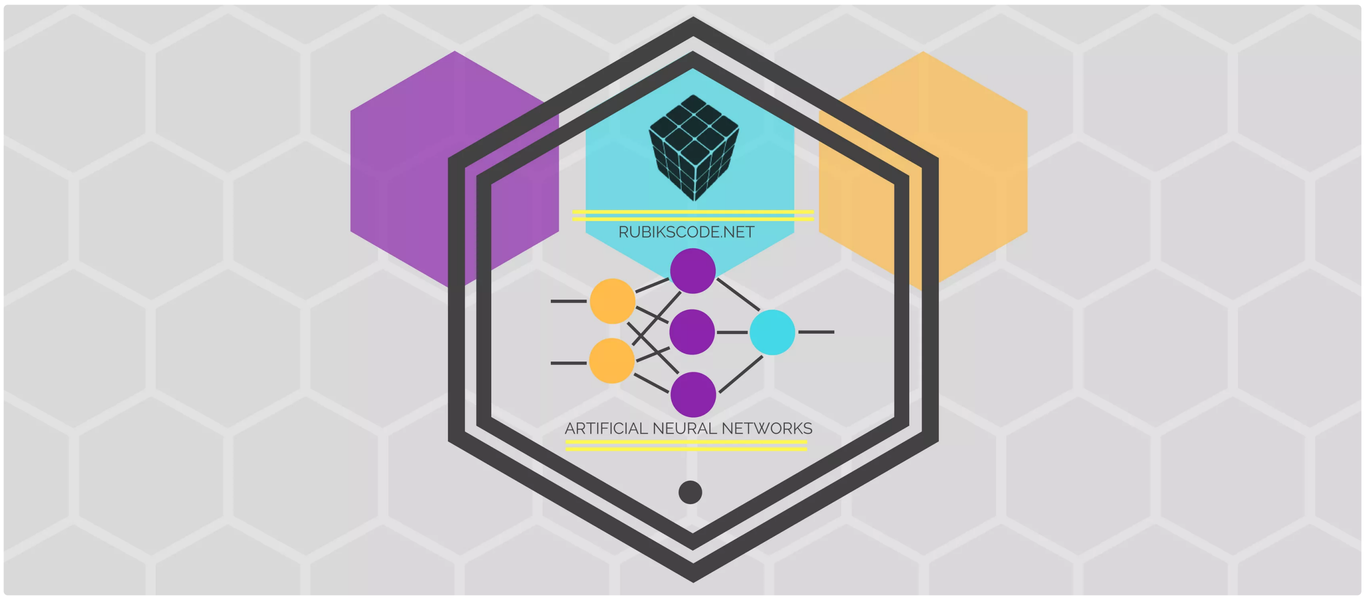
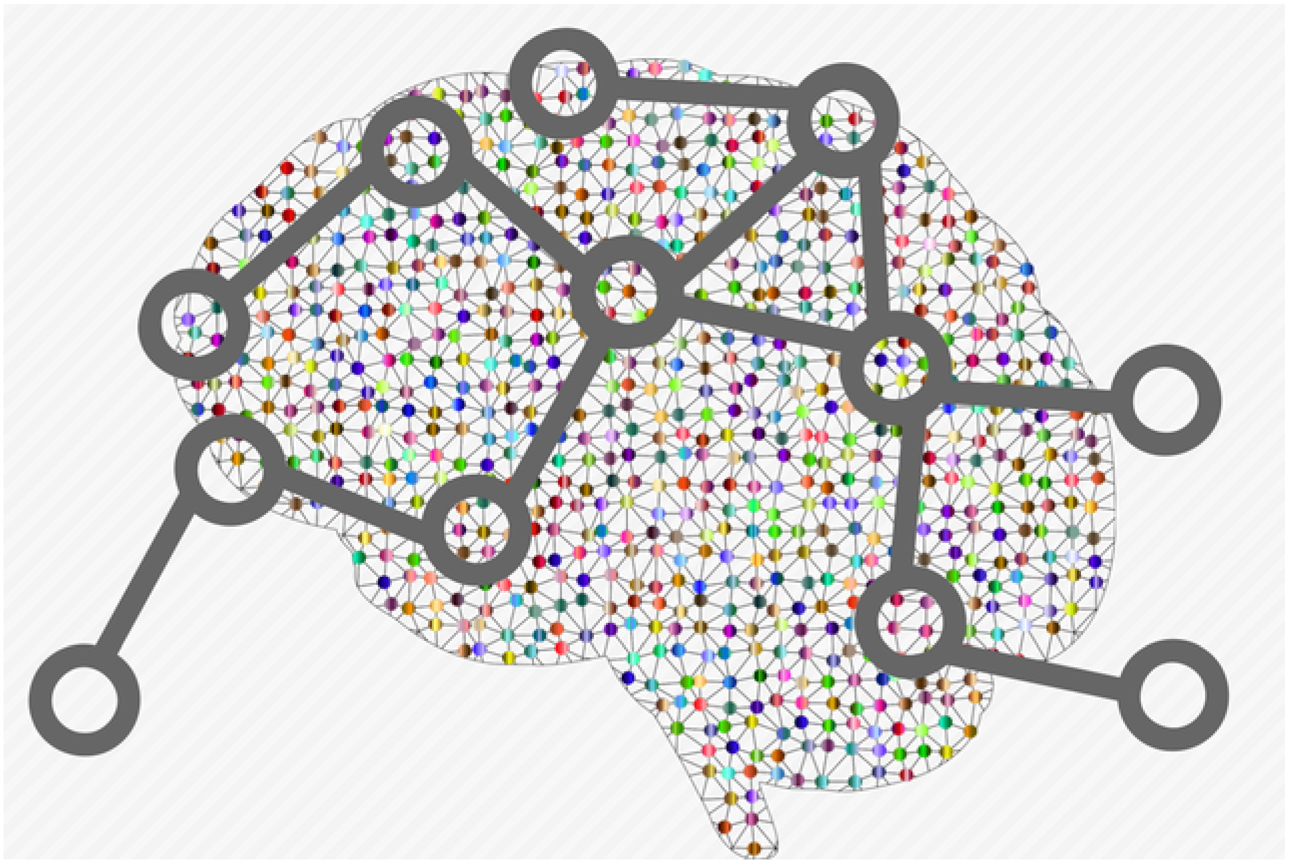
RUBIK'S CODE Freedom. Wisdom. Excellence.

Backpropagation Algorithm in Artificial Neural Networks

JANUARY 22, 2018 — 3 COMMENTS



In the **previous article**, we covered the learning process of ANNs using gradient descent. However, in the last few sentences, I've mentioned that some rocks were left unturned. Specifically, explanation of the backpropagation algorithm was skipped. Also, I've mentioned it is a somewhat complicated algorithm and that it deserves the whole separate blog post. So here it is, the article about backpropagation! Don't get me wrong you could observe this whole process as a black box and ignore its details. You will still be able to build Artificial Neural Networks using some of the libraries out there. However, knowing details will definitely put more light on the whole topic of whole learning mechanism of ANNs and give you a better understanding of it. So, let's dive into it!



Like the majority of important aspects of Neural Networks, we can find roots of backpropagation in the 70s of the last century. However, this concept was not appreciated until 1986. when David Rumelhart, Geoffrey Hinton, and Ronald Williams published their paper. Here they presented this algorithm as the fastest way to update weights in the ANNs, and today it is one of the most important components in ANNs learning process. One of the main tasks of backpropagation is to give us information on how quickly the error changes when weights are changed. Behind the scenes, it uses partial derivates of the cost function for each weight – ∂C/∂*w,* where *C* is the cost function, ie. function that we use to calculate the error, and *w* is the weight of the specific connection. Before we go any further let's set some assumptions regarding cost function necessary for this algorithm to work.

Cost Function Assumptions

In order to make this article easier to understand, from now on we are going to use specific cost function – we are going to use *quadratic cost* function, or *mean squared error* function:

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|y(x)-a\|^2.$$

where *n* is the total number of inputs in the training set, *x* is the individual input from the training set, *y(x)* is the corresponding desired output, *a* is the vector of actual outputs from the network when *x* is input. This function is most commonly used in ANNs so I will use it here for demonstration purposes too.

As mentioned, there are some assumptions that we need to make regarding this function in order for backpropagation to be applicable. To be exact there are two of them:

1. The cost function can be written as an average:
   $$C = \frac{1}{n} \sum_x C_x$$
   over cost functions *C(x)* for input *x*.
2. The cost function it can be written as a function of the outputs from the artificial neural network.

You can see that both of these assumptions are applicable to our choice of the cost function – *quadratic cost* function.

Backpropagation algorithm

We already established that backpropagation helps us understand how changing the weights and biases affects the cost function. This is achieved by calculating partial derivatives for each weight and for each bias, ie. ∂*C*/∂*w* and ∂*C*/∂*b.* Now, every neuron in the neural network generates some sort of an error. This error affects other neurons and ultimately it affects the global error, meaning it affects our cost function. The middle step of this whole process is calculating this value, and use it to align weights accordingly. By calculating the partial derivate of cost function by the input of each neuron, we define the error:

$$\delta = \frac{\partial C}{\partial net}$$

where *net* is the weighted input in the certain neuron.

We detoured a little bit, haven't we? Let's see what are the main steps of this algorithm. In a nutshell, backpropagation is happening in two main parts. First is called propagation and it is contained from these steps:

1. Initialize weights of the neural network
2. Propagate inputs forward through the network to generate the output values
3. Calculate the error

4. Propagation of the output back through the network in order to generate the error of all output and hidden neurons.

The second part of backpropagation updates weights of connections:

1. The weight's output error and input are multiplied to find the gradient of the weight.
2. A certain percentage, defined by learning rate (more on this a bit later) of the weight's gradient is subtracted from the weight.

Walk-trough

When we are propagation the error back to the neural network and when we are calculating errors, there is a difference depending on where the neuron is located in the network. Meaning, we use different equations to calculate the error of neuron in the output layer from equations that we use to calculate the error of the neurons in hidden layers. It is a little bit trickier and a little less obvious for neurons in hidden layers. Good thing is that, when the error is propagated back in the neural network, we start from the output layer. So, let's say that we want to calculate partial derivate of the cost function in respect to one of the weights of the neuron in output layer. Using chain rule we will get this equation:

$$\frac{\partial C}{\partial w^{ij}} = \frac{\partial C}{\partial o^{j}} \frac{\partial o^{j}}{\partial net^{j}} \frac{\partial net^{j}}{\partial w^{ij}}$$

where o^j is the output of the neuron *j*, net^j is the weighted input value of neuron *j* and w^{ij} is the weight of the connection to the neuron *j*. Notice that first two parts of this equation are actually δ, the error of the output neuron that I've mentioned in the previous chapter, if we apply a chain rule to it:

$$\delta = \frac{\partial C}{\partial o^{j}} \frac{o^{j}}{net^{j}}$$

Now, by solving each part of this equation we will get the final value of this derivate. The first part of the equation – derivate of the cost function with respect to the output of the neuron, is pretty straightforward to calculate. Since the output value of the output neuron is the output value of the neural network itself *a,* this can be rewritten like this:

$$\frac{\partial C}{\partial o^{j}} = \frac{\partial C}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} (y(x) - a) = a - y(x)$$

Meaning, this can be calculated by subtracting the expected output from the actual one in that neuron.

The second part of the equation – the derivative of the output of neuron *j* with respect to its input is simply the partial derivative of the activation function. We are going to assume that we are using the logistic function as the activation function of the output neuron:

$$\gamma(x) = \frac{1}{1+e^{-x}}$$

Then we can write this part of the equation like this:

$$\frac{\partial o^j}{\partial net^j} = \frac{\partial}{\partial net^j}\gamma(net^j) = \gamma(net^j)(1-\gamma(net^j))$$

And finally the third part of the equation – the derivative of weighted input to the neuron in respect to the weight of the input connection can be calculated like this:

$$\frac{\partial net^j}{\partial w^{ij}} = \frac{\partial}{\partial w^{ij}} w^{ij} o^j = o^j$$

Now, let's put this all together. The formula that we get for calculating the error for the neuron in the output layer is:

$$\delta = (a - y(x))a(1-a)$$

Calculating the error in the neuron that is in the hidden layer is a bit more complicated. Let's get back to the equation we started with:

$$\frac{\partial C}{\partial w^{ij}} = \frac{\partial C}{\partial o^j}\frac{\partial o^j}{\partial net^j}\frac{\partial net^j}{\partial w^{ij}}$$

The only adjustment we need to make is actually in the first part of the equation – derivate of the cost function with respect to the output of the neuron. The difference is that the output of the neuron in the output layer was already calculated, but for the neuron in the hidden layer, we need to work a little bit more. We can extend this first part, using chain rule again and considering that the cost function is a function of all inputs from neurons that are receiving input from neuron *j.* We will mark the set of that neurons with the letter *L.* Then this first part of the equation can be written like:

$$\frac{\partial C}{\partial o^j} = \sum_{l \epsilon L}\left(\frac{\partial C}{net^l}\frac{\partial net^l}{\partial o^j}\right) = \sum_{l \epsilon L}\left(\frac{\partial C}{\partial o^l}\frac{\partial o^l}{\partial net^l}w^{jl}\right)$$

This means that this whole derivative can be calculated if derivatives of all neurons that are closer to the output are known. Meaning, we can calculate the error of the neuron in the hidden layer using this formula:

$$\delta = \sum_{l \epsilon L}(\delta^l w^{jl})a(1-a)$$

Now, when the error is available, we can finally adjust our weight of connection. However, before we do that we need to pick the *learning rate* – η. This learning rate is quite important since it is dictating next step in the gradient descent process, and if it is not properly adjusted, this may cause minimum to be missed or learning process to be very slow. More on this gradient descent process you can read in the **previous article.** So, the value which will be used to modify weight value is:

$$\Delta w^{ij} = -\eta\frac{\partial C}{\partial w^{ij}} = -\eta y^i \delta^j$$

Meaning, that we can get this value by multiplying *learning rate,* with the output value of the input neuron of the connection with the error value of the output neuron of the connection. This value is multiplied by -1 because it should progress towards the minimum of the cost function.

As you can see there is quite some elegance and beauty to the whole process. Firstly, error for the output layer is calculated and sequentially that error is used to calculate the errors of all neurons in hidden layers. When we have this information it is quite easy to adjust weights.

Conclusion

Backpropagation is the tool that played quite an important role in the field of artificial neural networks. It optimized the whole process of updating weights and in a way, it helped this field to take off. And even thou you can build an artificial neural network with one of the powerful libraries on the market, without getting into the math behind this algorithm, understanding the math behind this algorithm is invaluable.

This article is a part of Artificial Neural Networks Series, which you can check out **here**.

Read more posts from the author at **Rubik’s Code**.