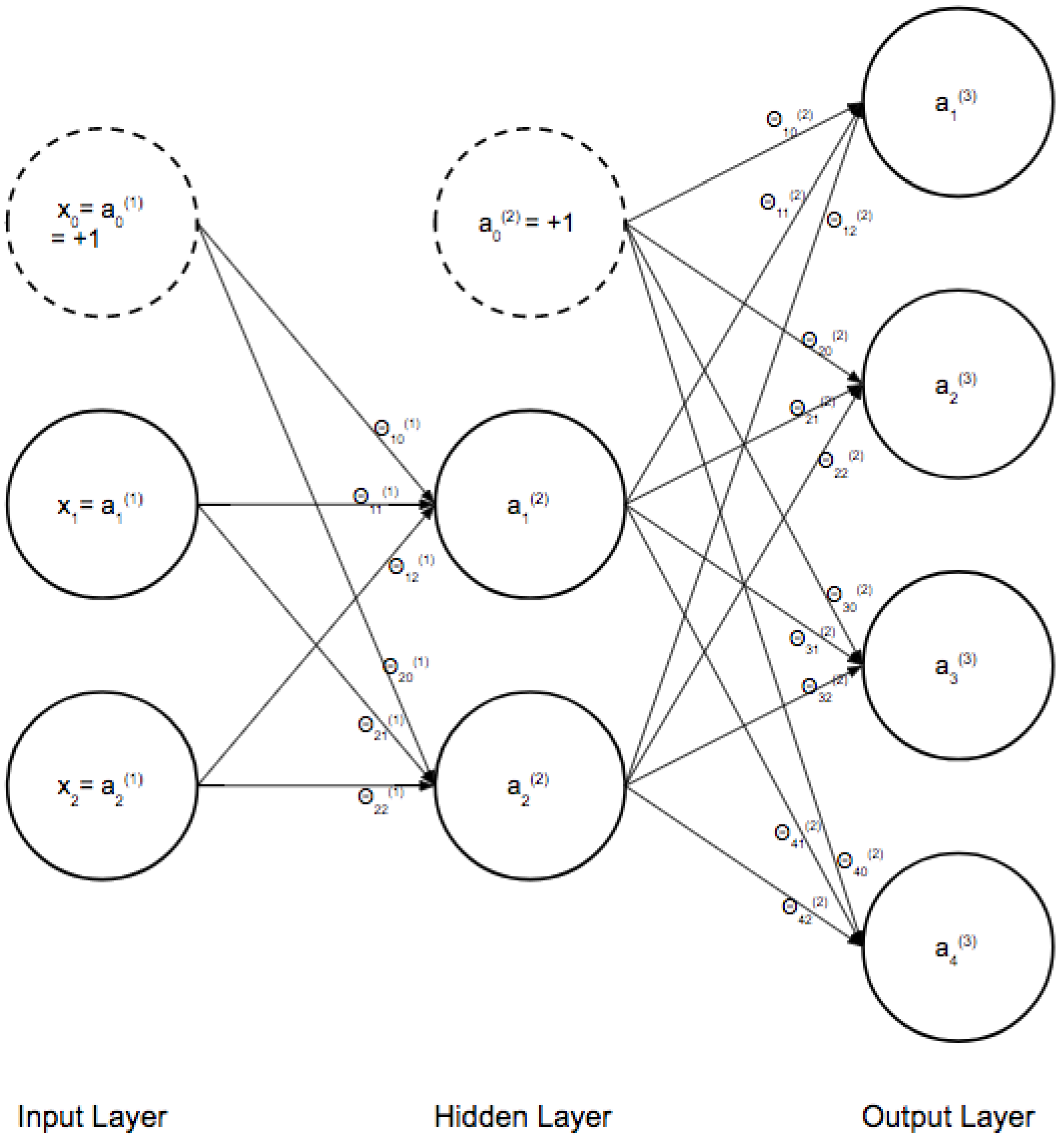
Under The Hood of Neural Network Forward Propagation — The Dreaded Matrix Multiplication

Matt Ross
Sep 10, 2017 · 11 min read

Introduction



This post was motivated by a frustrating bug in a neural network I was building that finally forced me to go under the hood and really understand the linear algebra at the heart of neural networks. I had found that I got by fine just making sure the inner dimensions of the the two matrices I was multiplying matched and when bugs would occur I would just kind of transpose a matrix here and transpose a matrix there until things worked out but this hid the truth that I didn't really understand each step of how the matrix multiplication worked.

We're going to go through each step of using forward propagation to compute the cost function of a rather simple neural network. Oh and if you are wondering the error my matrix multiplication ignorance caused, it was that I added my bias units (vector of 1's) as a column when it should have been a row. I did this because I didn't really understand the full output of the matrix multiplication prior to this step so didn't realize I had to make the change. To begin I'll explain the high level background of what is happening in forward propagation in a neural network, then we'll take a much closer look in a specific example, with indexed values and code to keep things clear.

So, neural networks are incredible at modelling complex relationships. We are only going to talk about the feed-forward propagation part of the network. Now, a neural network's input unit's could be anything. They could for example be the grayscale intensity (between 0 and 1) of a 20 pixel by 20 pixel image that represents a bunch of handwritten digits. In that case you would have 400 input units. Here we have 2 input units, plus our +1 bias unit (for an awesome explanation as to why have the bias unit at all, go here). Forward propagation is essentially taking each input from an example (say one of those images with a hand written digit) then multiplying the input values by the weight of each connection between the units/nodes (see Figure 5), then adding up all the products of all the connections to the node you are computing the activation of and taking that sum (z) and putting that through the sigmoid function (see below).

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

Figure 1: Sigmoid Function

That gives you the activation of each of the units of the hidden layer. Then you do the same to compute the next layer but you use the activations of the hidden layer as the "input" values this time. You multiply all the a^2 activations (i.e. the hidden layer) units by the second set of weights Theta2, sum each product connected to a single final output unit and pass that product through the sigmoid function to get yourself the final output activations a^3. g(z) is the sigmoid function and z is the product of the x input (or activation in hidden layers) and the weight theta (represented by a single arrow in the normal neural network diagram Figure 5).

$$h_\theta(x) = g(\theta^T x),$$

Figure 2: Hypothesis Function, using the Sigmoid Function.

Once you have all this, you want to calculate the cost of the network(Figure 4). Your cost function essentially calculates the cost/difference between the output hypothesis h(x) and the actual y values for the examples given. So, in the example I keep using, the y being the value of the actual digit represented by the inputs. If there is a image of a "4" feed through the network, the y is the value "4". Since there are multiple output units, the cost function compares h(x), the output, against a column vector where the 4th row is a 1, all the rest 0's. Meaning the output unit representing a "4" output it true, and all the rest false. For a output of, 1, 2 or n, see below.

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ \ldots \ \text{or} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

Figure 3: Our example data y values represented as logical TRUE/FALSE column vectors.

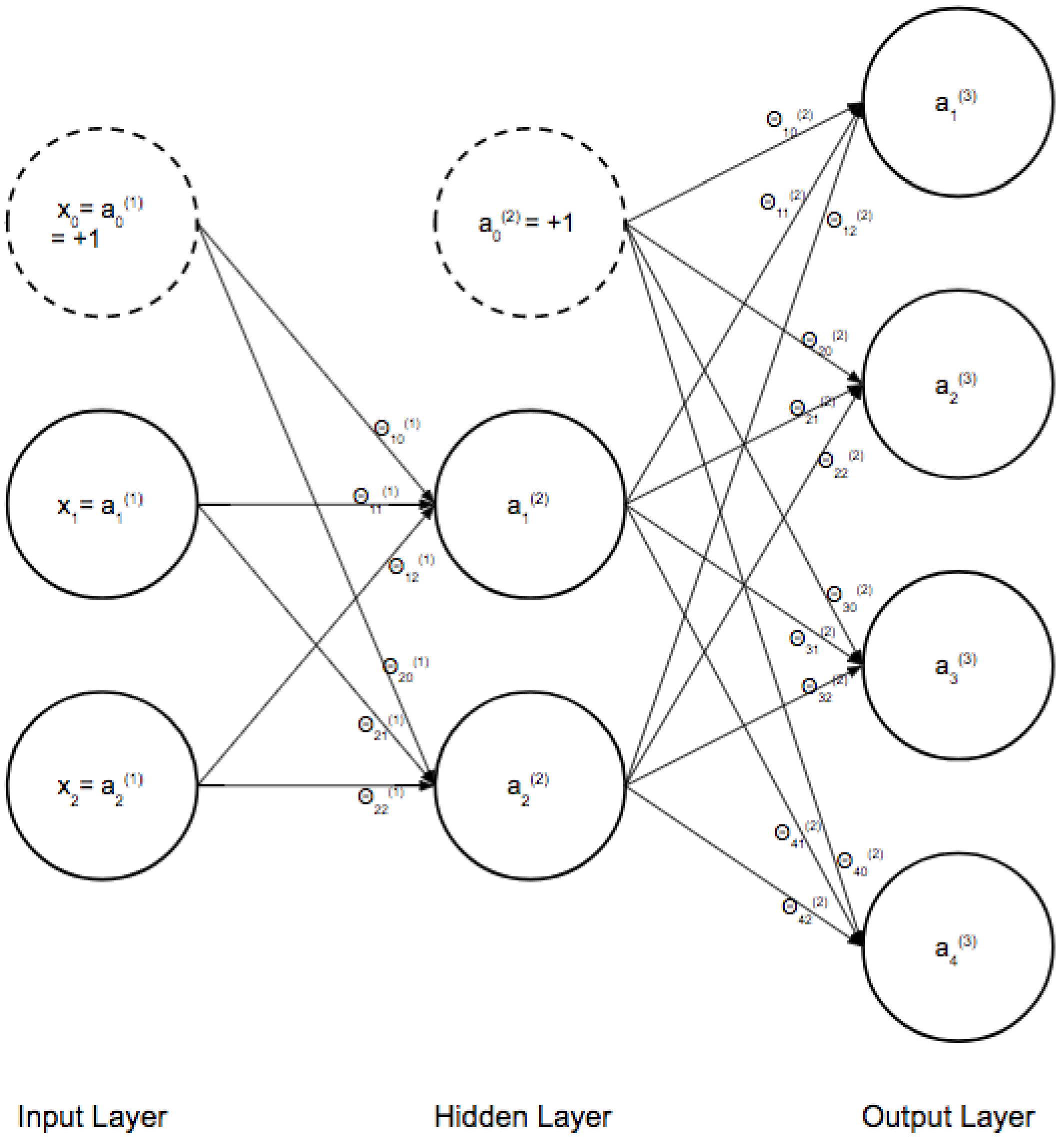
$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[-y_k^{(i)} \log((h_\theta(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right],$$

Figure 4: The Multi-Class Logistic Regression Cost Function

The two Sigma's in the cost function J(theta) above are there to sum up the cost for every single example you feed through the network (m) and for every single output class (K). *Now, you could do this by doing each computation individually, but as it turns out the way that humans have defined matrix multiplication makes it perfect for doing all of these forward propagation computations simultaneously.* Our friends in numerical computing have optimized the matrix multiplication functions so that a neural network can output a hypothesis extremely efficiently. To write our code so that we are doing all the computations simultaneously rather than say running everything in a for loop over all the input examples, is a process called vectorizing our code. It is VERY important in neural networks since they are already computationally expensive enough, we don't need any for loops in there slowing us down even more.

Our Network Example

In our network, we will have four classes, 1, 2, 3, 4 and will walk through each step of this computation. We'll assume the network is already trained and we have our Theta parameters/weights already trained through back propagation. It will be a 3 layer network (with two input units, 2 hidden layer units, and 4 output units). The network and parameters (a.k.a. weights) can be represented as follows.



$x_0 = a_0^{(1)} = +1$
$x_1 = a_1^{(1)}$
$x_2 = a_2^{(1)}$
$a_0^{(2)} = +1$
$a_1^{(2)}$
$a_2^{(2)}$
$a_1^{(3)}$
$a_2^{(3)}$
$a_3^{(3)}$
$a_4^{(3)}$
$\Theta_{10}^{(1)}$ $\Theta_{11}^{(1)}$ $\Theta_{12}^{(1)}$ $\Theta_{20}^{(1)}$ $\Theta_{21}^{(1)}$ $\Theta_{22}^{(1)}$
$\Theta_{10}^{(2)}$ $\Theta_{11}^{(2)}$ $\Theta_{12}^{(2)}$ $\Theta_{20}^{(2)}$ $\Theta_{21}^{(2)}$ $\Theta_{22}^{(2)}$
$\Theta_{30}^{(2)}$ $\Theta_{31}^{(2)}$ $\Theta_{32}^{(2)}$ $\Theta_{40}^{(2)}$ $\Theta_{41}^{(2)}$ $\Theta_{42}^{(2)}$
Input Layer
Hidden Layer
Output Layer

Figure 5: Our Neural Network, with indexed weights.

Before we go much farther, if you don't know how matrix multiplication works, then check out Khan Academy spend the 7 minutes, then work through an example or two and make sure you have the intuition of how it works. It is important to know this before going forward.

Let's begin with all our data. Our 3 pieces of example data and the corresponding y output values. This data doesn't represent anything, they are just numbers to show the calculations we'll be doing:

$$X = \begin{array}{cc|l} \text{Feature } x_1 & \text{Feature } x_2 & \\ \downarrow & \downarrow & \\ \hline 0.5403 & -0.4161 & \leftarrow \text{Example 1 input data, } x^{(1)} \\ -0.9900 & -0.6536 & \leftarrow \text{Example 2 input data, } x^{(2)} \\ 0.2837 & 0.9602 & \leftarrow \text{Example 3 input data, } x^{(3)} \end{array} \qquad y = \begin{array}{c|l} 4 & \leftarrow \text{Example 1 output data, } y^{(1)} \\ 2 & \leftarrow \text{Example 2 output data, } y^{(2)} \\ 3 & \leftarrow \text{Example 3 output data, } y^{(3)} \end{array}$$

Figure 6: Our data.

Of course as I mentioned, since there are 4 output units, our data must be represented as a matrix of logical vectors for each of the three example outputs. I'm working in MATLAB so to turn our y vector into a matrix of logical vectors:

```
yv=[1:4] == y;   %creating logical vectors of y values
```

$$yv = \begin{array}{cccc|l} 0 & 0 & 0 & 1 & \leftarrow \text{Logical Vector of Example 1 output data, } y^{(1)} \\ 0 & 1 & 0 & 0 & \leftarrow \text{Logical Vector of Example 2 output data, } y^{(2)} \\ 0 & 0 & 1 & 0 & \leftarrow \text{Logical Vector of Example 3 output data, } y^{(3)} \end{array}$$

Figure 7: Matrix of Example Output y data turned into logical vectors.

Also, notice that our X data doesn't have enough features. In our Figure 5 Neural Network, we have that dotted line bias unit x(0) that is necessary when we compute the product of the weights/parameters and the input value. This means we need to add our bias units to the data. Meaning we add a column to the beginning of the matrix:

```
X = [ones(m,1),X];
```

$$X = a^{(1)} = \begin{array}{c c c l} \text{Bias} & \text{Feature} & \text{Feature} & \\ x_0 & x_1 & x_2 & \\ \downarrow & \downarrow & \downarrow & \\ 1.0000 & 0.5403 & -0.4161 & \leftarrow \text{Example 1 input data, } x^{(1)} \\ 1.0000 & -0.9900 & -0.6536 & \leftarrow \text{Example 2 input data, } x^{(2)} \\ 1.0000 & 0.2837 & 0.9602 & \leftarrow \text{Example 3 input data, } x^{(3)} \end{array}$$

Figure 8: Data with bias added. Bias represented by dotted-line unit/node in Neural Net Figure 5

The data X is defined as the first activation values of the first input layer a^1 so if you ever see a^1 in the code (line 3), it is just referring to the initial input data. Our weights or parameters of each of the connections/arrows in the network are as follows:

$$\text{Theta1} = \begin{bmatrix} \underset{\Theta_{10}^{(1)}}{0.1} & \underset{\Theta_{11}^{(1)}}{0.3} & \underset{\Theta_{12}^{(1)}}{0.5} \\ \underset{\Theta_{20}^{(1)}}{0.2} & \underset{\Theta_{21}^{(1)}}{0.4} & \underset{\Theta_{22}^{(1)}}{0.6} \end{bmatrix} \begin{array}{l} \leftarrow \text{Weights Making up 1st Activation Unit of 2nd Layer} \\ \leftarrow \text{Weights Making up 2nd Activation Unit of 2nd Layer} \end{array}$$

Figure 9: First set of weights/parameters for our neural network with indices that match those on the arrows of the Figure 5 Neural Network diagram.

Below is the full code we will use to compute our logistic cost function, we've tackled line 2 and 9 but we will slowly break down the matrix multiplication and important matrix manipulations in the rest of this code:

```
1:  m = size(X, 1);
2:  X = [ones(m,1),X];
3:  a1 = X;
4:  z2 = Theta1*a1';
5:  a2 = sigmoid(z2);
6:  a2 = [ones(1,m);a2];
7:  z3 = Theta2*a2;
```

```
8:  a3 = sigmoid(z3);
9:  yv=[1:4] == y;
10: J = (1/m) * (sum(-yv’ .* log(a3) — ((1 — yv’) .* log(1 — a3))));
11: J = sum(J);
```

Let’s do the first step of the forward propagation, line 4 in the code above. Multiplying the input value for each example by their corresponding weights. I always imagine the input value flowing in and along the arrow in our network Figure 5, getting hit/multiplied by the weight then waiting at the activation unit/node for the other arrows to do their multiplication. Then the whole activation value for a particular unit is made up by first the sum of each of these arrow (input x weight), then that sum is passed through the sigmoid function (see Figure 1 above).

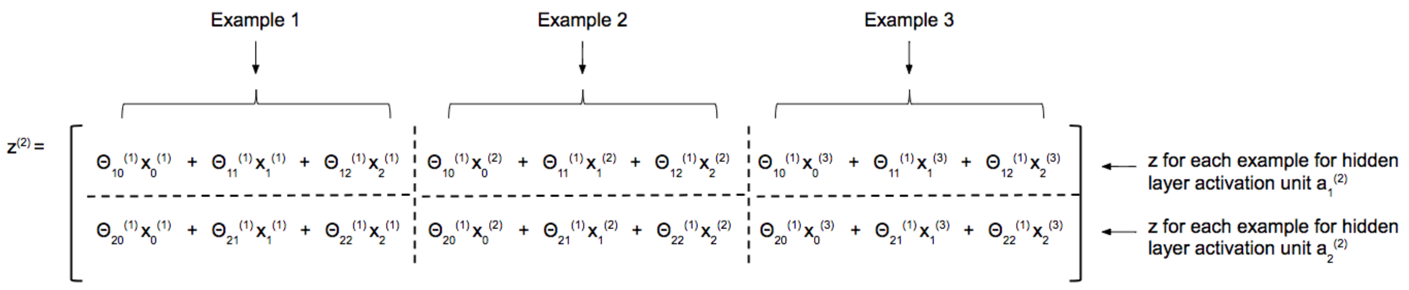
So here it’s easy to make your first matrix multiplication mistake. Since our data with bias units added to X (also called a1 here) is a 3x3 matrix and our Theta1 is a 2x3 matrix. It would be easy to simply matrix multiply the two together since the two inner dimensions of Theta: 2x3 and X: 3x3 are the same and so viola that should work right and give us our 2x3 resultant matrix? Wrong!

```
z2 = Theta1 * a1;      %WRONG! THIS DOESN'T GIVE US WHAT WE WANT
```

Though running this computation will output a matrix of the correct dimension that we would expect and need for the next step, all the values computed will be wrong and thus all computations will be wrong from here on in. Plus, since there was no computer error, it will be hard to tell why your network cost function is computing the wrong cost, if you even notice it. Remember, when you do matrix multiplication, each element ab of the resulting matrix is the dot product sum of the row in the first matrix **row a** by column of the second matrix **column b**. If we used the above code for computing z^2 above, this first element in the resulting matrix would result from multiplying our 1st row of Theta’s [0.1 0.3 . 0.5] with an entire column of bias units, [1.000; 1.000; 1.000], which is useless to us. This means we need to transpose our matrix of example input data so that the matrix will multiply each theta with each input correctly:

```
z2 = Theta1*a1';
```

The matrix multiplication of this is as follows:


Example 1
Example 2
Example 3
z for each example for hidden layer activation unit $a_1^{(2)}$
z for each example for hidden layer activation unit $a_2^{(2)}$

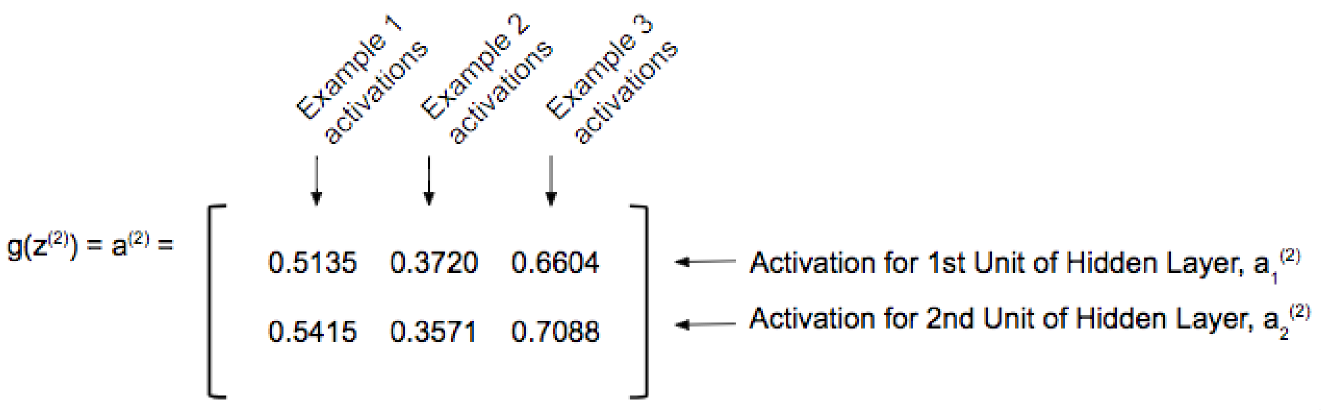
$$z^{(2)} = \begin{bmatrix} \Theta_{10}^{(1)}x_0^{(1)} + \Theta_{11}^{(1)}x_1^{(1)} + \Theta_{12}^{(1)}x_2^{(1)} & \Theta_{10}^{(1)}x_0^{(2)} + \Theta_{11}^{(1)}x_1^{(2)} + \Theta_{12}^{(1)}x_2^{(2)} & \Theta_{10}^{(1)}x_0^{(3)} + \Theta_{11}^{(1)}x_1^{(3)} + \Theta_{12}^{(1)}x_2^{(3)} \\ \Theta_{20}^{(1)}x_0^{(1)} + \Theta_{21}^{(1)}x_1^{(1)} + \Theta_{22}^{(1)}x_2^{(1)} & \Theta_{20}^{(1)}x_0^{(2)} + \Theta_{21}^{(1)}x_1^{(2)} + \Theta_{22}^{(1)}x_2^{(2)} & \Theta_{20}^{(1)}x_0^{(3)} + \Theta_{21}^{(1)}x_1^{(3)} + \Theta_{22}^{(1)}x_2^{(3)} \end{bmatrix}$$

Figure 10: The indexed symbolic representation of the matrix multiplication. The resultant elements in the columns each representing a single example, and the rows being the different activation units in the hidden layer. 2 hidden layers results in two values (or rows) per example.

Then we element-wise apply the sigmoid function to each of the 6 elements in the z^2 matrix above:

```
a2 = sigmoid(z2);
```

This just gives us a 2x3 matrix of the hidden layer activation values for both hidden units for each of the three examples:


Example 1 activations
Example 2 activations
Example 3 activations
Activation for 1st Unit of Hidden Layer, $a_1^{(2)}$
Activation for 2nd Unit of Hidden Layer, $a_2^{(2)}$

$$g(z^{(2)}) = a^{(2)} = \begin{bmatrix} 0.5135 & 0.3720 & 0.6604 \\ 0.5415 & 0.3571 & 0.7088 \end{bmatrix}$$

Figure 11: The activation values of the hidden units.

Because this was done as matrix multiplication, we were able to compute the activation values of the hidden layers all at once, rather than using say a for loop over all the examples which becomes extremely computationally expensive when you work with larger data sets. Not to mention then needing to do back propagation as well.

Now that we have the values of our activation units in the second layer, these act as the inputs into the next and final layer, the output layer. This layer has a new set of weights/parameters, Theta2, for each of the arrows in Figure 5 between the 2nd and 3rd layer, and we will do the same things we did above. Multiply the activation value (the input) by the weight connected to each activation node, separately sum up the products connected to each activation node, then run each activation node sum through the sigmoid function to get your final outputs. Our a^2 above acts as our input data and our weights/parameters are as follows:

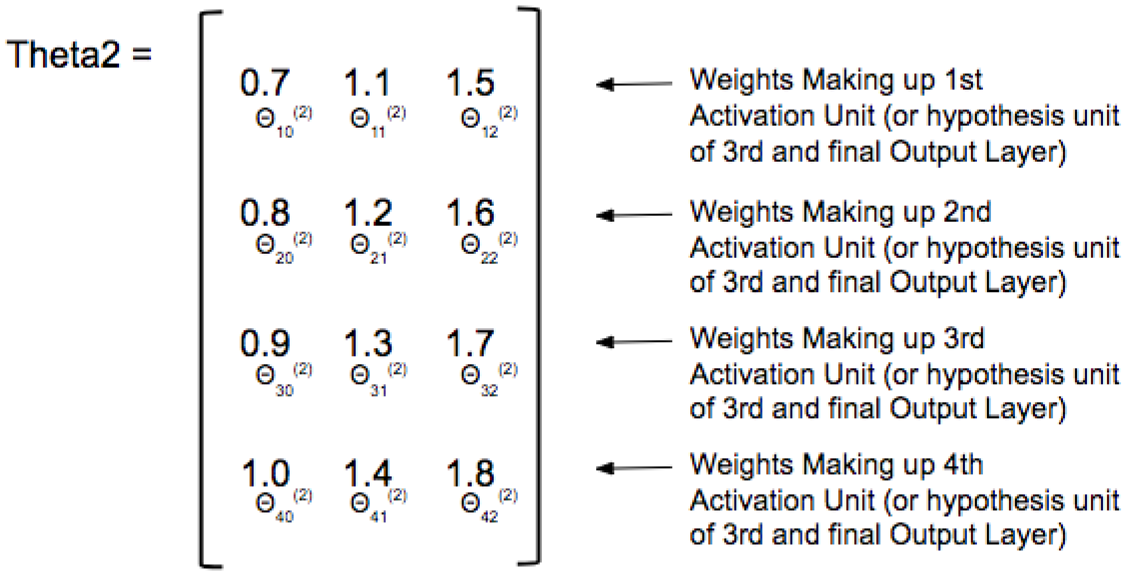

Theta2 =
0.7 $\Theta_{10}^{(2)}$ 1.1 $\Theta_{11}^{(2)}$ 1.5 $\Theta_{12}^{(2)}$
0.8 $\Theta_{20}^{(2)}$ 1.2 $\Theta_{21}^{(2)}$ 1.6 $\Theta_{22}^{(2)}$
0.9 $\Theta_{30}^{(2)}$ 1.3 $\Theta_{31}^{(2)}$ 1.7 $\Theta_{32}^{(2)}$
1.0 $\Theta_{40}^{(2)}$ 1.4 $\Theta_{41}^{(2)}$ 1.8 $\Theta_{42}^{(2)}$
Weights Making up 1st Activation Unit (or hypothesis unit of 3rd and final Output Layer)
Weights Making up 2nd Activation Unit (or hypothesis unit of 3rd and final Output Layer)
Weights Making up 3rd Activation Unit (or hypothesis unit of 3rd and final Output Layer)
Weights Making up 4th Activation Unit (or hypothesis unit of 3rd and final Output Layer)

Figure 12: Theta2 weights/parameters with indices. Each row represents a different output unit with the weights contributing to each output unit across each column.

We want to do the following computation:

```
z3 = Theta2*a2;
```

But before we do that, we must again add our bias units to our data, in this case the hidden layer activations a^2. If you notice again in Figure 5, the dotted line circle in the hidden layer, a(0), the bias unit which is only added when we do the next computation. So, we add this to the activations matrix seen in Figure 11 above.

This is where I introduced my bug that was the motivation for this post. To forward propagate the activation values, we will multiply each element of a row in Theta with each element of a column in a^2 and the sum of these products will give us a single element of the resulting z^3 matrix. Often the way data is structured has you adding the bias unit as a column, but if you did that (which I stupidly did), this would give us the wrong result. So we add the bias units as a row to a^2.

```
a2 = [ones(1,m);a2];
```

$$
\begin{array}{lcccl}
 & \text{Example 1 activations} & \text{Example 2 activations} & \text{Example 3 activations} & \\
 & \downarrow & \downarrow & \downarrow & \\
g(z^{(2)}) = a^{(2)} = \left[\right. & 1.0000 & 1.0000 & 1.0000 & \left.\right] \leftarrow \text{Activation for 0th bias Unit of Hidden Layer, } a_0^{(2)} \\
 & 0.5135 & 0.3720 & 0.6604 & \leftarrow \text{Activation for 1st Unit of Hidden Layer, } a_1^{(2)} \\
 & 0.5415 & 0.3571 & 0.7088 & \leftarrow \text{Activation for 2nd Unit of Hidden Layer, } a_2^{(2)}
\end{array}
$$

Figure 13: Adding the bias row to the a² activations.

Before we run our matrix multiplication to compute z^3 notice that where before in z^2 you had to transpose the input data a^1 to make it "line up" correctly for the matrix multiplication to result in the computations we wanted. Here, our matrices are lined up the way we want, so there is no transpose of the a^2 matrix. This is another common mistake and it is easy to do it if you don't understand the computation at the heart of it all (I've been very guilty of this in the past). Now we can run our matrix multiplication on a 4x3 and 3x3 matrix, resulting in a 4x3 matrix of output hypotheses for each of the 3 examples:

```
z3 = Theta2*a2;
```

$$
\begin{array}{c}
\begin{array}{ccc}
\text{Example [1]} & \text{Example [2]} & \text{Example [3]} \\
\downarrow & \downarrow & \downarrow
\end{array}
\\
z^{(3)} = \left[\begin{array}{c:c:c}
\Theta_{10}^{(2)} a_0^{(2)[1]} + \Theta_{11}^{(2)} a_1^{(2)[1]} + \Theta_{12}^{(2)} a_2^{(2)[1]} & \Theta_{10}^{(2)} a_0^{(2)[2]} + \Theta_{11}^{(2)} a_1^{(2)[2]} + \Theta_{12}^{(2)} a_2^{(2)[2]} & \Theta_{10}^{(2)} a_0^{(2)[3]} + \Theta_{11}^{(2)} a_1^{(2)[3]} + \Theta_{12}^{(2)[3]} a_2^{(2)} \\ \hdashline
\Theta_{20}^{(2)} a_0^{(2)[1]} + \Theta_{21}^{(2)} a_1^{(2)[1]} + \Theta_{22}^{(2)} a_2^{(2)[1]} & \Theta_{20}^{(2)} a_0^{(2)[2]} + \Theta_{21}^{(2)} a_1^{(2)[2]} + \Theta_{22}^{(2)} a_2^{(2)[2]} & \Theta_{20}^{(2)} a_0^{(2)[3]} + \Theta_{21}^{(2)} a_1^{(2)[3]} + \Theta_{22}^{(2)[3]} a_2^{(2)} \\ \hdashline
\Theta_{30}^{(2)} a_0^{(2)[1]} + \Theta_{31}^{(2)} a_1^{(2)[1]} + \Theta_{32}^{(2)} a_2^{(2)[1]} & \Theta_{30}^{(2)} a_0^{(2)[2]} + \Theta_{31}^{(2)} a_1^{(2)[2]} + \Theta_{32}^{(2)} a_2^{(2)[2]} & \Theta_{30}^{(2)} a_0^{(2)[3]} + \Theta_{31}^{(2)} a_1^{(2)[3]} + \Theta_{32}^{(2)[3]} a_2^{(2)} \\ \hdashline
\Theta_{40}^{(2)} a_0^{(2)[1]} + \Theta_{41}^{(2)} a_1^{(2)[1]} + \Theta_{42}^{(2)} a_2^{(2)[1]} & \Theta_{40}^{(2)} a_0^{(2)[2]} + \Theta_{41}^{(2)} a_1^{(2)[2]} + \Theta_{42}^{(2)} a_2^{(2)[2]} & \Theta_{40}^{(2)} a_0^{(2)[3]} + \Theta_{41}^{(2)} a_1^{(2)[3]} + \Theta_{42}^{(2)[3]} a_2^{(2)}
\end{array}\right]
\begin{array}{l}
\leftarrow \text{z for each example for output layer activation unit} \\
\leftarrow \text{z for each example for output layer activation unit} \\
\leftarrow \text{z for each example for output layer activation unit} \\
\leftarrow \text{z for each example for output layer activation unit}
\end{array}
\end{array}
$$

Figure 14: The indexed symbolic representation of the matrix multiplication. The resultant elements in the columns each representing a single example, and the rows being the different activation units of the output layer, with four output units. In a classification problem this would mean four classes/categories. Also notice the [m] superscript index on all the a's in each element is the example number.

Then we element-wise apply the sigmoid function to each of the 12 elements in the z^2 matrix:

```
a3 = sigmoid(z3);
```

This just gives us a 4x3 matrix of the output layer activations (a.k.a. hypothesis) for each of the output units/classes:

$$g(z^{(3)}) = a^{(3)} = \begin{array}{cccc} \text{Example 1 activations} & \text{Example 2 activations} & \text{Example 3 activations} & \\ \downarrow & \downarrow & \downarrow & \\ 0.8887 & 0.8382 & 0.9234 & \leftarrow \text{Activation for 1st Unit of Output Layer, } a_1^{(3)} \\ 0.9074 & 0.8603 & 0.9386 & \leftarrow \text{Activation for 2nd Unit of Output Layer, } a_2^{(3)} \\ 0.9233 & 0.8798 & 0.9509 & \leftarrow \text{Activation for 3rd Unit of Output Layer, } a_3^{(3)} \\ 0.9366 & 0.8969 & 0.9609 & \leftarrow \text{Activation for 4th Unit of Output Layer, } a_4^{(3)} \end{array}$$

Figure 15: The activation values for each of the output units of the network, for each example. If you were doing a for loop over all your examples, this would be a column vector rather than a matrix.

From here, you are just computing the cost function. The only thing to note is that you have to transpose the matrix of y vectors to make sure the element-wise operations you are doing in the cost function line up properly with each example and with the output units.

$$yv = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \xrightarrow{yv'} \begin{array}{ccc} \text{Example 1 y} & \text{Example 2 y} & \text{Example 3 y} \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{array}$$

Figure 16: Transpose of the logical y vectors matrix.

Then we put it all together to compute the cost function:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[-y_k^{(i)} \log((h_\theta(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right],$$

Figure 4: The Multi-Class Logistic Regression Cost Function

```
J = (1/m) * (sum(-yv’ .* log(a3) — ((1 — yv’) .* log(1 — a3))));
J = sum(J);
```

This gives us our cost, notice the double sum to account for summing over all the classes as well as over all the examples. And that’s all folks. The matrix multiplication can make this code very clean and efficient, no need to have for loops slowing things down, but it is essential you know what is happening in matrix multiplication so that you can adjust the matrices appropriately, whether it be order of multiplication, transposing when necessary and adding the bias units to the correct area of the matrix. Once you break it down, it is much more intuitive to grasp and I highly recommend going through an example slowly like this yourself if you are still unsure, it always comes down to some really simple fundamentals.

I hope this was helpful at walking through and demystifying the linear algebra necessary for forward propagation. Also, this is one of my first posts and definitely my first technical post, so any feedback, questions, comments, etc, is greatly appreciated.

)

Machine Learning Neural Networks Artificial Neural Network Artificial Intelligence
Linear Algebra

About Help Legal