Neural Networks II: the Neuron and Forward Propagation

Vinay Bhaip[*]

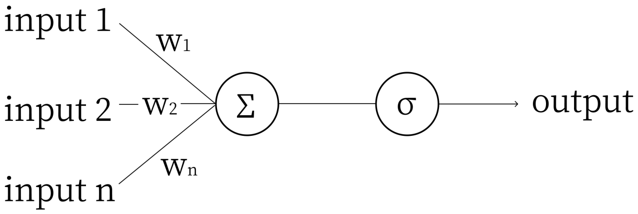
December 2019

1 Introduction

In the last lecture, we discussed the basis of neural networks: the perceptron. This lecture looks at the neuron, which is the basic unit of modern neural networks. This will cover how to pass in data through a neural network in a process known as forward propagation.

2 The Neuron

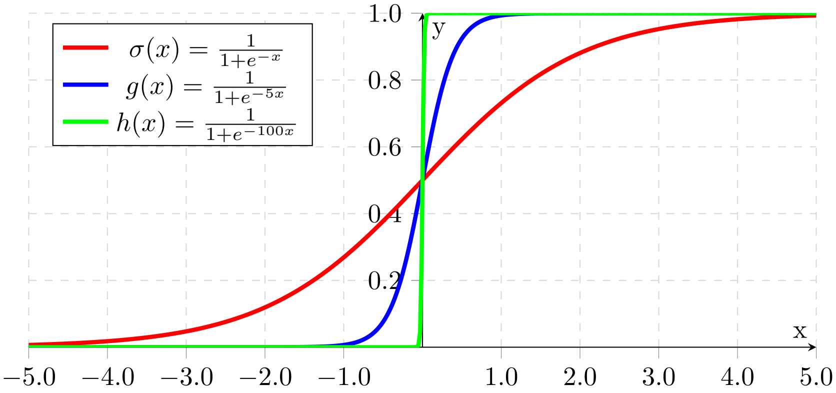
A neuron is similar to the perceptron in every way but one: the activation function. Recall that the activation function is what we pass our output into after multiplying each input by a weight and summing them and a bias together. The neuron looks as follows:



In the perceptron, we used the step function activation, which meant that if the input into it was negative, it would output 0 and if the input into it was positive, it would output 1. The new activation function is known as the Sigmoid function, represented by σ. The mathematical equation for this is $\sigma(x) = \frac{1}{1+e^{-x}}$.

Why is it better to use Sigmoid over the step function for our activation function? The main advantage of Sigmoid is that it is differentiable, which is crucial for letting our models learn in backpropagation.

[*] Based off Nikhil Sardana's Forward/Backpropagation Lecture



Notice how when the coefficient for x approaches infinity, the function approaches the step function we used in the perceptron. There are other activation functions like hyperbolic tangent and the Rectified Linear Unit, but we'll discuss those later on.

Besides this change in the activation function, everything else in the neuron is the same as perceptrons.

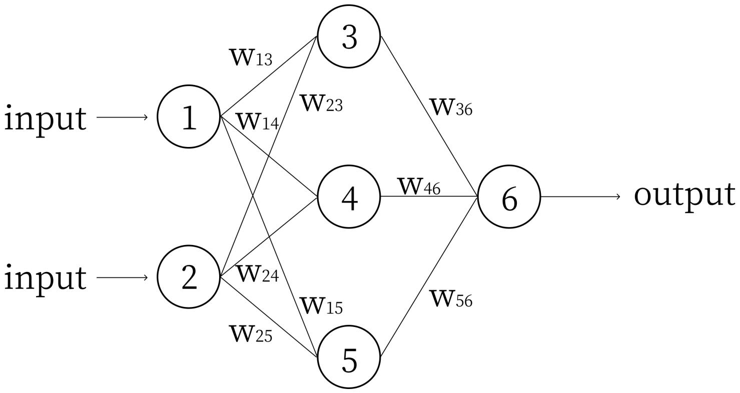
3 Forward Propagation

Now that we know how a neuron works, we can combine multiple of these neurons together, similar to what we did with the Multi-Layer Perceptron in the last lecture, to create a neural network. There are two main parts to learn about a neural network: forward propagation and backpropagation. Forward propagation is the way that a neural network computes its output. It passes the original input through layers of neurons until it reaches the model's prediction.

3.1 Non-Vectorized Forward Propagation

Non-vectorized forward propagation just means that we are trying to find the output of a neural network without the use of matrices. Matrices are extremely helpful in practice, but it is helpful to see how non-vectorized forward propagation works to understand the process conceptually.

For forward propagation, we must compute all the values of the neurons in the second layer before we begin the third, but we can compute the individual neurons in any given layer in any order. Consider the following network:



At each node where you see a number, there is also a bias. As a side note, you may wonder why we need a bias if we'll be adjusting the weights anyways. A good analogy to think of this is to compare this to a perceptron that has the line equation $y = mx + b$. Without the bias, all equations would go through the origin, so it is important to include.

For the sake of this lecture, we denote the weight w_{ab} as the weight connecting the input node a to the output node b.

We denote the value of node i as n_i, and the bias of node i as b_i. To find the value of a given neuron, we simply sum up the weighted values of the inputs and add the bias. We then pass this into our sigmoid function to get the output. Computing the network using these variables, we get:

$$n_3 = \sigma(w_{13}n_1 + w_{23}n_2 + b_3)$$

$$n_4 = \sigma(w_{14}n_1 + w_{24}n_2 + b_4)$$

$$n_5 = \sigma(w_{15}n_1 + w_{25}n_2 + b_5)$$

$$n_6 = \sigma(w_{36}n_3 + w_{46}n_4 + w_{56}n_5 + b_6)$$

Continuing this example of forward propagation, let's assign some numbers and compute the output of this network. Let $n_1 = 0.2$ and $n_2 = 0.3$. Let $w_{13} = 4, w_{14} = 5, w_{15} = 6, w_{23} = 5, w_{24} = 6, w_{25} = 7, w_{36} = 9, w_{46} = 10$ and $w_{56} = 11$, just so they are easy to remember. Let all the biases $b_{3..6} = 1$ (input nodes do not have biases, the "input nodes" are simply values given to the network). In practice, weights and biases of a network are initialized randomly between -1 and 1. Given these numbers, we compute:

$$n_3 = \sigma(4 * 0.2 + 5 * 0.3 + 1) = \sigma(3.3) = 0.964$$

$$n_4 = \sigma(5 * 0.2 + 6 * 0.3 + 1) = \sigma(3.8) = 0.978$$

$$n_5 = \sigma(6 * 0.2 + 7 * 0.3 + 1) = \sigma(4.3) = 0.987$$

$$n_6 = \sigma(9 * 0.964 + 10 * 0.978 + 11 * 0.987 + 1) = \sigma(30.313) \approx 1$$

This example actually illustrates one of the weak points of the Sigmoid function: it quickly approaches 1 for large numbers. We will discuss the implications of this as well as the solution for this later on. The reason for using the Sigmoid function will be shown in the section on backpropagation.

3.2 Vectorized Forward Propagation

Now that we understand how forward propagation works, we can see if we can find a more succinct way to compute a network's output. Let's look again at these nodes of the network:

$$n_3 = \sigma(w_{13}n_1 + w_{23}n_2 + b_3)$$

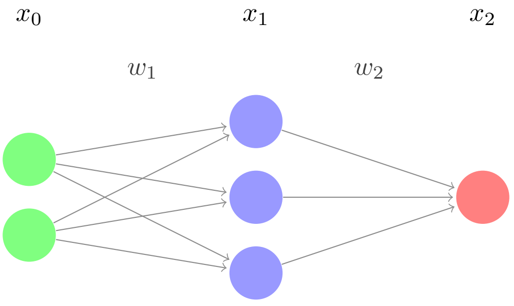
$$n_4 = \sigma(w_{14}n_1 + w_{24}n_2 + b_4)$$

$$n_5 = \sigma(w_{15}n_1 + w_{25}n_2 + b_5)$$

We can rewrite this as

$$\begin{bmatrix} n_3 \\ n_4 \\ n_5 \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \\ w_{15} & w_{25} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix} \right)$$

The matrix with the weights and the matrix with the neuron values are multiplied together using matrix multiplication. The advantage of this is that this process can now be parallelized, since computing each item in the resulting matrix does not depend on the computation of another item. Notice how the nodes in each layer of the network are in their own column vector, in the order they appear. Let's relabel this network by layers:



Here, x_0 and x_2 represent the input and output layers, and x_1 is the middle layer (called a hidden layer). Mathematically speaking, these are represented as column vectors of dimension $n \times 1$, where n is the number of nodes in the layer. Thinking back to the non-vectorized network in Section 3.1,

$$x_0 = \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} \qquad x_1 = \begin{bmatrix} n_3 \\ n_4 \\ n_5 \end{bmatrix} \qquad x_2 = \begin{bmatrix} n_6 \\ n_7 \end{bmatrix}$$

w_1 and w_2 are the weight matrices. Again, referring back to the non-vectorized network in Section 3.1, w_1 corresponds to

$$\begin{bmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \\ w_{15} & w_{25} \end{bmatrix}$$

and w_2 refers to

$$\begin{bmatrix} w_{36} \\ w_{46} \\ w_{56} \end{bmatrix}$$

Each layer (except the input) also has a bias vector, which has the same dimension as the layer itself (each node has a bias). Again thinking back to the non-vectorized network in Section 3.1, we define b_1 to be

$$\begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

and b_2 to be

$$\begin{bmatrix} b_6 \\ b_7 \end{bmatrix}$$

We can now rewrite the forward propagation formula in a far more compact form. In any n layer network, for a given layer x_{i+1} (assuming $0 \leq i < n-1$):

$$x_{i+1} = \sigma(w_i x_i + b_{i+1})$$

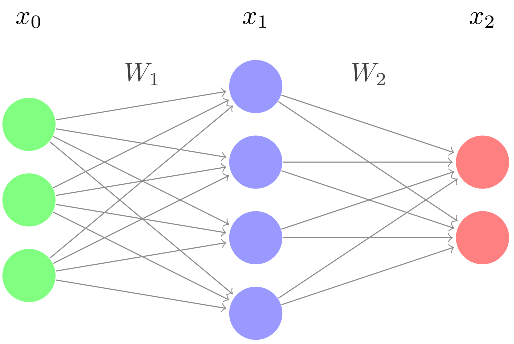
4 Introduction to Backpropagation

This lecture will not go into the specific math behind backpropagation, but will attempt to give a conceptual understanding behind it.

Backpropagation is how neural networks actually learn. This is the process in which neural networks are able to adjust weights and biases. Backpropagation attempts to minimize the error function of the model. What does we mean by this? When we build a neural network, we often have thousands of data points, which each have features and some ground truth label. We can pass all these data points into the network and see whether our model was right or wrong. The degree to which our model was wrong is known as the error. Backpropagation attempts to see how modifying the weights and biases can reduce the error and ultimately make the model perform as accurately as possible.

4.1 Error

Consider the following network:



For the input x_0, let y represent the target vector, or the ground truth. x_2 is our model's prediction. We define the error as

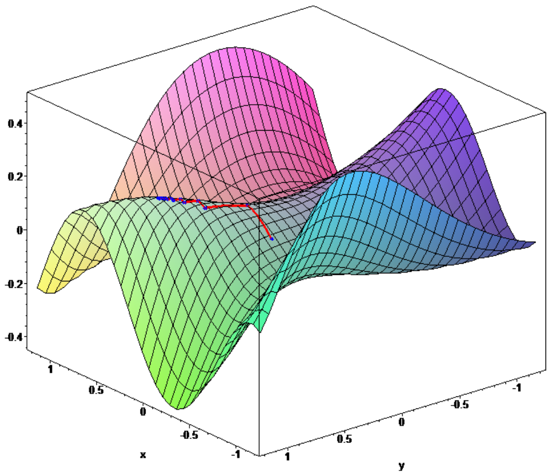
$$E = \frac{1}{2}||x_2 - y||^2$$

Essentially, this is the magnitude of the difference between the target and the network's output. The reason we choose to define the error in this way is so that the derivative of E with respect to y is just $x_2 - y$. In order for a network to become more accurate, we want to minimize this error.

Let's think of E as a function. Only x_2 can vary, and we can only control this by changing the weight matrices and the bias matrix. Thus, for a neuron with n weights and a bias, the error can be graphed as an $n + 2$ dimensional function. This is because there are n weights, 1 bias, and an original input (X_0). For the network depicted above, there are 20 total weights ($3*4+4*2 = 20$) and 6 biases to determine the error, which means that there are a lot of dimensions. If we can find the correct values for these weights and biases, we can get to the minimum of this function. This means we will have minimized the error and trained the network.

4.2 Gradient Descent

Most of the time, we will be dealing with many dimensions, so let's pretend we have a very simplistic error function that only deals with three dimensions. How do we get to the minimum? We use gradient descent, of course!

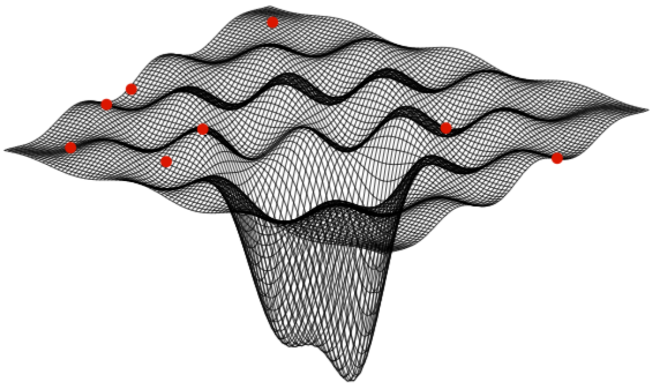


A multi-dimensional function. Look how you can see the minimum!

Gradient descent is simple: Starting at some point, we move in the direction of steepest decline for a certain length. Then, at our new point, we again compute the direction of steepest decline, and move in that direction for a certain length. We repeat this process over and over until every single direction is an incline, at which point we are at the minimum. One way you can think of this is as if we were to drop a ball at our location and watch it roll down until we reach a minimum.

This has three issues. First, how do we know how long our steps are? Take a step too long, and we could overshoot the minimum. Take a step too short and it will take us many steps to reach the minimum. This is essentially the same thing as the learning rate α we used in the perceptron lecture. The step length is actually just a constant set by the programmer, and normally ranges from 0.1 to 0.0001. Adjusting the constant to get the best result is an important practical topic for getting the best result, and we will discuss this in a later lecture. For now, just know its a constant.

Secondly, gradient descent is not guaranteed to get us to a minimum. What if there are multiple minima, and we just happen to land in a local minimum, like the many in the function below?



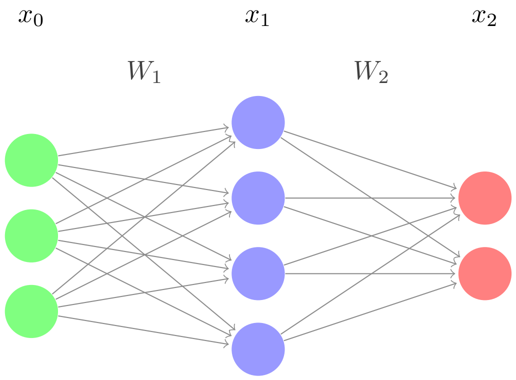
Getting out of local minima to reach the global minimum is another important machine learning topic. Different optimizers can help the network pop out of local minima using momentum, but this topic is complex and modern, so it is covered in a later lecture. For the purposes of explaining gradient descent, we'll just pretend we're working with an error function with one minimum.

The third final question is how we know which direction is the direction of steepest descent. To do this, we must calculate the gradient. We will cover the mathematics of this next time.

A quick note: you may be asking why we can't just calculate the minimum instead of going through gradient descent. With the error function, we only know information of what the derivative of the function is at that one point. Additionally, it is complicated to find the minimum given many dimensions that are given with a complicated neural network.

5 Practice Problems

1. Given the following network:



with the weight matrices, bias vectors and input as follows:

$$W_1 = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 1 & 2 \\ 3 & 5 & 1 \\ 2 & 3 & 4 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 1 & 4 & 2 & 2 \end{bmatrix}$$

$$x_0 = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \qquad b_1 = \begin{bmatrix} 4 \\ 1 \\ 1 \\ 2 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Instead of using the Sigmoid activation function, use a linear function $y = x$, which always has a derivative of 1. Compute the output of one forward pass:

$$t = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \qquad \alpha = 0.1$$

2. Write out the forward propagation algorithm in Python. Use the Numpy library for matrices. This will be really helpful for the neural network competition later on, so we highly recommend that you get this working.