Neural Networks

Andresito Ponce

April 15, 2020

The general idea behind neural networks is to maintain a **basis function** whose parameters can change during testing. Then the function can know how to perform "better" as the testing goes on.

Previously, we had only considered a linear combination of a fixed basis function, which usually took the form

$$y(x,w) = f(\sum_{j=1}^{M} w_j \phi_j(x))$$

However, our goal is to make some sort of these functions depend on certain parameters.

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

where a_j is the **activation** value.[1]

A common activation function is $y = \frac{1}{1+e^{-x}}$ We then transform the activation value by another nonlinear function h, and are left with the final result $z_j = h(a_j)$.

To summarize the basic structure: supposing the result fits into K classes, we then sum the weights of the D nodes in the first layer and the M nodes of the second layer, and we get a formula in the end resembling[2]

$$y_k(x,w) = \phi(\sum_{j=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{(2)})$$

[1] Remember the activation is the final output of this node. In classification, it determines which class we eventually assign to an input. The (1) superscript indicates they are the first layer of the NN.

[2] This equation basically says that the final value of class k is the result of adding all the nodes from all the layers that feed into k for the two levels?

Feed-Forward Neural Networks

In this scheme, there are a set of **input nodes**, which only connect to the nodes on the next layer, and do not form cycles. There might be some **hidden layers** connected which lie between the input layers and the final output layer.

Out of this simple model arises **deep learning**, where we have more than one hidden layers, **convolutional neural networks**, where nodes of one layer are not necessarily connected to the nodes of a different layer(in contrast to multilayer perceptron). [3]

For the problem of **regression**, using only one layer, we would also have to minimize the sum of the squares between the target vector and the output of our model $y(x,w)$. If we wanted a function with K

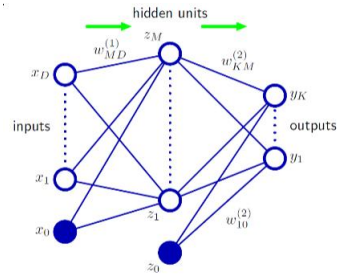

Figure 1: The input layers feed into the hidden layers, the combinations of weights and basis functions can then determine the results of the output layers.

[3] The lack of full connectedness means that the model is less prone to overfit data. Instead of matrix multiplication, CNNs also use **convolution** in at least one layer.

dimensions, our final layer could have K output nodes, one for each dimension.

For a classification problem, we might again use the sigmoid function to model our confidence of $p(C_1|x)$ for $p(C_2|x)$, by measuring the probability of one, say $y(x, w) = p(C_1|x)$ and doing $1 - y(x, w)$.

For multiclass classification, we can still optimize the set of weights w[4] and then apply the **softmax** function [5]

$$y_k(x, w) = \frac{e^{a_k(x,w)}}{\sum_j e^{a_j(x,w)}}$$

[4] does this refer to optimizing for *every* node in the network?
[5] This would be the probs an input belongs to class K right?

When utilizing gradient descent in a model such as this, we also need to calculate the gradient as well. Remember when looking to minimize the error function, we need to move in the direction where the error diminishes by the greater amount. When we calculate the derivative for the parameters of the function, this is what we are doing. We want to move in the direction the error decreases the fastest.

However, under the direct approach, we would have to use all the values of the input vector. With **stochastic gradient descent**, we could calculate the gradient using only some values from the input to speed up the calculation. This would reduce the time required, especially for large inputs. We only choose one data point to calculate the vector at a time.

In **gradient descent**, we want to adjust the set of weights along their derivatives. Thus, we move always in the direction the curve is *descending* the greatest. If we use sensible values during the calcualtion, we can avoid getting "stuck" on a local minimum, and reach the absolute lowest point of a fucntion.

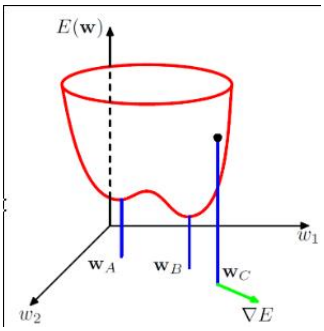

Figure 2: Visual description of gradient descent. We want to move the weights along the derivative to the global minimum of the entire function space.

Error Backpropagation

The main idea behind back propagation(BP) is that we start from the last layer in our diagram, let's call it layer k. Then we travel back along the layers and along the nodes and calculate the error regressively along the network. Thus we *propagate the error backwards*. Remember the basic notaion when we talk about *nodes* in a neural network:[6]

$$w_{ji}^{(1)}$$

Our main goal in BP is to calculate

$$\frac{\partial E}{\partial w_{ji}^{(2)}}$$

which means we want to calculate the change in the gradient relative to the weights. With this we are continuing our pattern of gradient descent, since we want to find the direction that minimizes the error.

[6] Here, the w is the weight, 1 refers to the layer where we find this node, j refers to the j^{th} node of the previous layer, and i is the i^{th} node of the current layer.

If we actually go through with the partial derivatives formula, we reach that

$$\frac{\partial E}{\partial w_{ji}} = \partial_j z_i$$

which basically means that the derivative of the gradient is just multiplying the error in node j (∂_j is the **error** here) times the z value at the input of the next node.

By making use of some earlier results, we can calculate the final **backpropagation formula** as follows:[7]

$$\partial_j = h'(a_j) \sum_k w_{kj} \partial k$$

[7] notice ∂k is from the node to which j connects. Thus, by summing the error in the previous layer, we can basically use the same formula to calculate the error for j as we did with linear regression.