Neural Networks and Activation Functions

Aaron Noyes

Neural Networks and Activation Functions

Neural networks are used increasingly to solve computing problems where an explicit ruleset is not a feasible solution. Many traditional artificial intelligence solutions are rule based. These can be powerful strategies which is illustrated alpha-beta pruning for adversarial problems where a computer creates a search space for potential moves and uses heuristics to decide which potential move has the least cost associated with it. Some problems like image feature detection, however, are not as cut and dry.

A computer sees an image as a series of pixels, so a rule-based solution to recognizing objects in images requires a precise set of rules based on an array of pixels that describes all of the complex objects that could possibly be depicted. This is a cumbersome task for even simple recognition tasks like identifying a dog in a photo. There are so many features unique to specific breeds of dogs that a rule set describing how the pixels should be sequenced in an image that contains a dog would be unruly.

Neural networks are a great solution for this problem. At their core neural networks take an input, multiply pieces of it by weights that approximate their importance, measure how far off they were, and use calculus to change the values of those decision-making weights to nudge the predicted output slightly in the right direction. This process is then repeated thousands of times in order to achieve minimal error. Below is a less generalized explanation of this process. This strategy makes neural networks a sort of "black box" where inputs are fed through and in a very difficult to understand way use the input to approximate an answer. Although this might seem like an arbitrary and unpredictable solution, it has proven itself to be quite powerful.

Neural Networks Overview

Neural networks are often visualized as a fully-connected graph of nodes, although in reality they are a series of matrix multiplications. For simplicity they will be described first as a graph and then related back to the matrix math behind them.

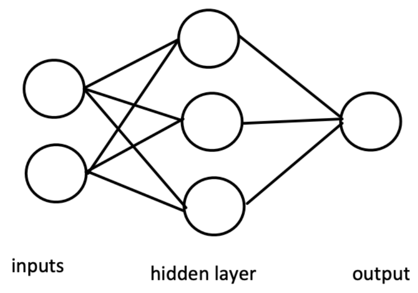

Figure 1: Neural Network Visualization

In the neural network depicted in Figure 1 each column represents a layer in the network and each circle a node or a neuron. The weights mentioned in the introduction are located on each of the edges in the graph.

Forward Pass

The first step in training a neural network is the same process by which the network makes predictions. The values in the input nodes are multiplied by the weights of the edges, fed through an activation function to make their result non-linear, and then summed into their respective nodes in the hidden layer. This process is then repeated between the hidden and output layers resulting in the predicted output from the network based on the inputs (Svovil, Kvasnicka, Pospichal, p. 44, 1997). Activation functions are usually simple mathematical functions that aim to make output from nodes non-linear which is important for the minimization phase in the training process. A more detailed explanation of activation functions is contained in the proceeding section. The process described above is typically referred to as forward propagation

Back-Propagation

Svovil, Kvasnicka, Pospichalp (1997) describe the basic process through which neural networks learn. For neural networks to learn they be provided training data in the form of inputs and their associated expected output. An error function is used to measure how close the predicted output from the forward pass was to the expected output. Each weight in the network is then updated in a way that minimizes that error function. The partial derivative of the error function with respect to each weight is used in order to find out how much to each must change to decrease the overall error of the network. Specifically $w' = w - \lambda(\frac{\delta E}{\delta w})$ where w' is the updated weight, w is the previous weight, lambda is the learning rate, $(\frac{\delta E}{\delta w})$ is the gradient, and E is the error function (p. 45-46). One of the most commonly used error functions is mean squared error.

Training

The training process cycles through the forward and backwards pass described above repeatedly for a set number of trials referred to typically as epochs. The main challenge in the training process is identifying suitable activation functions, learning rates, and error functions appropriate for the task at hand. This is a complex subject that requires understanding of the specific problem neural networks are being applied to. Typically, these meta-problems are referred to as hyperparameters.

Hyperparameters

Learning Rate

One of the most important pieces of a neural network is the learning rate. In backpropagation it is that scaler multiplied by the gradient of each weight. The gradient represents the slope, providing the direction the weight should be adjusted to move towards a minimum in the error function.

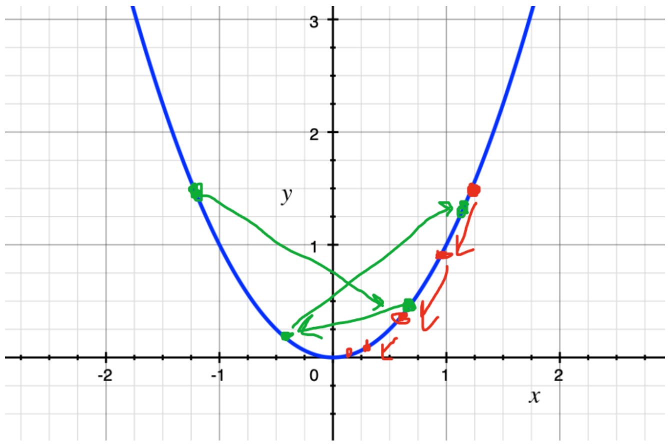

Figure 2: Different Learning Rates

Figure 2 depicts the effects of the two learning rates on a simplified error function. The goal is to move the points towards the local minimum of the blue error function. The red dots represent a smaller learning rate. As the gradient is multiplied by the learning rate the red dots slowly approach y=0. The green dots represent a learning rate that is too large. Since each step is such a large jump the continually over-shoot the minimum and never successfully minimize the error function.

A good learning rate would make the largest possible move without overshooting, thus limiting computation time and minimizing error.

Activation Function

Activation functions aim to make neuron outputs non-linear. This helps to smooth out the decision-making process by making neurons less sensitive to input changes. If there were no activation functions, then the outputs in successive layers would grow exponentially and the impact of earlier nodes would be insignificant. Karlik and Olgac (2011) describe a variety of these functions (p. 112).

The uni-polar sigmoid function ($\sigma = \frac{1}{1+e^{-x}}$) "can interestingly minimize the computation capacity for training" (Karlic & Olgac, 2011, p. 112). The hyperbolic tangent function is defined as $\sigma = \frac{\sinh(x)}{\cosh(x)}$ (Karlic & Olgac, 2011, p. 112).

Neural Networks: An Example

To completely understand the process by which a network learns an example will be stepped through below. The network depicted in Figure 1 will be assigned inputs and weights, as well as an expected output. To add substance to this example this network will approximate the output of the logical "AND" operator.

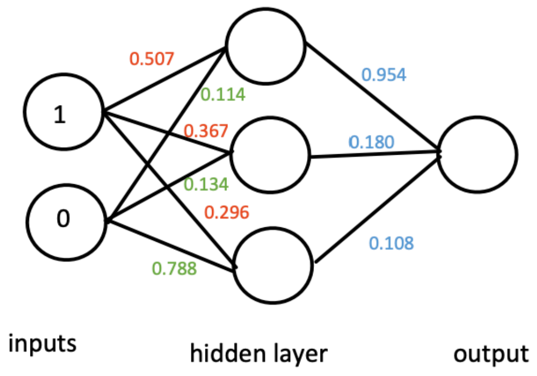

Figure 3: Neural Network with Weights

Figure 3 depicts a neural network with randomly initiated weights on each edge. Red weights connect the input’s first neuron with the hidden layer, those in green connect the input’s second neuron to the hidden layer, and blue weights connect the hidden layer neurons to the output layer. This is where matrix math becomes necessary for efficiency. To find the values passed to the hidden layer neurons, rather than adding up individual values, a matrix multiplication will be performed between the input layer and initial weights. The result is shown below.

$$[\ 1\ \ 0\ ] \cdot \begin{bmatrix} 0.507 & 0.367 & 0.296 \\ 0.114 & 0.134 & 0.788 \end{bmatrix} = [0.508\ 0.367\ 0.296]$$

Figure 4: Neural Network Weight Multiplication

The values of the hidden nodes will be calculated by applying the activation function element wise to the resulting matrix in Figure 4. The sigmoid activation function is used to compute the values [0.624 0.591 0.573]. This process is repeated once more for the hidden layer to the output layer resulting in an output of 0.682. The desired output for 1 AND 0 is 0 so the mean squared error is 0.583.

Weights are then updated using the backpropagation algorithm detailed above. These equations are omitted here but can be found in the associated python program nn.py in lines 77 and 78.

Comparing Activation Functions

In order to gain insight into the impact of hyperparameters and their effect on the learning process for a neural network a python program has been created to create adaptable neural networks. The *NeuralNetwork.py* file contains a neural network based on that described by James Loy (2018). The class has been updated to have flexible learning rate, a properly implemented training method, and different activation functions. In the case of networks using activations other that sigmoid, the activation function is used only in the hidden layer and the output layer continue to use the sigmoid function in order to keep outputs between 0 and 1. This class is quite flexible and will create appropriately sized weights for a three-layer neural network. The class accepts two arrays, one containing inputs, and the other containing corresponding expected outputs. A summary method is included that prints input, expected output, predicted output, mean squared error, and trials required to reach that error.

Another program titled *nn.py* contains methods for printing summaries off different neural networks. When run the program will create neural networks with varying hyperparameters and then output graphs comparing the error over training trials and the number of trials required for each network.

An initial comparison creates four neural networks using sigmoid, relu, tanh, and softmax activations respectively. In this trial each network was initialized with a learning rate of 0.3 and trained until it met an error better than 0.002. At a glance it seems the relu activated network learned significantly more quickly than the others.

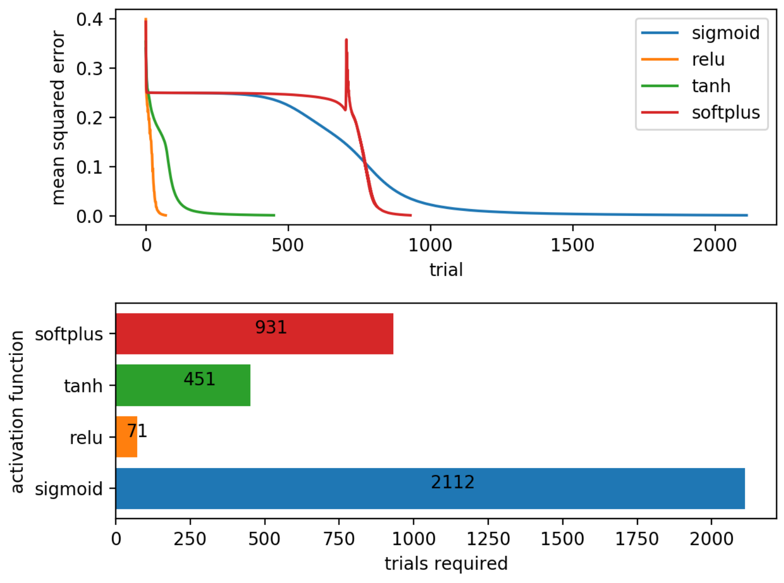

Figure 5: Networks with Learning Rate 0.3

When approached with a deeper analysis it is actually the case that the given hyperparameters seemed to favor relu, rather than it being a superior method. One notable feature is the sudden increase in error using the softplus activation. It seems to jump at a certain point which is difficult to understand. Each network does approximate the desired output quite well in very quick time.

Another trial using the same activations with a larger learning rate illustrates the importance of hyper parameters.

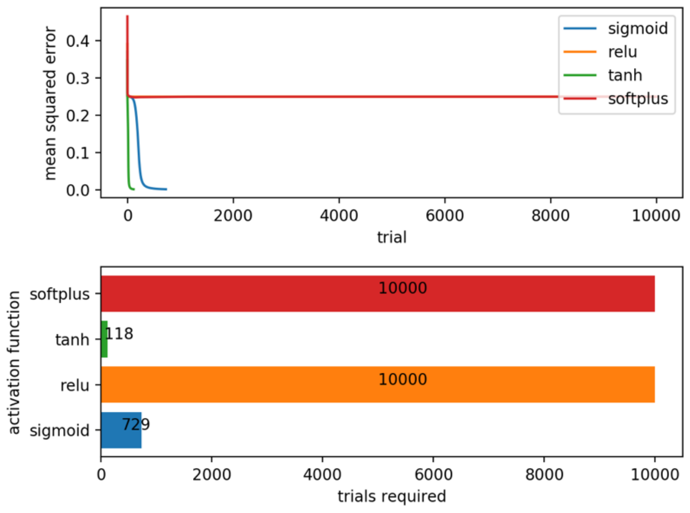

Figure 6: Networks with Learning Rate 1

In this trial each network was given a large learning rate of 1. In the case of the tanh and sigmoid functions this rate was reasonably efficient and did not overshoot the minimum. Both reduced their error significantly faster than with the smaller learning rate. Softplus and relu however gave up after 10000 and did not improve their approximations after the first handful of epochs.

One final comparison gave each network a relatively small learning rate of 0.0001. In this case each network decreased its error in a consistent fashion but failed to converge to the goal error of 0.002.

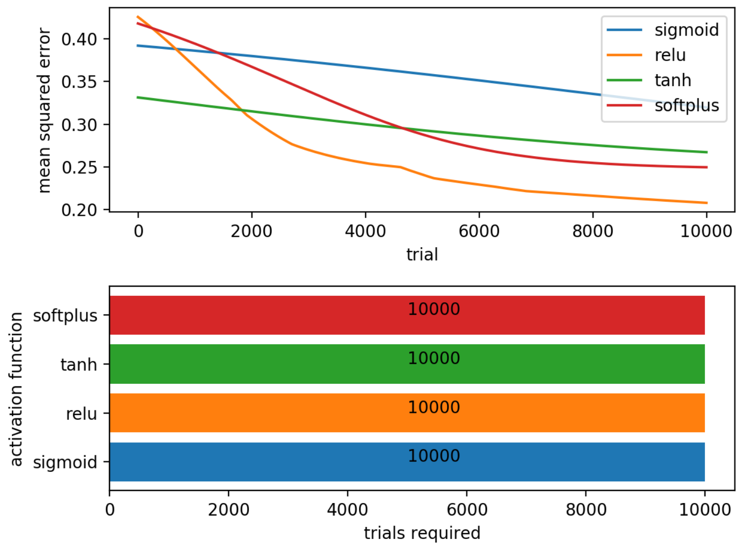

Figure 7: Networks with Learning Rate 0.0001

In this final trial it is clear that with a small enough learning rate each network with a small enough learning rate can produce a consistent decrease in error. Although with a learning rate of such a small magnitude it is much more computationally intensive to arrive at a sufficiently small error. In a test using similar learning rates and no limit to learning epochs the program failed to exit without running out of memory.

Analysis

Neural networks are a powerful tool. They are able to predict outcomes without being given explicit rules about the problem at hand. The magic behind these networks comes from some pretty straightforward calculus and linear algebra, but the result is incredibly versatile in many fields namely image recognition.

There is no universal solution as to what hyperparameters work in any network for any given problem. Learning rate, activation function, number of hidden layers, and many more variables must be carefully picked and tested when implementing any solution. In these cases balance is key and many variations must be attempted in order to meet the desired output goals for the network and be efficient computationally.

Note and Credits

All figures present in this report were created by me using Microsoft OneNote and the MacOS program titled *grapher***.** The algorithms described are widely used but specific reference is made to formulas found in the referenced works. The general idea for this program is based on the work of James Loy (2018). The referenced article from Karlic & Olgac (2011) provides a much more scientifically inclined analysis of several activation functions in neural neworks.

The mentioned programs are attached to the submission of this report, as well as a README file containing credits for third party libraries used, the original source of the neural network class, as well as a more basic description of what changes were implemented in it. The source code for Loy's (2018) program can be found on github at the following address: https://gist.githubusercontent.com/jamesloyys/ff7a7bb1540384f709856f9cdcdee70d/raw/8f769d4336569500683a3e27aae661e85bd70435/neural_network_backprop.py.

Thank you as well to stackoverflow user "cheersmate" who helped to find a bug in the initial implementation of the NeuralNetwork class with the relu activation. This report was incredibly fun to put together and helped gain a deeper conceptual understanding of neural networks.

References

Karlik, B., & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4), 111-122.

Loy, J (2018, May 14) How to build your own Neural Network from scratch in Python. Retrieved from https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6

Svozil, Kvasnicka, & Pospichal. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1), 43-62.