Network Report

fsv7eq

August 2019

1 Overview

The network operates like a traditional network during back propagation. The difference from a traditional network is how the cost propagated for each training example is calculated. While in a traditional network there would be examples of known expected values, in this case the network needed to train to fit 3 unknown parameters using only the expected value from an equation with those 3 unknown parameters as input.

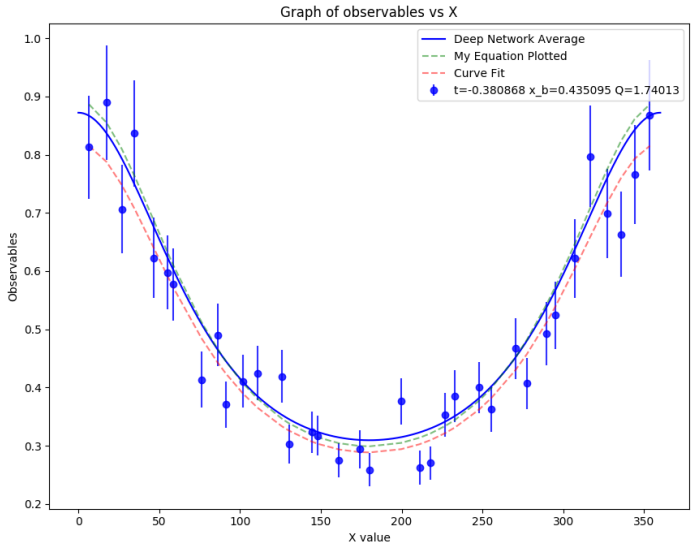
First attempts of back-propagating the difference from the equation estimated value and expected value did not perform well.

Eventually a shrinking window sampling method was settled on. In this method each parameter (1, 2, 3) has a low value, l_i, and an upper value, u_i where i is the parameter number. Each mini-batch, 5 values are selected through uniform distribution between l_i and u_i for each i. These 3 sets of 5 values are then used to create 125 permutations of possible parameters. Each of the 125 sets of parameters are then used to approximate each of the points in the mini-batch. The parameters that best fit the points in the mini-batch are then the ones that are back-propagated. Then the parameter ranges close around the parameter values chosen through an operation like $u_i = 0.99 * u_i + 0.01 * p$ where p is the chosen parameter value.

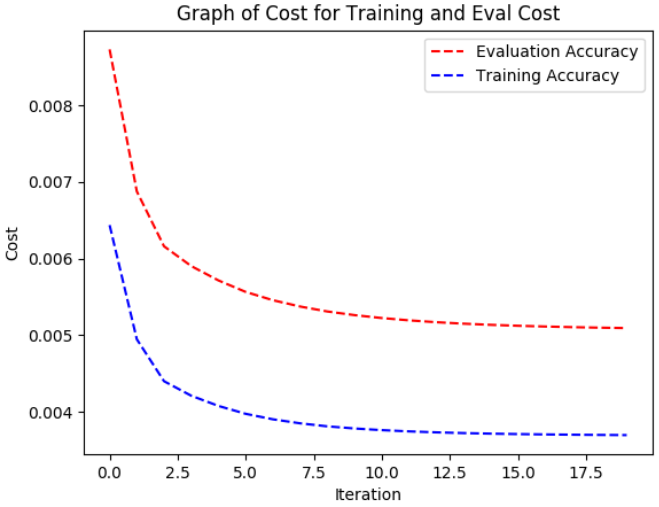
2 Results

The network performed somewhat well on the data but suffered from the same problem that the curve fit suffered from: The line fits well but the parameter values are not correct. Looking at the results in Figure 1, the network fits the points quite well. The cost graph shown in Figure 1b shows the network is minimizing the cost and successfully fitting the line.

However, the problem arises when the estimated values are compared to the actual values. Looking at Table 1 (which shows the numerical results for the network pictured in Figure 1b), the problem can be seen. While the curve fits the points quite well, the same cannot be said about the estimated values for the parameters. The actual values for ReH, ReE, and ReHT are 0.677, 1.054, and 0.908 respectively. The trained network estimates these parameters to be 1.13,



(a) Results of the 3 network ensemble

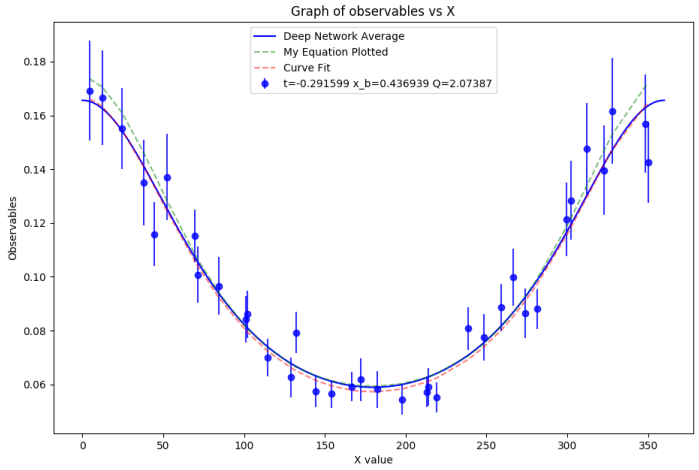


(b) Cost graph for one of the networks in the ensemble

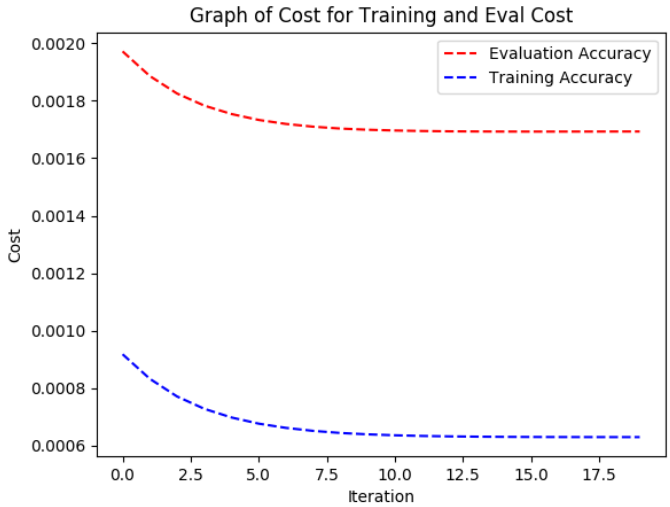
Figure 1: Results for 3 network ensemble trained on data with Kinematic values $x_b = 0.435$, $t = -0.3808$, $Q = 1.74013$

1.1006, and 0.86. While parameter's 2 and 3 are close to their actual values, parameter 1 is not. This is a problem that is fairly consistent across different sets of kinematics.

Table 1: Network 1 Results			
Type	ReH	ReE	ReHT
Correct	0.677287	1.0538	0.908049
Curve Fit	$0.4964 +/- 0.5279$	$1.2793 +/- 0.78$	$0.12243 +/- 3.918$
Deep Network	$1.1296 +/- 0.014$	$1.1006 +/- 0.031$	$0.8609 +/- 0.0939$



(a) Results of the 3 network ensemble



(b) Cost graph for one of the networks in the ensemble

Figure 2: Results for 3 network ensemble trained on data with Kinematic values $x_b = 0.4368$, $t = -0.2916$, $Q = 2.0738$

Table 2: Network 2 Results			
Type	ReH	ReE	ReHT
Correct	0.631324	0.95597	0.85238
Curve Fit	$0.59504 +/- 0.055$	$1.027 +/- 0.167$	$1.25 * 10^{-23} +/- 1.49$
Deep Network	$0.7979 +/- 0.04$	$0.718 +/- 0.049$	$0.2071 +/- 0.0025$

Looking at Table 2 for the results of Network 2, the same issue can be seen. While the Curve Fit fitted the parameters fairly well in this example, the same can not be said about the network. Although the line fits well when looking at Figure 2, the parameter values estimated by the network are not accurate. Additionally, the standard deviations of the network parameters are far too small to be of any use.

3 Methods

There were many different methods explored when developing the network. Initially, the network was trained by back propagating the error between estimated cross section and actual cross section values as the error for each of the three output nodes. This method resulted in poor fits where, after training, each of the three outputs predicted nearly exactly the same value as one another and failed to minimize the RMSE between the estimated cross section values and actual values. The common results of training like this can be seen in Figure 3.

This result led to the implementation of a regularization action which measured the length of the curve using the Euclidean distance between the points

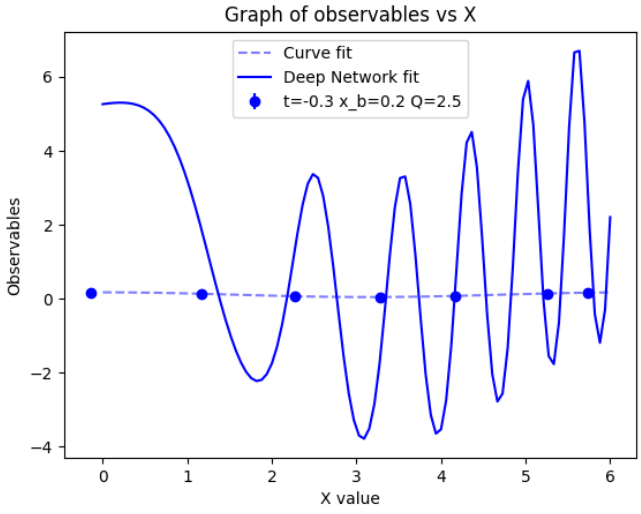


Figure 3: Results of training the network purely on cross section error

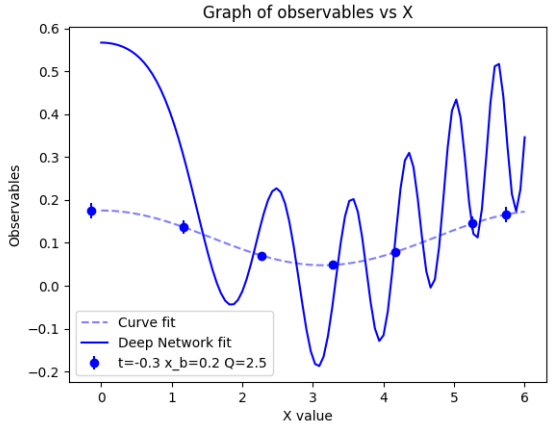


Figure 4: Results of training the network with length minimization and error shrinking

of the line over the X axis space. Additionally, a parameter was added that would shrink the error being back propagated to ensure the network was not getting stuck by consistently "jumping over" the cost minimum. The results of these changes were promising when compared to the results represented in Figure 3. The curve resulting from this stage, as seen in Figure 4, show that the network was beginning to learn the curve. The magnitude of the errors had dropped significantly between the first trial and the one shown above with the estimations by the network now within the same order of magnitude as the expected values.

Following this, derivatives were implemented into the error scaling with great success. Since at the time these tests were still being performed on the simpler model of cross sections, the derivative could be added to the model with relative ease. The error that was being back propagated for each of the output neurons was now the cross section error multiplied by the derivative with respect to each parameter in the equation. The results of this addition can be seen in Figure 5.

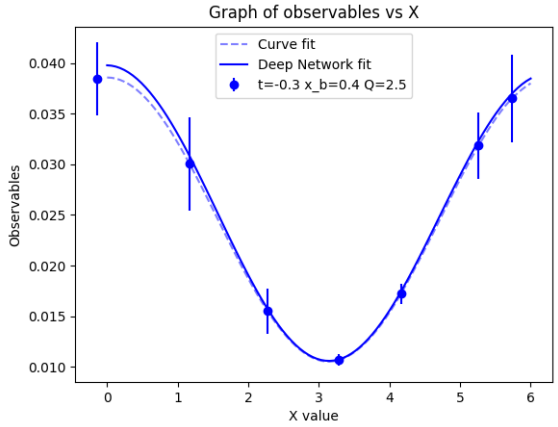


Figure 5: Results of training the network with parameter derivatives included

After implementing the derivative and obtaining the results seen in Figure 5, the data for the more complex equation was ready and tests began to determine the effectiveness of the previous methods on the newer data. Since the TotalUUXS equation was very complex, getting the analytical derivative, as had been done with the previous models, was no longer an option. Additionally, the network was suffering from a problem where the outputted parameters were experiencing a runaway effect where as the error between the estimated cross section increased, the training would cause the parameters to become even less accurate and cause the error to increase even more thus causing the parameters to become less accurate.

To solve these problems many randomized iterations were run to understand how the network trains. It was discovered that if the estimated parameters were within the same order of magnitude as the correct parameters, then the training would work but if they were too large or too small, the runaway would occur. The solution was to implement a window of "allowed" parameter estimations. The idea was to essentially force the network to estimate parameters within the range where regular training could occur. If the network estimated a parameter value outside of this range, the value back propagated was the edge of the boundary. For example: the parameter 1 neuron returns a value of -5.0, then the network will take notice of this and continuously back propagate values within the range for that neuron. Additionally, the windows would shrink around the estimated values as the network trained to force the network to find a minimum that could produce estimations that were stable for all sets of inputs.

The windows approach was also used to solve the derivatives problem. Although the derivatives themselves were no longer a concern once the TotalUUXS equation was implemented in TensorFlow and the partial derivative functionality of TensorFlow could be used, for the Numpy based version this problem still existed. The problem was that without the derivatives, the network would be forced to train on just the error from the estimated cross section vs the actual. This was not an effective way of training and would consistently result in a parameters that were a general minimum across all curves, not a minimum for the current curve. There had to be some way to give some form of error relative to each parameter. The windows implemented before were used to solve this problem, as these windows would be used to generate sets of permutations of potential parameter values within the range for that neuron. Specifically, each parameter (ReH, ReE, and ReHTilde) had its own range and 5 values would be selected randomly from each. These would then be used to create 125 (5x5x5) sets of potential parameters. Each of these parameters would then be used to calculate estimated cross section values and get an error value when compared to the actual value. The parameters with the lowest error were chosen to be back propagated, and the windows would shrink around the value picked for each parameter.

After implementing these into the network, the effect was substantial. Looking at Figure 6, the accuracy of the network fit is higher than that of a traditional Chi Squared curve fit. The results shown in the figure were fairly standard and clearly shows that the network is able to fit the curve to the points. How-

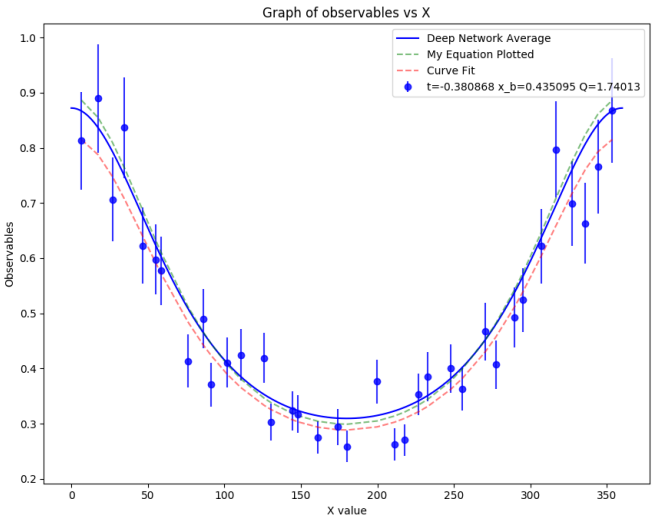


Figure 6: Results of training the network with length minimization and error shrinking

ever, as described in the Results section, the parameter predictions were still not consistently similar to the correct parameters and were at times completely wrong.

4 Further Problems

As touched on in the methods section and in the results section, the parameter predictions themselves are still a major problem for this approach. It is unclear if this error is coming from bias in the network itself or is a result of the inherent error in the data. Considering the network was able to fit the curve to the data points quite well, it seems the network itself and the approach being taken are not to blame. Further testing should be done on data with techniques applied to reduce noise or on artificial data with no error added to determine if this is the source of error.