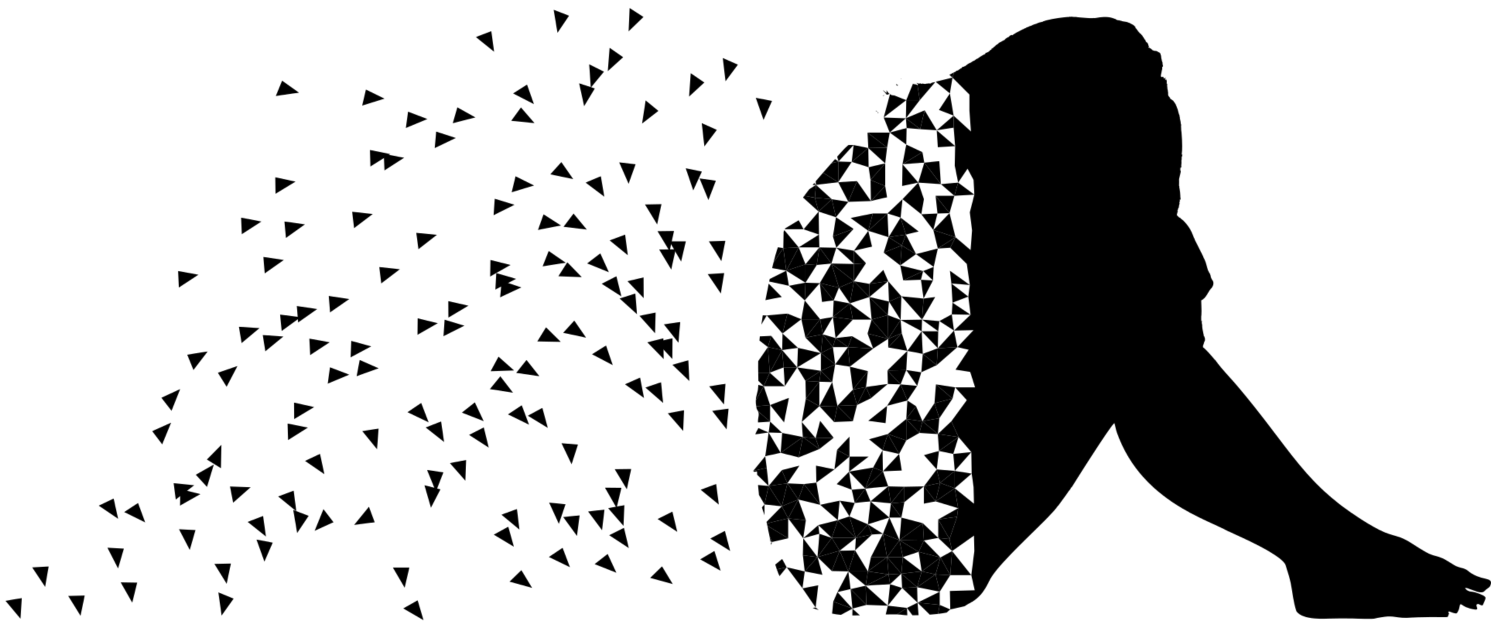


Under The Hood of Neural Networks. Part 1: Fully Connected.

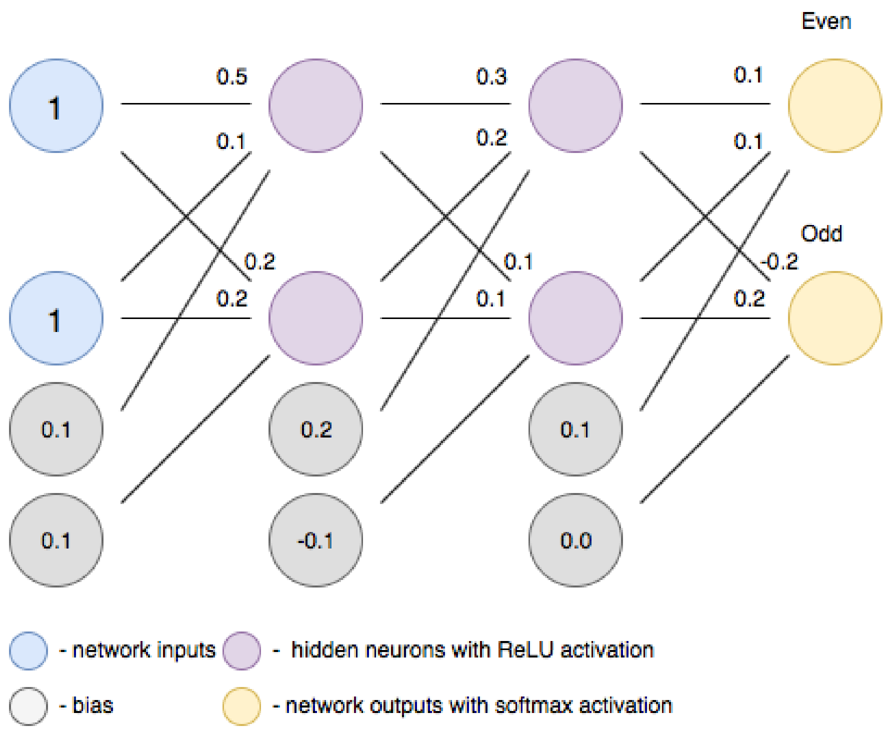
Deep Learning is progressing fast, incredibly fast. One of the reasons for having such a big community of AI developers is that we got a number of really handy libraries like TensorFlow, PyTorch, Caffe, and others. Because of that, often implementation of a Neural Network does not require any profound knowledge in the area, which is quite cool! However, as the complexity of tasks grows, knowing what is actually going on inside can be quite useful. This knowledge can help you with the selection of activation functions, weights initializations, understanding of advanced concepts and many more. So in this set of articles, I’m going to explain the mathematics behind the inference and training processes of different types of Neural Networks.

This post I will devote the most basic type of Neural Networks: Fully-Connected Networks. In spite of the fact that pure fully-connected networks are the simplest type of networks, understanding the principles of their work is useful for two reasons. First, it is way easier for the understanding of mathematics behind, compared to other types of networks. Second, fully-connected layers are still present in most of the models.

Here I will explain two main processes in any Supervised Neural Network: forward and backward passes in fully connected networks. The focus of this article will be on the concept called backpropagation, which became a workhorse of the modern Artificial Intelligence.

Forward Pass

Forward pass is basically a set of operations which transform network input into the output space. During the inference stage neural network relies solely on the forward pass. Let's consider a simple neural network with 2-hidden layers which tries to classify a binary number (here decimal 3) as even or odd:



Here we assume that each neuron, except the neurons in the last layers, uses ReLU activation function (the last layer uses softmax). Activation functions are used to bring non-linearity into the system, which allows learning complex functions. So let's write down the calculations, carried out in the first hidden layer:

neuron 1 (top):

$$\begin{aligned} h^{in} &= 1 * 0.5 + 1 * 0.1 + 0.1 = 0.7 \\ h^{out} &= \mathrm{ReLU}(0.7) = 0.7 \end{aligned}$$

neuron 2 (bottom):

$$\begin{aligned} h^{in} &= 1 * 0.2 + 1 * 0.2 + 0.1 = 0.5 \\ h^{out} &= \mathrm{ReLU}(0.5) = 0.5 \end{aligned}$$

Rewriting this into a matrix form we will get:

$$\begin{aligned} h_{in} &= \begin{bmatrix} 1 & 1 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.2 \\ 0.1 & 0.2 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} \\ h_{out} &= \mathrm{ReLU}(\begin{bmatrix} 0.7 & 0.5 \end{bmatrix}) \end{aligned}$$

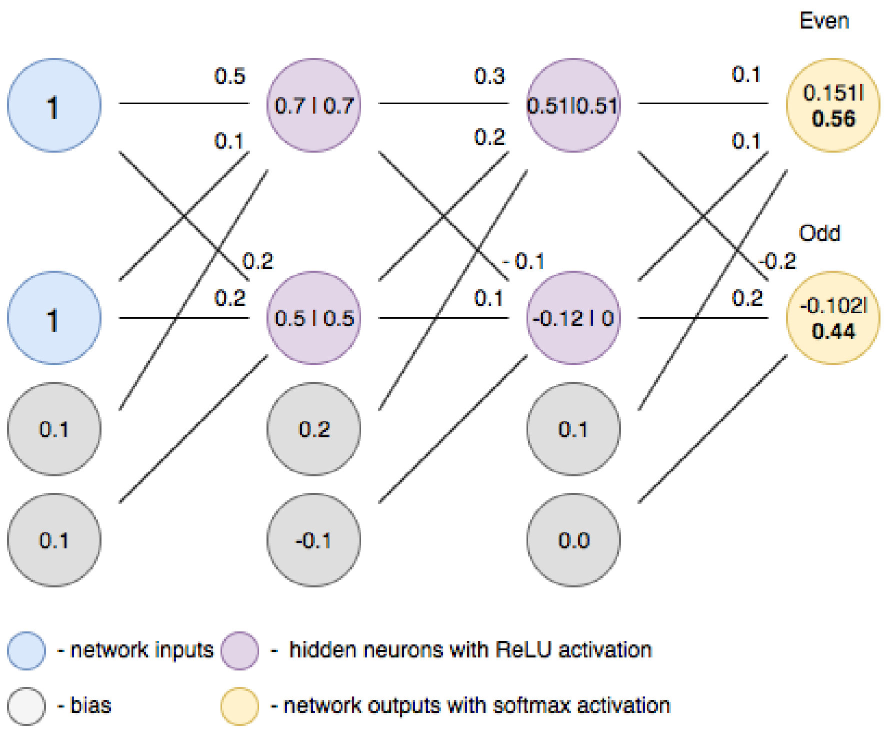
Now if we represent inputs as a matrix I (in our case it is a vector, however if we use batch input we will have it of size Number_of_samples by Number_of_inputs), neuron weights as W and biases as B we will get:

$$\begin{aligned} h^{in} &= I * W + B \\ h^{out} &= \mathrm{ReLU}(h_{in}) \end{aligned}$$

Which can be generalizaed for any layer of a fully connected neural network as:

$$\begin{aligned} h_0^{out} &= I \\ h_i^{in} &= h_{i-1}^{out} * W_i + B_i \\ h_i^{out} &= \mathrm{F}_i(h_i^{in}) \end{aligned}$$

where **i**—is a layer number and **F**—is an activation function for a given layer. Applying this formula to each layer of the network we will implement the forward pass and end up getting the network output. Your result should look as following:



Backward Pass

If we do all calculations, we will end up with an output, which is actually incorrect (as 0.56 > 0.44 we output Even as a result). So knowing this we want to update neuron weights and biases so that we get correct results. That's exactly where backpropagation comes to play. Backpropagation is an algorithm which calculates error gradients with respect to each network variable (neuron weights and biases). Those gradients are later used in optimization algorithms, such as Gradient Descent, which updates them correspondingly. The process of weights and biases update is called Backward Pass.

In order to start calculating error gradients, first, we have to calculate the error (in other words—loss) itself. We will use standard classification loss—cross entropy. However, the loss function could be any differentiable mathematical expression. The standard choice for regression problem would be a Root Mean Square Error (RMSE). The cross entropy loss looks as following:

$$L = -\sum_{j}^{M} y_j \ln p_j$$

where **M** is the number of classes, **p** is the vector of the network output and **y** is the vector of true labels. For our case we get:

$$L = -(1 * \ln 0.44 + 0 * \ln 0.56) = 0.82$$

Now, in order to find error gradients with respect to each variable we will intensively use chain rule:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

So starting from the last layer and taking partial derivative of the loss with respect to neurons weights, we get:

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial h_3^{in}} \frac{\partial h_3^{in}}{\partial W_3}$$

Knowing the fact that in case of softmax activation and cross-enthropy loss we have (you can derive it yourself as a good exercise):

$$\frac{\partial L}{\partial h_3^{in}} = p - y$$

now we can find gradient for the last layer as:

Proceeding with the layer 2:

$$
\begin{aligned}
\frac{\partial L}{\partial W_2} =& (p-y)\frac{\partial h_3^{in}}{\partial W_2} \\
=& (p-y)\frac{\partial h_2^{out} W_3 + B_3}{\partial W_2} \\
=& (p-y)\frac{\partial F_2(h_2^{in}) W_3}{\partial W_2} \\
=& \sigma_3 \frac{\partial F_2(h_1^{out} W_2 + B_2) W_3}{\partial W_2} \\
=& \sigma_3 W_3^T \circ F_2'(h_1^{out} W_2 + B_2) \frac{\partial (h_1^{out} W_2 + B_2)}{\partial W_2} \\
=& (h_1^{out})^T \sigma_3 W_3^T \circ F_2'(h_1^{out} W_2 + B_2) \\
\sigma_2 =& \sigma_3 W_3^T \circ F_2'(h_1^{out} W_2 + B_2) \\
\frac{\partial L}{\partial W_2} =& (h_1^{out})^T \sigma_2 \\
& \circ\text{- Hadamard (element-wise) product}
\end{aligned}
$$

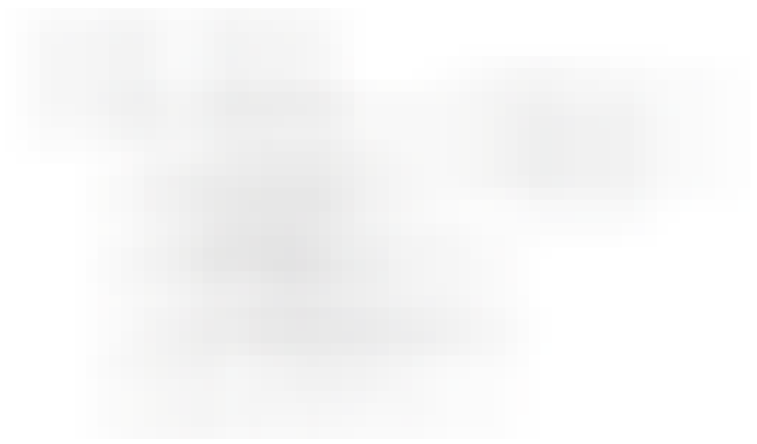
And the layer 1:

$$\begin{aligned}
\frac{\partial L}{\partial W_1} =& (p-y)\frac{\partial h_3^{in}}{\partial W_1} \\
=& \sigma_3 W_3^T \circ F_2'(h_1^{out}W_2 + B_2)\frac{\partial (h_1^{out}W_2 + B_2)}{\partial W_1} \\
=& \sigma_2 \frac{\partial (F_1(h_1^{in})W_2)}{\partial W_1} \\
=& \sigma_2 \frac{\partial (F_1(h_0^{out}W_1 + B_1)W_2)}{\partial W_1} \\
=& (h_0^{out})^T \sigma_2 W_2^T \circ F_1'(W_1 h_0^{out} + B_1) \\
\sigma_1 =& \sigma_2 W_2^T \circ F_1'(h_0^{out}W_1 + B_1) \\
\frac{\partial L}{\partial W_1} =& (h_0^{out})^T \sigma_1
\end{aligned}$$

Following the same procedure for biases:

$$\begin{aligned}
\frac{\partial L}{\partial B_3} =& (p-y)\frac{\partial h_3^{in}}{\partial B_3} \\
=& (p-y)\frac{\partial h_2^{out}W_3 + B_3}{\partial B_3} \\
=& (p-y) \\
=& \sigma_3
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial B_2} =& (p-y)\frac{\partial h_3^{in}}{\partial B_2} \\
=& (p-y)\frac{\partial h_2^{out}W_3 + B_3}{\partial B_2} \\
=& (p-y)\frac{\partial F_2(h_2^{in})W_3}{\partial B_2} \\
=& \sigma_3 \frac{\partial F_2(h_1^{out}W_2 + B_2)W_3}{\partial B_2} \\
=& \sigma_3 W_3^T \circ F_2'(h_1^{out}W_2 + B_2)\frac{\partial (h_1^{out}W_2 + B_2)}{\partial B_2} \\
=& \sigma_3 W_3^T \circ F_2'(h_1^{out}W_2 + B_2) \\
=& \sigma_2
\end{aligned}$$



Now we can track a common pattern, which can be generalized as:

$$\begin{aligned}\sigma_L &= (p - y)\\ \sigma_i &= \sigma_{i+1} W_{i+1}^T \circ F_i'(h_i^{in})\\ \frac{\partial L}{\partial W_i} &= (h_{i-1}^{out})^T \sigma_i\\ \frac{\partial L}{\partial B_i} &= \sigma_i\end{aligned}$$

which are the matrix equations for backpropagation algorithm. Having those equations we can calculate the error gradient with respect to each weight/bias. To reduce the error we need to update our weights/biases in a direction opposite the gradient. This idea is used in Gradient Descent Algorithm, which is defined as follows:

$$x_{t+1} = x_t - \alpha \frac{\partial L_t}{x_t}$$

where **x** is any trainable wariable (W or B), **t** is the current timestep (algorithm iteration) and **α** is a learning rate. Now, setting **α** = 0.1 (you can choose different, but keep in mind that small values assume longer training process, while high values lead to unstable training process) and using formulas for gradient calculations above, we can calculate one iteration of the gradient descent algorithm. You should get the following weight updates:

Applying this changes and executing forward pass:

we can see that performance of our network improved and now we have a bit higher value for the odd output compared to the previous example. Running the Gradient Descent Algorithm multiple times on different examples (or batches of samples) eventually will result in a properly trained Neural Network.

Summary

In this post I have explained the main parts of the Fully-Connected Neural Network training process: forward and backward passes. In spite of the simplicity of the presented concepts, understanding of backpropagation is an essential block in biulding robust neural models. I hope the knowledge you got from this post will help you to avoid pitfalls in the training process!

Don’t forget to clap if you found this article useful and stay tuned! In the next post I will explain math of Recurrent Networks.

Please leave your feedback/thoughts/suggestions/corrections in the comments below!

Thanks for reading!