Lecture 9 – Multi-Layer Neural Networks

***** Exam 1 will be given one week from today (Tuesday, Oct 19, 2021) in person (IESB 218) *****

Gradient Descent and Back propagation

So far, we talked about sigmoid neurons, how they could be used to form multi-layer feed forward networks, and dove into the mathematical details of how an input vector would lead to a cascade of activations flowing through the network.

The critical missing piece is how can these multi-layer networks of sigmoid neurons learn to recognize (or classify) input patterns? This question further breaks down into two parts:

(1) How can we quantify the error present in a network and figure out how that error can be reduced?

(2) Once we know the error and the overall change that needs to be made to correct the error, how can we apportion corrections to that error across the weights and biases that define the network? [This problem may seem intractable at first glance as there may be tens of thousands of weights, even in a network of modest size. For example the network we discussed earlier has 784 X 15 + 15 X 10 = 11,910 weights. If we increase the hidden layer to 30 nodes, as we will do to obtain better performance on the MNIST problem in our programming assignment, we get 23,700 weights.]

Gradient Descent

We will use gradient descent to quantify the error and determine how that error can be reduced.

Each time we present an input vector, **X**, to our network (where **X** is a 784 by 1 vector in our MNIST network) it will produce an output vector, **a** (where **a** is a 10 by 1 vector in our MNIST network).

For each input vector **X**, we also have a CORRECT classification, for that input (in our MNIST network each 784 by 1 input vector will have a 10 by 1 correct output vector).

So if **X** were a 784 by 1 input vector for a handwritten 5, the corresponding **Y** would be a 10 by 1 vector with 0's in all positions save one which would hold a 1. The position of this 1 indicates the output class:

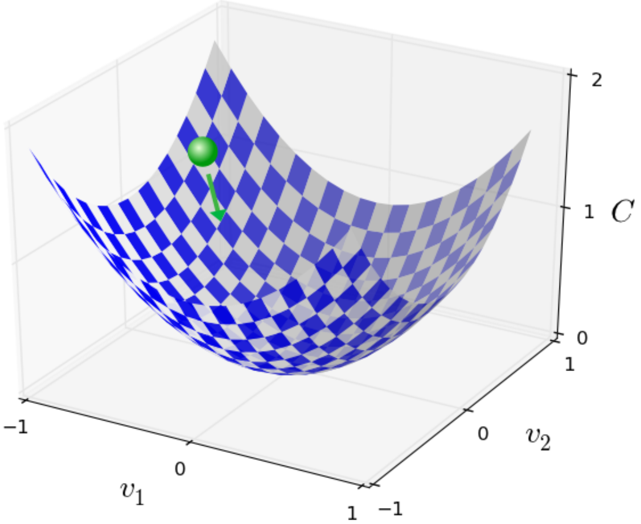
$$\begin{matrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{matrix}$$

Thus, when the input vector **X** belongs to the output class j then $\mathbf{Y}_j = 1$ [assuming zero based indexing]

To save vertical space I can write this vector as $[0,0,0,0,0,1,0,0,0,0]^T$ – where the T means “transpose” this vector. In other words, turn it on its side so it will still be a 10 by 1 vector, instead of 1 by 10.

When we first construct our network, the weights in the network will be randomly assigned so we expect that our output vector **a** will be quite different from our desired output vector **Y** for most inputs **X**. We will need some kind of function that will take **a** and **Y** and return a measure of error, or cost, which we will call **C**. We won’t be interested in C as much as we will be interested in the gradient of **C**.

A **gradient** is a vector that points in the direction of the greatest rate of increase of a function. The magnitude of the vector is the slope of the graph of the function in that direction. The *gradient descent algorithm* works by repeatedly computing the gradient ∇C, and then moving in the *opposite* direction pointed to by gradient vector by a small amount, called “eta”, allowing us to move down the slope in small increments. The end point (we hope) is a network with minimum error. This is NOT guaranteed.



In the *standard* **gradient descent** algorithm, we make a pass through all the training data, at each step incrementally computing the gradients and then, after we have completed a full pass through the entire training data set, we update our weights and biases. I say “incrementally computing” because each training input generates a delta to each $\text{weight\_gradient}_{j\,k}$ and bias_gradient_j and the final gradients are the sum of these. Updating the weights and biases can thus be done as follows:

$$\text{weight}_{j\ k\ \text{new}} = \text{weight}_{j\ k\ \text{old}} – (\text{learning\_rate} /\text{size\_of\_training\_data}) * \sum \text{weight\_gradient}_{j\ k}$$

$$\text{bias}_{j\ \text{new}} = \text{bias}_{j\ \text{old}} – (\text{learning\_rate} /\text{size\_of\_training\_data}) * \sum \text{bias\_gradient}_j$$

where we are summing over the individual gradients generated by each input in the training data set.

In contrast, the **Stochastic Gradient Descent (SGD)** algorithm trains on randomized subsets of the training data. (The word “stochastic” means “randomly determined”.) SGD can dramatically speed up the process of computing the gradients. If you choose your subset (we call them *mini-batches*) in a way that it is representative of the whole training data set you can usually reach valid conclusions about the entire set without examining all of the items.

The ***Stochastic Gradient Descent (SGD)*** algorithm:

1. Randomize the order of the items in the training set.
2. Divide the training set into equal sized mini-batches, (e.g., ten items in each mini-batch)
3. Using **back propagation** (see below) compute the weight gradients and bias gradients over the first [next] mini-batch
4. After completing the mini-batch update the weights and biases as follows:
   $\text{weight}_{j\ k\ new} = \text{weight}_{j\ k\ old}$ – (learning_rate /size_of_mini-batch) * ∑ $\text{weight\_gradient}_{j\ k}$
   $\text{bias}_{j\ new} = \text{bias}_{j\ old}$ – (learning_rate /size_of_mini-batch) * ∑ bias_gradient_{j}
   where the summation is over the weight_gradients and bias_gradients returned from back propagating each input in the mini-batch.
5. If additional mini-batches remain, return to Step 3
6. If our stopping criteria have not been met (e.g. fixed # of epochs; accuracy), return to Step 1

By randomizing the order of the training pairs and choosing a mini-batch size that is not too small the gradients computed from the mini-batch should be similar to the gradients you would have gotten from iterating over the entire training set. [We will divide our 50,000 MNIST training cases into 5,000 mini-batches, each containing ten X , Y pairs (where X is a 784 by 1 vector and Y is a 10 by 1 vector).]
We refer to a pass through all the mini-batches making up a training set as an "*epoch*".

Back propagation

The **back propagation** algorithm:
[Inputs: X (an input vector) and Y (the desired output vector)]
[Outputs: gradient values for each weight and bias in the network]

1. Using the current weights and biases [which are initially random] along with an input vector X, compute the activations (outputs) of all neurons at all layers of the network. This is the "feed forward" pass.
2. Using the computed output of the final layer together with the desired output vector Y, Compute the gradient of the error at the final level of the network and then move "backwards" through the network computing the error at each level, one level at a time. This is the "backwards pass".
3. Return as output the gradient values for each weight and bias in the network.

Back Propagation relies on four underlying equations – that easily simplify to three equations:

(1) one for the error associated with the nodes (biases) in the final layer of the network,
(2) one for the error associated with the nodes (biases) in the intermediate layers, and
(3) one for the error associated with the weights.

First, I will present the equations and we will talk about how to compute them. Later we will come back to each equation to discuss "where the heck these things came from".

Note that equations 1 and 2 are specific to sigmoidal neurons *(neurons that utilize the sigmoidal activation function). Equations 3 and 4 are general, in that they apply to any activation function.*

(1) An equation for the error in the final output layer – the L^{th} level (Capital "L").

Looking ahead to Equation (3) we see that this equation computes the BiasGradients for the final output layer.

$$\delta_j^L = (a_j^L - y_j) * a_j^L * (1 - a_j^L)$$

(2) An equation for the error in the l^{th} layer in terms of the error in the $l+1^{th}$ layer.

Looking ahead to Equation (3) we see that this equation computes the BiasGradients for the hidden layers.

$$\delta_k^l = (\sum_j w_{jk}^{l+1} * \delta_j^{l+1}) * a_k^l * (1 - a_k^l)$$

(3) An equation for the rate of change of the cost with respect to any bias in the network.

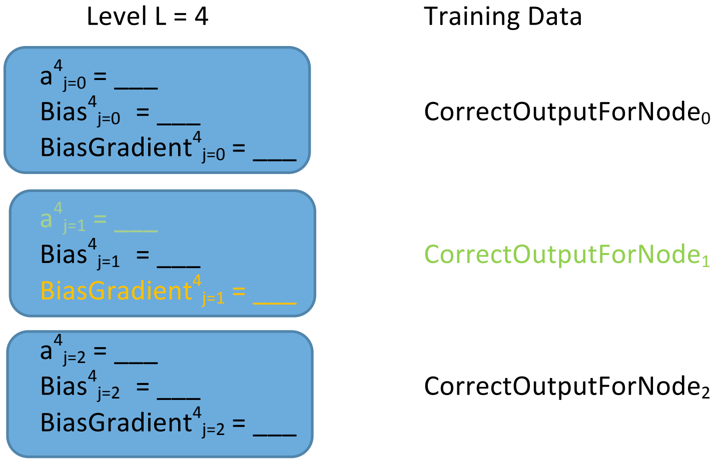
$$\text{BiasGradient}_j^l = \delta_j^l$$

(4) An equation for the rate of change of the cost with respect to any weight in the network.

$$\text{WeightGradient}_{jk}^l = a_k^{l-1}\ \delta_j^l$$

Below, I present figures that will help us visualize how these equations are computed in practice.

Visualizing the Computation of the Bias Gradient for Node 1 in Final (Output) Layer 4



Level L = 4

Training Data

$a^4_{j=0}$ = ____
$\text{Bias}^4_{j=0}$ = ____
$\text{BiasGradient}^4_{j=0}$ = ____

CorrectOutputForNode$_0$

$a^4_{j=1}$ = ____
$\text{Bias}^4_{j=1}$ = ____
$\text{BiasGradient}^4_{j=1}$ = ____

CorrectOutputForNode$_1$

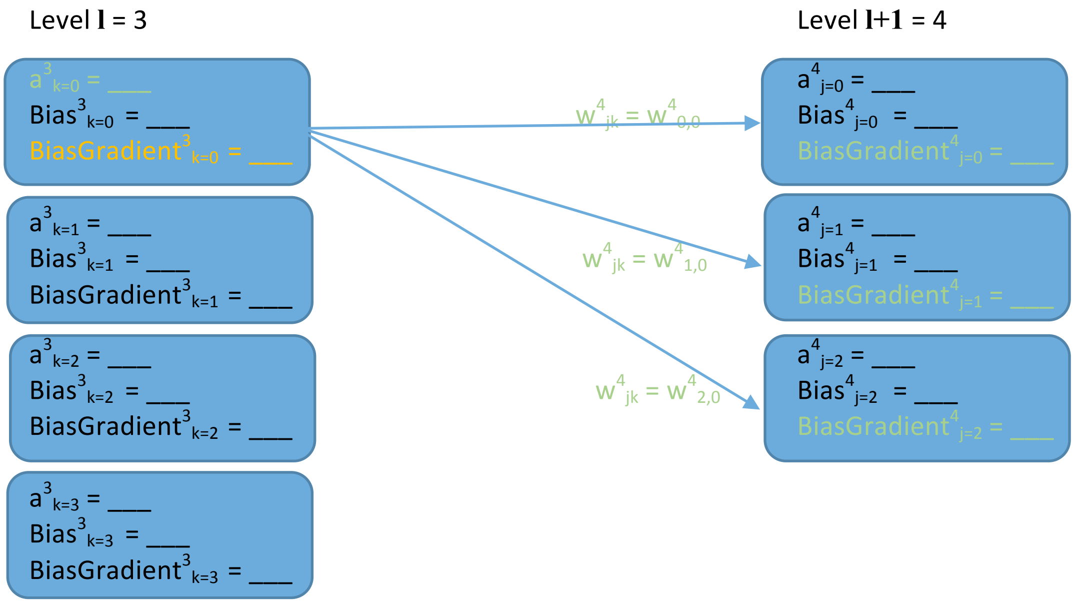
$a^4_{j=2}$ = ____
$\text{Bias}^4_{j=2}$ = ____
$\text{BiasGradient}^4_{j=2}$ = ____

CorrectOutputForNode$_2$

$$\textbf{BiasGradient}_j^L = (a_j^L - y_j) * a_j^L * (1 - a_j^L)$$

$$\textbf{BiasGradient}_1^{\,4} = (a_1^{\,4} - \text{CorrectOutputForNode}_1) * a_1^{\,4} * (1 - a_1^{\,4})$$

Visualizing the Computation of the Bias Gradient for Node 0 in Hidden Layer 3



Level **l** = 3

Level **l+1** = 4

$a^3_{k=0}$ = ____
$\text{Bias}^3_{k=0}$ = ____
$\text{BiasGradient}^3_{k=0}$ = ____

$a^3_{k=1}$ = ____
$\text{Bias}^3_{k=1}$ = ____
$\text{BiasGradient}^3_{k=1}$ = ____

$a^3_{k=2}$ = ____
$\text{Bias}^3_{k=2}$ = ____
$\text{BiasGradient}^3_{k=2}$ = ____

$a^3_{k=3}$ = ____
$\text{Bias}^3_{k=3}$ = ____
$\text{BiasGradient}^3_{k=3}$ = ____

$w^4_{jk} = w^4_{0,0}$

$w^4_{jk} = w^4_{1,0}$

$w^4_{jk} = w^4_{2,0}$

$a^4_{j=0}$ = ____
$\text{Bias}^4_{j=0}$ = ____
$\text{BiasGradient}^4_{j=0}$ = ____

$a^4_{j=1}$ = ____
$\text{Bias}^4_{j=1}$ = ____
$\text{BiasGradient}^4_{j=1}$ = ____

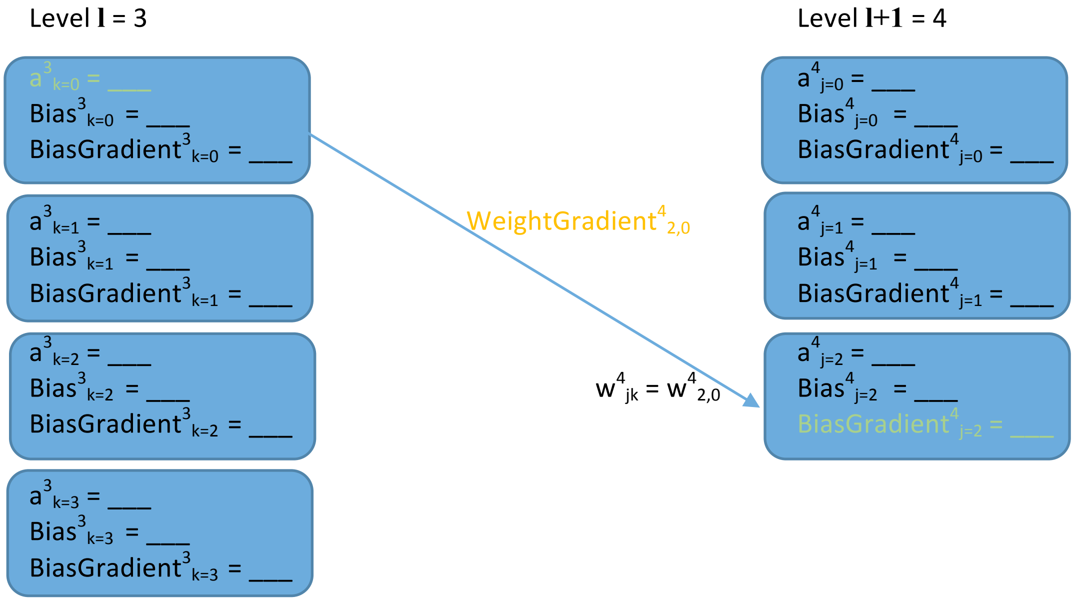
$a^4_{j=2}$ = ____
$\text{Bias}^4_{j=2}$ = ____
$\text{BiasGradient}^4_{j=2}$ = ____

$$\textbf{BiasGradient}_k^{\,l} = (\textstyle\sum_j w_{jk}^{\,l+1} * \text{BiasGradient}_j^{\,l+1}) * a_k^{\,l} * (1 - a_k^{\,l})$$

$$\textbf{BiasGradient}_0^{\,3} = (\textstyle\sum_{j=0 \text{ to } 2} w_{j0}^{\,4} * \text{BiasGradient}_j^{\,4}) * a_0^{\,3} * (1 - a_0^{\,3})$$

$$\textbf{BiasGradient}_0^{\,3} = (w_{0,0}^{\,4} * \text{BiasGradient}_0^{\,4} + w_{1,0}^{\,4} * \text{BiasGradient}_1^{\,4} + w_{2,0}^{\,4} * \text{BiasGradient}_2^{\,4}) * a_0^{\,3} * (1 - a_0^{\,3})$$

Visualizing the Computation of the Weight Gradient for the Weight from Node 0 in Level 3 to Node 2 in Level 4 ($W^4_{2,0}$) Written as WeightGradient$^4_{2,0}$


Level **l** = 3
Level **l**+**1** = 4
$a^3_{k=0}$ = ____
$\text{Bias}^3_{k=0}$ = ____
$\text{BiasGradient}^3_{k=0}$ = ____
$a^3_{k=1}$ = ____
$\text{Bias}^3_{k=1}$ = ____
$\text{BiasGradient}^3_{k=1}$ = ____
$a^3_{k=2}$ = ____
$\text{Bias}^3_{k=2}$ = ____
$\text{BiasGradient}^3_{k=2}$ = ____
$a^3_{k=3}$ = ____
$\text{Bias}^3_{k=3}$ = ____
$\text{BiasGradient}^3_{k=3}$ = ____
$\text{WeightGradient}^4_{2,0}$
$w^4_{jk} = w^4_{2,0}$
$a^4_{j=0}$ = ____
$\text{Bias}^4_{j=0}$ = ____
$\text{BiasGradient}^4_{j=0}$ = ____
$a^4_{j=1}$ = ____
$\text{Bias}^4_{j=1}$ = ____
$\text{BiasGradient}^4_{j=1}$ = ____
$a^4_{j=2}$ = ____
$\text{Bias}^4_{j=2}$ = ____
$\text{BiasGradient}^4_{j=2}$ = ____

$$\textbf{WeightGradient}^l_{jk} = a_k^{l-1}\ \textbf{BiasGradient}_j^l$$

$$\text{WeightGradient}^4_{2,0} = a_0^{\,3} * \text{BiasGradient}_2^{\,4}$$

Go Over the Programming Assignment.

Demo my Solution to the Programming Assignment.

Note Concerning Debugging of your Program.

One of the most difficult things in implementing a neural network architecture from scratch is working through the debug phase. What do you do if your program does not appear to be learning?

Well, the problem could be that the various parameters are just set so that learning the problem is difficult. (Are your initial weights really random? Are your inputs scaled in a way so that their relative size “makes sense” compared to the other parameters? Do you have enough layers? Do you have enough nodes in each of the hidden layers? Etc. etc. etc.)

Additionally, you may have made one or more errors in the underlying back propagation equations. So there could be other bugs in your code.

How do you figure out which is which?

To help you over this hump, I’ve implemented a simple three layer [4,3,2] fully connected, neural network of sigmoidal neurons as an Excel spreadsheet.

Excel Implementation of Stochastic Gradient Descent and Back Propagation

Walk through the Excel spreadsheet that implements a [4,3,2] three-layer network of sigmoid neurons and trains that network on a training set of 4 items subdivided into two mini-batches of size 2 each.

The Network attempts to learn that any input pattern with $X_0 = 1$ should output the vector $[1\ 0]^T$ and any input pattern with $X_3 = 1$ should output the vector $[0\ 1]^T$.

****Optional Material (if time). Otherwise cover in Thursday's lecture.****

YouTube Videos

A four video sequence (by 3Blue1Brown, available on YouTube) provides an introduction to neural networks using MNIST character recognition as an example. These videos provide a different "view" of the material we've covered over the last few lectures – and contain lots of illuminating animations.

https://www.youtube.com/watch?v=aircAruvnKk&feature=youtu.be (neural network intro)
https://www.youtube.com/watch?v=IHZwWFHWa-w (gradient descent)
https://www.youtube.com/watch?v=Ilg3gGewQ5U (back propagation introduction)
https://www.youtube.com/watch?v=tIeHLnjs5U8 (back propagation calculus)