ML Stuff

David Torpey, Shahil Mawjee, Ziyad Jappie

September 20, 2017

1 Perceptron

Perceptrons were the first architecture akin to a neural network that were introduced. They had an input layer, and an output neuron. The output neuron is known as a Binary Threshold Unit (BTU), since its output is defined as the following piece-wise function:

$$f(x) = \begin{cases} 1 & x \geq \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where ϵ is the threshold. This is known as the *activation function* of the neuron, and clearly has the form of a step function. The neuron is a binary unit since its output is either 1 or 0; the neuron is either on or off (i.e. it fires or does not fire).

Consider a perceptron with two input units (with associated values x_1 and x_2), and a single output unit h, and with weights w_1 and w_2, where $x_1, x_2, w_1, w_2 \in \mathbb{R}$. The output of the perceptron is then given by $h = f(x_1 w_1 + x_2 w_2) \in \mathbb{R}$. Notice that there is not bias term in the perceptron. It is useful to write the formula for h in a vectorised form: $h = f(\mathbf{w}^T \mathbf{x})$, where $\mathbf{w} = [w_1, w_2] \in \mathbb{R}^2$, and $\mathbf{x} = [x_1 x_2] \in \mathbb{R}^2$. The step is known as the forward propagation step.

The concept of the perceptron can be extended into what is known as the *multi-layer perceptron* (MLP), which generalises the architecture to contain as many layers, and neurons per layer as needed. Also, bias terms can be introduced into MLPs. Bias terms represent the intercept for the function the neural networks attempts to approximate, whereas the weights of the MLP are the gradients. The weights are orthogonal to the decision boundary learned by the MLP.

Up to now, we have only consider linear perceptrons. This introduces an inherent weakness applicability and usefulness of perceptrons, since they are limited to learning linear (or approximately linear) functions and decision boundaries. If the data being considered is highly non-linear, linear perceptrons are simply unable to effectively or accurately model them. An example of this is the XOR function, which cannot be solved by a linear perceptron.

Thus, activation functions are introduced in or to introduce non-linearity in the neural network, which then enables the network to model non-linear data and capture the non-linear relationships between predictors. There are some mild assumptions regarding correct activation functions, which are listed below:

1. Non-constant
2. Bounded
3. Monotonically-increasing
4. Continuous

The first and most common activation function is the sigmoid, defined as:

$$P(y = 1|x) = \sigma(x) = \frac{1}{1 + e^{-x}} \in [0, 1] \tag{2}$$

The sigmoid can be thought of as the firing rate of the neuron. Its output is a probability that a given output y will be equal to 1, given the input x. The $y = 1$ can intuitively be thought of as the class label in a binary classification setting. The reason the the sigmoid has fallen output of favour in recent times is due to the fact that is exacerbates and perpetuates the vanishing and exploding gradient problem. During backpropagation, the error derivatives need to be fed back to the early layers of the network so the weights can be updated accordingly. However, the sigmoid squashes these gradients with each passing layer, and thus does not allow the error information to reach these early layers.

This problem is best demonstrated by example. Consider a neural network with a single hidden layer with m neurons, and n input neurons. We then have weight matrices $\boldsymbol{W}^{(1)} \in \mathbb{R}^{n \times m}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{m \times 1}$, input vector $\mathbf{x} \in \mathbb{R}^n$, hidden layer activation vector $\mathbf{h}^{(1)} \in \mathbb{R}^m$, and output $\hat{z} \in \mathbb{R}$. If we assume a regression setting, we can defined the objective function J as the squared error: $J = (z - \hat{z})^2$. We assume a sigmoid activation function $\sigma(.)$ for all layers.

We know that the following are true:

$$\mathbf{h}^{(1)} = \sigma({\boldsymbol{W}^{(1)}}^T \mathbf{x}) \tag{3}$$

$$\hat{z} = \sigma({\boldsymbol{W}^{(2)}}^T \mathbf{h}^{(1)}) \tag{4}$$

Now, during backpropagation, we want to (WLOG) backpropagate the error derivatives to update all weights. Assuming we want to update $\boldsymbol{W}^{(1)}$, we would need to calculate $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}}$, which would require the chain rule:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial J}{\partial \hat{z}} \frac{\partial \hat{z}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \boldsymbol{W}^{(1)}} \tag{5}$$

Each term in the RHS of the above equation can be computed as follows:

$$\frac{\partial J}{\partial \hat{z}} = 2(z - \hat{z}) \tag{6}$$

$$\frac{\partial \hat{z}}{\partial \mathbf{h}^{(1)}} = {\boldsymbol{W}^{(2)}}^T \mathbf{h}^{(1)} (1 - {\boldsymbol{W}^{(2)}}^T \mathbf{h}^{(1)}) \boldsymbol{W}^{(2)} \tag{7}$$

$$\frac{\partial \mathbf{h}^{(1)}}{\partial \boldsymbol{W}^{(1)}} = {\boldsymbol{W}^{(1)}}^T \mathbf{x} (1 - {\boldsymbol{W}^{(1)}}^T \mathbf{x}) \mathbf{x} \tag{8}$$

Thus, we have the full expression for $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}}$:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = 2(z - \hat{z}) {\boldsymbol{W}^{(2)}}^T \mathbf{h}^{(1)} (1 - {\boldsymbol{W}^{(2)}}^T \mathbf{h}^{(1)}) \boldsymbol{W}^{(2)} {\boldsymbol{W}^{(1)}}^T \mathbf{x} (1 - {\boldsymbol{W}^{(1)}}^T \mathbf{x}) \mathbf{x} \tag{9}$$

We can see in the above equation that $\hat{z}$ and $\mathbf{h}^{(1)}$ (outputs of layers) appear multiple times. These outputs are the result of the sigmoid function, which means they are in the interval $[0, 1]$. The means that in the above backpropagation formula, many multiplications with terms smaller than 1 would occur, which means, given enough layers, the derivative would approach zero - thus vanishing the gradient.