the `logits` argument (which we conveniently named `logits` is a `batchSz` of logit values (i.e., a `batchSz` by `vocabSz` array of logits). The `labels` argument is a vector of correct answer. The function feeds the logits into `softmax` to get a column vector of probabilities `batchSz` by `vocabSz`. That is, the $s_{i,j}$, the i, j'th element of softmax. is the probability of word j in the i'th example in that batch. The function then locates the probability of the correct answer (from `answr`) for each line, computes its natural-log and outputs a batchSz by 1 array (effectively a column vector) of those log probabilities. The second line above takes that column vector and sum it to get the total loss for that batch of examples.

If you are curious, the use of the word "sparse" here is the same as (and presumably taken from) that in, e.g., *sparse matrix*. A sparse matrix is one with very few non-zero values so it is space efficient to only store the position and values of the non-zero values. Going back to our computation of loss in the first TF Mnist program (page 30) we assumed the correct labels for the digit images were provided in the form of one-hot vectors with only the position of the correct answer non-zero. In `tf.nn.sparse_softmax` we just give the correct answer. The correct answer can be thought of as a sparse version of the one-hot representation.

Returning to the language model, with this code in hand we do a few epochs over our training examples, and get embeddings that demonstrate word similarities like those in 4.2. Also, if we want to evaluate the language

model we can print out the total loss on the training set after every epoch. It should decrease with increasing epoch number.

In Chapter 1 (page 18) we suggested that within training epochs we print out the average per example loss, since if our parameters are improving the model, this should decrease (the numbers we see get smaller). Here we suggest a minor tweak on this idea. First note that in language modeling an "example" is assigning probabilities to possible next words, so the number of training examples is the number of words in our training corpus. So rather than talk about average per-example loss' we talk about *average per-word loss*. Next rather than print out average per-word loss, print out e raised to this power. That is, for a corpus d, with $|d|$ words, if the total loss is x_d, then print out:

$$f(d) = e^{\frac{x_d}{|d|}}. \tag{4.6}$$

This is called the *perplexity* of the corpus d. It is a good number to think about because it actually has a somewhat intuitive meaning: on average guessing the next word is equivalent to guessing the outcome of tossing a fair die with than number of outcomes. Note what this means for guessing the second word of our training corpus given the first word. If our corpus has a vocabulary size of 10,000, and we start with all of our parameters near zero, then the 10,000 logits on the first example will all be zero, and all of the probabilities will be 10^{-4}. Readers should confirm that this results in a perplexity that is exactly the vocabulary size. As we train the perplexity decreases, and, for the particular corpus you author used with a vocabulary size of about 7,800 words, after two training epochs with a training set of about 10^6 words the development set had perplexity 180, or so. With a four cpu laptop the model took about 3 minutes per epoch.

4.3 Improving Feed-Forward Language Models

There are many ways to improve the language model we just developed. For example, in Chapter 2 we saw that adding a hidden layer (with an activation function between the two layers) improved our Mnist performance from 92% correct to 98%. Adding a hidden layer here will improve the dev set perplexity from 180 to about 177.

But the most straight-forward way to get better perplexity is to move from a bi-gram language model to a *tri-gram model*. Remember that in going from Equation 4.2 to Equation 4.4 we assumed that the probability of a word only depended on the previous word. Obviously this is false. In general the choice of the next word can be influenced by words arbitrarily

far back, and the influence of the word two back is very large. So a properly trained model that bases its guess on the two previous words (called a *tri-gram* model because probabilities are based upon sequences of three words) gets much better perplexity than do bi-gram models.

In our bi-gram model we had one placeholder for the previous word index, `inpt`, and one for the word to predict (assuming a batch-size of one), `answr`. We now introduce a third placeholder that has index of the word two back, `inpt2`. In the TF computation graph we add a node that finds the embedding of `inpt2`,

```
embed2 = tf.nn.embedding_lookup(E, inpt2),
```

and then one for concatenating the two

```
both= tf.concat([embed,embed2],1).
```

Here the second argument specifies which axis of the tenser has the concatenation done to it. (Remember, in reality we are doing a batch-size of embeddings at the same time, so each of the results of the lookups is a matrix of size batch-size x embedding-size. We want to end up with a matrix of batch-size x (embedding-size *2), so the concatenation happens along axis 1, the rows (remember, the columns are axis 0). Lastly we need to change the dimensions of **W** from embedding-size x vocabulary-size to (embedding-size *2) x vocabulary-size.

In other words, we input the embeddings for two previous words, and the NN uses both in estimating the probability of the next word. Furthermore, the backward pass updates the embeddings of both words. This lowers the perplexity from 180 to about 140. Adding yet another word to the input layer lowers things still more, to about 120.

4.4 Overfitting

In Section 1.6 we discussed the iid assumption that lurks behind all of the guarantees that the training methods of our NNs do, in fact, lead to good weights. In particular we noted that as soon as we use our training data for more than one epoch our data all bets are off.

We did not, however, offer any empirical evidence to this point. The reason is that the data we used in our Chapter 1 examples, Mnist, is, as data sets go, very well behaved. What we want from training data, after all, is that it covers all of the possible things that might occur (and in the correct proportions) so when we look at the testing data there are no

Epoch	1	2	3	4	5	6	7	10	15	20	30
Train	197	122	100	87	78	72	67	56	45	41	35
Dev	172	152	145	143	143	143	145	149	159	169	182

Figure 4.3: Overfitting in a language model

Epoch	1	2	3	4	5	6	7	10	15	20	30
Dropout	213	182	166	155	150	144	139	131	122	118	114
L2 Reg	180	163	155	148	144	140	137	130	123	118	112

Figure 4.4: Language model perplexity when using dropout

surprises. With only ten digits, and 60,000 training examples, Mnist meets this criteria quite well.

Unfortunately, most data sets are not that complete, and written-language data sets in general, and the Penn tree-bank in particular, are far from ideal. Even if we restrict vocabulary size to 8,000 words, and only look at tri-gram models, there are a large number of tri-grams in the test set that do no appear in the training data. At the same time, repeatedly seeing the same (relatively small) set of examples causes the model to overestimate their probability. Figure 4.3 shows perplexity results for a two layer tri-gram language model trained on the Penn Tree-Bank. The rows give the number of epochs we have trained for, the average perplexity for each training example at each epoch, follow by the average over the development corpus. First, looking at the row of training perplexity, we see that it decreases monotonically with increasing epoch. This is as it should be.

The row of development perplexities tells a more complicated story. It too starts out decreasing, from 172 on the first epoch, to 143 on the fourth, but then it holds steady for two epochs, and starting on the seventh epoch the perplexity of the dev data increases. By the 20th iteration it is up to 169, and it reaches 182 on the 30th. The difference between the training and development results on the 30th epoch, 35 vs 182 is classic overfitting of the training data.

Regularization is the general term for modifications to fix overfitting. The simplest regularization technique is *early stopping*. We just stop training at the point where the development perplexity is the lowest. But while simple, early stopping is not the best method for correcting an overfitting problem. Figure 4.4 shows two much better solutions, *dropout* and *L2 regularization*.

In dropout we modify the network to randomly drop pieces of our computation. For example, the dropout data in Figure 4.4 came randomly

dropping the output of 50% of the the first layer of linear units. So the next layer sees zeros in random locations in its input vector. One way to think of this is that the training data no longer is identical at each epoch, since each time different units are dropped. Another way to see why this helps is to note that the classifier can not depend on the co-incidence of a lot of features of the data lining up in a particular way, and thus it should generalize better. As we can see from Figure 4.4 it really does help prevent overfitting. For one thing , the first line of Figure 4.4 shows no reversal in the perplexity of the development corpus. Even at 30 epochs perplexity is decreasing, albeit at a glacier like rate (about 0.1 units per epoch). Furthermore the absolute lower value using dropout is much better than we can achieve by early stopping — a perplexity of 114 vs.145.

The second technique we showcase in Figure 4.4 is L2 Regularization. L2 starts from the observation that overfitting in many kinds of machine learning is accompanied by the learning parameters getting quite large (or quite small for weights below zero). We commented earlier that seeing the same data repeated times causes the NN to overestimate the probabilities of what it has seen at the expense of all the examples that could occur, but did not, in the training data. This overestimation is achieved by weights with large absolute values or, almost equivalently, large squared values. In L2 regularization we add to the loss function a quantity proportional to the sum of the squared weights. That is, if before we were using cross-entropy loss, our new loss function would be:

$$\mathcal{L}(\Phi) = -log(\Pr(c)) + \alpha\frac{1}{2}\sum_{\phi\in\Phi}\phi^2. \tag{4.7}$$

Here α is a real number that controls how we weight the two terms. It is typically small. In the above experiments we set it to .01, a typical value. When we differentiate the loss function with respect to ϕ the second term adds $\alpha\phi$ to the total of $\frac{\partial\mathcal{L}}{\partial\phi}$.This encourages both positive and negative ϕ to move closer to zero.

Both forms of regularization work about equally well in this example, though in general dropout seems to be the preferred method. They are both easy to add to a TF network. To drop out say, 50% of the values coming out of the first layer of linear units, (e.g., `w1Out`) we add to our program:

```
keepP= tf.placeholder(tf.float32)
w1Out=tf.nn.dropout(w1Out,keepP)
```

Note that we made the keep probability a placeholder. We typically want to do this because we only do dropout when training. When testing it is not needed, and indeed is harmful. By making the value a placeholder we can feed in the values 0.5 when we train and 1.0 when testing.

Using L2 regularization is just as easy. If we want to prevent the values of, e.g. `W1` the weights of some linear units, from getting too large we simply add:

```
.01 * tf.nn.l2_loss(W1)
```

to the loss function we use when training. Here .01 is a hyper-parameter to weight how much we count the regularization compared to the original cross entropy loss. If your code computes perplexity by raising e to the per/word loss, be sure to separate the combined loss used in training from the loss used in the computation of perplexity. For the latter we only want the cross entropy loss.

4.5 Recurrent Networks

A *recurrent neural network* or *RNN* is, in some sense, the opposite of a feed-forword NN. It is a network where the output of the network contributes to it's own input. In graph terminology it is a directed cyclic graph, as opposed to feed-forwards directed acyclic graph. The simplest version of an RNN is illustrated in Figure 4.5. The box labeled $\mathbf{W_r b_r}$ consists of a layers of linear units with weights $\mathbf{W_r}$ and biases $\mathbf{b_r}$ plus an activation function. Input comes into to it from the bottom left and the output o, comes out on the right and splits. One copy circles back to itself. It is this circle that makes this recurrent and not feed-forward. The other copy goes to a second layer of linear units with parameters $\mathbf{W_o}, \mathbf{b_o}$ that is responsible for computing the output of the RNN, and the loss.

Recurrent networks are appropriate when we want previous inputs to the network to have an influence arbitrarily far into the future. Since language works this way, RNNs are frequently used in language-related tasks in general, and language-modeling in particular. Thus here we assume the input is the embedding of the current word w_i, the prediction is w_{i+1} and the loss is our standard cross-entropy loss.

Computing the forward pass of the NN works pretty much as it does in a feed-forward NN except that we remember o from the previous iteration and concatenate it with the current word embedding at the start of the forward pass. The backward pass, however, is not so obvious. Earlier in explaining

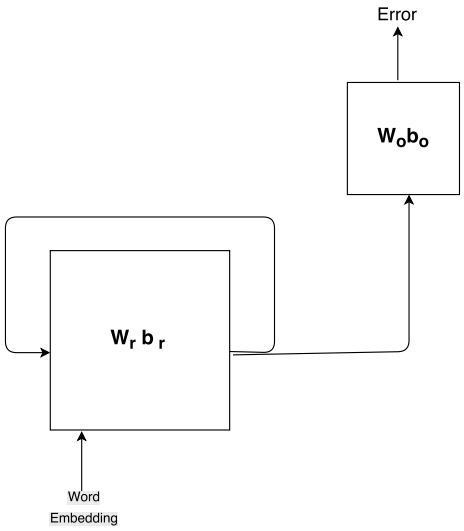

Figure 4.5: A graphical illustration of a recurrent NN

how it is that the parameters in word embeddings are also updated by TF we said TF works backward from the loss function, tracing back, continuing to look for parameters that have an effect on the error, and differentiate the error with respect to those parameters. In Chapter 1's NN for Mnist this took us back through the layer with $\mathbf{W}$ and $\mathbf{b}$, but then stopped when we encountered only the image pixels. The same is true for convolutional NNs, thought the ways in which parameters enter into the error function computation is more complicated. But now there is potentially no limit on how far back we need go in the backward pass. We first compute the error due (presumably) to not predicting w_{i+1} with probability one, we find, e.g., $\mathbf{W_o}$ and since its inputs were $\mathbf{W_r}$'s outputs, the latter's parameters must be modified as well,. But $\mathbf{W_r}$'s inputs were computed on the last iteration for $\mathbf{W_r}$ itself, so they computed to the error, and so forth and so on back. Computing effect of the weights $\mathbf{W_r}$ is an unbounded computation.

We solve this problem by brute force. We simply cut off the computation after some arbitrary number of iterations backward. The number of iterations is called the *window size* and it is a system hyper-parameter. The overall technique is called *back propagation through time* and is illustrated in Figure 4.6 where we assume a window size of three. (A more realistic value for window size would be, say, twenty.) In more detail Figure 4.6 imagines we are processing a corpus that starts with the phrase "It is a small world

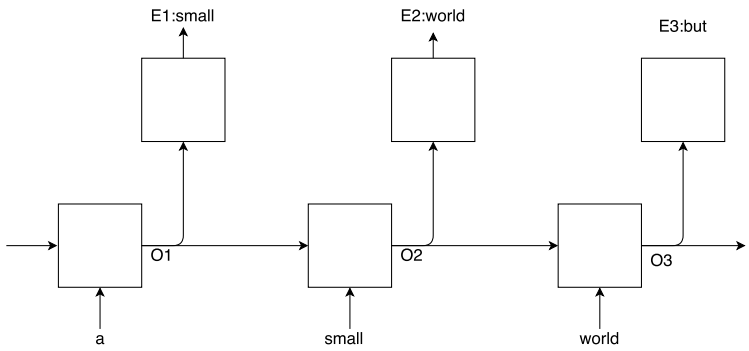

Figure 4.6: Back-propagation through time with window size equal to three

but I like it that way ." along with sentence padding. Back-prop though time treats Figure 4.6 as if it were a feed forward network taking in not a single word, but window-size (i.e., three) of them and then compute the error on the three, For our short "corpus" the first call to training would take "STOP It is" as the input words, and "it is a" as the three words to predict.

Figure 4.6 imagines we are on the second call, where the incoming words are "a small world" and they are to predict "small world but". At the start of the second forward pass the output from the first call comes in at the left (O0) and it is concatenated with the embedding of "a", and passed through the RNN to where it becomes O1 feeding the loss at E1.

But besides going to E1, O1 also goes on to be concatenated with the second word, "small". We compute the error there as well. Here we compute the effect of both **W** and **b** (not to mention embeddings), on the error in predicting "small". But **W**, and **b** cause the error in two different ways — most directly from the error that leads from them to E2, but also from how they contributed to O1. Naturally when we next compute E3, **W** and **b** effect the error three ways: directly from O3, from O1 and O2. So the parameters in those variables are modified 6 times (or equivalently, the program keeps a running total and modifies them once).

Figure 4.6 ignores the issue of batch-size. As you might expect, TF RNN functions are built to allow for simultaneous batch-size training (and testing). So each call to the RNN takes a batch-size by window-size array of input, to predict a similarly sized array of prediction words. As noted before, the semantics of this is that we are working on batch-size groups in parallel, so the last words predicted from the first training call are the first

STOP	It	is	a	small	world
but	I	like	it	that	way

STOP	It	is
but	I	like

a	small	world
it	that	way

Figure 4.7: Allotting words when batch size is two and window size is three

input words to the second.

To make this work out we need to be careful how we create the batches. Figure 4.7 illustrates what happens for the mock corpus "STOP It is a small world but l like it that way STOP" . We assume a batch size of two and a window size of three. The basic idea is to first divide the corpus into two and then fill each batch from pieces from each part (in this case half) of the corpus. The top window of 4.7 shows the corpus divided into two pieces. The next pair of windows shows the two input batches that are created from this. Each batch has three word segments from each half. We also need to batch up the prediction words to feed into the network. This is exactly like the above figure, but each word is one further along in the corpus. So the top line of the prediction diagram would read "It is a small world but".

Since the "corpus" is 14 words each half consists of six words. To see why six and not seven, concentrate on the predictions for the second batch. Go through carefully with seven words per half and you find that we do not have a prediction word for the last input. Thus the corpus is initially divided into S sections, where for a corpus of size x and a batch size b $S = \lfloor (c-1)/b \rfloor$ (where "$\lfloor x \rfloor$" is the *floor function* —the largest integer smaller than x). Here the "minus one" gives the last input word its corresponding prediction word.

We have said nothing so far about what we do at the end of a sentence. The easiest thing is to simply plow on to the next. This means that a given window-size segment fed to the RNN can contain pieces of two different sentences. However, we have put the padding STOP word between them, so the RNN should, in principle, learn what that means in terms of what sorts of words are coming up — e.g. capitalized ones. Furthermore, there can be good clues about subsequent words from the last words of the previous sentence. If we just concerned with language modeling, separating sentences

with STOP but otherwise not worrying about separation when training or using RNNs seems to be sufficient.

Let us review RNNs by looking again at Figures 4.5 and 4.6 and thinking about how we program the RNN language model. As we just noted, the code taking us from our word corpora to model input needs to be slightly revamped. Previously the input (and predictions) was a batch size vector, now it is a batch-size by window-size array. (Google has some code in TF that can do this for you.) We also need to turn each word into its word embedding, but this is unchanged from the feed-forward model. j Next the word is fed into the RNN. The key TF code for the creation of the RNN is as follow:

```
rnn= tf.contrib.rnn.BasicRNNCell(rnnSz)
initialState = rnn.zero_state(batchSz, tf.float32)
outputs, nextState = tf.nn.dynamic_rnn(rnn, embeddings,
                initial_state=initialState)
```

The first line here adds the RNN to the computation graph. Note that the width of the RNNs weight array is a free parameter, the rnnSz (you may remember when we added an extra layer of linear units to the Mnist model at the end of Chapter 2 we had a similar situation). . The last line is the call to the RNN. It takes three arguments, and returns two. The inputs are, first the RNN proper, second, the words that the RNN is going to process (there will be batch-size by window-size of them), and the `initial_state` that it gets from the previous run. Since on the first call to `dynamic_rnnn` there is no previous state, we create a dummy on with function call on the second line `rnn.zero_state`.

`tf.nn.dynamic_rnn` has two outputs. The first, which we named `outputs`, is the information that is going feed the error computation. In Figure 4.6 these are the outputs O1, O2, O3. So `output` has the shape [batch-size, window-size, hidden-size]. The first dimension packages up batch-size examples,. Each example itself consists of O1, O2, and O3, so the second dimension is window-size. Lastly, e.g, O1 is a vector of rnn-size floats that comprise the RNN output from a single word.

The second output from `tf.nn.dynamic_rnn` we called `nextState` and it is the last output (O3) from this pass though the RNN. The next time we call `tf.nn.dynamic_rnn` we will have `initial_state = nextState` Note that `nextState` is, in fact, information that is present in `outputs` since if is the collection of O3 from the batch-size examples. For example, Figure 4.8 shows `next_state` and `outputs` for batch-size of three, window size of two, and rnn size of five; Note that with window size of two, every other

```
[[-0.077  0.022 -0.058 -0.229  0.145]
 [-0.167  0.062  0.192 -0.310 -0.156]
 [-0.069 -0.050  0.203  0.000 -0.092]]

[[[-0.073 -0.121 -0.094 -0.213 -0.031]
  [-0.077  0.022 -0.058 -0.229  0.145]]
 [[ 0.179  0.099 -0.042 -0.012  0.175]
  [-0.167  0.062  0.192 -0.310 -0.156]]
 [[ 0.103  0.050  0.160 -0.141 -0.027]
  [-0.069 -0.050  0.203  0.000 -0.092]]]
```

Figure 4.8: `next_state` and `outputs` of an RNN

line of the output is a next state line. It is somewhat convenient to have the next state packaged up for us separately, but the real reason for having this repetition will become clear in the next section.

The last piece of the language model is the loss. This is computed in the upper right-hand side of Figure 4.5. As we see there, the RNN output is first put through a layer of linear units to get the logits for softmax, and then we compute the cross entropy loss from the probabilities. As just discussed, the output of the RNN is a 3D tensor with shape [batch-size, window-size, rnn-size]. Up until now we have only passed 2D tensors, matrices, through our linear units, and we have done so with matrix multiplication — e.g., `tf.matmul(inpt, W)`.

The easiest way handle this is by changing the shape of the RNN output tensor to make it a matrix with the correct properties.

```
output = tf.reshape(output,[batchSz*windowSz, rnnSz])
logits = matmul(output,W)
```

Here $\mathbf{W}$ is the linear layer ($\mathbf{W_o}$) that takes the output of the RNN and turns it into the logits in Figure 4.5. We then can hand this to `tf.nn.sparse_softmax_cross_entropy_with_logits` that returns a column vector of loss values that reduce to a single value with `tf.reduce_mean`. This final value can be exponentiated to give us our perplexity.

Changing the shape of the RNN output was convenient here for pedagogical reasons (it allowed us to reuse `tf.matmul`) and computational ones (it put things into the shape required by `sparse_softmax`. In other situations the downstream computation might require the original shape, For

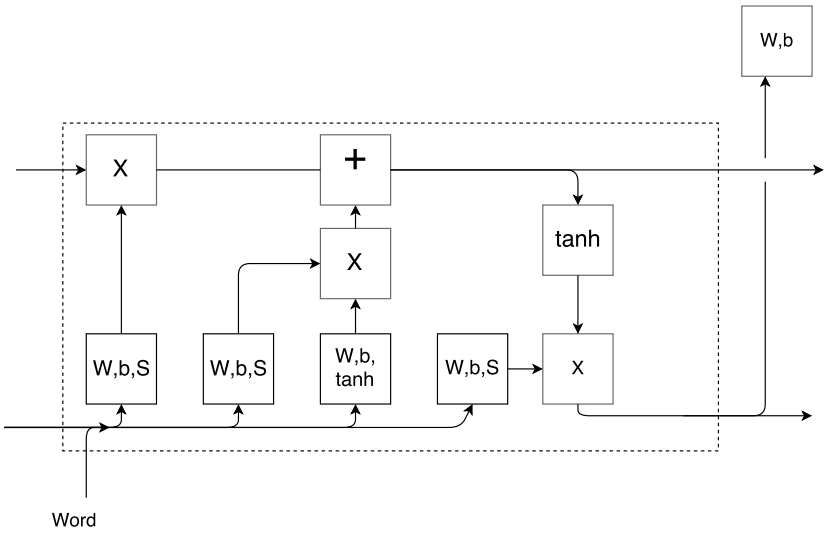

Figure 4.9: The architecture of LSTMs

this we can turn to one of many TF functions that handle multi dimensional tensors, Here the one we would use would be

```
tf.tensordot(outputs, W,  [[2], [0]])
```

This is a generalization of `tf.matmul` in the sense that `tf.matmul(A,B)` is the same as `tf.tensordot(A,B,[[1], [0]])`. Matrix multiplication can be thought of repeatedly taking the dot product of the entries in the rows of A with those of the columns of B. In tensor notation the columns of A are the first dimension of the tensor A, while rows are the second dimension. This is what the third `tf.tensordot` specifies. In particular `[[1], [0]]` says to do matrix multiplication — repeated dot products of the rows of A with the columns of B.

4.6 Long Short-Term Memory

A *long short-term memory* NN (*LSTM*) is a particular kind of RNN that almost always outperforms the standard RNN presented in the last section. The problem with RNN's is that while the goal is to remember things from far back, in practice they seem to quickly forget. In Figure 4.9 everything in the dotted box corresponds to a single RNN unit. Obviously LSTMs elaborate the architecture quite significantly. First note that we have shown

Figure 4.10: The tanh function

one copy of an LSTM in a back-prop though time diagram. So on the left we have information coming in from when we processed the previous word (using two tensors of information rather than one). From the bottom we have the next word coming in. On the right we have two tensors going out to inform the next time unit, and, as we did with plain RNN's, we have this information going "up" in the diagram to predict the next word, and the loss (upper right-hand side).

The goal is to improve the RNNs memory of past events by training it to remember the important stuff, and forget the rest. To this end LSTMs pass two versions of the past. The "official" selective memory is at the top, and a more local version at the bottom. The top memory time line is called the *cell state* and abbreviated "c". The lower line is called "h" (history?).

Figure 4.9 introduces several new connectives and activations functions. First, as we look along the memory line, it is modified at two locations before being passed on to the next time unit. They are labeled times, and plus. The idea is that memories are removed at the times unit, and added at the plus unit.

Why do we say this? Look now at the current word embedding coming in at the bottom left. It is goes through a layer of linear units followed by a sigmoid activation function as indicated by the $\mathbf{W}, \mathbf{b}, \mathbf{S}$ annotation. $\mathbf{W}, \mathbf{b}$ make up the linear unit, and $\mathbf{S}$ is the sigmoid function. We showed the sigmoid function in Figure 2.3, you might want to review it because a few of its specifics matter in the following discussion. The output of the sigmoid is then multiplied point-wise with the memory coming in top left. (By "pointwise we mean that, e.g., the $x[i, j]$'th element of one array is multipled (or added, etc) by the $y[i, j]$ the element of the other.) Given that sigmoids are bounded by zero and one, the result of the multiplication must be a reduction in the absolute value at each point of the main memory. This corresponds to "forgetting." Overall this configuration, sigmoid feeding a multiplicative layer, is a common pattern when we want "soft" gating.

Contrast this with the going's on at the additive unit that the memory next encounters Again the next word embedding has come in from bottom left, and this time it goes separately through two linear layers, one with sigmoid activation, one with a *tanh activation function* , shown in Figure 4.10 Tanh stands for *hyperbolic tangent*. It is important than as opposed to the sigmoid function, tanh can output both positive and negative values so it can express new material as opposed to just scale. The result of this is

added to the cell state at location B. After this the cell memory line spits. One copy goes out the right, and one copy goes through linear plus tanh to become the new history line on the bottom. This is to be concatenated with the next word embedding, and the process repeats. The point to be emphasized here is that the cell-memory line never directly goes though linear units. Things are de-emphasized (e.g., forgotten) at , things at the "X" unit, and added at "+", but that is it. Thus the logic of the LSTM mechanism.

As for the program, only one small change in needed to the TF version:

```
tf.contrib.rnn.BasicRNNCell(rnnSz)
```

becomes

```
tf.contrib.rnn.LSTMCell(rnnSz).
```

Note that this change affects the state that is passed from one time unit to the next. Previously, as shown in Figure 4.8 the states had shape [batchSz, rnnSz]. Now it is [2, batchSz, rnnSz], one [batchSz, rnnSz] tensor for the c-line, one for the h-line.

As for performance the LSTM version is much better, at the cost of taking longer to train. Take an RNN model such as that we developed in the last section, give it plenty of resources (word-embedding vectors of size 128, hidden size of 512) and we get a respectable perplexity of 120 or so. Make the single function call change and our implementation went down to 101.

4.7 Written Exercises

Exercise 4.1: A *linear gated unit* (LGU) is a variant of LSTMs. Referring back to Figure 4.9 we see that the latter has one hidden layer that controls what gets removed from the main memory line, and a second that controls what is added. In both cases the layers take the lower line of control as input, and produce a vector of numbers between zero and one that are multiplied with the memory line (forgetting) or added (remembering).tt LGUs differ in replacing these two layers by a single layer with the same input. The output is multiplied time the control line as before. However, it is also subtracted from one, multiplied by the control layer, and added to the memory line. In general LGUs work as well as LSTMs, andM having one fewer linear layer, are slightly faster. Modify Figure 4.9 so it represents the workings of a LGU.

Chapter 5

Sequence to Sequence Learning

Sequence to sequence learning (typically abbreviated *seq2seq*) is a deep learning technique for mapping a sequence of sym?bols to another sequence of symbols, when it is not possible (or at least we cannot see how) to perform the mapping on the basis of the individual symbols themselves. The prototypical application for seq2seq is *machine translation* (abbreviated *MT*) — having a computer translate between natural languages such as French and English.

It has been recognized since around 1990 that expressing this mapping in a program is quite difficult, and that a more indirect approach works much better. We give the computer an *aligned corpus* — many examples of sentence pairs that are mutual translations of each other and require the machine to figure out the mapping for itself. This is where deep learning comes in. Unfortunately the deep learning-techniques we have learned for natural-language tasks, e.g., LSTMs, by themselves are not sufficient for MT.

Critically, language-modeling , the task we have concentrated on in the last chapter, proceeds on a word by word basis. That is, we put in a word and we predict the next one. MT does not work like this. Consider some examples from the *Canadian Hansard*, the record of everything that has been said in the Canadian parliament which by law must be published in Canada's two official languages, French and English. The very first pair of sentences of a section I happen to have at hand is:

> edited hansard number 1
> hansard rèvisè numero 1

An early lesson for English speakers learning French (and presumably vice-versa) is that adjectives typically go before the noun they modify in English, and after in French. So here the the adjectives "edited" and "rèvisè" ' are not in the same positions in the translations. The point is that we cannot work our way left to right in the *source language* (the language we are translating from) spitting out one word at a time in the *target language*. In this case we could have input two words and output two, but the observed sequential mismatches can grow much larger. The following is from a few lines below.

> this being the day on which parliament was convoked by proclamation of his excellency ...
> parlement ayant ètè convoquè pour aujourd ' hui , par proclamation de son excellence ...

The word by word translation of the French would be "parliament is convoked for today, by proclamation of his excellency", and in particular "this being the day" is translated into "aujourd 'hui". (Indeed, generally the lengths of the sentences in the pair are not the same.) Thus the requirement for sequence to sequence learning, where a sequence is generally taken to be a complete sentence.

5.1 The Seq2Seq Paradigm

The diagram for a very simple seq2seq model is shown in Figure 5.1. It shows the process over time (as usual time running from left-to-right) of translating "hansard rèvisè numero 1" into "edited hansard number 1". The model consists of two RNNs. As opposed to LSTMs we are assuming an RNN model that passes a single memory line. We could use `BasicRNNCell` , however a better choice would be using a newer competitor to the LSTM, the *Gated Recurrent Network*, or *GRU* that only passes a single memory line between time units.

The model operates in two passes, each having its own GRU. This first pass is called the *encoding pass* and is represented in the lower half of Figure 5.1. The pass ends with the output from the last French word (in this case the number "1") being passed on to the second half of the process. The goal of this pass is to produce a vector that "summarizes" the sentence. This is sometimes referred to as a *sentence embedding* by analogy to word embeddings. .

The second half of seq2seq learning is called the *decoding pass*. As seen in the figure, it is a pass through the target language (here English) sentence.

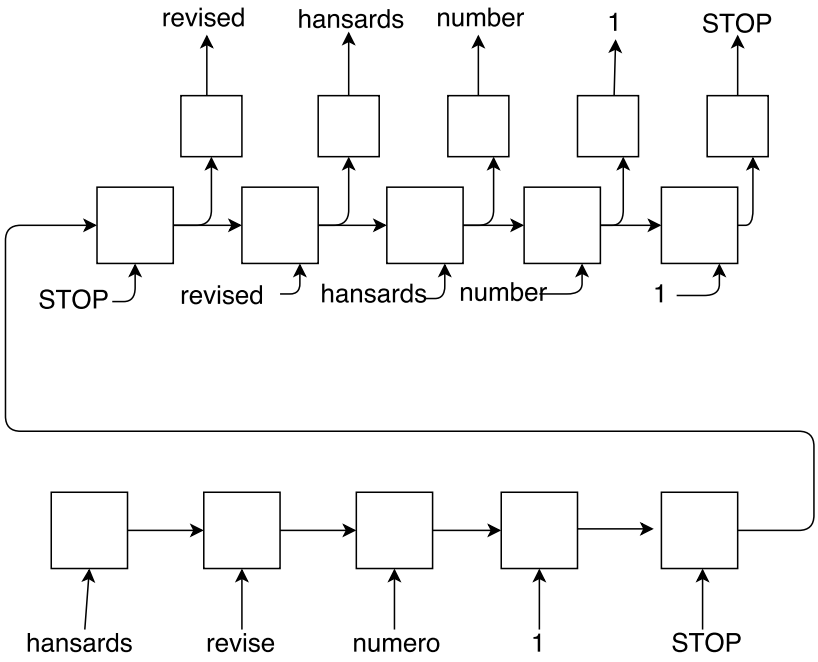

Figure 5.1: A illustration of a simple sequence to sequence learning model

(Perhaps it is time to make explicit that we are talking here about what happens during training, so we know the English sentence.) This time the goal is to predict the next English word after each word is input. The loss function is the usual cross-entropy loss. .

The terms *encode* and *decode* come from communication theory. A message needs to be encoded to form the signal that is sent, and then decoded back again from the signal that is received. If there is noise in the communication the signal received will not necessarily be identical to the one sent. Imagine that the original message was English, and the noise converted it to French. Then the process of translating it (back) to English is "decoding".

Note that the first input word for the decoding pass is the padding word STOP. It is also the last word to be output. Were we to use the model for real French-to-English MT the system does not have the English available. But it can assume that we start the processing with STOP. Then to generate each subsequent word we give the LSTM the previous predicted word. It stops processing when the LSTM predicts that the next "word" should be STOP again. Naturally we should also work this way when testing. Then we know the English, but we only use that information for evaluation. You should generally expect that the programs ability to predict the next French word is going to be much worse when we use the last predicted word rather than the real last word in the English corpus to predict the next English word since if it makes a bad choice that tends to make the next word a bad choice as well. In this chapter we ignore the complexity of real MT, by evaluating our program's ability to predict the correct next English word given the correct previous word.

Of course in real MT there is no single correct English translation for a particular French sentence, so just because our program does not predict the exact word used in the translation in our parallel corpus does not mean it is wrong. Objective MT evaluation is a topic of importance, but one we ignore here.

One last simplification before we look at writing an NN MT program. Figure 5.1 is laid out as a back-propagation though time diagram, so all of the RNN units in the bottom row are actually the same recurrent unit, but shown at successive times. Similarly for the units in the top row. As you may remember back-prop though time models have a window size hyper parameter. In MT we want to process an entire sentence in one fell swoop, but sentences come in all sizes. (In the Penn treebank they range from one word to over one hundred fifty). We simplify our program by only working on sentences where both the French and English are less than 12 words long, or 13 including a STOP word; We then make each sentence length thirteen

```
1  with tf.variable_scope("enc"):
2    F = tf.Variable(tf.random_normal((vfSz,embedSz),stddev=.1))
3    embs = tf.nn.embedding_lookup(F, encIn)
4    embs = tf.nn.dropout(embs, keepPrb)
5    cell = tf.contrib.rnn.GRUCell(rnnSz)
6    initState = cell.zero_state(bSz, tf.float32)
7    encOut, encState = tf.nn.dynamic_rnn(cell, embs,
8                                         initial_state=initState)
9
10 with tf.variable_scope("dec"):
11   E = tf.Variable(tf.random_normal((veSz,embedSz),stddev=.1))
12   embs = tf.nn.embedding_lookup(E, decIn)
13   embs = tf.nn.dropout(embs, keepPrb)
14   cell = tf.contrib.rnn.GRUCell(rnnSz)
15   decOut,_ = tf.nn.dynamic_rnn(cell, embs, initial_state=encState)
```

Figure 5.2: TF for two rnns in an MT model

by adding extra padding STOP's. Thus the program may assume that all sentences have the same length of 13 word.

5.2 Writing a Seq2Seq MT program

Let us start by reviewing the RNN models we in covered in Chapter 4, with a slight twist. So far in this book we have not paid much attention to good software engineering practices. Here however TF, for reasons to be explained, forces us to clean up our act. Since we are creating two, nearly identical rnn models we introduce the TF construct `variable_scope`. Figure 5.2 shows the TF code for the two rnns we need in our simple seq2se2 model.

We have divided the code into two pieces, the first creates the encoding rnn, and the second for the decoding. Each section is enclosed within a TF `variable_scope` command. This function takes one argument, a string to serve as the name for the scope. The purpose of `variable_scope` is to allow us to package together a group of commands in such a way as to avoid variable-name conflicts. For, example, both the top and bottom segments use the variable name `cell` in such a way that if we had not created two separate scopes they would have stepped on each other with very bad results.

Also, as we mentioned earlier, even if we had been careful and given each of our variables unique names, this code *still* would not have worked correctly. For reasons buried in the details of the TF code, when `dynamic_rnn`

creates the material to insert into the TF graph, it always uses the same name to point to it. Unless we put the two calls in separate scopes (or unless the code is set up not to mind that the two calls are, in fact, one and the same), we get an error message.

Now consider the code within each variable scope. For the encoder we first create space, `F`, for the French-word embeddings. We assume a place holder named `encIn` which accepts a tensor of French-word indices with shape batch size by window size. The lookup function then returns a three dimensional tensor of shape batch size, by window size, by embedding size (line 3), to which we apply dropout with the probability of keeping a connection set to `keepProb` (line 4). We then create the rnn cell, this time using the GRU variant of the LSTM. Line 7 then makes use of the cell to produce the outputs and the next state.

The second GRU is parallel to the first, except that the call to `dynamic_rnn` takes the state output of the encoder rnn, rather than a zero-valued initial state. This is the `state=encState` in line 15. Again consulting Figure 5.1 the decoder rnn's word by word output feeds into a linear layer. The figure does not show, but the reader should imagine, the layer's output (the logits) feeding into a loss computation. The code would look like this:

```
W = tf.Variable(tf.random_normal([rnnSz,veSz],stddev=.1))
b = tf.Variable(tf.random_normal([veSz,stddev=.1))
logits = tf.tensordot(decOut,W,axes=[[2],[0]])+b
loss = tf.contrib.seq2seq.sequence_loss(logits, ans,
                                        tf.ones([bSz, wSz]))
```

The only thing new here is the call to `seq2seq_loss`, a specialized version of cross entropy loss in cases when logits are three dimensional tensors. It takes three arguments, the first two standard — the logits and a 2D tensor of correct answer (batch size by window size). The third argument allows for a weighted sum — for situations where some errors should count more toward the total loss than others. In our case we want every mistake to count equally, so the third argument has all of the weights equal to one.

As we said earlier, the whole idea of the simplest seq2seq model in Figure 5.1 that the encoding pass creates a "summary" of the French sentence by passing the French though the GRU and then using the final GRU state output as our summary. There are, however, a lot of different ways to create such sentence summaries, and a significant body of research has been devoted to looking at these alternatives. Figure 5.3 shows a second model that for MT is slightly superior. The difference in implementation is small, rather than pass the encoder final state to the decoder as start state, we rather take the sum of all of the encoder states. Since we padded all of the

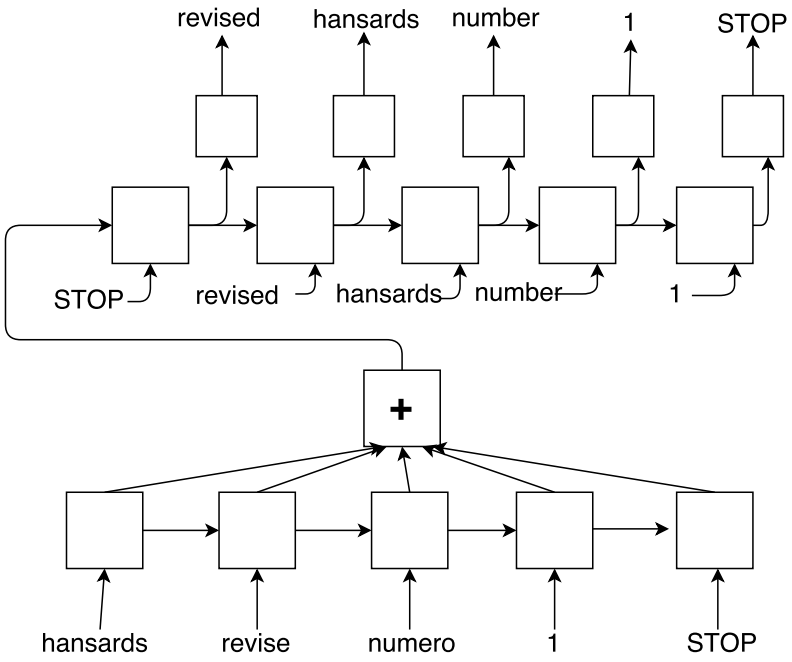

Figure 5.3: Seq2seq sentence summarization by addition

French and English to be of length 13, this means that we take all 13 states and sum them. The hope is that this sum is more informative than the one final vector, which, in fact, seems to be the case.

Actually, your author chose to take the mean of the state vectors as opposed to the sum. If you go back to chapter one and look at the forward pass computation, you will remember that taking mean rather than sum makes no difference to the final probabilities, as softmax will wash any multiplicative differences away, and taking the mean just corresponds by dividing by window size (13). Furthermore, the direction of the parameter gradient does not change either. What can and does change is the magnitude of the change we make in the parameter values. In the situation we currently face, taking the sum is roughly equivalent to multiplying the learning rate by 13. As a general practice it is better in such situations to keep parameter values near zero, and modify the learning rate directly.

5.3 Attention in Seq2seq

Figure 5.4 illustrates a small variation on the summing seq2seq mechanism of Figure 5.3 in which the summary plus the English word embedding is fed into the decoder cell at each window position, rather than just the English embedding. This might or might not work any better than just feeding the summary in at the beginning of the English decoding. Rather we present this variation as a way of introducing the idea of *attention* in seq2seq models.

Attention models, like Figure 5.4 feed a summary of the source sentence into the decoder GRU at each window position, but unlike Figure 5.4 the summaries are all different, depending on what is happening at that point in the translation. To take one of our standard examples, suppose the English so far is exactly tracking the French, and the previous word was "le" which we correctly translated as "the". We might next see a French noun. But before we make the next English a corresponding noun the program should give almost equally high weight to the word following the French noun to see if it is an adjective, in which case it should be output first.

We start with something more modest which we label *position-only attention*. While the point of using seq2seq technology is that translations are seldom word for word, it is nevertheless the case that words at the start, middle, or end of source sentence do translate into words at the start, middle or end of the target. This is particularly true for English and French, which are very similar as languages go. (It is not a co-incidence that the French took over England back in 1066.) But even languages which have

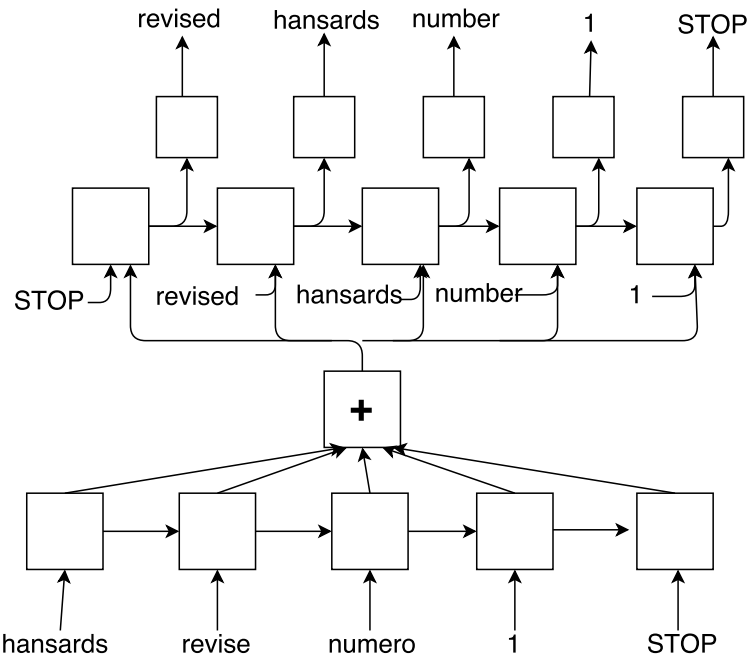

Figure 5.4: Seq2seq where the encoder summary is fed directly to each decoder window position

1/3	1/6	0	0	0	0
1/3	1/3	1/6	1/6	0	0
1/6	1/3	1/3	1/3	1/6	1/6
1/6	1/6	1/3	1/3	1/3	1/6
0	0	1/6	1/6	1/3	1/3
0	0	0	0	1/6	1/3

Figure 5.5: A possible weight matrix for weighting corresponding French/English positions more highly

no obvious commonalities have this property. The reason is the *given new distinction*. It seems to be the case in all languages that when saying something new about things that we have already been talking about (and this is usually what happens in coherent conversation or writing) we first mention the "given" — what we have been talking about, and only then mention the new material. So in a conversation about Jack we might say, "Jack ate a cookie", but if we were talking about a batch of cookies, "One of the cookies was eaten by Jack." So we are going to build an attention scheme where, say, the attention an English word at position i pays to the state of the French encoding at position j just depends on i and j. The closer i and j, the higher the attention.

From this point of view, the last model, shown in Figure 5.4 pays equal attention to all 13 states, since it multiplies each state by $\frac{1}{13}$before summing them together. So we give our program a weight matrix something like that in Figure 5.5, expect it should really be 13 by 13 rather than 6 by 6. Here we assign $W[i, j]$to be the weight given to the ith french state when used to predict the j'th English word. So the total weights for any English word is the column sum, which we have made 1. Looking at the first column we see that the first two states each count for one third of the weights (in our imagined window size of six), the next two count for one sixth, and the last two are ignored. For the moment we assume that the 13 by 13 version of Figure 5.5 is a tensor flow constant in out program.

Next, given the 13 by 13 weight matrix, how do we use it to vary the attention for a particular English output? Figure 5.6 shows the tensor flow, and sample numerical calculations for the situation where batch size is 2, window, size 3, and rnn size is 4. At the top we see an imaginary encoder outputm `encOut`. The batch size is two, and to make things simple we made the two batches identical. Within each batch we have the three vectors of lengh 4, each being the lengh four (rnn size) vector for the rnn output at

```
eo=  (((  1, 2, 3, 4),
       (  1, 1, 1, 1),
       ( -1, 0,-1, 0)),
      ((  1, 2, 3, 4),
       (  1, 1, 1, 1),
       ( -1, 0,-1, 0)))
encOut=tf.constant(eo, tf.float32)

wadj = (  ( .6, .333, .333 ),
          ( .2, .333, .333 ),
          ( .2, .333, .333 ) )
wAdjust = tf.constant(wadj,tf.float32)

encOT =tf.transpose(encOut,[0,2,1])
decIT = tf.tensordot(encOT,wAdjust,[[2],[0]])
decI=tf.transpose(decIT,[0,2,1])

sess= tf.Session()

print sess.run(encOT)
''' [[[ 1.  1. -1.]
      [ 2.  1.  0.]
      [ 3.  1. -1.]
      [ 4.  1.  0.]]
     ...] '''

print sess.run(decIT)
''' [[[ 0.60000002  0.333       0.333     ]
      [ 1.4000001   0.99900001  0.99900001]
      [ 1.79999995  0.99900001  0.99900001]
      [ 2.60000014  1.66499996  1.66499996]]
     ...] '''

print sess.run(decI)
''' [[[ 0.60000002  1.4000001   1.79999995  2.60000014]
      [ 0.333       0.99900001  0.99900001  1.66499996]
      [ 0.333       0.99900001  0.99900001  1.66499996]]
     ...] '''
```

Figure 5.6: Simplified attention calculations with bSz = 2, wSz = 3, and rnnSz = 4

that window position. So, for example, in batch 0, the first (0 based) state vector is (1, 1, 1, 1).

Next we have our made-up weight vector, `wAdjust` of dimensions 3 by 3 (the window size). It is French state position by English word position. So the first column says, in effect, that the first English word should be given a state vector that is 60% the first French state vector, and 20% each of the other two rnn state vectors. It is arranged so that the weights for each English word add up to 100%.

Next come the three TF commands that allow us to take in the unweighted encoder states, and produce the weighted versions for each English word decision. First we re-arrange the encoder output tensor, from [bSz, wSz, rnnSz] to [bSz, rnnSz, wSz]. This is done by the `tf.transpose` command, The transpose command takes two arguments, the tensor to be transposed, and a bracketed vector of integers specifying the transpose to be performed. Here we asked for [0,2,1] — keep the 0'th dimension in place, but the "2" says to make the dimension what was originally the second become dimension 1, and the "1" coming last makes the last dimension what use to be the first. We show the result of this transform where we executed `print sess.run(encOT)`,

We effected the transposition to make it easy to do the matrix multiplication in the next step (the `tensordot`). In fact, if we do not have the complication of batch size, we are multiplying tensors of shape [rnnSz, wSz] * [wSz,wSz], and we could use standard matrix multiplication (`matmul`). The extra dimension induced by batch size nixes this possibility and we fall back on

```
decIT = tf.tensordot(encOT,wAdjust,[[2],[0]]).
```

Lastly we reverse the transposition we did two steps ago to put the encoder output states back into their original form.

It is worth looking a bit and comparing the final result at the bottom of Figure 5.6 with our imaginary encoder output at the top of the figure. The column (.6, .2, .2) says to hand the first English Word a vector composed of 60% "state" zero, which is [1,2,3,4]. So we expect the resulting state to increase as we go from left to right, which (.6, 1.4, 1.8, 2.6) does. The second state of all ones does not have much effect (it adds .2 to each position). But the last state (0, -1, 0. -1) should cause the the result to have an up and down pattern, which it does, sort of. (They components of (.6, 1.4, 1.8, 2.6) all increase, but the first and third increases are larger than the second.)

Once we have the re-weighted encoder states to feed the decoder we concatenate each with the English word embedding that we were already

-6.3	**1.3**	.37	.13	.06	.04	.11	.10	.02
-.66	-..44	**.64**	.26	.16	.02	.03	.04	.06
-.38	-.47	-.04	**.63**	.18	.10	.07	.06	.12
-.30	-.44	-.35	-.15	**.48**	.24	.06	.13	0
-.02	-.16	-.35	-.37	-.23	.12	**.32**	.22	11
.05	-.11	-.11	-.35	-04	-.22	.05	**.26**	.24
.10	.02	-.04	-.23	-.32	-33	-.25	-.01	**.28**
0	.03	.01	-.18	-.21	-.26	-.30	-.1.1	-.17

Figure 5.7: Attention weights for the 8x9 top left corner of the 13x13 attention weight matrix

feeding into the decoder rnn. This completes our simple attention MT system. However, one thing is important to finish this example. That is, we can trivially have our program learn the attention weights by just making our 13 by 13 attention array a TF variable, rather than a constant. The idea is similar to what we did when in Chapter 3 we had the NN learning the convolution kernels. Figure 5.7 shows some of the weights learned in this fashion. The bold numbers are the highest numbers in their row. They show the expected correlation between positions between related words in the French and English versions a sentence.

5.4 More Advanced Models

5.4.1 Multi-Length Seq2Seq

In the last section we restricted our translation pairs to examples where both sentences are 12 words or less (13 including a STOP padding). In real MT such limitations would not be permitted. On the other hand, having a window since of, say 65, that would allow the translation of virtually all sentences that came our way would mean that a more normal sentence would end up padded with 40 or 50 STOPs. The solution that has been adopted in the NN MT community is to create models with multiple window lengths. We now show how this works.

Here are a few lines of TF pulled out of Figure5.2

```
5    cell = tf.contrib.rnn.GRUCell(rnnSz)
7    encOut, encState = tf.nn.dynamic_rnn(cell, embs,
8                                         initial_state=initState)
```

From our current perspective, it is striking that neither of the two TF commands primarily responsible for setting up the encoder rnn mentions the window size. GRU creation needs to know the rnn size, after all, it is in charge of allocating space for the GRUs variables. But window size does not enter into this activity. On the other hand, `dynamic_rnn` certainly *does* need to know the window size, since it is responsible for creating the pieces of the TF graphic that execute back-propagation through time. And it does get the information, in this case by way of the variable `embds`, which is of size [bSz, wSz, embedSz]. So suppose we decide to support two different window size combinations, the will, say, handle all sentences where, say, the French is 14 words or less, and the English 12 words or less. The second we make double this, 28 and 24. If either the French is larger than 28 or the English larger than 24 we through out the example. If either the French or English is larger than 14, or 12, respectively, but smaller than the 28, 24 limits, we put the pair into the larger group. We then create one GRU to use in both `dynamic_rnn`'s as follows

```
cell = tf.contrib.rnn.GRUCell(rnnSz)
encOutSmall, encStateS = tf.nn.dynamic_rnn(cell, smallerEmbs,
                                           initial_state=initState)
encOutLarge, encStateL= tf.nn.dynamic_rnn(cell, largerEmbs,
                                           initial_state=initState)
```

Note that while we have two (or potentially 5 or 6) `dynamnic_rnn`;s, depending on the range of sizes we want to accommodate, they all share the same GRU cell so they learn and share the same knowledge of French. In a similar fashion we would create one English GRU cell, etc.

5.4.2 Complicated Attention

5.4.3 Multiple-Language Machine Translation

5.5 Programming Exercise

Assignment still in progress. Official version to be posted on Piazza.

This chapter has concentrated on NN technology used in MT, so here we endeavor to built a translation program. Unfortunately in the current state of deep learning this is very difficult. While recent progress has been impressive, the programs that boast the good results require about a billion training examples, and days of training, if not longer. This does not make for a good student assignment.

Instead we are going to use about a million training examples — some of the Canadian Hansards text restricted to French/English training examples

where both sentences are twelve words or less (thirteen including the STOP padding). We also set our hyperparameters on the small side: embedding size of 30, rnn size of 64, and we only train for one epoch. We set the learning rate to .005.

As noted earlier, evaluating MT programs is difficult, short of going through and grading its translations by hand. We adopt a particularly simple-minded scheme. We go through the correct English translation until we hit the first STOP. A machine generated word is considered correct if the word in the same position in the Hansard English is identical. To repeat, we stop scoring after the first STOP. For example,

the	law	is	very	clear	.	STOP
the	*UNK*	is	a	clear	.	STOP

would count as 5 correct out of 7 words. At the end we divide the total number of correct words by the total of all words in the development set English sentences.

With this metric your authors implementation scored 59% correct on the test set after one epoch (65% after the second, and 67% after three), Whether this sounds good or bad depends on your prior expectations. Given our earlier comments about needing a billion training examples, perhaps your expectations were low, certainly ours were. However, an examination of the translations produced shows that even 59% is a misleadingly optimistic number. We ran the program printing out the first sentence in every 400th batch using a batch size of 32. The first two training examples of inputs correctly translated were:

```
Epoch 0 Batch 6401 Cor: 0.432627
* * *
* * *
* * *

Epoch 0 Batch 6801 Cor: 0.438996
le trs hon. jean chrétien
right hon. jean chrétien
right hon. jean chrétien
```

Here "* * *" is inserted between sessions of parliament by the editor. 14,410 lines of the file of 351,846 lines consist solely of this marking. By this point in the first epoch (it is half way through) the program has no down memorized the corresponding "English" (which is, of course, identical). In a similar vein, the name of the next speaker is always added to the Hansard before

what they say. Jean Chrétien was the prime minister of Canada at period for this volume of the Hansards, and he seems to have spoken 64 times. So the translation of this French sentence was also memorized. Indeed, one might ask if any of the correct translations are *not* memorized. The answer yes, but not that many. Here are the last six from the 22,000 example test set.

19154 the problem is very serious .
21191 hon. george s. baker :
21404 mr. bernard bigras (rosemont , bq) moved :
21437 mr. bernard bigras (rosemont , bq) moved :
21741 he is correct .
21744 we will support the bill .

These are from a run with double attention for corresponding words, a rnn size of 64 learning rate of .005, and one epoch. The accuracy metric described earlier was 68.6% for the test set. We printed out any test example that was completely correct, and did not correspond to any English training sentence.

It is interesting/useful to get some idea of how the state changes between words of a sentence. In particular, the first seq2seq model used the encoder final state to prime the English decoder. Since we just took the state at word 13, no matter the length of the the original French (maximum of 12 words) we are assuming that we have not lost much by taking the state after, say 8 STOPs if the original French were five words. To test this we looked at the 13 states produced by the encoder and for each state computed the cosine similarity[between successive states. The following is from a training sentence being processed in the third epoch.

English: that has already been dealt with .
Translation: it is a . a . .
French word indices:[18, 528, 65, 6476, 41, 0, 0, 0, 0, 0, 0, 0, 0]
State similarity: .078 .57 .77 .70 .90 1 1 1 1 1 1 1 1 1

You might first notice the terrible quality of the "translation." (two words correct out of 8, ".", and STOP). The state similarities however look reasonable. In particular, once we hit the end of the sentence at word five (in the French) all of the state similarities are 1.0 — so the state does not change at all due to the padding, as we hoped.

The least similar states are the first compared to the second. From there the similarity increases almost monotonically. Or in other words, as we progress through the sentence there is more past information worthy of

preserving, so more of the old state hangs around, making the next state similar to the current one.

Chapter 6

Deep Reinforcement Learning

Reinforcement learning (abbreviated *RL*) is the branch of machine learning concerned with learning how an *agent* should behave in an *environment* in order to maximize a *reward*. Naturally *deep* reinforcement learning restricts the learning method to deep learning.

Typically the environment is defined mathematically as a *Markov decision process* (*MDP*). MDPs consist of a set of states $s \in S$ that the agent can be in (e.g., locations on a map), a finite set of actions ($a \in A$), a function $T(s, a, s') = \Pr(S_{t+1} = s' \mid S_t = s, A = a)$ that takes the agent from one state to another), a reward function from a state, action, and subsequent state to the reals $R(s, a, s')$, and a *discount* $\gamma \in [0, 1]$ (to be explained momentarily.) In general actions are probabilistic, so T specifies a distribution over the possible resulting state from taking an action in a particular state. The models are called *Markov* decision processes because they make the *Markov assumption* — if we know the current state, the history (how we got to the current state) does not matter.

In MDPs time is discrete. At any time the agent is in some state, takes an action that leads to it a new state, and receives some reward, often zero. The goal is to maximizes its *discounted future reward* as defined by

$$\sum_{t=0}^{t=\infty} \gamma^t R(s_t, a_t, s_{t+1}), \tag{6.1}$$

If $\gamma < 1$ then this sum is finite. If gamma is missing (or equivalently equal to one) then the sum can grow to infinity, which complicates the math. A typical γ is .9. The quantity in Equation 6.1 is called *discounted* future reward

because the repeated multiplication by a quantity less than one causes the model to "discount" (value less highly) rewards in the future compared to rewards we get right now. This is reasonable because nobody lives forever.

Our goal is to *solve* the MDP — we also speak of finding an optimum *policy*. A policy is a function $\pi(s) = a$ that for every state s specifies the action a the agent should take. A policy is optimum, denoted $\pi^*(s)$, if the specified actions lead to the maximum expected discounted future reward. The *expected* here means, "on average". Since actions are not deterministic, the same action may end up giving quite different rewards. More formally we speak of the *expected value* of a numerically valued random variable as

$$E[X] = \sum_x \Pr(X = x) \cdot x \tag{6.2}$$

the sum over values of the possible outcomes times their probability. A standard example is the expected value of a roll of a fair six sided die is 3.5

$$E[R] = \frac{1}{6} * 1 + \frac{1}{6} * 2 + \frac{1}{6} * 3 + \frac{1}{6} * 4 + \frac{1}{6} * 5 + \frac{1}{6} * 6 = 3.5 \tag{6.3}$$

So this chapter is concerned with learning optimal MDP policies: first using so-called *tabular* methods, then using their deep learning counterparts.

6.1 Value Iteration

A basic question we need to answer before we talk about solving MDPs is whether we assume the agent "knows" the functions T and R or has to wander around the environment learning them as well as creating its policy. It simplifies things greatly if we know T and R so we start with that case.

Value iteration is about as simple as policy learning in MDP gets. (In fact, it is not really a learning algorithm at all in the sense that it does not need to get training examples, or interact with the environment.) The algorithm is given in Figure 6.1. V is a *value function* a vector of size $\mid s \mid$ where each entry $V(s)$ is the best expected discounted reward we can hope for when we start in state s. Q (simply called the Q *function*) is a table of size $|s|$ by $|a|$ in which we store our current estimate of the discounted reward when taking action a in state s. The value function V has a real number value for every state. The higher the number, the better it is to reach that state. Q is more fine-grain, it gives the values we can expect for each state action pair. If our values in V were correct then line 2(a)i would set $Q(s, a)$ correctly. It says that the value for $Q(s, a)$ consists of the

1. For all s set $V(s) = 0$
2. Repeat until convergence:
   (a) For all s:
      i. For all a, set $Q(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V(s'))$
      ii. $V(s) = max_a Q(s,a)$
3. return Q

Figure 6.1: The value iteration algorithm

0:S	1:F	2:F	3:F
4:F	5:H	6:F	7:H
8:F	9:F	10:F	11:H
12:H	13:F	14:F	15:G

S	staring location
F	frozen location
H	hole
G	goal location

Figure 6.2: The frozen-lake problem

immediate reward $R(s, a.s')$, plus the value for the state we end up in, as specified by V. Since actions are not deterministic we have to sum over all possible states. This gives us the expectation.

Once we have the correct Q we can determine the optimal policy π by always picking the action $a = \arg\max_{a'} Q(s,a')$. Here $\arg\max_x g(x)$ return the value of x for which $g(x)$ is maximum.

To make this concrete we consider a very simple MDP — the *frozen-lake problem*. The game is one of many that are part of the *Open AI Gym* — a group of computer games with uniform API's convenient for reinforcement learning experimentation. We have a 4 x 4 grid (the lake) shown in Figure 6.2. The goal of the game is to get from the start position (state 0 in the upper left) to the goal (lower right) without falling through a hole in the ice. We get a reward of one whenever we take an action and we end up in the goal state. All other state-action-state triples have zero reward. If we end up in a hole state (or the goal state) the game stops and if we play again

0	0	0	0
0	0	0	0
0	0	0	0
0	0	.6	0

0	0	0	0
0	0	0	0
0	0	.32	0
0	.32	.72	0

Figure 6.3: State values after the first and second iterations of value iteration

we go back to the start state. Otherwise we go left(l) down(d), right(r), or up(u), (the numbers zero to three, respectively) with some probability of "slipping" and not going in the intended direction. (In fact, the way the Open AI Gym game is programmed an action, e.g., right, takes us with equal probability to any of the immediately adjacent states except the exact opposite (e.g., left), so it is *very* slippery.)

Consider the case where we are in the start state. To keep the computation simple assume there are two possible actions, r and d. Let's say that $T(0, r, 1) = .8$, and $T(0, r, 4) = .2$ (when we are trying to go right we actually go right 80% of the time, and down 20%). Let $\gamma = .9$.) So line 2(a)i first computes

$$Q(0, r) = .8 \cdot (0 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) \tag{6.4}$$

We are summing over the two action possibilities. The first, with probability 8 is we end up in state 1. There we sum the reward, 0, and discounted (.9) future reward ($V(0)$ is 0. The second has a probability of .2, a reward of 0 and a future reward of 0. So the value of $Q(0, r)$ stays zero. The reader should see that $Q(0, d)$ is going to stay zero as well, and thus in line 2(a)ii $V(0)$ willl continue to be zero.

Because most all of the rewards and V values are 0, on the first iteration everything stays zero until we get to state 14. Suppose there are three possible moves out of state 14 and for all of them the probability of going in the correct direction is .6 and .2 for each of the other directions.

$$\begin{aligned} Q(14, r) &= .6 \cdot (1 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) = .6 \\ Q(14, l) &= .2 \cdot (0 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) = 0 \\ Q(14, u) &= .2 \cdot (1 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) + .2 \cdot (0 + .9 \cdot 0) = .2 \end{aligned}$$

And $V(14) = .6$.

The left half of Figure 6.3 shows the table of V values after the first iteration. Value iteration is one of several algorithms that work toward an

```
0  import gym
1  game = gym.make('FrozenLake-v0')
2  for i in range(1000):
3     st = game.reset()
4     for stps in range(99):
5          act=np.random.randint(0,4)
6          nst,rwd,dn,_=game.step(act)
7          # update T and R
9          if dn: break
```

Figure 6.4: Collecting statistics for an Open AI Gym game

optimum policy by keeping tables of the best estimates of function values. Thus the name *tabular methods*.

On iteration two, again most values stay zero, but this time states 10 and 13 are also going to get non-zero Q and V entries because from them we can go to state 14, and as just observed $V(14) = .6$ now. Another way to think about value iteration is that every change to V (and Q) incorporates the exact information about what is going to happen one move into the future (we get reward R) but then falls back to the initially inaccurate information already incorporated into these funcitons. Eventually the functions include more and more information about states we have not yet reached.

6.2 Q-learning

Value iteration assumes the learner has access to the complete details of the model environment. We now consider the opposite case — *model-free learning*. The agent can explore the environment by making a move, and it gets back information about the reward, and the next state, but it does not know the actual movement probabilities or reward function T, R

Assuming that our environment is a Markov decision process the most obvious way to plan in a model-free environment is to wander around the environment randomly, collect statistics on T, R, and then create a policy based upon the Q table as described in the last section. Figure 6.4 shows the highlights of a program for doing this. Line 1 creates the frozen lake game. To start a game (from the initial state) we call `reset()`. A single run of the frozen lake game ends when we either fall in a hole or reach the

goal state. So the outer loop (line 2) specifies that we are going to run the game 1000 times. The inner loop (line 4) says that for any one game we cut off the game at 99 steps. (In practice this never happens, we fall into a hole or reach the goal long before then.) Line 5 says that at each step we first randomly generate the next action. (There are 4 possible actions, left, down, right, and up: the numbers 0 to 3 respectively.). Line 6 is the critical step. The function `step(act)` takes one argument (the action to be taken) and returns 4 values. The first is the state in which the action has left the game (in FL an integer from 0 to 15) and the second is the value of the reward we receive (in FL typically 0, occasionally 1). The third state, named `dn` above, is a true-false indicator if the run of the game is terminated (i.e. did we fall into a hole, or reached the goal). The last argument is information about the true transition probabilities, which we ignore if we are doing model-free learning.

If you think about it, wandering at random in the game is a pretty bad way to collect our statistics. Mostly what happens is that we wander into a hole and go back to the start and keep collecting statistics about what happens at the states near the start state. A much better idea is to learn and wander at the same time, and allow the learning to influence where we go. If we do in fact, glean useful information in the process then as we progress we get further and further into the game, thus learning more about more different states. In this section we do this by choosing according to the probability ϵ to either (a) choose a move at random, or (with probability $(1-\epsilon)$) to base our decision on the knowledge we have gleaned so far. If ϵ is fixed this is called an *epsilon-greedy strategy*.

It is also common to have epsilon decrease over time (an *epsilon-decreasing strategy*. One simple way to do this is to have an associated hyper parameter E, and set $\epsilon = \frac{E}{i+E}$ where i is the number of times we have played the game. (So E is the number of games at which we go from mostly random to mostly learned.) As you might expect, how we choose whether to explore or base our choice on our current understanding of the game, can have a big effect on how fast we learn the game, and has it's own name — the *exploration-exploitation tradeoff*. (When we use game knowledge we are said to be *exploiting* the knowledge we have already picked up.)

Another popular way to combine exploration and exploitation is to always use the values given by the Q function but turn them into a probability distribution and then pick an action according to that distribution, rather than always picking the action with the highest value. (The latter is called the *greedy algorithm*.) So if we had three actions, and their Q values were $[4, 1, 1]$ we would pick the first 2/3's of the time etc.

Q-learning is one of the first and most popular algorithm for model-free learning combining exploration and exploitation. The basic idea is not to learn R and T but to learn the Q and V tables directly. From the viewpoint of Figure 6.4 we need to modify lines 5 (we no longer act completely randomly) and line 7 where we we modify Q, V not R and T. We have already explained what to do at line 5, so we turn to line 7

Our Q-learning update equations are

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R(s,a,n) + \gamma V(n)) \quad (6.5)$$

$$V(s) = \max_{a'} Q(s,a'), \quad (6.6)$$

where s is the state we were occupying, a, is the action we took, and a' is the state we now occupy having just taken a step in the game in line 6 of Figure 6.4.

The new value of $Q(s,a)$ is a mixture controlled by α of its old value and the new information — sort of a learning rate. Typically α is small. To make it clear why it is needed, it is useful to contrast these equations with lines 2(a)i and 2(a)ii from the value iteration algorithm in Figure 6.1. There, since the algorithm was given R and T we could sum over all possible outcomes of the action we took. In Q learning we cannot to this. All we have is the last outcome from taking a step. The new information is based upon just one move in our exploration of the environment. Suppose we are state 14 of Figure 6.2. but unbeknownst to us there is a very small probability(.0001) that if we move down from that state we get a "reward" of -10. The odds are this is not going to happen, but if it does it is going to throw things very badly out of wack. The moral is, the algorithm should not put to much emphasis on a single move. In value iteration we know both T and R and between the two of them the algorithm factors in both the possibility of a negative reward, and the low probability of it happening.

6.3 Basic Deep Q-Learning

With tabular Q learning under our belt we are now in position to understand *deep Q learning*. As with the tabular version we start with the schema of Figure 6.4. The big change this time is we represent the Q function not as a table but using a NN model.

It is common in courses such as this to characterize machine learning as a *function-approximation* problem — finding a function that closely matches some target functions., e.g., the target function might map from pixels to

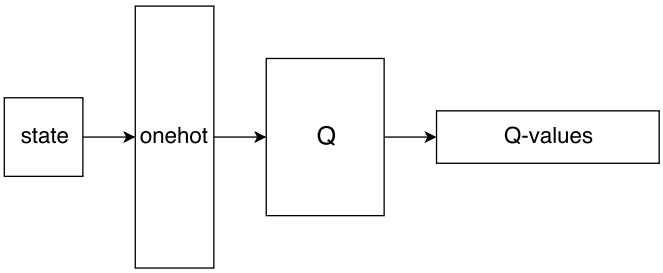

Figure 6.5: Frozen-lake deep Q-learning NN

one of ten integers, where the pixels are from an image of the corresponding digit. We are given the value of the function for some inputs and the goal is to create a function that is a closely matches its output for all. This author does not usually find this characterization illuminating, but in the case of deep Q learning it is completely apt — we are going to approximate our (unknown) Q function using NNs by wandering around the Markov decision process learning along the way.

We should emphasize that the change from tabular to deep learning models is *not* motivated by the frozen-lake example. It is exactly the sort of problem for which tabular Q learning is suited. Deep Q learning is needed when there are too many states to create a table for them.

One of the events in the re-emergence of NNs was the creation of single NN model that was able to apply deep Q learning to many Atari games. This program was created by *Deep Mind*, a startup that was purchase by Google. Deep Mind was able to get a single program to learn a bunch of different games by representing the games in terms of the pixels in the images that the games generate. Each pixel combination was a state. Off hand I do not remember the image size they used, but even if it were as small as the 28x28 images we used for Mnist, and each pixel was either on or off, that would be 2^{784} possible pixel value combinations — so in principle that number of states would be needed in the Q table. At any rate, way too many for a tabular scheme to cover. (I looked it up, an Atari game window is 210x160 RGB, and the Deep Mind program reduced this to 84x84 black and white) We return later to discuss cases more complicated than frozen-lake.

Replacing the Q table by a NN function boils down to this: to get a movement recommendation, rather than look in the Q table, we in effect call the Q table by feeding the state into a one layer NN as shown in Figure 6.5. The TF code for creating just the Q function model parameters is given

```
inptSt = tf.placeholder(dtype=tf.int32)
oneH=tf.one_hot(inptSt,16)
Q= tf.Variable(tf.random_uniform([16,4],0,0.01))
qVals= tf.matmul([oneH],Q)
outAct= tf.argmax(qVals,1)
```

Figure 6.6: TF model parameters for the Q learning function

in Figure 6.6. We feed in the current state (the scaler `inptSt`) which we turn into the one-hot vector `oneH` that is transformed by a single layer of linear units `Q`. `Q` has the shape 16x4 where 16 is the size of the one-hot vector of states, and 4 is the number of possible actions. The output `qVals` are the entries in Q(s), and `nAct`, the maximum of the Q table entries, is the policy recommendation.

Implicit in Figure 6.6 is the assumption we are only playing one game at a time, and thus when we feed in an input state (and get out a policy recommendation) there is only one of them. From our normal handling of NNs, this corresponds to a batch size of one. For example, the the input state is a scaler — the number of the state in which the actor finds itself. From this it follows that `oneH` is a vector. Then, since `matmul` expects two matrices, we call it with `[oneH]`. This in turn means that `qVals` is going to be a matrix of shape [1,4], i.e., it is only going to have the Q values for one action (up, down, etc.). Lastly then `outAct` is of shape [1], so the action recommendation is outAct[0]. (You should see why we go into this detail when we present the rest of the code for deep Q learning in Figure 6.7.)

As in tabular Q learning the algorithm either chooses an action at random (at the beginning of the learning process), or on the basis of the Q-table recommendation (near the end). In deep Q learning we get the Q table recommendation by feeding the current state s into the NN of Figure 6.5 and choosing an action u, d, r, or l, according to which of the four is the highest. Once we have the action, we call `step` to get the result, and then learn from it. Naturally, to do this in deep learning we need a loss function.

But now that we mention it, *what is the loss function of deep Q learning*? This is the key question, because, as has been evident all along, as we make moves, particularly in the early learning stages, we do not know if the moves we are making are good or bad! However, we do know the following: on average

$$R(s,a) + \gamma \max_{a'} Q(s',a') \tag{6.7}$$

(where as before, s' is the state we end up in after a in s) is a more accurate estimate of $Q(s, a)$ than the current value, because we are looking one move ahead. So we make the loss

$$(Q(s, a) - (R(s, a) + \gamma \max_{a'} Q(s', a')))^2, \tag{6.8}$$

the square of the difference between what just happened (when we took the step) and predicted values (from the Q table/function). This is called the *squared error loss* or *quadratic loss*. The difference between the Q calculated by the network (the first term) and the value we can compute by observing the actual reward for the next action plus the Q value one step in the future (the second term) is called the *temporal difference error*, or *TD(0)*. If we looked two steps into the future it woud be TD(1).

Figure 6.7 gives the rest of the TF code (following on from that in Figure 6.6). The first 5 lines builds the remainder of the TF graph. Now skim the rest of the code with emphasis on lines 7, 11, 13, 14, 19, and 25. They implement the basic AI Gym "wandering" . That is, they correspond to all of Figure 6.4. We create the game (line 7) and play 2000 individual games(line 11) each one starting with `game.reset()` (line 13). Each episode has a maximum of 99 moves (line 14). The actual move is made in line 19. The game is over as indicated by the flag we named `dn` (line 25). This leaves two gaps lines 15-17 (choose next action), and 20-22. Line 15 is the forward pass in which we give the NN the current state and get back a vector of length one (which the next line turns into the a scaler — the number of the action. We also always give the program a small probability of taking a random action (line 18). This ensures that we eventually explore all of the game space.

Lines 20-22 are concerned with computing the loss and performing the backward pass to update the model parameters. This is also the point of lines 1-5, which create the TF graph for loss computation and updating.

The performance of this program is not as good as that for tabular Q learning, but as we said, tabular methods are quite suitable for the frozen-lake MDP.

6.4 Policy Gradient Methods

We now turn to an Open AI Gym problem that cannot be handled by standard tabular methods, *cart pole*, and a new deep RL method, *policy gradients*.

A "cart pole", as shown in Figure 6.8 is a cart on a one dimensional

```
1  nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)
2  loss = tf.reduce_sum(tf.square(nextQ - qVals))
3  trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
4  updateMod = trainer.minimize(loss)
5  init = tf.global_variables_initializer()

6  gamma = .99
7  game=gym.make('FrozenLake-v0')
8  rTot=0
9  with tf.Session() as sess:
10  sess.run(init)
11  for i in range(2000):
12    e = 50.0/(i + 50)
13    s=game.reset()
14    for j in range(99):
15     nActs,nxtQ=sess.run([outAct,qVals],feed_dict={inptSt: s})
16     nAct=nActs[0]
17     if np.random.rand(1)<e: nAct= game.action_space.sample()
19     s1,rwd,dn,_ = game.step(nAct)
20     Q1 = sess.run(qVals,feed_dict={inptSt: s1})
21     nxtQ[0,nAct] = rwd + gamma*(np.max(Q1))
22     sess.run(updateMod,feed_dict={inptSt:s, nextQ:nxtQ})
23     rTot+=rwd
24     if dn: break
25     s = s1
26 print "Percent games succesful: ", rTot/2000
```

Figure 6.7: Remainder of deep Q learning code

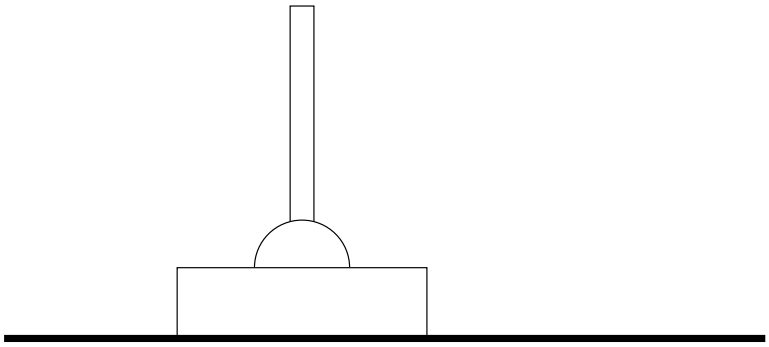

Figure 6.8: A cart pole

track. It has a pole attached to it by a sticky joint so that when the cart is propelled in one direction or another the top of the pole moves left or right according to the dictates of Newton's laws. A state consists of four values — the postion of the cart and the angle of the pole after the previous and current move. We give values at consecutive times to enable the program to figure out the direction of motion. There are two actions the player can make — propel the cart to the right, or propel left. Should the cart move too far to the right or left, or should the top of the pole move too far from perpendicular, `step` signals that the current game is over, and we need to `reset` to start a new one. We get one unit of reward for every move we make before failing. Naturally the goal is to keep the cart and pole well positioned for as long as possible. Since the state corresponds to a four-tuple of real numbers, the number of possible states is infinite so tabular methods are ruled out.

Up until now we have used our NN models to approximate the Q function for our MDP. In this section we show a method where the NN models the policy function directly. Again we are concerned with model-free learning, and again we adopt the paradigm of wandering around the game environment initially choosing actions mostly at random, but moving over to using the NN recommendation. As pretty much everywhere in this chapter, the burning problem is finding an appropriate loss function since we do not know the correct actions we ought to be taking.

In deep Q learning we made one move at a time, and we depend on the fact that, having made the move, received a reward, and ended up in a new state, our knowledge of the current local environment has improved. Our loss was the difference between what wes predicted (e.g., the Q function) on the basis of the old knowledge and what, in fact, happened.

Here we try something different. Suppose we play an entire iteration of a game without making any modification to our network — e.g., we make 20 moves (directions to the cart) before the pole tips over. This time we handle exploration/exploitation by choosing actions according to a probability distribution derived from the Q function, rather than taking the Q function maximum.

Under this scenario we can compute the discounted reward for the first state ($D_0(\mathbf{s}, \mathbf{a})$)) when it is followed by all of the states and actions we just tried out:

$$D_0(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} \gamma^t R(s_t, a_t, s_{t+1}) \tag{6.9}$$

Furthermore we can do the same for all of the other states we go through by just changing the limits on the summation. For the state at time t:

$$D_t(\mathbf{s}, \mathbf{a}) = \sum_{t'=t}^{n-1} \gamma^t R(s_{t'}, a_{t'}, s_{t'+1}). \tag{6.10}$$

That is, the discounted future reward for, e.g. the fourth state in the sequence of states we move thought (when taking action a) is D_4. Again note we have gained information here. For example, before we tried the first random sequence of moves we had no idea of what a possible reward was. Afterword we know that, say 20 is possible (and indeed reasonable for a random action sequence). Or then again, we now know if we fell over on move 20, then $Q(s_{19}, a_{19}) = 0$

A good loss function that captures these facts, and many others is:

$$L(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} D_t(\mathbf{s}, \mathbf{a})(-\log \Pr(a_t \mid s))) \tag{6.11}$$

To unpack this, first note that the right-most term is the cross entropy loss, and by itself has the effect of encouraging the net to respond with action a_t when it is in state s_t. Of course, by itself this is pretty useless, since, particularly at the beginning of learning we chose the actions randomly.

Next consider how the D_t values affect this. In particular, suppose a_0 was a bad reaction to s_0. For example, suppose the cart is centered, and the pole is learning to the right at the start, and we chose to go left, which causes the pole to lean still further to the right. The reader should see that everything else being equal, the value of D_0 is smaller in this case than it world have been if we had chosen to move right — the reason being that (everything else

```
state= tf.placeholder(shape=[None,4],dtype=tf.float32)
W =tf.Variable(tf.random_uniform([4,8],dtype=tf.float32))
hidden= tf.nn.relu(tf.matmul(state,W))
O= tf.Variable(tf.random_uniform([8,2],dtype=tf.float32))
output= tf.nn.softmax(tf.matmul(hidden,O))

rewards = tf.placeholder(shape=[None],dtype=tf.float32)
actions = tf.placeholder(shape=[None],dtype=tf.int32)
indicies = tf.range(0, tf.shape(output)[0]) * 2 + actions
actProbs = tf.gather(tf.reshape(output, [-1]), indicies)
loss = -tf.reduce_mean(tf.log(actProbs)*rewards)
optimizer = tf.train.AdamOptimizer(learning_rate=1e-2)
trainOp = optimizer.minimize(loss)
```

Figure 6.9: TF graph instructions for cart-pole policy gradient NN

equal) if the first move is good the pole and cart should remain in bounds longer (n is larger) and the D values are larger. So Equation 6.11 gives a higher loss to a bad a_0, than a good one, thus training the NN to prefer the good one.

Now let's put this loss function to use. Figure 6.9 gives TF code for creating the policy gradient NN using the loss function in Equation 6.11. and Figure 6.10 gives pseudo code for using the NN to learn a policy, and act in the game environment. (This algorithm/loss-function combination is known as *REINFORCE*.)

First consider the pseudo code. Note that the outermost loop (line 2) has us playing 3001 sessions of the game. The inner loop (line b) has us playing the game session until `step` tells us we are done (line D) or until we have moved 999 times. We choose a random action according to probabilities derived from our NN (lines i, ii) and then execute the action in the game. If the action leads to a final state we then update the model parameters.

We see from Figure 6.9 that `output` is computed by taking the current state values `state` and running them through a two layer NN with linear units `W` and `O` respectively, naturally separated by a `tf.relu` and then fed into a `softmax` to turn the logits into probabilities. As should be familiar from previous uses of multilayer NN's, the first layer has dimensions [input-size, hidden-size], and the second [hidden-size, output-size] where hidden-size is a hyper-parameter (we chose 8).

Because we have designed a new loss function here, and not used a

1. totRs=[ ]
2. for i in range(3001):
   (a) st=reset game
   (b) for j in range(999):
      i. actDist = sess.run(output, feed_dict=state:[st])
      ii. select act randomly according to actDist
      iii. st1,r,dn,_=game.step(act)
      iv. collect st,a,r in hist
      v. st=st1
      vi. if dn:
         A. disRs = [D_i(states, actions from hst) | i = 0 to j]
         B. create feed_dict with state=st, actions, from hist and rewards=disRs.
         C. sess.run(trainOp,feed_dict=feed_dict)
         D. add j to end of totRs
         E. break
      vii. if i%100=0: print out average of last 100 entires in totRs

Figure 6.10: Pseudo code for a policy gradient training NN for cart-pole

$$\begin{array}{cc} \Pr(\texttt{l} \mid s_1) & \Pr(\texttt{r} \mid s_1) \\ \Pr(\texttt{l} \mid s_2) & \Pr(\texttt{r} \mid s_2) \\ & \\ \Pr(\texttt{l} \mid s_n) & \Pr(\texttt{r} \mid s_n) \end{array} \quad \rightarrow \quad \begin{array}{c} Pr(a_1 \mid s_1) \\ Pr(a_2 \mid s_2) \\ \\ Pr(a_n \mid s_n) \end{array}$$

Figure 6.11: Extracting action probabilities from the tensor of all probabilities

standard one from the TF library, the loss computation had to be built up from more basic TF functions (second half of Figure 6.9). For example, in all of our previous NNs the forward and backward pass were inextricably linked insofar as there were no computations from outside of TF involved. Here we are getting the values of `reward` from the outside — `reward` is a placeholder, fed in according to lines A, B, and C from Figure 6.10. Similarly, `actions` is a placeholder.

Turning to the last three lines of Figure 6.9 things look more familiar. `loss` is just computing the quantities from Equation 6.11. For `optimizer` we have used the *Adam optimizer*. We could have used our familiar gradient descent optimizer by just substituting it in and doubling the learning rate, and would have achieved almost as good performance, but not quite. The Adam optimizer is slightly more complicated, and generally considered superior. It differs from gradient descent in several ways, the most fundamental being the use of *momentum*. As the name suggests an optimizer that uses momentum tends to keep moving a parameter value up/down if it has been moving up/down recently — more so than gradient descent would have.

This leaves the middle two lines of Figure 6.9, the ones setting `indicies` and `actProbs`. First, ignore how they work, and concentrate on what they need to do. What is needed is the transformation shown in Figure 6.11 On the left we see the output of a forward pass that computed the probability that each of the possible actions `r` and `l` is the best one to take. If this were Chapter 1, and we had full supervision, we would multiply this by a batch size tensor of one-hot vectors to get the probabilities of the actions we should take according to the supervision. This is, in in fact, what we show on the right of Figure 6.11.

To enact this transformation we depend on `gather`, which takes two arguments,

```
tf.gather(tensor, indicies)
```

and pulls out the elements of the tensor specified by the numeric indices and

puts them together in a new tensor. For example, if tensor is ((1,3), (4,6), (2,1), (3,3,)), and indices is (3,1,3), then the output is ((3,3), (4,6), 3,3)). In our case we turn turn action probability matrix on the left of Figure **??** into a vector (1D) of probabilities, and depend on the previous line to set `indicies` to the correct list so `tf.gather` collects the probabilities of just the actions specified by the vector `actions`. Showing that `indicies` is set correctly is left as an exercise for the reader (Exercise 6.4).

It is useful to go back and look more carefully at how Q-learning and REINFORCE relate to one-another. First they differ in how the collect environment information to inform the NN. Q-learning moves one step, and then looks to see if the NN prediction of the outcome is close to what actually occurred. Looking back at Equation 6.8, the Q-learning loss function, we see that if the prediction and outcome are the same, then there is nothing to update. REINFORCE on the other hand, we play an entire episode before changing any NN parameters, where an *episode* is a complete run of game, from the initial state until the game signals that it is done. Notice that we could have done something like Q learning but using the REINFORCE parameter modification schedule. This slows down the learning in so far as we make parameter changes much less often, but in compensation we make better changes because we are computing the *actual* discounted reward.

6.5 Actor-Critic Methods

Having just looked at the differences between Q learning, and REINFORCE, now concentrate on the similarities. In both the NN is either computing a policy, or in the case of Q learning, a function that can be trivially used to create a policy (for any state s always take the action a that maximizes $Q(s, a)$. Thus in both cases our NN is approximating a single function, that tells us how to act. We call such RL programs *actor* methods. In this section we consider programs that have two NN subcomponents, each with their own loss functions: one as before an actor program, and the second a *critic* program. As you might guess we call this type of RL, *actor-critic methods*. In particular in this section we are going to cover *advantage actor-critic* method, or *a2c*. It is a good choice for us because (a), it is a small change from REINFORCE, and (b) it really does work quite well.

The advantage of a state-action pair is the difference between the state-action Q value, and the state's value:

$$A(s, a) = Q(s, a) - V(s) \tag{6.12}$$

Intuitively we expect the advantage to be a negative number because $V(s)$ is computed by doing an $\arg\max_a$ over the possible actions. However, for good actions, A is large negative numbers go so A measures how good an action is in a particular state compared to the state overall.

Next we define loss for a2c after exploring a sequence of actions from a start state to the end of a game as follows:

$$L(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} A_t(\mathbf{s}, \mathbf{a})(-\log \Pr(a_t \mid s))) \tag{6.13}$$

This is very close to the REINFORCE loss of Equation 6.11 but we have replaced $D_t(s, a)$, the discounted reward, by $A_t(s, a)$ We remember that REINFORCE's loss is meant to encourage actions that lead to larger reward. Now we are encouraging actions that that are better than alternative actions from the same state. While this is somewhat reasonable, why should this be better than encouraging rewarding actions directly?

The answer has to do with the *variance* of $A(s, a)$. Formally, the variance of a function is the expectation of the square of the difference between between the functions value and its mean value. Intuitively this means that functions that vary a lot have high variance, and compared to Q, A should have much lower variance. Look at cart-pool, assuming the game gives us reasonable response in terms of moving left or right compared to how fast the pole moves, the difference between a move right and a move left will be small, and thus A is small in virtually all parts of the state space. Contrast this with Q. In cart-pole, Q with a random policy is in the teens or twenties, whereas with an even moderately good policy is in the multiple hundreds.

Add now a second fact, everything else equal, it is easier to approximate a function with low variance than one with high. A constant function, with zero variance, is the easiest of all. So if A is much easier to estimate, that could overcome the disadvantage incurred by maximizing A rather than Q directly. This seems to be the case. Of course, we don't know how to compute A at this point. So that is next on our agenda.

As you should remember, in REINFORCE we follow a path based upon our current policy to the end of a game, and use the discounted reward $D_t(s.a)$ from Equation 6.10 to estimate $Q(s, a)$. We now put this to double duty as our estimate of Q when computing A (Equation 6.12).

As for $V(s)$, we now introduce a new NN to compute it as in Figure 6.12 We have created a two layer fully-connected NN with the `V1` layer, multiplying the (`state` placeholder from Figure 6.9. The output of the second layer is compared to the actual reward using a quadratic loss function.

```
V1 =tf.Variable(tf.random_normal([4,8],dtype=tf.float32,stddev=.1))
v1Out= tf.nn.relu(tf.matmul(state,V1))
V2 =tf.Variable(tf.random_normal([8,1],dtype=tf.float32,stddev=.1))
vOut= tf.matmul(v1Out,V2)
vLoss=tf.reduce_mean(tf.square(rewards-vOut))
loss=loss + vLoss
```

Figure 6.12: TF code added to Figures 6.9 and 6.10 for a2c

. This is the critic loss. `vLoss` which is added to the loss computed by the actor portion of the NN for REINFORCE. Note also that it is now necessary, in Figure 6.10 line 2(b)viA to compute $D_t(s.a) - V(a)$ for the loss computation, rather than just $D_t(s, a)$.

6.6 Experience Replay

We mentioned early on that a major catalyst in the rebirth of NN's was Deep Mind's success with a program that could play multiple Atari games at an expert level. The NN technology used there is known as *DQN* (*Deep Q Network*). This particular RL scheme has been largely replaced by actor-critic methods, but the program also introduce several improvements that are orthogonal to use of actor, vs. actor-critic methods. One, in particular, is *experience replay.*

As you might expect, RL is a big component of the current push toward self-driving cars. One big problem in the application of RL to this domain is the acquisition of training data. Current RL requires a lot of it, and compared to computers the real world, and in particular the streets and highways, move very slowly. Actually, if you start timing Open AI Gym games, even computer simulations can be slow — a large fraction of the time spent in RL is in the execution of the game. If we could speed up the world, we could learn even faster, but we can't.

In experience replay we use the same training data multiple times. It is simplest to explain in the context of Open AI Gym. Going back to REINFORCE, as we played the game we used a variable `hist` to record the history of a play of the game — each state we occupied, the action we took, the state in which we ended up, and the reward received. We needed this at the end of the game play to compute the D_t's, but having computed them we threw the history away With experience replay for each time t we save $< s_t, a_t, s_{t+1}, D_t >$. With these numbers we can do another forward and

backward pass of our data and get more "juice" out of it. And there is a second benefit as well: we can play, and then replay, each time step in a random order. You may remember the iid assumption mentioned in Section 1.6 where we noted how RL could be particularly problematic as the training examples were correlated from the get-go. Taking random actions from several different game plays reduces the problem significantly.

Of course, we pay a price. Seeing an old training example is not as informative as a new example. Furthermore, the data can sort of get out of date. Suppose we having data from early in our training before we knew not to, say, move left when the pole is leaning far to the right. And suppose since then we have learned better. This means that we are uselessly relearning from the old data what to do in state s_{old}, when, in fact, the our current policy never allows us to arrive at that state. So instead we do something like this: keep a buffer of 50 game plays, corresponding to, say, 5000 state-action-state-reward four-tuples (we are averaging 100 moves before failing). We now pick e.g., 400 states at random to train from. We then replace the oldest game in the buffer with a new game played using the new policy based upon the up-to-date parameters.

6.7 Written Exercises

Exercise 6.1: Show that the V table show on right-hand side of Figure 6.3 gives the correct values (to two significant digits) for the state values after the second pass of value iteration.

Exercise 6.2: Equation 6.5 has a parameter α, but our TF implementation in Figures 6.6 and 6.7 seemingly make no mention of α. Explain where it it "hiding", and what value did we give it?

Exercise 6.3: Suppose in the training phase of the cart-pole REINFORCE algorithm it only took three actions (l,l,r) to reach termination, and $\Pr(l \mid s_1) = .2$, $\Pr(l \mid s_2) = .3$, $\Pr(l \mid s_1) = .9$. Show the values of `output`, `actions`, `indicies`, and `actProbs`.

Exercise 6.4: The TF function `tf.range` when given two arguments

```
tf.range(start, limit)
```

creates a vector of integers starting at `start`, and going up to (but not including) `limit`. Unless the named variable `delta` is set the integers differ by one. Thus its use in Figure 6.9 produces a list of integers in range 0

to batch-szie.. Explain how when combined with the next line of TF they accomplish the transformation in Figure 6.11.

Chapter 7

Unsupervised Neural-Network Models

This book has been following an unacknowledged path from *supervised learning* problems such as Mnist, to *weakly-supervised learning* problems such as re-enforcement learning. Our digit-recognition problem is said to be fully supervised because each training example comes along with the correct answer. In our re-enforcement learning examples the training examples are unlabeled. Instead we get a week form of labeling in so far as the rewards we get from Open AI Gym guide the learning process. In this chapter we consider *unsupervised learning* where we get no labels or other forms of supervision. We want to learn the structure of our data from only the data itself. In particular we look at *autoencoders*, and *generative adversarial networks*.

7.1 Basic Autoencoding

An auto-encoder is a neural net whose output is, if working correctly, almost identical to the input. To make this non-trivial, we place obstacles in its way. The most common method of doing so is *dimensionality reduction*. Figure 7.1 shows a simple two layer autoencoder. The input (say a 28x28 pixel Mnist image) is passed though a layer of linear units and is transformed into the intermediate vector which is significantly smaller than the original input — e.g., 256 compared to the original 784. This vector is then itself put through a second layer, and the goal is for the output of the second layer to be identical to the input of the first. The reasoning goes that to the degree we can reduce the dimensions of the middle layer compared to the input,

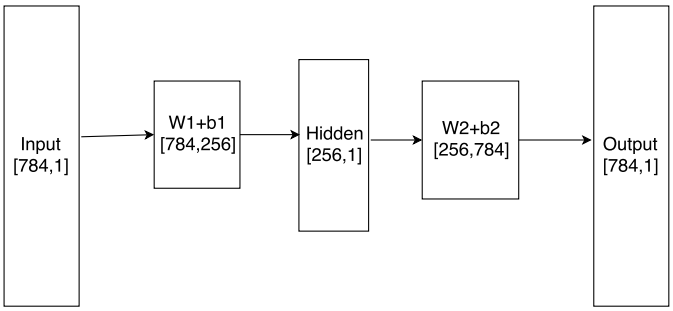

Figure 7.1: A simple two layer autoencoder

Figure 7.2: Using a stacked autoencoder for pretraining

the NN has encoded information in the middle layer about the structure of Mnist images. To put this at a great level of abstraction, we have:

$$\text{input} \rightarrow \text{encoder} \rightarrow \text{hidden} \rightarrow \text{deocder} \rightarrow \text{input}$$

where the *encoder* looks like a task-oriented NN, and the *decoder* looks like the encoder in reverse,.

We care about autoencoders for several reasons. One is simply theoretical, and perhaps psychological. Except in school, people get little in the way of supervision for learning, and by the time we enter school we have learned the most important of our skills: vision, spoken-language, motor tasks, and at it's most basic, planning. How can this be possible?

A more practical reason is the use of *pre-training*. Labeled training data is usually in short supply, and our models almost always work better the more parameters they have to manipulate. Pre-training is the technique of training some of the parameters first on a related task, then starting the main training cycles not with our now standard random initialization, but rather from values reached when trained on the related task. A paradigmatic example is shown in Figure 7.2. On the left we show a *stacked autoencoder*. A stacked autoencoder is one with several layers of units between the input and the middle layer where each layer has fewer units than the previous. After the middle layer we get the mirror image of layers until the final one where the reconstruction can be compared to the input. On the right of the figure we have a supervised classifier. It has three layers before the softmax. However, the first two layers are initialized using the weights obtained from autoencoding. Pretraining is now a standard technique for making less training data go further.

	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	3	14	15	1	0	0	0	0	0	0	0	0	0	0
7	187	221	205	151	74	11	1	0	0	2	8	23	55	69
8	237	249	251	250	249	239	221	225	197	214	216	236	237	228
9	92	194	232	219	217	225	245	251	251	249	241	237	249	250
10	1	8	7	17	31	49	100	126	106	45	37	81	242	251
11	0	0	0	0	1	9	13	7	2	0	1	43	239	247
12	0	0	0	0	0	2	2	0	0	0	2	151	247	215
13	0	0	0	0	0	0	0	0	0	1	61	246	248	57
14	0	0	0	0	0	0	0	0	1	32	207	253	185	10
15	0	0	0	0	0	0	0	0	9	176	251	237	31	1
16	0	0	0	0	0	0	0	0	47	237	252	67	2	0
17	0	0	0	0	1	0	1	9	171	249	237	9	0	0
18	0	0	2	7	1	1	5	100	243	251	138	1	0	0
19	0	0	0	2	1	0	19	217	253	222	19	0	0	0
20	0	0	0	0	0	2	107	246	241	44	1	0	0	0
21	0	0	0	0	1	48	220	247	168	4	0	0	0	0
22	0	0	0	0	18	196	251	233	42	0	0	0	0	0
23	0	0	0	1	98	249	250	140	2	0	0	0	0	0
24	0	0	0	14	237	254	242	40	0	0	0	0	0	0
25	0	0	5	116	252	254	205	8	0	0	0	0	0	0
26	0	0	16	158	253	249	56	4	2	0	1	0	0	0
27	0	0	0	0	2	1	0	0	0	0	0	0	0	0

Figure 7.3: Reconstruction of Mnist test example in Figure 1.1

A third reason for studying autoencoding is a variant called *variational autoencoders*. They are like standard ones, except they are designed to return random images in the style of those on which the autoencoder was trained.

Autoencoders typically use the sigmoid function (S) as the activation function between layers, so the autoencoder shown in Figure 7.1 can be expressed as:

$$\mathbf{h} = S(S(\mathbf{x}\mathbf{E}_1 + \mathbf{e}_1)\mathbf{E}_2 + \mathbf{e}_2) \tag{7.1}$$

$$\mathbf{o} = S(\mathbf{h}\mathbf{D}_1 + \mathbf{d}_1)\mathbf{D}_2 + \mathbf{d}_2 \tag{7.2}$$

$$L = \sum_{i=1}^{n}(x_i - o_i)^2 \tag{7.3}$$

Figure 7.3 shows the output of the autoencoder when the input is the image of a seven at the very start of Chapter 1. The reconstruction is remarkably good.

You might wonder why we used a sigmoid activation function rather than our almost standard `relu`. The reason is that up until now we have not been much concerned with the actual values that get passed through the network. At the end they are all passed through the softmax function and mostly all that remains are their relative values. An autoencoder, in contrast compares the absolute values of the input against those of the output. As you may

remember when we discussed the data normalization of our Mnist images (17), we divided the raw pixel values by 258 to normalize their values from 0, to 1. As we saw in Figure

The basic idea of a convolution autoencoder fits the model we laid out above, $\rightarrow$ encoder $\rightarrow$ decoder, etc.

7.2 Written Exercises

Exercise 7.1: Mnist digits (at least all we have seen) always have rows of zeros around the edges. What would this imply (everything else the same) about the values of the first-level weights?

Index

a2c, 107
activation functions, 33, 71
actor, 107
actor-critic methods, 107
Adam optimizer, 106
advantage actor-critic, 107
agent, 91
aligned corpus, 73
apply a filter, 40
argmax, 31, 93
Atari games, 98
attention, 80
autoencoders, 111
average per-word loss, 60
axis, in Tensorflow, 31
axon, 3

back propagation through time, 65, 76
backward pass, 13
`BasicRNNCell`, 68, 74
batch size, 16, 21
bias, 3
bigram model, 55
binary classification problem, 3
broadcasting, 22

Canadian Hansard, 73
cart pole, 100
`cast`, 31
cell body, 3
cell state, 71
chain rule, 13, 53
channels, 44
checkpointing, 35
classification problem, 2
communication theory, 76
`concat`, 61
`constant`, 25
`conv2D`, 44
convolutional filter, 40
convolutional kernel, 40
convolutional neural networks, 39
corpora, 4
cosine similarity, 57, 88
critic, 107
cross-correlation, 40
cross-entropy loss, 10, 76

data normalization, 17, 114
decode, 76
decoder, 112
decoding pass, 74
deep learning, 1
Deep Mind, 98
deep Q learning, 97
Deep Q Network , 109
deep reinforcement learning, 91
deep RL, 22
dendrites, 3
development corpus, 17
development set, 4
dimensionality reduction, 111
discount, 91

discounted future reward, 91
discretize, 1
dot product, 4
DQN, 109
dropout, 62
`dropout`, 63

early stopping, 62
`embedding_lookup`, 58
embedding layer, 56
encode, 76
encoder, 112
encoding pass, 74
environment, 91
epoch, 6
epsilon-decreasing strategy, 96
epsilon-greedy strategy, 96
`equal`, 31
expectation, 92
expected value, 92
experience replay, 109
exploration-exploitation tradeoff, 96

features, 2
feed forward neural networks, 8
floor function, 43, 67
forward pass, 13
frozen-lake problem, 93
fully connected neural nets, 39
function-approximation, 97

Gated Recurrent Network, 74
`gather`, 106
generative adversarial networks, 111
given new distinction, 82
`global_variable_initializer`, 28
Google, 25, 98
GPUs, 21
gradient descent, 8, 13, 15, 106
gradient operator, 20
`GradientDescentOptimizer`, 28
grammatical structure, 54
graphics processing units, 21
greedy algorithm, 96
GRU, 74
`gym.make`, 95

held-out set, 4
heuristics, 2
hidden-size, 35
hyper parameter, 5, 96
hyper-parameter, 65
hyperbolic tangent, 71

iid assumption, 22, 61
image feature, 41
independent identically distributed, 22
information theory, 12
instability, 22

L2 regularization, 62
`l2_loss`, 64
labels, 2
language modeling, 73
layers, 8
learning rate, 9, 16, 35, 80
LGU, 72
light intensity, 1
linear algebra, 19
linear gated unit, 72
linear units, 4
logits, 11, 21
long short-term memory, 70
loss functions, 8
LSTM, 70, 74

machine learning, 2
machine translation, 73
Markov assumption, 91
Markov decision process, 91, 98
`matmul`, 29, 30, 99

matrix, 19
maximum likelihood estimate, 54
MDP, 91
Mnist, 1, 98, 111
model-free learning, 95, 102
momentum, in optimization, 106
MT, 73
multi-class decision problems, 7

National Institute of Standards, 1
neural nets, 1
neuron, 3
Newton's laws, 102
NIST, 1
NN, 8
Numpy, 21

one dimensional convolution, 44
one-hot vectors, 30, 59
`onehot`, 99
Open AI Gym, 93, 95, 111

padding, 42
parameters, 3
Penn Tree-bank Corpus, 54
perceptron, 3
perceptron algorithm, 4, 17
perplexity, 60, 69
pixel values, 1, 17
`placeholder`, 26
pointwise operation, 71
policy, 92
policy gradients, 100
position-only attention, 80
pre-training, 112
probability distribution, 10, 53
PTB, 54
Python, 25

Q function, 92
Q-learning, 97

quadratic loss, 100, 108

`randint`, 95
random variables, 53
`random_normal`, 28
`range`, 110
RBG color, 40, 44
re-enforcement learning, 111
rectified linear unit, 33
recurrent neural network, 64
`reduce_mean`, 28
regularization, 62
REINFORCE, 104
reinforcement learning, 91
relu, 33
`reset` in AI Gym, 95
`reshape`, 69
reshape, 45
reshape, in numpy, 45
reward, 91
RL, 91
RNN, 64
rnn size, 68, 72

same padding, 42
`save` method in TF, 36
`Saver`, 35
saver objects, 35
self-driving cars, 109
sentence embedding, 74
sentence padding, 55, 66
seq2seq, 73
`seq2seq_loss`, 78
sequence to sequence learning, 73
`Session`, 25
sigmoid function, 33, 71, 113
soft functions, 10, 71
softmax, 10, 80, 113
solving MDPs, 92
source language, 74

sparse matrix, 59
`sparse_softmax_cross_entropy`, 59, 69
squared error loss, 100
stacked autoencoder, 112
`step` in AI Gym, 96
stochastic gradient descent, 13, 16
STOP padding, 88
stride, 41
sum of squares loss, 24
supervised learning, 111

tabular MDP methods, 92
tabular methods, 95
tanh activation function, 71
target language, 74
TD(0), 100
temporal difference error, 100
`tensordot`, 70
Tensorflow, 21, 25
tensors, 27
test set, 4
TF, 25
three dimensional convolution, 44
training examples, 4
training set, 4
`transpose`, 84
transpose, of a matrix, 20
tree bank, 54
tri-gram model, 60
`run`, 25

underscore, in Python, 32
unknown words, 54
unsupervised learning, 111

valid padding, 42
validation set, 4
value function, `V`, 92
value iteration, 92
vanishing gradient, 34
`Variable`, 27
variable initialization, 28
`variable_scope`, 77
variance, 108
variational autoencoders, 113
vocabulary, of English, 54

weakly-supervised learning, 111
weight initialization, 17
weights, 3
Wikipedia, 14
window size, 65, 76
word embedding, 74
word-embedding size, 72

zero-one loss, 9