Lecture 8 – Multi-Layer Neural Networks

Multi-layer Feed Forward Networks of Sigmoid Neurons

Useful networks consist of multiple layers with each layer consisting of multiple neurons.

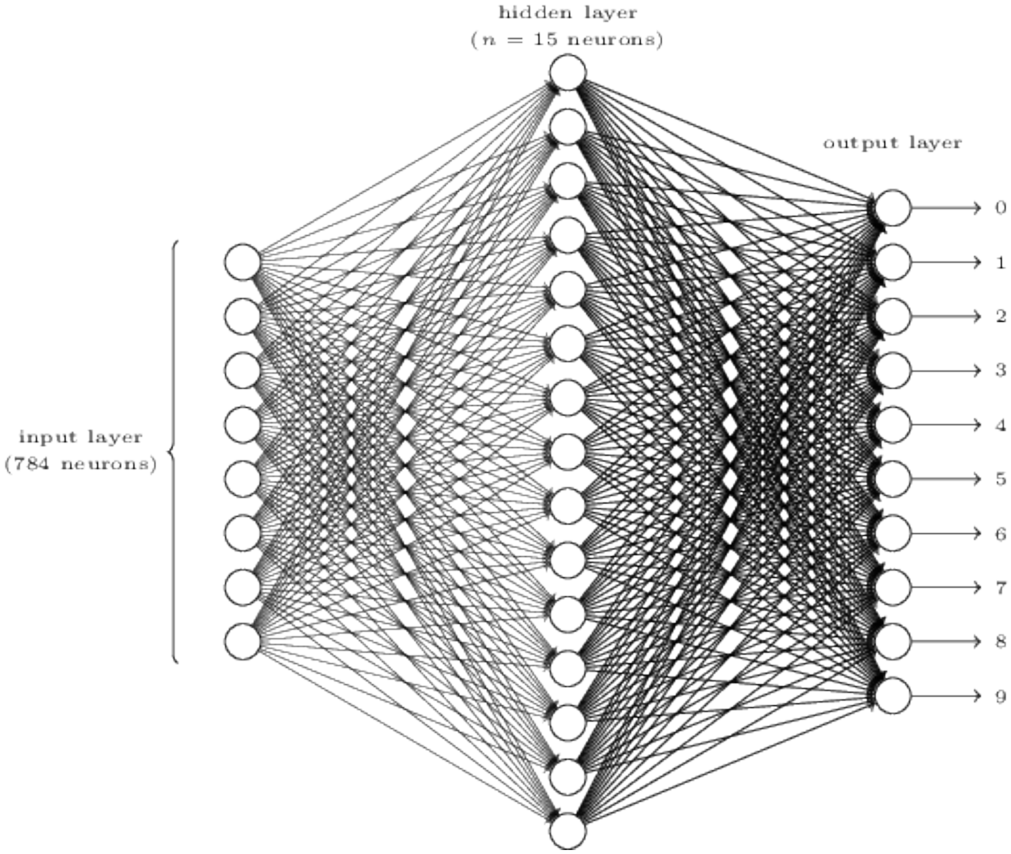
To help focus our discussion we will concentrate on a particular network architecture designed to solve the MNIST handwritten digit recognition problem.

MNIST data may be broken down into a training set of 50,000 images along with their correct classification (0-9), a testing set of 10,000 images with their correct classification (0-9), and a validation set of 10,000 images with their correct classifications.

Each input image is 28 by 28 gray scale pixels – a total of 784 pixels per image.   The input layer to our neural net will thus consist of 784 inputs.

The output layer will consist of 10 neurons, one for each category 0-9. Since the neurons are sigmoid neurons we will expect an output where one of the ten neurons is near 1 and the other 9 neurons are near zero. We could use a value of 0.5 or greater to indicate “yes” and below 0.5 to indicate “no” or simply go with the output neuron with the greatest value.

There will also be a single hidden layer in our particular network with 15 neurons in this level. (Ours is just one of the many different possible architectures for a MNIST character recognizers.)



For the moment, let's assume that this network has been pre-trained.

Let's trace what will happen when an input image is presented to the network.

First, recall that our definition of the sigmoid neuron activation function FOR **ONE** NEURON is:
$y = \boldsymbol{\sigma}(z)$ where $\boldsymbol{\sigma}(z)$ is defined as $1 / (1 + e^{-z})$ and where $z = (\sum (i=0 \text{ to } (n-1)) W_i X_i) + b$
or simply $y = \boldsymbol{\sigma}(w \bullet x + b)$ in vector notation.

$$y = \sigma([w_0\ w_1\ w_2\ \cdots\ w_{\ n-1}\ ] \boxed{?} [x_0\ x_1\ x_2\ \vdots\ x_{n-1}\ \ ] + b)$$

When we realize that in a multi-layer network the outputs (the activations) of one level of the network become the inputs for the next level we get the following formula for computing the activation of the j^{th} neuron in the L^{th} layer of the network:

$$a_j^L = \boldsymbol{\sigma} (\textstyle\sum_{(k)} w^L_{j\,k}\ a_k^{L-1} + b_j^L)$$

where a_j^L is the activation of the j^{th} neuron on the L^{th} level.
$w^L_{j\,k}$ is the weight **FROM** the **k^{th}** neuron on the L-1^{th} level **TO** the **j^{th}** neuron on the L^{th} level**
a_k^{L-1} is the activation of the k^{th} neuron on the L-1^{th} level
b_j^L is the bias of the j^{th} neuron on the L^{th} level

(** Yes, I know this sounds backwards. The order is necessary to enable the matrix multiplication.)

It's very easy to get lost here, so let's think about our concrete example neural network and ask ourselves some simple questions before we try walking through the math that represents the pattern of neuron activations when an input is presented to the network.

(1) **How many input 'units' does this network possess?** 784 (one for each pixel in the input image). So when we pass an image to the network we can imagine that image residing in a 784 by 1 "column" vector. We will call this vector a^0 (because it occurs at the 1^{st} (input layer) and computer scientist like starting their numbers at zero, odd ducks that they are). We will refer to the individual elements of this vector as a_j^0 where j can be any number between 0 and 783 inclusive.

(2) **Are there any incoming weights or biases associated with the input units?** No. Even though we sometimes call these input units "input neurons" they aren't real neurons with biases and incoming weights. (They do, of course, have outgoing values [see (1) above] and outgoing weights [see (6) below].

(3) **How many neurons do we have in the hidden layer?** 15.

(4) **How many output (activation) values will be generated by the hidden layer?** 15, one for each neuron. These activations can be stored in a 15 by 1 "column" vector, which we will call a^1 (because the hidden layer is the second layer of the network and we started counting at zero). The individual elements of this vector are denoted as a_j^1 where j can be a value between 0 and 14 inclusive.

(5) **How many bias values will exist in the hidden layer?** 15, one for each neuron. These can be stored in 15 by 1 "column" vector, which we will call b^1 with individual elements of this vector denoted as b_j^1 where j can be a value between 0 and 14 inclusive.

(6) **How many weights will be incoming to level 1?** In other words **how many weights will connect the 784 input 'units' to the 15 hidden level neurons**? In our "fully connected" network, 784 times 15 (11,760) since each of the 15 neurons will have a weighted connection to the each of 784 'units' in the input layer. It would seem to make sense to store these level 1 weights in a 784 by 15 array. However, in order to make the matrix multiplication work out we will actually use a 15 by 784 matrix called w^1 for the weights coming into level 1 (the second level of the network). Each individual element of w^1 will be referred to as $W^1_{j\,k}$ where the rows of this matrix represent individual neurons at level 1 (so there are 15 rows) and the columns represent the 784 individual input units from level 0. So, for example $W^1_{4\,499}$ would represent the weight of the 500th input unit in level 0 to the 5th neuron at level 1 (assuming zero based indexing).

(7) **How many neurons do we have in the final layer?** 10.

(8) **How many output (activation) values will be generated by the final layer?** 10, one for each neuron. These activations can be stored in a 10 by 1 "column" vector, which we will call a^2 and the individual elements of this vector can denoted as a_j^2 where j can be a value between 0 and 9 inclusive.

(9) **How many bias values will we have in the final layer?** 10, one for each neuron. Stored in a 10 by 1 "column" vector, b^2 with individual elements denoted as b_j^2 where j can be a value between 0 and 9 inclusive.

(10) **How many weights will be coming into level 2?** Each of the 10 neurons in Level 2 will have a weighted connection from each of the 15 neurons in the hidden layer (level 1), giving 150 weights. Thus, w^2 will be a 10 by 15 array. Each row of the matrix will represent a level 2 (output layer) neuron while each column will represent a level 1 (hidden layer) neuron, and the individual weights themselves will be $W^2_{j\,k}$ where j can range from 0 to 9 and k from 0 to 14.

Now that we have identified the various vectors and matrices needed to implement this three level neural network, we are finally in a position to trace the pattern of activations that flow through the network.

Output of Layer 0 – the input (1st) layer

$$\begin{array}{c} a^0 \\ \left|\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{782} \\ a_{783} \end{array}\right| \\ 784 \text{ x } 1 \end{array}$$

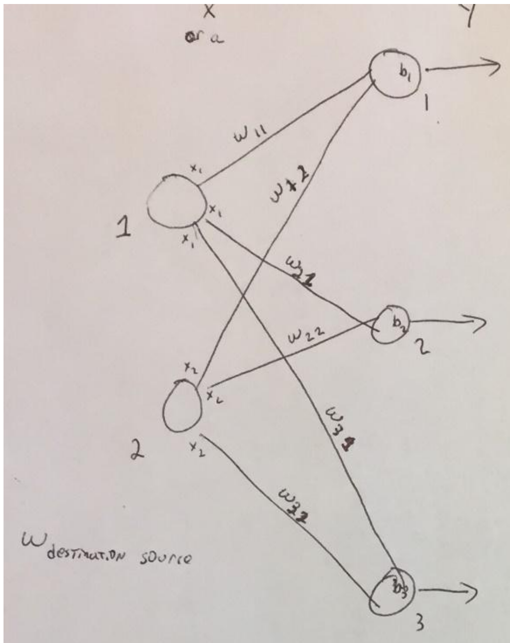
Output of Layer 1 – the “hidden” (2nd) layer

$$\sigma\left(\begin{array}{c} w^1 \\ \left|\begin{array}{ccccc} w_{0\,0} & w_{0\,1} & w_{0\,2} & \dots & w_{0\,783} \\ w_{1\,0} & w_{1\,1} & w_{1\,2} & \dots & w_{1\,783} \\ \dots & \dots & \dots & \dots & \dots. \\ w_{14\,0} & w_{14\,1} & w_{14\,2} & \dots & w_{14\,783} \end{array}\right| \\ 15 \text{ X } 784 \end{array} \bullet \begin{array}{c} a^0 \\ \left|\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{782} \\ a_{783} \end{array}\right| \\ 784 \text{ X } 1 \end{array} + \begin{array}{c} b^1 \\ \left|\begin{array}{c} b_0 \\ b_1 \\ \dots \\ b_{14} \end{array}\right| \\ 15 \text{ X } 1 \end{array} \right) = \begin{array}{c} a^1 \\ \left|\begin{array}{c} a_0 \\ a_1 \\ \dots \\ a_{14} \end{array}\right| \\ 15 \text{ X } 1 \end{array}$$

Output of Layer 2 – the final output (3rd) layer

$$\sigma\left(\begin{array}{c} w^2 \\ \left|\begin{array}{ccccc} w_{0\,0} & w_{0\,1} & w_{0\,2} & \dots & w_{0\,14} \\ w_{1\,0} & w_{1\,1} & w_{1\,2} & \dots & w_{1\,14} \\ \dots & \dots & \dots & \dots & \dots. \\ w_{9\,0} & w_{9\,1} & w_{9\,2} & \dots & w_{9\,14} \end{array}\right| \\ 10 \text{ X } 15 \end{array} \bullet \begin{array}{c} a^1 \\ \left|\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{13} \\ a_{14} \end{array}\right| \\ 15 \text{ X } 1 \end{array} + \begin{array}{c} b^2 \\ \left|\begin{array}{c} b_0 \\ b_1 \\ \dots \\ b_9 \end{array}\right| \\ 10 \text{ X } 1 \end{array} \right) = \begin{array}{c} a^2 \\ \left|\begin{array}{c} a_0 \\ a_1 \\ \dots \\ a_9 \end{array}\right| \\ 10 \text{ X } 1 \end{array}$$

At this point you should be able to see, and implement in code, the overall structure of this network and the pattern of activations that flow through the network when it is presented an input image.



$$\sigma(w \cdot x + b) = y$$

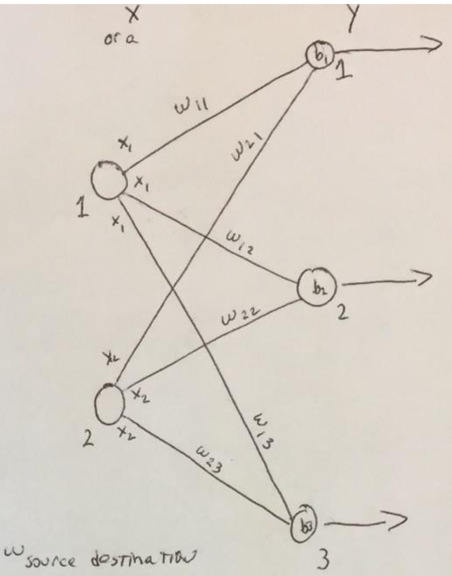
$$\sigma\left(\underset{3\times 2}{\begin{bmatrix} w_{11} & w_{12}\\ w_{21} & w_{22}\\ w_{31} & w_{32}\end{bmatrix}} \cdot \underset{2\times 1}{\begin{bmatrix} x_1\\ x_2\end{bmatrix}} + \underset{3\times 1}{\begin{bmatrix} b_1\\ b_2\\ b_3\end{bmatrix}}\right) = \underset{3\times 1}{\begin{bmatrix} y_1\\ y_2\\ y_3\end{bmatrix}}$$

$$\underset{3\times 1}{\begin{bmatrix} w_{11}\cdot x_1 + w_{12}\cdot x_2\\ w_{21}\cdot x_1 + w_{22}\cdot x_2\\ w_{31}\cdot x_1 + w_{32}\cdot x_2\end{bmatrix}}$$

$$\sigma(w_{11}\cdot x_1 + w_{12}\cdot x_2 + b_1) = y_1$$

$$\sigma(w_{21}\cdot x_1 + w_{22}\cdot x_2 + b_2) = y_2$$

$$\sigma(w_{31}\cdot x_1 + w_{32}\cdot x_2 + b_3) = y_3$$



$$\sigma(x \cdot w + b) = y$$

$$\sigma\left(\underbrace{\underset{1\times 2}{\begin{bmatrix} x_1 & x_2\end{bmatrix}} \cdot \underset{2\times 3}{\begin{bmatrix} w_{11} & w_{12} & w_{13}\\ w_{21} & w_{22} & w_{23}\end{bmatrix}}}_{} + \underset{1\times 3}{\begin{bmatrix} b_1 & b_2 & b_3\end{bmatrix}}\right) = \underset{1\times 3}{\begin{bmatrix} y_1 & y_2 & y_3\end{bmatrix}}$$

$$\begin{bmatrix} x_1\cdot w_{11} + x_2\cdot w_{21} & x_1\cdot w_{12} + x_2\cdot w_{22} & x_1\cdot w_{13} + x_2\cdot w_{23}\end{bmatrix}$$

$$\sigma(x_1\cdot w_{11} + x_2\cdot w_{21} + b_1) = y_1$$

$$\sigma(x_1\cdot w_{12} + x_2\cdot w_{22} + b_2) = y_2$$

$$\sigma(x_1\cdot w_{13} + x_2\cdot w_{23} + b_3) = y_3$$

A Note about Activation Functions

When discussing multi-layer neural networks, I am concentrating on using neurons with a sigmoidal activation function. I am doing this for a number of reasons.

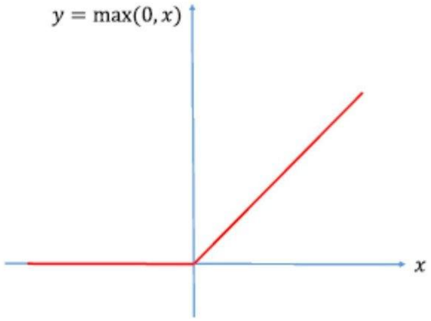
For one, the text that I'm referring to (Michael Neilsen's) uses the sigmoid activation function and I wanted his examples to match what we are doing – especially since his text includes a reference version of the program you are going to implement.

The main reason, however, is historical. Sigmoidal neurons were one of the first activation functions to enable learning in multi-layer neural networks, and they are still used to this day. As we discussed earlier, the sigmoid function was chosen because its output mimics the overall shape of the perceptron output (0 when less than the threshold and 1 when greater than or equal to the threshold); inputs far below the threshold are nearly zero, input far above the threshold are nearly one.

However, Today, one of the, if not the, most popular activation functions is the rectifier. A unit that implements the rectifier is often called a ReLU (Rectified Linear Unit). ReLU's have some advantages over sigmoids. For example, networks using ReLUs tend to converge faster than those using sigmoids. A bonus with using ReLUs is that they are VERY easy to compute. F(X) = MAX (0,X). Much nicer than the sigmoid would you not agree?

Thus, ReLU's are both better (in the results they produce) and more efficient (requiring fewer compute resources) than sigmoids.

Here is what the ReLU activation function looks like:



Gradient Descent and Backpropagation

So far, we talked about sigmoid neurons, how they could be used to form multi-layer feedforward 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:

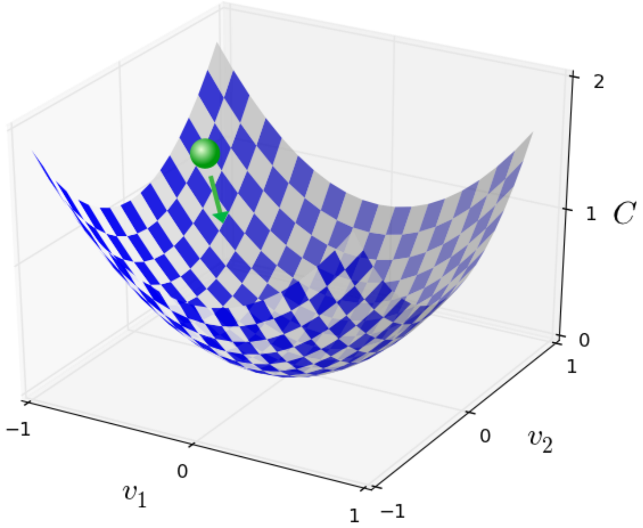
$$\begin{matrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{matrix}$$

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 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 **backpropagation** (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:
   $\text{weight}_{j\,k\,\text{new}} = \text{weight}_{j\,k\,\text{old}} - (\text{learning\_rate} / \text{size\_of\_mini-batch}) * \sum \text{weight\_gradient}_{j\,k}$
   $\text{bias}_{j\,\text{new}} = \text{bias}_{j\,\text{old}} - (\text{learning\_rate} / \text{size\_of\_mini-batch}) * \sum \text{bias\_gradient}_{j}$
   where the summation is over the weight_gradients and bias_gradients returned from backpropagating 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 times; 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*".