Introduction to Deep Learning

Eugene Charniak

Contents

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 data* (pronounced "em-nist") In this data each image is at 28 by 28 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 $[8, 8]$ is dark. Given the shape of a '7' this is quite reasonable. Similarly sevens will often have a light patch in the middle – i.e. pixel $[13, 13]$ has a

	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

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 positive integers.

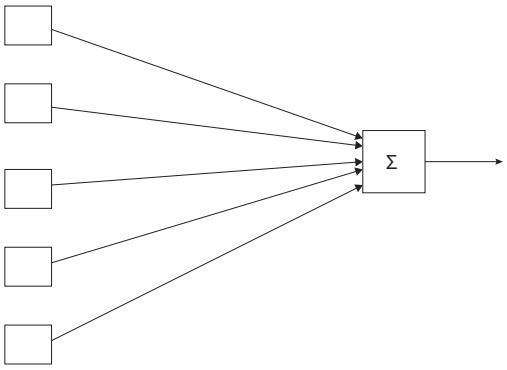

Figure 1.2: Schematic diagram of a perceptron

Figure 1.3: A typical neuron

1.1 Perceptrons

We start, however, with a simpler mechanism for 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 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 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 features 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 $\mathbf{x}^k = [x_1^k \ldots x_l^k]$ 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 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

1. set b and all of the $\mathbf{w}$'s to 0.
2. for N iterations, or until he 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

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 will be 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 will 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 *meta-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 algoriithm will increase the weights, and next time $f(x^k)$ will return a larger value —- it will be "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.

Lets 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 will be different. Good practice is to randomize the order in which the training data is presented each epoch. This way we do not tune the model to an accidental feature of the data, the input order. More seriously a fixed order may actually decrease our performance. However, for students just coming to this material for the first time, we can give ourselves some latitude here and omit this niceity.

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

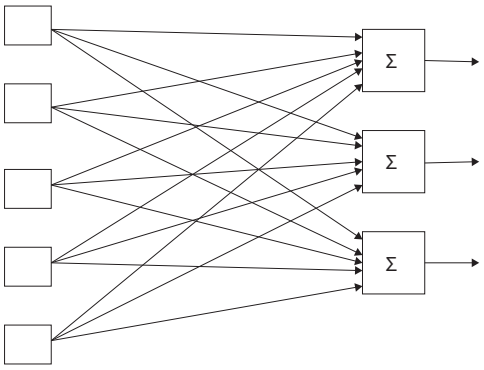

Figure 1.5: Multiple perceptrons for identification of multiple classes

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 1.6. where we show 3 perceptrons for identifying an image as being of one of three classes of objects.

While Figure 1.6 looks very interconnected, in actuality this is simply three separate perceptrons which share the same inputs. Each perceptron is 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, the zero to fourth perceptrons will be expected to return zero (and their weights changed if they do not), the fifth will be trained to return one, and the sixth through ninth to also return zero.

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 1.6 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

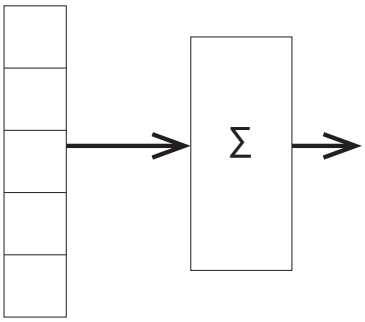

Figure 1.6: NN showing layers

layer. Figure 1.5 is a revised version of Figure 1.6 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 decent*

Before we can talk about gradient decent,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 loss*. Zero-one loss has the advantage of being pretty obvious, so obvious that we never bothered to justify their use. However, they have disadvantages. In particular they do not work well with gradient decent 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

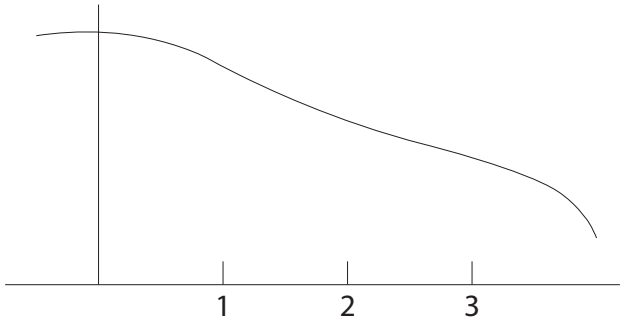

Figure 1.7: Loss as a function of ϕ_1

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 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 will 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 $f(x)$ will change its value, and we will 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 will 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.

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

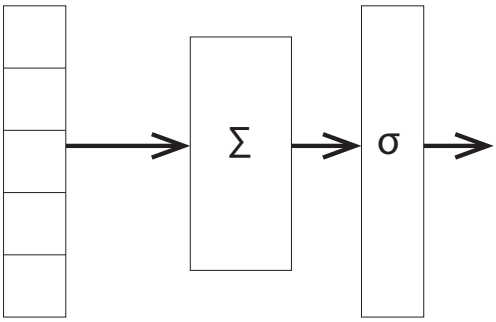

Figure 1.8: A simple network with a softmax layer

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.

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 there is a property of probability distributions called their *cross-entopy* and our function X is computing an estimate of this number. However we will not have need to go deeper into information theory in this book, so we leave it with this shallow explanation.

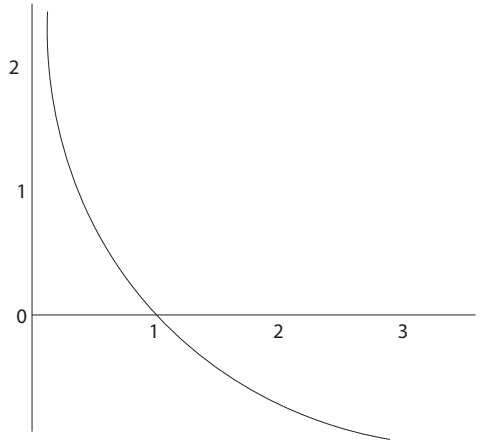

Figure 1.9: Graph of -ln(x)

1.3 Derivatives and Stochastic Gradient Decent

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. This method is called *gradient decent* because we are looking at the slope of the loss function (its *gradient*), and then having the system lower its loss (desend) by following the gradient.

Let's 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:

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

This uses the chain rule to say the first part of the above comment — 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

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

where $w_{j,i}$ is the i'th weight of the j'th linear unit. Since the only thing in $b_j + \sum_i x_i w_{i,i}$ 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:

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

where p_i is the probability assigned to class i by the nettwork. 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 will 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 will 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

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

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 weiights is:

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

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

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

(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

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

We have now derived how the parameters of our model should be adjusted in light of a single training example. The *gradient decent* 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.

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 decent, but rather *stochastic gradient decent* 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 will 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. 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 $w_{j,i}$ 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

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, or whatever the input values to the network may be, 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. One place we can see how this makes sense 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 will 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 (which I will often refer to as *jitter*) This means all of the logit values are very close to zero, so all of the probabilities will be very close to $\frac{1}{10}$. (See the discussion on page 14) 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 willl be 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 avery, 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 will be 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 will be 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 matricies 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 matricies $\mathbf{X} = \mathbf{YZ}$ 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 also express the backward pass more compactly. First, we define

$$\nabla_{\mathbf{l}} X(\Phi) = \left(\frac{\partial X(\Phi)}{\partial l_1} \dots \frac{\partial X(\Phi)}{\partial l_m} \right) \tag{1.24}$$

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 $\mathbf{x}$ It is called the *gradient operator*. 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 transform of a matrix — making the rows of the matrix into columns, and vice versa.

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ & & \dots & \\ x_{l,1} & x_{l,2} & \dots & x_{l,m} \end{pmatrix}^T = \begin{pmatrix} x_{1,1} & x_{2,1} & \dots & x_{l,1} \\ x_{1,2} & x_{2,2} & \dots & x_{l,2} \\ & & \dots & \\ x_{1,m} & x_{2,m} & \dots & x_{l,m} \end{pmatrix} \tag{1.25}$$

With these we can rewrite Equation 1.20 as

$$\mathbf{\Delta W} = -\mathcal{L}\mathbf{X}^T \nabla_{\mathbf{l}} X(\Phi) \tag{1.26}$$

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 dimensions. 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 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 will have n (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.

Chapter 2

Tensorflow

2.1 Tensorflow Preliminaries

Tensorflow is an open-source programming language developed by Google that is specifically designed to make programming 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 a Python program, that is because it is. In fact Tensorflow (hense 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". As you might expect, the above program prints out this string.

It is instructive to contrast this behavior with what would have happened if we replaced the last line with `print(x)` This will print 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. Below we write down a simple Tensorflow program for Mnist digit recognition. The primary TF program will take an image and run the forward NN pass to get the networks solution to what digit we are looking at. Also, during the training phase it will run the backward pass and modify the programs parameters. To hand the program the image the image we would 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 a 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 initializing 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.initialize_all_variables())
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 creates a piece of the TF graph that will create 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 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):

```
[-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.

Initializing the variables is a separate step because there are other ways this can occur — most notably, by saving values from a previous run of the program and reading them in.

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 decent will modify 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.

```
 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.initialize_all_variables())
20 #-----------------------------------------------
21 for i in range(1000):
22   imgs, ans = mnist.train.next_batch(batchSz)
23   sess.run(train, feed_dict={img: imgs, ans: ans})

25 sumAcc=0
26 for i in range(1000):
27    imgs, ans= 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

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 variation of what we just saw in our discussion of TF variables. Next, we make placeholders for the data we will be feeding 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 24) 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 decent. 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`). Simlarly, 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 compuations. 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 recommended that when looking at code like this it is a good idea to 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 of 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 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 decent 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.

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 Mnsit 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 `GradentDecentOptimizer` 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 for the `accuracy` computation itself, it does what one would expect, count the number of correct answers and divides by the number of images

processed. `tf.argmax(prbs,1)` finds returns an array of maximum probabilities for each of the images, and the `tf.equal` sees if they correspond to the correct answer for the image. `tf.equal` returns an array of boolean values, 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.

2.3 Multi-layered NNs

The program we have designed, first generally then 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 In all cases activation are applied piecewise to the

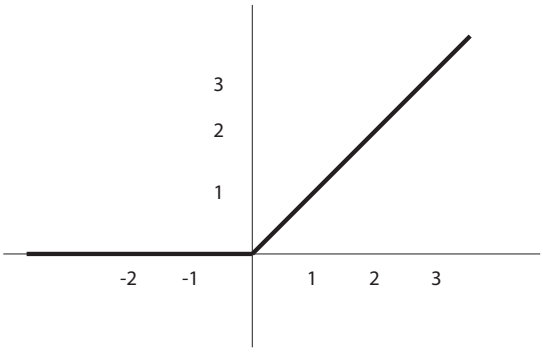

Figure 2.2: Behavior of relu

Figure 2.3: The sigmoid function

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

Let's do this in TF. In Figure 2.4 we replace the definitions of **W** and **b** in lines 5 an 6 with lines 1 through 4 above, and replace the computation of `prbs` in line 11 with lines 5 though 7 above. This turns our code into a multi-layered NN. 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

```
1 U = tf.variable(tf.random_normal([784,784], std_dev=.1))
2 bU = tf.variable(tf.random_normal([784], std_dev=.1)
3 V = tf.variable(tf.random_normal([784,10], std\_dev=.1))
4 bV = tf.variable(tf.random_normal([10], std_dev=.1)
5 l1Output = matmul(img,U)+bu
6 l1Output=tf.relu(l1Output)
7 prbs=tf.softmax(matmul((l1Ouput,V)+bv)
```

Figure 2.4: TF replacement code for multi-level digit recognition

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!

Chapter 3

Word Embeddings and Language Models

3.1 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 change you have never seen this particular sentence, and unless you read this book again you will never see it a second time. Whatever it's probability is, 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 iet $(E_{1,n}) = (E_1 \dots E_n)$ 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 us

$$P(\text{We live in a small world}) = P(\text{We})P(\text{live}|\text{We})P(\text{in}|\text{We live})\dots \quad (3.1)$$

More generally

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{1,j-1} = e_{1,j-1}) \tag{3.2}$$

Before we go on, we should go back a bit to where we said "breaking the strings into a sequence of words." This process 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 what space definition the word that appears in the above sentence is "1066." , that is "1066" with a period after it. So we are going to also going to 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 by a special word "*UNK*". So this sentence would now appear in our corpus as "The population of Providence is *UNK* ."

With that out of the way let us return to Equation 3.2. If we had a very large amount of English text we might be able to estimate the probabilities on its right-hand side (at least for small n) 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(in|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:

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{j-1} = e_{j-1}) \tag{3.3}$$

This is called a *bigram model* — where bigram means "to word". It is called this because each probability is only depending on a sequence of two words.

Now we want to use deep learning to estimate these bigram probabilities. That is, we give the deep network a word, w_i and the output is a probability distribution over possible next words w_{i+1}. To do this 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. Depending on the application one typically sees values from 20, to 200, and sometimes larger. 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 word embeddings so that if, say, "the" has index 5, the 5'th row of $\mathbf{E}$ 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 3.1. The small square on the left is the input to the network — the integer index of the current word, w_i. On the right are the probabilities assigned to possible next words w_{i+1}, and the cross-entropy loss function is $-\ln P(w_c)$ 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 looking up the w_i'th row in $\mathbf{E}$. From that point on all NN operations are on the word embedding.

A critical point is that $\mathbf{E}$ is a parameter of the model. That is, initially the numbers in $\mathbf{E}$ are random with mean zero and small standard deviation, and their values are modified according to stochastic gradient decent. 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 the same direction. in 30-dimensional space, and neither is very close to, say, "computer' (which will be closer to "machine').

With a bit more thought, however, perhaps this is not so amazing. As

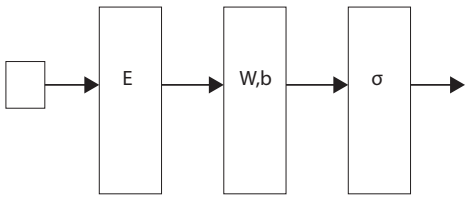

Figure 3.1: A feed-forward net for language modeling

already stated the loss function is the cross entropy loss. Initially all the logit values will be about equal since all of the model parameters are about equal (and new zero). But there is some random jitter. 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. If we now see "recalls that' moving the embedding for "recalls' to look more like says will similarly make "that" have higher probability, so that is what the model is going to do.

Figure 3.2 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

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{(\sqrt{(}\sum_{i=1}^{i=n} x_i^2)(\sqrt{(}\sum_{i=1}^{i=n} y_i^2)} \tag{3.4}$$

In Figure 3.2 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

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

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 the 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, mostly they 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.

3.2 Building Language Models

Now let us build a TF program for computing bigram probabilities. It is very similar to that 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 input an image the NN takes a word index i where $0 \leq i < |V|$ and the firrst thing is to find $\mathbf{E}[i]$ 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],
                                          std_dev = 0.1))
embed = tf.nn.embedding_lookup(E, inpt)
```

We assume that the unshown code for reading the words in and replacing the characters by unique word indices packages up `batchSz` of them in a column vector. `input` poits to this vector. (The correct answer for each word (the next word of the text) is a similar column vector, `answr`. Next we created the embedding lookup array `E`. The function `tf.nn.embedding_lookup`creates the necessary TF code and puts in into the computation graph. Future manipulations (e.g., `tf.mat_mul` 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 will use a built-in TF function to compute the cross-entropy loss:

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

The TF function `tf.nn.sparse_softmax_cross_entropy_with_logits` takes as its first argument a]tt batchSz of logit values (i.e., a `batchSz` by `vocabSz` array of logits) that it feeds into `softmax` to get a column vector of probabilities `batchSz` by `vocabSz` vector of probabilities. so an element of the output array $e_{i,j}$ 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 tt 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 is going to take that column vector and sum it to get the total loss for that batch of examples.

At this point we do a few epochs over our training examples, and get embeddings that demonstrate word similarities like those in 3.2. Also, if we want to evaluate the language model we can print out the total loss on the training set after every epoch. What you should see that for the first few epochs it will decrease, though the exact numbers that get spit out are rather hard to interpret. So researchers in language modeling print a related number called the *perplexity*. Up until now we have been dealing with digit images, and the natural evaluation metric for for our models is

how accurate they are. But nobody cars about actually predicting next words (we are rarely be able to do so) but rather we want a measure of how well we are doing overall in prefering reasonable sequences of words to unreasonable ones. Perplexity does this for us.

The perplexity of a corpus d (we typically measure it on our development corpus) with $|d|$ words, and total cross-entropy x Iis e raised to the negative per-word cross-entropy.

$$f(d) = e^{-\frac{x_d}{|d|}} \tag{3.5}$$

In Chapter 1 in our debugging discussion (page 21) we suggested computing the average per example loss. When the loss is the cross-entropy loss, as it was then, then we are, in fact, computing the average cross-entropy loss.

More to come.