Back Propagation Based Neural Network
A Step by Step Example

Sari Saba Sadiya

October 22, 2016

1 Introduction:

The following is an implementation of a simple Neural Network (NN) in python, with a back propagation method for training. As the reader might have discovered, despite NN being extremely popular there seem to be a lack of beginner level guides. Worry not, just keep a stiff upper lip and you will be an expert in no time.

Back propagation is a common method for training NN (of-course, it should be used in conjunction with an optimization method such as gradient descent). The method calculates the gradient of a loss function with respect to all the weights in the network, so that the gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function. This requires a forward pass through the network, in the end of which we have the output. We then calculate the error (In this implementation we use $E = \|u - y\|_2^2 = \sum_i (u_i - y_i)^2$ where y is our output and u is the ground truth). We then calculate the gradients $\frac{\partial E}{\partial w_{i,j}^k}$ for every weight by going backwards through the k layers.

A word of warning, this is meant to be an example of a simple back propagation based Neural Network, due to efficiency issues I strongly discourage the use of this code for any non pedagogical goals.

This guide is still a work in progress.

2 License:

3 The Neural Network:

The code itself is in the file BackPropagationNN.py. In addition, the Matlab file input.mat contains sample input for the program.

3.1 The Neural Network Data Structure:

The data structure is fairly straight forward, there are k hidden layers, each contains a number n_k of neurons. Each neuron, for example the jth Neuron at layer k, will have an array of weights w_{ij}^k (one from each neuron at the previous level) and a bias b_j^k. Note that since we get W^k as a vector of weights $w_{i,j}$ we transpose the matrix to get all the i input weights for a specific neuron j, see

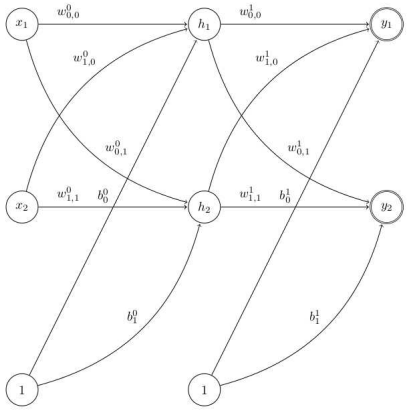
```
for k in range(0,self.n_hidden):
    h_layer_k = NeuronLayer(weights[k][0].T,biases[k][0],activation_func)
    self.layers.append(h_layer_k) #initialize output layer
```

Finally, in addition to the hidden layer we add an output layer which can potently receive an activation function different from the one specified for the hidden layers.

3.2 The Forward and Backward Passes:

The forward and backward passes are implemented as methods of the Neural-Network class. The forward pass is fairly straightforward, we simple calculate the pre-activations (net values, the sum of all the inputs) for each layer, the activation (outputs), and then we feed the results to the next layer.

The backwards pass requires some math, consider the following network with one hidden layer:



We know that our loss function is $\|u - y\|_2^2$ and we want the gradient $\frac{\partial E}{\partial w_{0,0}^1}$

. From the chain rule we get that:

$$\begin{aligned} & \frac{\partial E}{\partial w_{0,0}^1} \\ = & \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial w_{0,0}^1} = \\ = & \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial out(y_1)} \cdot \frac{\partial out(y_1)}{\partial net(y_1)} \cdot \frac{\partial net(y_1)}{\partial w_{0,0}^1} = \end{aligned}$$

Where $net\,(y_1) = \sum_i w_{i,1}^1 h_i + b_1^1$ and $out\,(y_1) = g\,(net\,(y_1))$ for the activation function of the output layer. This gives us $\frac{\partial E}{\partial w_{0,0}^1} = -2 \cdot (u_1 - y_1) \cdot g'\,(net\,(y_1)) \cdot out\,(h_1)$ or in python:

```
dWij = -2*(target[j]-activations[k][j])*out_func_grad(nets[k][j])*activations[k-1][i]
```

In addition, for the layer $k-1$ we need to calculate $\frac{\partial E}{\partial out(h_1)} = \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial out(h_1)} = \sum_i \frac{\partial (u_i - y_i)^2}{\partial out(h_1)}$ we can calculate the different elements separately and get

$$\begin{aligned} & \frac{\partial (u_1 - y_1)^2}{\partial out(h_1)} \\ = & \frac{\partial\left((u_1 - y_1)^2\right)}{\partial out(y_1)} \cdot \frac{\partial out(y_1)}{\partial net(y_1)} \cdot \frac{\partial net(y_1)}{\partial out(h_1)} = \\ = & -2 \cdot (u_1 - y_1) \cdot g'\,(net\,(y_1)) \cdot w_{0,0}^1 = \end{aligned}$$

Similarly we get:

$$\begin{aligned} & \frac{\partial (u_2 - y_2)^2}{\partial out(h_1)} \\ = & \frac{\partial\left((u_2 - y_2)^2\right)}{\partial out(y_2)} \cdot \frac{\partial out(y_2)}{\partial net(y_2)} \cdot \frac{\partial net(y_2)}{\partial out(h_1)} = \\ = & -2 \cdot (u_2 - y_2) \cdot g'\,(net\,(y_2)) \cdot w_{0,1}^1 = \end{aligned}$$

In python this is equation is expressed as:

```
dOij = -2*(target[j]-activations[k][j])*out_func_grad(nets[k][j]) \\
*self.layers[k].weight(i,j)
```

Now we move on to the hidden layer. We want to calculate:

$$\begin{aligned} & \frac{\partial E}{\partial w_{0,0}^0} \\ = & \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial w_{0,0}^0} = \\ = & \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial out(h_1)} \cdot \frac{\partial out(h_1)}{\partial net(h_1)} \cdot \frac{\partial net(h_1)}{\partial w_{0,0}^0} = \\ = & \frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial out(h_1)} \cdot g'\left(net\left(h_1\right)\right) \cdot out\left(x_1\right) = \end{aligned}$$

We have calculated $\frac{\partial\left(\sum_i (u_i - y_i)^2\right)}{\partial out(h_1)}$ in the previous layer, thus in python we get:

```
dWij = Ograds[-1][j]*activation_func_grad(nets[k][j])*activations[k-1][i]
```

4 Gradient Check:

We can check the gradient numerically since if all other values remain constant E can be written as a function of $w_{i,j}^k$. We get that $\frac{\partial E}{\partial w_{i,j}^k} = \lim_{h \to 0} \frac{E\left(w_{i,j}^k + h\right) - E\left(w_{i,j}^k\right)}{h}$ the function does exactly this $\forall k, i, j$ and checks if there is any significant difference in the gradients.