Neural Networks

1 What a Neural Network Computes

To begin with, we will discuss fully connected feed-forward neural networks, also known as multilayer perceptrons. A feedforward neural network consists of layers of computational units. This network transforms a vector of input features into a vector of outputs. The input layer consists of a set of units that each represent one of the input features. Each subsequent layer receives input from the previous layer; each unit receiving input from all of the units in the previous layer. The output layer represents a vector of the outputs.

We can divide the computation of a unit into a linear and nonlinear component. First, the unit takes a linear combination of all its inputs, along with a (scalar) bias term. Then it applies a nonlinear function to the result of this linear combination. For simplicity, we will suppose that the nonlinear function is the same throughout the network. In practice, the nonlinear function is typically the same for all units in the same layer, but may be different for different layers.

We can think of this graphically, with each computational unit a node in a graph, and edges denoting the input to each node from the previous layer. We can provide weights at each edge which indicate the coefficient of the linear combination of values leading to a unit. We let w^l_{jk} denote the weight of the edge from the k'th neuron in layer $l-1$ to the j'th unit in layer l. We use b^l_j to denote the bias value for neuron j in layer l. a^l_j denotes the activation of this unit (that is, its output). We can write this as:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_k\right)$$

Here σ denotes the nonlinear function that we use.

We can write this in matrix/vector form. Let w^l denote a matrix containing all the weights in layer l. Each row of w^l gives the weights for one unit. We let b^l be a vector of all the bias values, and a^l denote a vector of the outputs at layer l. We also abuse notation, by applying σ to a vector, producing a vector output in which we apply σ componentwise to the elements of the vector. Then we can write:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

It will also be convenient to abbreviate $z^l = w^l a^{l-1} + b^l$, so that $a^l = \sigma(z^l)$. So z_l is just the output before we apply the non-linearity.

1.1 Non-linear functions

There are a wide range of possible non-linear functions that can applied. First, we can use threshold type functions, that turn z^l into a set of binary values. For example, the Sign function turns positive values to +1 and negative values to -1. We used this in the perceptron, and it is natural at least at the output layer when we want the output of the

network to be zero. This has also been inspired by real neurons, which tend to not fire at all, when their output is below a certain level, and then to begin to fire above some threshold. However, one problem with these functions is that they are not differentiable and not suitable for gradient descent. Actually, the problem isn't so much that they are not continuous at 0, but that they saturate, so that they have zero derivative almost everywhere.

A second long-used non-linearity is the sigmoid function:

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

Note that this takes on values from 0 (when $z = -\infty$) to 1 (when $z = \infty$). Most of the action occurs in a small range, when z is between about -4 and 4. A closely related nonlinear function is $\tanh$.

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

These are related as:

$$tanh(z) = 2\sigma(2z) - 1$$

This is a little prettier, in that its values go from -1 to 1, instead of 0 to 1. Sigmoid and tanh also saturate, making gradient based learning more difficult.

More recently, RELU has become very popular.

$$RELU(z) = \max(0, z)$$

The feeling is RELU is better for learning because it doesn't saturate so easily. It does have the odd property that it is unbounded, so that if z is large, $RELU(z)$ is large.

There are other types of nonlinear functions that can be used. Notably, we will see max pooling, which instead of linearly combining the inputs, combines them non-linearly by taking the max. We will look at this when we discuss convolutional neural nets. Goodfellow also says that many other nonlinearities are tried that work about as well as the ones discussed above. These generally don't get discussed in the literature, because they are not regarded as interesting contributions.

1.2 What if we use a linear function?

It's also worth considering what happens if instead of using a nonlinear function, we ignore this step in one of the layers, so that:

$$a^l = w^l a^{l-1} + b^l$$

Let's consider doing this for the case of a three layer network with one hidden layer. We will keep the final nonlinearity, but remove the nonlinearity in the hidden layer. We then have:

$$a^2 = \sigma(w^2 a^1 + b^2) = \sigma(w^2(w^1 a^0 + b^2) + b^1)$$

where a^0 is just the input. We can write this as:

$$a^2 = \sigma(W a^0 + b)$$

where $W = w^2 w^1$ and $b = w^2 b^2 + b^1$. Note that this is equivalent to a two-layer network.

So we might think that there is no point to having the hidden layer in this case. Actually, this isn't necessarily the case, if the hidden layer doesn't have too many units. Let's suppose the input vector is length n, the output is length m, and the hidden layer has k units. Then w^1 is a $k \times n$ matrix, and w^2 is $m \times k$. If $k < m, n$ then W has rank k. This means the set of possible outputs has lower dimension that the input or output space. We can think of this network as projecting the inputs into a lower-dimensional linear subspace of the output space. Potentially, we use many fewer parameters with such a network than we would with a two-layer network that connected n inputs to m outputs. In fact, if we use a loss function that attempts to match the output to the input, this network will learn to perform Principle Component Analysis.

1.3 Example: XOR

Let's work through a simple but classic example that shows the value of a hidden layer. XOR. We have inputs, and their labels, of $(0,0), 1$, $(1,0), 0$, $(0,1), 0$ and $(1,1), 1$. Clearly, we can't handle this with a perceptron. because this computes a linear function of the input followed by a threshold. So the decision boundary is linear. But these inputs are not linearly separable.

To handle this with a three layer network with RELU, let's first consider what it is that one of the hidden units computes. We have:

$$a_1^1 = max(0, w_{1:}^1 a^0 + b_1^1)$$

where $w_{1:}^1$ is the first row of w^1. In this case, $z_{1:}^1$ is a linear function of the input. Then we take the max of this and zero. Note that $z_{1:}^1 = 0$ defines a hyperplane that divides the input space into two halves. $z_{1:}^1$ is zero over one halfspace, and linear over the other half space. The gradient of $z_{1:}^1$ is orthogonal to this hyperplane.

So suppose, for example, that $w_{1:}^1 = (-1 - 1)$ and $b_1^1 = .5$. Then the first hidden unit is associated with the line $x_1 + x_2 = .5$. It's output is zero on this line, and everywhere above the line. Below the line, its output is proportional to the distance from the line to the input. This line divides the point $(0,0)$ from the other three points. Similarly, we can separate $(1,1)$ from the other points with $w_{2:}^1 = (1,1)$ and $b_2^1 = -1.5$. This corresponds to a line at $x_1 + x_2 = 1.5$, with the non-negative part above the line.

Then, if we just add the outputs of these two hidden units, ie $w_{1:}^2 = (1,1), b_1^2 = 0$ we get a function that is zero for the inputs $(1,0)$ and $(0,1)$ and positive for $(0,0)$ and $(1,1)$.

1.4 A geometric interpretation of a three layer network with RELU (Advanced and Optional)

We can generalize this geometric interpretation to any three layer network. We can think of each unit as defining a hyperplane in which the z value of the unit (its output before RELU) is equal to zero. Then the unit will have zero output on one side of

this hyperplane, and will compute a linear function on the other side. Moreover, the gradient of this linear function will be orthogonal to the hyperplane.

We can now think of these hyperplanes as forming an arrangement in the input space. That is, they divide the input space into convex regions bounded by hyperplanes. Inside a given region, all the hidden units compute linear functions (some of which are identically zero). This means that if we take a linear combination of the hidden units, this will also be a linear function within the region. So the z value computed by an output unit will be linear within each region, and piecewise linear over the whole input space. It will also be continuous over the input space, since it is the sum of continuous functions.

Note that while the number of weights grows linearly with the number of hidden units, the number of regions can grow much faster than that. For example, with a 2D input space, we can still get $O(n^2)$ regions, and we can get more in higher dimensional spaces. This means from equation counting that the number of possible piecewise linear functions we can produce in these regions is much larger than the number of parameters we have. So not all piecewise linear functions can be acheived by a given network. In particular, if we have n hidden units we can form any arrangement of regions from any n hyperplanes, but we are constrained as to the linear functions we can compute in these regions. In general, knowing the linear functions in $O(n)$ regions will determine the functions that can be computed in the rest. There's not really a good simple representation of this constraint, though.

Suppose we wish to use a network to perform classifcation of the input into two classes, using an output layer that just thresholds the output values into two classes, just like the Perceptron. Then we can show as a lemma, that if a network perfectly classifies this data, then every region defined by the hidden layers must be linearly separable. This follows from the fact that the output unit computes a linear function in each region. Unfortunately, because of the constraints on the linear functions we can compute across different regions, this is a necessary but not a sufficient condition for perfect classification.

We can use this to get some bounds on classification problems. For example, suppose we have a triangle of positive examples, surrounded by negative examples. What is the fewest hidden units we can use to get perfect classification performance.

It's pretty easy to see that one unit isn't enough. No one unit can divide the data into two linearly separable regions. But two units can. However, can we actually achieve perfect classification? The answer is, not really. Consider a line through the triangle, and graph the z value of the output unit along this line. It must be linear inside the triangle, and linear outside, with three linear pieces. Unless the function is constant inside the triangle, it's impossible to threshold it to perfectly divide the inside from the outside.

However, we can get classification with three lines. Consider lines that are the sides of the triangle and facing outward. All negative examples have positive values for the hidden units, while the positive examples create zero outputs for the hidden units. We can combine these to get outputs of different signs for the positive and negative examples.

Suppose instead of one triangle, we have n triangles. If they are in general position, we can see that we need at least $O(n)$ hidden units to classify things perfectly. It's

not clear that this bound can be acheived, though. For example, we can't just apply the construction we had for one triangle to two. The units for one triangle will also compute non-negative values for the other triangle.

2 Training

Now we consider how to train a neural network. As with the Perceptron, the idea is to define a loss function, and then perform gradient descent on the weights of the network to minimize the loss. We first consider some loss functions, then the approach to gradient descent known as back propagation.

2.1 Loss Functions

We'll start with the simplest loss function.

2.1.1 Quadratic Loss

Above I referred to the input as a^0. I'll also just call this x. The desired output will then be called $y(x)$. Given an input x, we can denote the output of the network as $a^L(x)$, where L denotes the number of layers. So quadratic loss is:

$$C_{quadratic} = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Note that n is the number of training examples.

This loss is often used for regression. That is, we have some continuous value that we want the network to produce, so we penalize according to the sum of square difference between the desired output and the actual output. This is, for example, the loss that we would typically minimize with linear regression.

2.1.2 Cross-entropy

See around Eq. 6.12 in Goodfellow, or Chapter 3 in Nielson.

First, recall the entropy of a random variable, x, is:

$$E(-log(p(x))) = \int_x -p(x) \log(p(x)) dx$$

If log is base 2, this is the average number of bits needed to code instances of this random variable.

For the cross-entropy, we take

$$E(-log(p_{model}(y|x))) = \int_{x,y} -p(x,y) \log(p_{model}(y|x)) d(x,y)$$

That is, we assume we have a model (ie a network) that given x can produce an estimate of the probability distribution of the output, y. Then we take the expected value of the

log of these probabilities. To estimate this from a sample, we take:

$$C_{cross-entropy} = -\frac{1}{n}\sum_{x,y} \log(p_{model}(y|x)$$

Suppose, to make this concrete, that we have a single output node, so that a^L is a scalar. And suppose that our label is binary, so y is 0 or 1. We can interpret a^L as the probability that y is 1. Then our loss would be:

$$C = -\frac{1}{n}\sum_{x,y} y \log a^L + (1-y)\log(1-a^L)$$

So the difference between this and the quadratic loss is that with the quadratic loss, if we want an output of 1 and get an output of y instead, the loss is $(1-y)^2$, whereas with the cross-entropy loss, the loss would be $-\log(y)$.

Notice that the cross-entropy is 0 when the network output matches the label, and gets bigger the more they are mismatched. When a is far from y, the cross-entropy grows much more rapidly than the quadratic loss, which means that there is a much bigger gradient when we are far from the right answer, which can help convergence (see Nielson, chapter 3).

2.1.3 Other loss functions

There are lots of other loss functions, many of which we'll see throughout the course. Sometimes, the main contribution of a paper is to design a loss function that is appropriate for the problem being solved. I'll just mention one class of loss functions, which is to regularize the solution. This can be done, for example, by adding a term based on the norm of the weights, so we not only try to fit the training data, but try to do this with as small weights as possible. This can be viewed as a version of trying to find a simple solution.

2.2 Backpropagation

Now we consider how to find the weights that minimize the loss over the training data. The main idea is to do this using gradient descent. So at each iteration, we try to change the weights to reduce the loss as much as possible. To do this, we need to compute:

$$\frac{\partial C}{w^l_{jk}}$$

for all l, j, k.

2.2.1 The Chain Rule

In writing the cost above, we haven't written it directly as a function of the weights, we've written it as a function of a^L, the output of the network. a^L depends on the weights and the input, so we could write out the loss as a function of all of these, and

then take partial derivatives. But this would be extremely messy. After all, we use the a values to simplify our description of the network with modularity. But also, this will make the partial derivatives simpler, by avoiding dupication of effort when we compute closely related partials. For example, w_{11}^1 and w_{12}^1 both influence z_1^1. But then, the way changes in z_1^1 influence the cost is the same for both. So we save effort by only computing the effect of this change once for all the weights that influence

The chain rule states:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial x}$$

As an example, suppose $y = x^2 + 7$ and $z = \cos y$. We could say:

$$z = \cos(x^2 + 7)$$

Then when we compute:

$$z' = -2x\sin(x^2 + 7)$$

we are using the chain rule. We are saying:

$$\frac{\partial z}{\partial y} = -\sin(y)$$

and

$$\frac{\partial y}{\partial x} = 2x$$

so:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = (-\sin(y))2x = -\sin(x^2 + 7)2x$$

2.2.2 Neural Nets

Ok, so let's see what happens when we use the chain rule on a fragment of a neural network. To simplify notation, I'll just use a simple set of variables. Let's say:

$$z = \sum_i w_i x_i + b$$

where w_i are the weights, which can change, and x_i are the inputs, which are held fixed while we compute the partial derivatives. Next, let's say:

$$a = \sigma(z)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$. And let's say the loss is $C = (y - a)^2$. Here y is the training label, so it's also fixed. We want to compute $\frac{\partial C}{\partial w_i}$. We have:

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i}$$

Then we can compute:

$$\frac{\partial C}{\partial a} = -2(y - a)$$

We also have:

$$\frac{d\sigma(z)}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = \left(\frac{1+e^{-z}-1}{1+e^{-z}}\right)\left(\frac{1}{1+e^{-z}}\right) = (1-\sigma(z))\sigma(z)$$

So, that is:

$$\frac{\partial a}{\partial z} = (1-a)a$$

Finally, we have:

$$\frac{\partial z}{\partial w_i} = x_i$$

So to compute $\frac{\partial C}{\partial w_i}$ we first run the network, which gives us the value of a. Then we can compute:

$$\frac{\partial C}{\partial w_i} = -2(y-a)(1-a)ax_i$$

Note that if we want to compute this for multiple values of w_i, we don't need to recompute $-2(y-a)(1-a)a$. That computation can be shared. In fact, if we have a more complex network, we can share a lot more. We can work this out for a full network with a hidden layer in Figure 1.

Note that I haven't written out all the intermediate values that we can use. For example, we can compute $\frac{\partial C}{\partial z_1^2} = \frac{\partial L}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}$, and then whenever we need this, which we will over all the nodes in the previous layer, we don't have to recompute the product.

This is an example. We can now write this in a more general form. First we'll introduce the abbreviation, $\delta_j^l = \frac{\partial C}{\partial z_j^l}$. This captures how sensitive the cost is to changes in the output of each neuron (before its non-linearity). We can also talk about the vector of these, δ^l.

We can write:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

This is just use of the chain rule. We can write this in vector form as:

$$\delta^L = \nabla_a C \circ \sigma'(z^L)$$

Here $\circ$ is the Hadamard (componentwise) product, and a denotes a^L.

Next, we can write each δ^l in terms of subsequent ones. This gives us:

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \circ \sigma'(z^l)$$

This is also just application of the chain rule. That tells us that $\delta^l = \delta^{l+1}\frac{\partial z^{l+1}}{\partial a^l}\frac{\partial a^l}{\partial z^l}$. The first partial gives us the w^{l+1}.

Using these, we can then get:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \qquad \frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$

This is again through the chain rule, and because $\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$ because the z value is just the weighted sum of the a values of the previous layer (and with the bias term, the weight is applied to just a 1 instead of an a value).

We can think of these equations as propagating the δ values backwards, recursively.

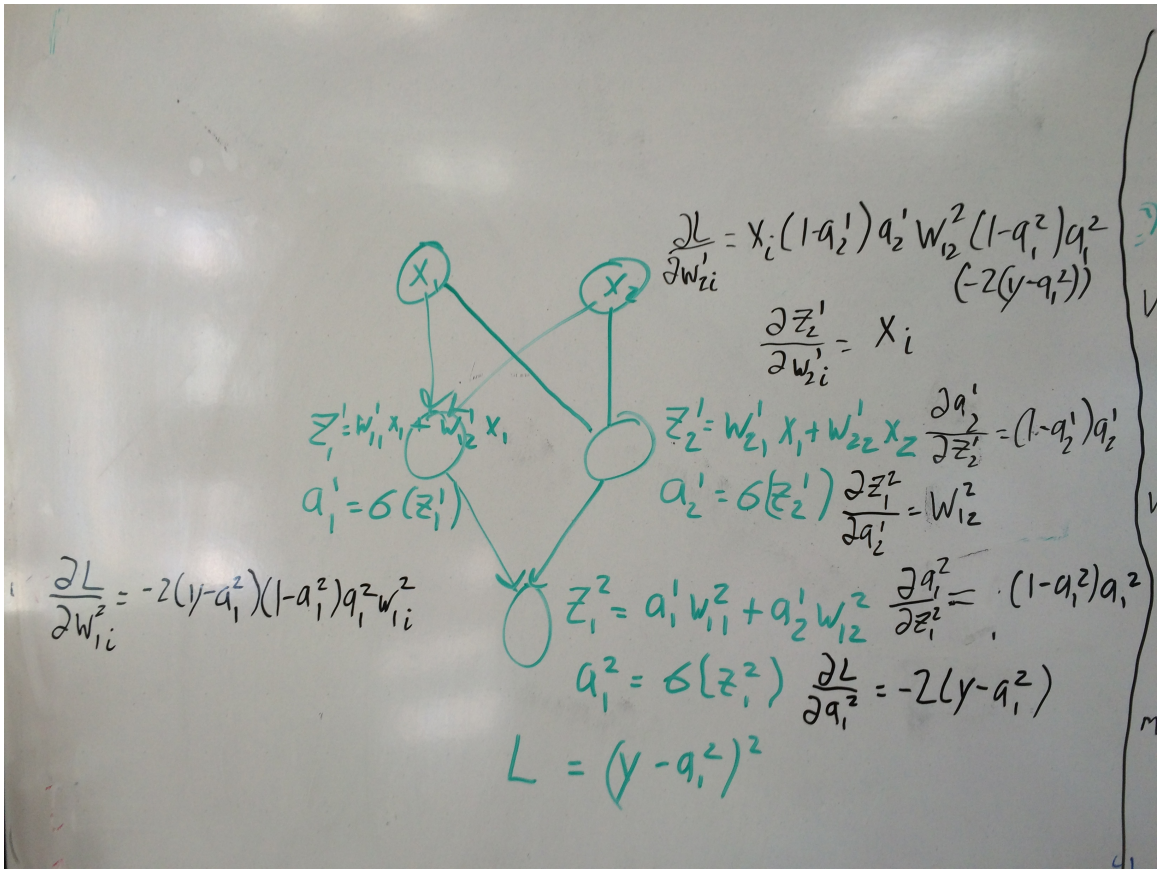

Figure 1: Backpropagation

2.3 Optimization

2.3.1 Stochastic Gradient Descent

Computing the gradient requires looking at all the training examples. Instead, we divide the data into mini-batches, and compute the gradient just based on the mini-batch. This gives us a random gradient whose expected value is the true gradient.

2.3.2 Step size and momentum

The learning rate determines how big a step you take in the direction of the gradient. Generally, the learning rate needs to be decreased over time. Intuitively, as you get closer to a minimum, you need to take smaller steps to avoid overshooting it. This is generally adjusted in a heuristic way.

When using momentum, you move in the direction of the weighted average of

previously computed gradients. This can reduce the noise in the gradient direction. Noise can be due to the *stochastic* gradient descent, or to a cost function with an ill-conditioned Hessian, so the gradient changes very rapidly. This can cause the gradient you compute to change very rapidly.

2.3.3 Initialization

One principal is to avoid symmetries in initialization, avoiding saddle points or local maxima. Generally initialize weights randomly.

Example: what happens when we initialize weights to be zero.

2.3.4 Local Minima

They can occur. Note that one reason is that two networks can be equivalent (eg., they have different hidden units that compute the same thing). These aren't a problem. What's a problem is when we hit local minima with much more error that the global minimum. We'll discuss this problem in a subsequent class.

2.4 Regularization

2.4.1 Early Stopping

Use a validation set. If error on the training set continues to go down, but error on the validation set starts to go up, use the trained network with the lowest error on the validation set. This can be shown in some cases to be a kind of regularization, but really it's a useful heuristic.

2.4.2 Dropout

When training the network, randomly drop out some units, by setting their output to zero. The claim is that this is somewhat like training a large ensemble of networks, and combining the results. To use the trained network, you multiply each unit's output by the probability that it hasn't dropped out.

2.4.3 Batch Normalization

- Motivation: note that when we computed gradients for backpropagation, the gradient with respect to a weight often depended on the value that that weight was being multiplied by (the output of the previous layer). This makes it difficult to decide on a step size, since if the outputs get big (small) the step size implicitly gets bigger (smaller) for that layer, relative to others. Normalizing each layer's output gives us more uniform steps.

- Add a nonlinearity between layers that subtracts the mean of the outputs and scales them to have unit variance. Therefore, the inputs to each layer from each unit at the previous layer will be zero mean unit variance, over the mini-batch.

- Don't make the network output have identity covariance matrix, since this would require too much data from the mini-batch.
- Normalization is taken into account when performing a gradient descent step, which is straightforward since the normalization can also be differentiated.
- This would result in having the inputs to a layer always lie near 0, which might not be desirable. So they also add an affine transformation to each layer that adds an offset and scales the values output by that layer (just two parameters for the whole layer). This would, in theory, allow the network to learn to undue the normalization.
- At test time, we also have to do normalization, but we may just have one test item. So we freeze the normalization, computing statistics based on all the training data.

2.5 Other Considerations

2.5.1 Dataset Augmentation

Generally, the more data you have the better. If your data is limited, sometimes you can use it to generate new data. Often, this is done in a simple way, such as jittering images of characters with small amounts of translation. This is good, because the new data is quite similar to data that you might have collected, if you had more time. But it doesn't necessarily show the full range of variation that truly new data would have. If you only have examples of one type of "4", this won't give you other types.

Sometimes people do much more sophisticated things for dataset augmentation. For example, you can try to deform your examples, or change their colors.

2.5.2 Grid Search for Hyperparameters

2.5.3 Fine Tuning