NEURAL NETWORK PACKAGE MODULE 2

MATRICIES AND FORWARD PROPAGATION

QMIND EDUCATE

ANDREW FARLEY

TABLE OF CONTENTS

MATRICIES

What is a Matrix?

When we are doing different operations with Neural Networks, such as Forward Propagation, we often have massive amounts of values which would be hard to store individually. This is where the idea of a **matrix** helps us out. A matrix is an **n-dimensional** structure which contains many values called **elements**, and is very similar to an *array* in programming. For this package we will mostly be working with 2-D matrices (which are a lot easier to comprehend then, say, 10-D matrices). By utilizing matrices and their associated operations, we will make the process of forward and back propagation much more efficient and easier to compute.

Dimensions

Every matrix has an associated **set of dimensions**. In our case, that set will always be of size 2 because we are working with 2 dimensional arrays. These dimensions are usually written in the form *n x m*. Let's suppose we had a matrix that looked like this:

$$\begin{bmatrix} 2 & 1 & 5 \\ 4 & 2 & 8 \end{bmatrix}$$

We call this a **2x3** matrix because it has 2 rows and 3 columns. The *n* always refers to the **number of rows** (or the height) and the *m* always refers to the **number of columns** (or the width). Make sure you remember this as it will be important later.

Matrix Operations

Just like regular numbers, matrices have **operations** associated with them. These *operations* may be thought of as addition, subtraction, and dot products (the dot product being analogous to multiplication). Any number can be thought of as a 1x1 matrix and can have operations applied to it such as addition, subtraction, and multiplication by other numbers (or matrices!). You can think of operations as basic math with normal numbers, but with **larger dimensions** associated (n x m dimensions to be exact).

ADDING AND SUBTRACTING

When we add and subtract matrices we will just add/subtract any numbers that have **the same position** in each of the matrices. For this reason, any matrices that are using these operations must have the **same dimensions**. Let's take the matrix we had before and add this new matrix to it:

$$\begin{bmatrix} 2 & 1 & 5 \\ 4 & 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & -3 & 1 \\ -6 & 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & -2 & 6 \\ -2 & 9 & 9 \end{bmatrix}$$

Looking above and following the process of adding the two matrices by adding numbers in the same position, you can see that 4 + (-6) = -2. Note that the number -2 also is placed in the same position as the 4 and (-6) added initially! This is completed for each of the matrix positions until all positions have been added. This same process is followed for subtraction of matrices.

As you can also see, the resulting matrix will have the **same dimensions** as the previous two. Remember, only elements at the **same position** in the matrix will affect each other with addition and subtraction.

An aside:

To show we have simply expanded the definition of adding/subtracting when completing addition/subtraction of matrices, take the equation:

$$2 + 3 = 5$$

If we just make each number in the equation a 1x1 matrix it will still return the same result! As was stated before, normal numbers can just be thought of as 1x1 matrices. These addition and subtraction operations will still hold true and be valid for this, and any size of matrix.

THE DOT PRODUCT

Instead of multiplying, matrices have an operation called the **dot product**. While it may seem daunting at first, some practice will make you a pro. This online resource will let you visualize the process one would take to perform the dot product between two matrices. I recommend you use it as you follow along this section of the module to clarify any questions you have about what is happening.

Let's take two new matrices:

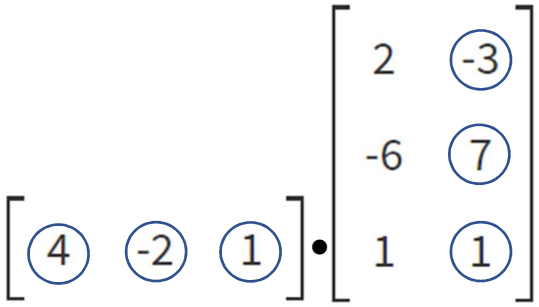
$$\begin{bmatrix} 4 & -2 & 1 \end{bmatrix} \bullet \begin{bmatrix} 2 & -3 \\ -6 & 7 \\ 1 & 1 \end{bmatrix}$$

Notice that in the example above, the matrix dimensions aren't the same. When using the dot product with two matrices of dimensions $n_1 \times m_1$ and $n_2 \times m_2$, m_1 must equal n_2 (the **inner dimensions must match**). This because we will be doing a summation of the products of individual rows and columns. These one-dimensional vectors must be the *same size* to do a proper summation of products. To compute the dot product we are going to product-sum every column-row vector pair we can without repeating any. The resulting number will be the new value at the intersecting coordinate in the resultant matrix. For the example above we would first determine the dimensions of the new matrix. This dimension should be $n_1 \times m_2$, or *1x2* in our case. We then take the row vector [4, -2, 1] and the column vector [2, -6, 1]. We notice that their summed product hasn't been computed yet, so we calculate it.

$$\begin{bmatrix} \textcircled{4} & \textcircled{-2} & \textcircled{1} \end{bmatrix} \bullet \begin{bmatrix} \textcircled{2} & -3 \\ \textcircled{-6} & 7 \\ \textcircled{1} & 1 \end{bmatrix}$$

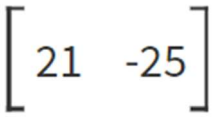
$$New\ Value = 4 * 2 + -2 * -6 + 1 * 1 = 21$$

Therefore the resultant vector so far looks like [21, ?]. We now compute the only other row-column pair, row vector [4, -2, 1] and column vector [-3, 7, 1].



$$New\ Value = 4 * -3 + -2 * 7 + 1 * 1 = -25$$

Therefore the final resultant vector is:



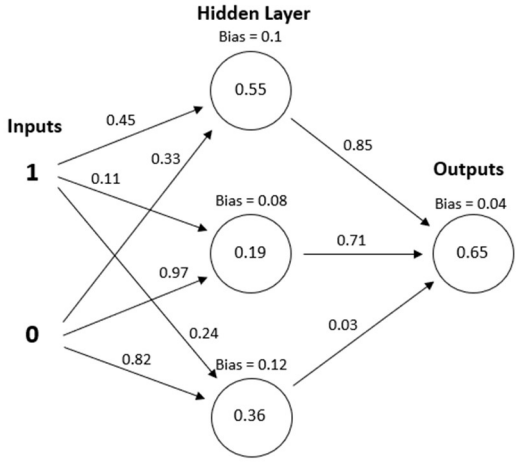
If you setup the matrices here with the same numbers as above, you can see this happen in action. Play around with different matrices to see how it will react. Come up with different dot product problems and see if you can figure out the answer. Use the site to double check!

HOW MATRICES APPLY TO NEURAL NETWORKS

So, what does this have to do with neural networks? How are matrices used to make things more efficient and easier to work with? The idea of computing a sum of products should be setting off alarm bells in your head, where did we just use that? Let's take the set of inputs as a row matrix (meaning it will have a dimension *1 x n*) and the set of weights as a matrix where the column number represents a neuron in the current layer and the row number represents a connected neuron from the previous layer. The value at the intersection of these will be the weight between them. If we compute the **dot product** of these two matrices and **add the bias**, represented as a row vector, we will get what values each of those neurons will output to the next layer! Of course, we will need to apply the activation function to every value in the output.

Matrix Multiplication and Layer Output

Let's look at a neural network similar to one of the last modules and assume it has the sigmoid activation function:



Looking at the different components as matrices:

Inputs	Hidden's Weights	Hidden's Bias	Output's Weights	Output's Bias
$\begin{bmatrix} 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.45 & 0.11 & 0.24 \\ 0.33 & 0.97 & 0.82 \end{bmatrix}$	$\begin{bmatrix} 0.1 & 0.08 & 0.12 \end{bmatrix}$	$\begin{bmatrix} 0.85 \\ 0.71 \\ 0.03 \end{bmatrix}$	$\begin{bmatrix} 0.04 \end{bmatrix}$

We can now compute the output of the first layer using the formula:

$$Layer\ Output = f(Layer\ Input \bullet Layer\ Weights + Layer\ Bias)$$

Dot product the inputs with the weights:

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \bullet \begin{bmatrix} 0.45 & 0.11 & 0.24 \\ 0.33 & 0.97 & 0.82 \end{bmatrix} = \begin{bmatrix} 0.45 & 0.11 & 0.24 \end{bmatrix}$$

Add that result with the bias:

$$\begin{bmatrix} 0.45 & 0.11 & 0.24 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.08 & 0.12 \end{bmatrix} = \begin{bmatrix} 0.55 & 0.19 & 0.36 \end{bmatrix}$$

Apply the activation function to every value and get the output matrix of the hidden layer:

$$f(0.55) = 0.63 \quad f(0.19) = 0.55 \quad f(0.36) = 0.59$$

$$\begin{bmatrix} 0.63 & 0.55 & 0.59 \end{bmatrix}$$

We can now feed this result to the next layer and keep **propagating** our input throughout the network. This process is exactly the same as the one described in the first module, just in a data structure that computers can understand easily and compute efficiently. If you are still confused on how matrices work, I urge you to check out this video by The Coding Train on YouTube. It is very well done and one of the best resources I've found on matrices.

An Aside:

We want to represent our networks as matrices for multiple reasons. The first being that they are more compact. Once you understand matrices better, they can also be easier to read. Instead of having to draw out an entire network someone can just give you a list of matrices and you will instantly have an idea of how to process the inputs and produce outputs. You also can see the number of layers as well as the input and output sizes. The second, and arguably more important, reason is that computers are very good at doing matrix math. The idea of a matrix (or vector/array) is something that computers have been optimized to work with. The screen you are viewing this on right now does thousands of matrix operations a second to produce what is to be displayed. Instead of looping through many lists of numbers, we can just give the computer the matrix computations above to calculate and it will do it quickly. All Neural Network libraries are built upon the concept of the matrix, make sure you understand them!

NEURAL NETWORK OUTPUT

After this massive chain of multiplying input and weight matrices and adding the bias we should get an output on the last layer. From the example before, the **final output** should look like:

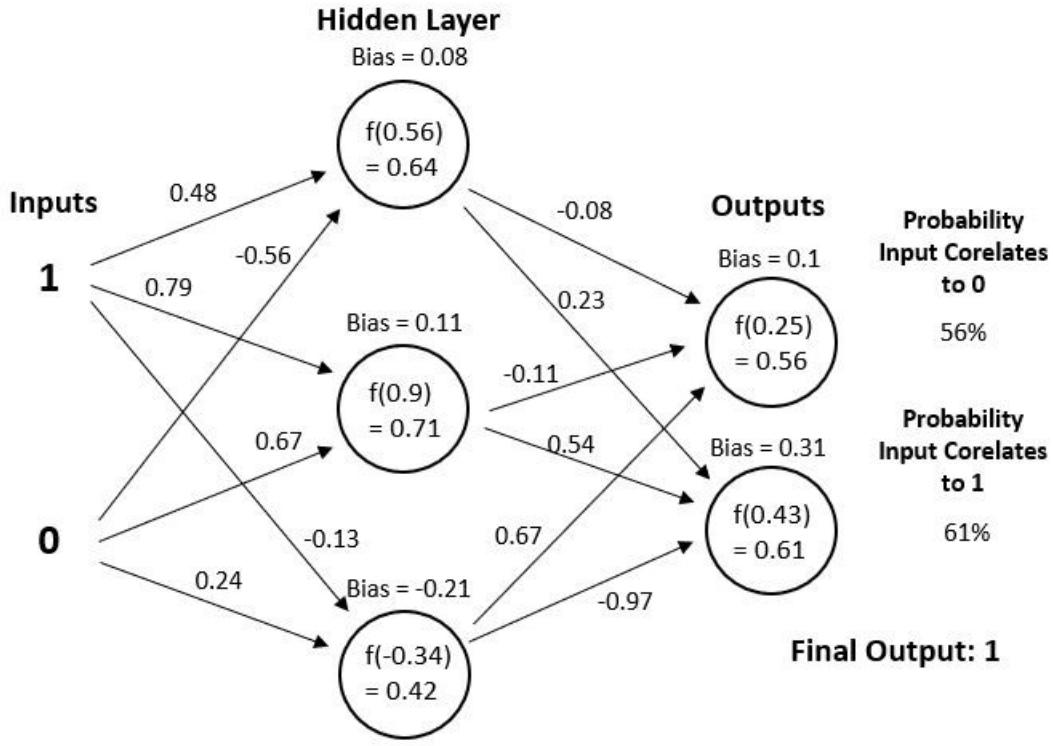
$$\begin{bmatrix} 0.73 \end{bmatrix}$$

In this case we only have **one output**. The value given would represent a **confidence level** of what the output is. When we are doing something, like using the network to classify images, we expect the possible output to be from a predefined set. For example, let's say we had a bunch of pictures that were cats, dogs, and turtles and we wanted them classified. We would train a network using back propagation (discussed more in-depth next module) and pass our images through it. The

network used for this would have three outputs: one for the confidence it's a dog, one for the confidence it's a cat, and one for the confidence the image is a turtle. By using the *sigmoid function*, the confidence levels are **independent** of one another. This can be both good and bad. If we get a high confidence for two different values (i.e. the network is 98% confident the input is a dog and 97% confident it's a cat) then it might get confused and output the wrong answer. If every layer output is normalized in some fashion, this will cause a bigger discrepancy between the confidences and lead to a more accurate result. As stated in the previous module, we can use a different activation function to change this, SoftMax is a good example. Because our outputs are independent, we select the output that has the highest confidence level as our final result. In other words, we find the index of the biggest value in the output matrix, which is how our network will always give one final output.

'AND' OPERATOR EXAMPLE

Let's take the neural network that models the 'AND' operator from the last module.



We can setup a table of the different matrices for each layer just like we did for the above network.

Inputs	Hidden's Weights	Hidden's Bias	Output's Weights	Output's Bias

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0.48 & 0.79 & -0.13 \\ -0.56 & 0.67 & 0.24 \end{bmatrix} \quad \begin{bmatrix} 0.08 & 0.11 & -0.21 \end{bmatrix} \quad \begin{bmatrix} -0.08 & 0.23 \\ -0.11 & 0.54 \\ 0.67 & -0.97 \end{bmatrix} \quad \begin{bmatrix} 0.1 & 0.31 \end{bmatrix}$$

Notice how if you look at the dimensions of the input layer and the hidden layer's weights (*1x2* and *2x3* respectively) the inner dimensions are the same, they are both 2! We can now multiply these matrices to produce a *1x3* matrix which is the same dimension of our bias, allowing us to add them with no issues. This will continue for the rest of the network. The hidden layer's output and output layer's weights are *1x3* and *3x2* respectively which will produce a *1x2* matrix, the same size as the output layer's bias allowing us to add them. Seeing that all matrices will not cause issues with the forward propagation, we can compute the output of every layer, left-to-right, to produce what our network is thinking the input [1 0] means! To begin we multiply the input layer with the hidden layer's weights. This should produce an output of:

$$\begin{bmatrix} 0.48 & 0.79 & -0.13 \end{bmatrix}$$

We add the bias and apply the activation function to every number to produce the hidden layer's output:

$$\begin{bmatrix} 0.64 & 0.71 & 0.42 \end{bmatrix}$$

The hidden layer output is just the input to the next layer, the output layer. We can multiply this matrix by the output layer's weights to produce this new matrix:

$$\begin{bmatrix} 0.15 & 0.12 \end{bmatrix}$$

Again, adding the bias and applying the activation function will give us our final network outputs:

$$\begin{bmatrix} 0.56 & 0.61 \end{bmatrix}$$

You might not realize this, but you already know what this matrix means! The first number is the *confidence* the network has the input results in a 0, and the second number is the confidence the network has the input results in a 1. Choosing the *maximum confidence* (just like you would when you make a decision) our network's final decision is that this input should result in a 1. As stated before, this is *incorrect* as 1 and 0 is 0. As such, the network must be trained so it produces the correct output as often as possible.

OTHER RESOURCES

Do look at the resources I mentioned throughout this module; it is really helpful having another perspective and a set of examples to work with as you try and understand these concepts. The video series I mentioned last module covers everything in this module and the next one. They are very good and so are many others on YouTube and the internet alike.

REFERENCES

[1] G. Sanderson, "Deep Learning (Video Series)," 3blue1brown, 2017.

[2] D. Shiffman, "Introduction to Neural Networks (Video Series)," Coding Train, 2017.