Neural Networks III: Backpropagation

Vinay Bhaip*

December 2019

1 Introduction

Last week, we went over how forward propagation in neural networks works. Forward propagation, as you should recall, is the process in which we feed a data point into a neural network and see what our network classifies the data point as. The next question then becomes how do we create the model to accurately classify the data points.

In this lecture, we'll go over how backpropagation works. Backpropagation is how neural networks are able to update weights and biases to optimize their ability to classify data.

The first section in this lecture was covered in the last lecture as well, but is in this lecture as a conceptual review for the math of backpropagation.

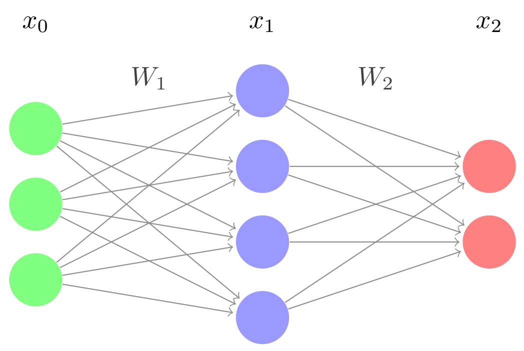
2 Introduction to Backpropagation

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.

2.1 Error

Consider the following network:

*Based off Nikhil Sardana's Forward/Backpropagation Lecture



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

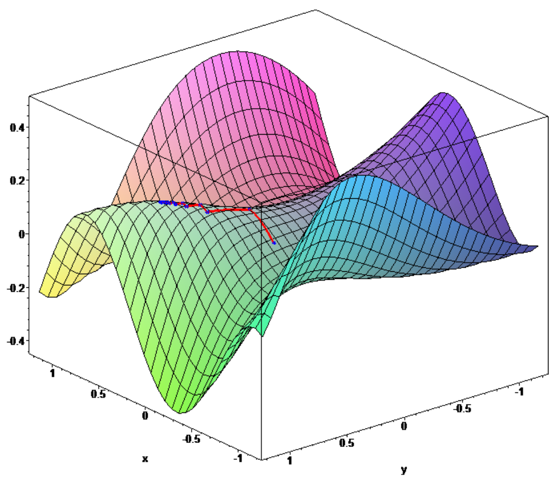
$$E = \frac{1}{2}||x_2 - y||^2$$

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.

2.2 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!

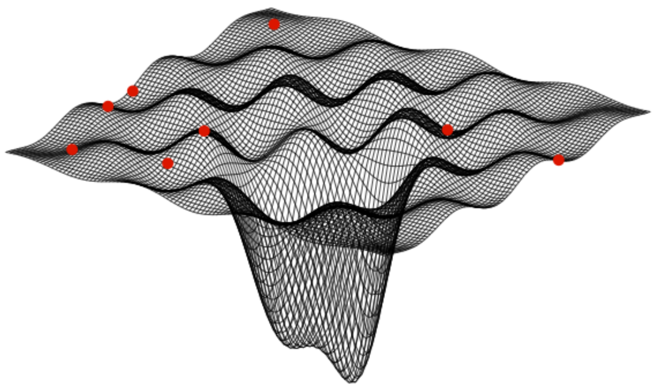


A multi-dimensional function. Look how you can see the minimum!

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 that it's 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?



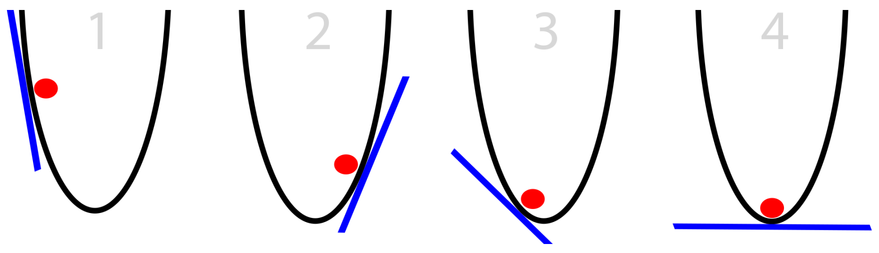
Getting out of local minima to reach the global minimum is another important machine learning topic. Different optimizers can help the network pop out of local minima using momentum, but this topic is complex and modern, so it is covered in a later lecture. For the purposes of explaining gradient descent, 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.

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.

2.3 Gradients

The gradient is an extremely useful technique to find the direction of steepest ascent. Let's consider a simple two-dimensional parabola:



From elementary calculus, we know that:

$$f(x) = x^2$$
$$f'(x) = 2x$$

The derivative gives us the instantaneous rate of change for any x. If we have a function in terms of x and y, we can take the derivative of $f(x, y)$ with respect to x to find the rate of change in the x direction, and the derivative with respect to y to find the rate of change in the y direction. These are called partial derivatives. We treat the other variables like we would any other constant.

Let's do an example. Given $f(x, y) = 2x^2 + 3xy + y^3$, the partial derivatives are:

$$\frac{\partial f}{\partial x} = 4x + 3y$$

$$\frac{\partial f}{\partial y} = 3x + 3y^2$$

The gradient of $f(x, y)$, or $\nabla f(x, y)$ is just the vector:

$$(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

For our example, the gradient is:

$$(4x + 3y, 3x + 3y^2)$$

This is the direction of steepest ascent. This lecture will not go into the mathematics as to why this is true, but we strongly encourage you to look into why this is the case.

With backpropagation, we want to use gradient descent. The gradient itself points in the direction of steepest ascent, so naturally taking the negative of the gradient points in the direction of steepest descent.

Now that we've got that covered, in order to find the minimum of a multi-dimensional function, we just need to compute the gradient, move in the negative of that direction for a certain length, and repeat until the gradient is 0. There remains one more problem: how do we compute the gradient for our network? Our function is

$$E(W, b) = \frac{1}{2}||o - t||^2$$

Where o is the network output at t is the target. Since the error is in terms of the weights and biases, that means that we need to compute:

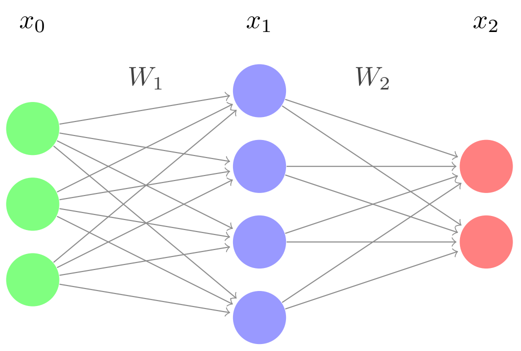
$$(\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2}, ..., \frac{\partial E}{\partial b_n})$$

This is why backpropagation is a fundamental concept in machine learning. It allows us to compute this gradient in a computationally efficient manner.

2.4 Vectorized Backpropagation

Non-vectorized backpropagation becomes extremely convoluted, so we'll be looking at vectorize backpropagation to begin with. Remember, vectorized backpropagation means that we will be using matrices.

Consider the following network:



Ignoring biases (which we will see follow a relatively simple rule), we know from forward propagation that:

$$x_1 = \sigma(W_1 x_0)$$

$$x_2 = \sigma(W_2 x_1)$$

And the error is, assuming some 2×1 target vector y:

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

Let's first take the partial derivative of E with respect to W_2. This is just like taking a normal derivative (using the chain rule).

$$\frac{\partial E}{\partial W_2} = (x_2 - y)\frac{\partial(\sigma(W_2 x_1))}{\partial W_2}$$

$$\frac{\partial E}{\partial W_2} = [(x_2 - y) \odot \sigma'(W_2 x_1)]\frac{\partial W_2 x_1}{\partial W_2}$$

Here, $\odot$ is the Hadamard product, or element-wise multiplication. (Remember, these are all vectors). For the sake of simplification, lets define

$$\delta_2 = (x_2 - y) \odot \sigma'(W_2 x_1)$$

Then, we can rewrite the partial as

$$\frac{\partial E}{\partial W_2} = \delta_2 \frac{\partial W_2 x_1}{\partial W_2} = \delta_2 x_1^T$$

Note that x_1^T means that the x_1 vector has been transposed (i.e. it is a row vector). This is essential for the dimensions to work out, which we can check now.

Since the whole point is to update the weights by some factor every time we backpropagate in the direction of fastest descent to minimize the error, we want to subtract the partial matrix (since it is in the direction of fastest ascent):

$$W_i = W_i - \alpha \frac{\partial E}{\partial W_i}$$

where alpha is the learning rate. This requires $\frac{\partial E}{\partial W_i}$ to be the same dimensions as W_i. Using W_2 as an example, we know that

$$x_2 = \sigma(W_2 x_1)$$

where x_2 is a 2×1 vector, x_1 is a 4×1 vector, so W_2 is a 2×4 matrix. Thus, both $\frac{\partial E}{\partial W_i}$ and $\delta_2 x_1^T$ are also 2×4 matrices. Since $\delta_2 = (y - \sigma(W_2 x_1)) \odot \sigma'(W_2 x_1)$, and we know y is a 2×1 matrix, δ_2 has dimensions 2×1. If δ_2 is 2×1, then it must be multiplied by a 1×4 vector to create a 2×4 matrix. Since x_1 is 4×1, it must be transposed to become 1×4.

Let's continue to the next weight matrix.

$$\frac{\partial E}{\partial W_1} = (x_2 - y) \frac{\partial(\sigma(W_2 x_1))}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = [(x_2 - y) \odot \sigma'(W_2 x_1)] \frac{\partial W_2 x_1}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = \delta_2 \frac{\partial W_2 x_1}{\partial W_1} = W_2^T \delta_2 \frac{\partial x_1}{\partial W_1}$$

Substituting in for x_1, we get:

$$\frac{\partial E}{\partial W_1} = W_2^T \delta_2 \frac{\partial(\sigma(W_1 x_0))}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = [W_2^T \delta_2 \odot \sigma'(W_1 x_0)] \frac{\partial W_1 x_0}{\partial W_1}$$

Again, we simplify this:

$$\delta_1 = W_2^T \delta_2 \odot \sigma'(W_1 x_0)$$

and we finish with

$$\frac{\partial E}{\partial W_1} = \delta_1 \frac{\partial W_1 x_0}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = \delta_1 x_0^T$$

We can generalize this for any layer. The only difference is the delta for the last layer:

$$\delta_L = (x_L - y) \odot \sigma'(W_L x_{L-1})$$

The delta for every other layer is:

$$\delta_i = W_{i+1}^T \delta_{i+1} \odot \sigma'(W_i x_{i-1})$$

And the gradient for every weight matrix are calculated and the weight matrices are updated as follows:

$$\frac{\partial E}{\partial W_i} = \delta_i x_{i-1}^T$$

$$W_i = W_i - \alpha \frac{\partial E}{\partial W_i}$$

For biases, the rule is simpler:

$$b_i = b_i - \alpha \delta_i$$

That is the essence of backpropagation. Note that these formulas work for any activation function. The reason we use sigmoid to teach this is because its derivative is fairly straightforward:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

3 Conclusion

The past three lectures should give you a solid foundation on how neural networks work. The best way to fully understand these concepts is through practice. For this reason, there are two things we recommend you do:

1. Complete the problem set on neural networks. This is due next week.
2. Work on the Kaggle competition (https://www.kaggle.com/c/nn4) to create a neural network from scratch to classify images of hand-written digits. This is a famous dataset known as MNIST. Since this is the most hands-on coding competition you'll be doing, you'll have much more time to work on it. The competition is due after break (January 15th), though you shouldn't need to work on it during break to get it done.