COSC 420 Project 4: Artificial neural net back-propagation system characteristics and utilities

Timothy R Lovvorn

April 6, 2016

Abstract

Back-propagation is a computational method derived to take a neural network of varying sizes and attempt to "teach" the network to recognize certain repeating characteristics within a given data set, generally when one is unaware of the implicit relationship of the inputs/outputs in the data set in question. The network is trained upon a large sample size of inputs and outputs all sharing some form of relationship, be it a mathematical computation, a mapped value, a specific type, and so on, before being validated and testing upon a smaller subset of the same data. The relationship between the number of layers within the architecture, the number of neurons within each layer, and the overall error of the network when tested upon our test data will be explored herein.

I Introduction

The idea of the neural network is an attempt to recreate the general architecture, and thereby computational brilliance, of the human brain. Similar in fashion to the ability of our brain to learn based upon various stimuli, the back-propagation neural net seeks to perform a similar learning ability. By first forward propagating through a neural net with a set number of inputs, one is able to determine a singular output for the network. After this is done, a back-propagation can be performed by determining by what approximate percentage the output received by the network is similar to the expected output. Based upon this, a backward iteration through the network is performed, changing the weights and thresholds upon each neuron in the network based upon their relation to the next layer of neurons and a user set learning rate. With each

movement through the net, the weights will be adjusted to approximate an output closer to the expected values until, the net is able to correctly match the output value with a very low margin of error. The network has then effectively, through a trial and error process of various iterations (epochs), learned how the given inputs are related to the outputs. This has been used for a variety of purposes form categorizing similarities in images, such as tanks for military purposes, to creating mathematical computation tools when the inputs/outputs are known but the function itself is not. Our experimentation will focus highly upon the latter, utilizing the network to effectively create a method of effectively approximate the value of inputs subjected to a function based upon known input/outputs of the function without actually hard-coding the functionality of said function in the program itself.

2 Methods

Experimentation upon a learning neural net via backpropagation was achieved by creating a program in the C++ computing language to take files containing input/output values and run them through a back propagation artificial neural net of dynamic architecture to determine the effects different architecture build have on the ability of the net to effectively approximate given inputs to expected values. The program consisted of a single h file, L_net.h and two .cpp files, L_net.cpp and Learning_sim.cpp all compiled through MinGW into a single windows executable run.exe which takes command line arguments for the number of hidden layers, neurons per layer, learning rate, train_data.txt, validation_data.txt, test_data.txt, number of epochs, and problem number. This allowed for manipulation of architecture between runs of the simulation. Weights for the neurons, as well as sigma values, delta values, and h values were maintained in two and three dimensional vectors. All program files are appended to this report. The run.exe executable will call an instance of L_net, which will create out vectors, zero out the sigma, delta, and h vectors, while randomizing the values in the hidden and outer weights vectors between -0.1 and 0.1 with the threshold being represented as the zeroth weights between each layer. A training function is then called which iterates through every input and output value, propagates forward through the network, back propagates to determine the gradient of the root mean square error calculated for each input/output pair to calculate delta values used to update the weights, and then forward propagate using said deltas to update the weights. After each propagation a validation is run as well. This continues until all epochs have been achieved, then a final test function is called to test the values against a final set of input/output pairs to determine the effectiveness of our training. The RMSE values for each thousandth epoch are outputted for graphing purposes. For problem 1, the values of out input/output are based upon the follow equation:

$$\mathrm{f(x,y)} = \frac{\left(1 + \sin\left(\frac{\pi x}{2}\right)\cos\left(\frac{\pi y}{2}\right)\right)}{2}, \mathrm{x} \in (-2,2), \mathrm{y} \in (-2,2)$$

And for problem 2:

$$f(x,y,z) = \frac{3}{13}\left(\frac{x^2}{2} + \frac{y^2}{3} + \frac{z^2}{4}\right), x \in (-2,2), y \in (-2,2), z \in (-2,2)$$

All calculations for each complete cycle of the net are as follows. Because of the high level of symbolism required to demonstrate the mathematical equations for each step of determining partial derivatives in order to find the gradient, all equations will be demonstrated in a pseudo code shorthand for ease of reading.

Forward propagation was performed as followed:

In each layer you must compute the local sum of the weighted values and their sigma values. In the case of the first layer, the summation, h, is simply each weight multiplied by the input values, then adding the bias to the overall h value.

H_{sum} += $weight_{hidden}$ * input

$H_{total} = H_{sum} + bias$

The sigma value for determing whether or not there is a sufficient value of h to change the overall value of the neuron is calculated by the following equation:

$\sigma = 1/(1 + \exp(-H_{total}))$

For all non-first hidden layers, the h value is computed as the sum of each weight multiplied by the corresponding previous sigma value:

H_{sum} += $weight_{hidden} * \sigma_{previous}$

The final output of the net for the forward propagation is calculated as the sum of the output weights multiplied by the final sigma value of the last hidden layer, then adding the bias to the sum.

H_{sum} += $weight_{out} * \sigma_{last}$

$H_{out} = H_{sum} + bias$

And the final sigma is calculated in the same manner as all sigma values shown above using the output value:

$\sigma_{final} = 1(1 + \exp(-H_{out}))$

Back-propagation was performed as follows:

Starting from the last hidden layer, you begin computing the delta values for each layer to use in later shifting the weight values accordingly. For the last hidden layer, delta is computed as the final sigma multiplied by 1 minus the final sigma, multiplied by the difference between the expected output given in the data file, and the actual sigma output:

$\delta_{first} = \sigma_{final} * (1-\sigma_{final}) * (output-\sigma_{final})$

The delta for each hidden layer is computed as the sigma value for the current node multiplied by 1 minus the sigma value for the current node * multiplied by the weight associated with the proceeding layer (or output in the case of the last layer) multiplied by the delta value of the

proceeding layer for the corresponding node. It is important to use the proceeding layer values as this is a backward propagation:

$\delta_{sum} += \sigma_{current} * (1-\sigma_{current}) * \delta_{next} * weight_{next}$

This is continued for all hidden layers of the net, forward weight updating propagation is as follows:

For the first layer of the net, the weights are updated by summing the product of the learning rate, the delta value for the current node, and the inputs:

$Weight_{first} += Lrate * \delta_{current} * input$

For every other hidden layer the weights are updated by summing the product of the learning rate, the delta value of the current node, and the sigma value of the previous corresponding node:

$Weight_{hidden} += Lrate * \delta_{current} * \sigma_{previous}$

The bias weights of each hidden layer are updated by adding the sum of the learning rate multiplied by the delta value of the current node:

$Weight_{biashidden} += Lrate * \delta_{current}$

The weights of the output layer are updated by adding the products of the learning rate, the first delta value calculated, and the corresponding previous sigma value.

$Weight_{out} += Lrate * \delta_{first} \ \sigma_{previous}$

For the bias weight of the output layer, it is calculated by summing the products of the learning rate, the first tabulated delta value.

$Weight_{biasout} += Lrate * \delta_{first}$

After each successful run of approximately 1,000 epochs, to calculate the change in the overall error of the calculations of the net, a root mean square error was calculated and output to note the change over several thousand epochs. After completing a validation, or test pattern sequence to the net, a running sum is kept of the square of the difference of the expected output and the actual sigma output:

$Sum += (Output-\sigma_{final})^2$

After all patterns have been completed, the RMSE value is taken with the following equation:

$$RMSE = \sqrt{\frac{sum}{2*numTestingPatterns}}$$

This RMSE value is useful to determine how effective the current architecture for the net is and how each subsequent change of either number of hidden layers, neurons per layer, learning rate, or possibly number of epochs effects the overall effectiveness of the net. Various scenarios computed by varying the aforementioned variables are discussed below.

3 Results

In total, after several test runs were performed to gain a general understanding of the behavior, 7 simulations were conducted to gather data for extrapolation and graphing. The first 2 simulations, as illustrated in Figure 1, were to measure how varying the learning rate will affecting the rate of change of the RMSE. A further two simulations, illustrated by Figure 2, were used to measure how the total number of epoch will affect the overall trend of RMSE as well. The final three experimentations were performed to determine the effect of varying the number of neurons within the respective hidden layers of the net, with the results shown in the graph of Figure 3. For each set of simulations, variables within the architecture which were not being examined were held in constant in a manner described within the caption below each figure to observe the effect of a single type of change upon the network.

4 Discussion

The first note, perhaps the most important of this entire paper, is to note all experimentation should be considered inherently flawed for my simulations. We can base this off our general knowledge of how a neural net should perform when utilizing backpropagation to train upon a specific set of data. Over epochs, we should see a gradual decrease and stabilization of the RMSE as the training of the net reaches closer and closer in its ability to approximate the relationship between the inputs and outputs passed through the net. This was not the case in all of my experimentation. As noted in Figures 2 and 3, regardless of how we changed our variables in those relationships, we noticed a steady *increase* in the RMSE over time. While such phenomena is to be occasionally expected for certain outlier configurations, in general you should always have an overall decrease in your RMSE to an eventually stability (unless training is continued to a point you will increase RMSE by overspecializing the net to the training data). While it is possible this is because by sheer random chance, all the randomized values for weights at the beginning of experimentation, as well as the specific configurations I chose when performing experiments happened to be specialized outliner cases, the probability of this being the case is almost immeasurably low. The likely cause is logic errors within the simulation program itself. While programming my simulation, I spent a great deal of time over the course of two and a half weeks attempting to debug my code to fit within our parameters for error. Despite many, many hours spent manipulating the program, the primary issue of a RMSE far above acceptable levels or indeed even increasing, was a persistent issue. After reworking and printing various portions of the results over several days and hours, my best theory is the issue is caused by the calculations of the local field. It would appear some portion of the logic calculating the h value causes the h to trend very rapidly toward one and remain there, causing the same occurrence with the sigmoid value. This would cause the overall change in the weights to steadily

decrease to an almost negligible amount before the network was wholly successful in discovering the relationship between our input/output data. I have an extreme personal interest in this particular computational machine and fully intend to continue to manipulate my simulation until I achieve success, but for the purposes of this experimentation, time simply ran short. Because of this likely fallacy in the data, our conclusion drawn from this experimentation will have to be generalities, but there is still much discussion to be had upon the source material.

4.1 Limitations

As noted above, the most glaring issue with our experimentation is error in the functionality of the simulation itself. Because of this error, we cannot reliably simply trust the data gathered alone when drawing conclusions about the behavior of these backpropagation nets. The work around here for anyone wishing to perform similar experimentation on backpropagation networks would be quite concise in this case. By studying the data of my colleagues, as well as their own conclusions based upon their experimentation, it would be possible to aggregate all there data and methods together to draw conclusions from their results as well as check for consistency of data across all experimentation. In addition, the subject of back propagation neural networks, being a rather traditional form of artificial intelligence, has also been heavily researched, in particular by universities and monolithic technological enterprises such as Google and IBM. Investigating the data of those which access to such near infinitesimal resources to study the subject matter would be no doubt very informative.

Of smaller note, but a continual consideration in any such computer generated program using randomization, is the issue of outlier cases. Computers, when performing a random number generation such as C++'s rand() function, effectively create a pseudo random number each time because repeatability is critical to the theory of computation, and computers' seed randomizer generators for this purpose. This is mitigated by causing each seed call to be an instantaneous moment in time, as each would be unique on our current universe model and near impossible to be subject to a recurrent pattern. There is still an issue of randomized values possibly causing outlier cases, making collected data misrepresentative to the average behavior of our network; however, as our data is already unreliable due to logic errors within the simulation, this is not a particular concern.

A final issue would be sheer computational power. With my processor on my Windows box I was able to simultaneously run 3 simulations to gain my data, but this was an absolute limit as it put a tremendous amount of strain on my computer to perform this feat. Experimental runs themselves, despite being conducted on relatively small networks of only a few layers with a few dozen neurons per layer at most, took extremely long amounts of time to perform an appropriate number

of epochs. The runtime of a rather median case, a 3 layer net with 15 neurons per layer and 30,000 epochs, took approximately 30 minutes to run on my system. This put a tremendous time crunch on the project, as even with an ideally running system and simulation only a small number of experimental runs, I would estimate with my computing power to be a maximum of 20 per day (assuming parallel running of different simulations simultaneously), is possible. This issue can be mitigated by extreme principle and other resources, but is impractical for a collegiate undergrad level project on an already well-researched topic.

4.2 Conclusions

Despite the unreliable nature of our data, based upon our general knowledge of the networks we can draw some conclusions about the general behavior of backpropagation networks. The first would be while neural nets are very powerful, there is a degree of unreliability in there use dependent on what exactly you are trying to teach. Our networks are told to find a relationship between the inputs/output values it is given, and to attempt to fit itself to simulate this relationship. Because of this anything you wish to teach a network to do, say perform a complex computation, recognize a certain image across a gallery of photos, or determine the most prolific amoeba in an experimental solution based upon several microscopic images taken, you have to be very careful to control for other variables. Say in the above example perhaps amoeba A is shown to be the most prolific amoeba in the solution as all the images demonstrate is has changed position the most, yet when you observe the solution yourself you notice amoeba B is moving about far more often than amoeba A. Further experimentation reveals whenever the scanning electron microscope began to take a picture, its radiant heat caused the solution to warm, and in the warmer environment amoeba B became far more lethargic so all images of amoeba B seemed to be relatively stationary compared to the unaffected amoeba A. The network would have no notion of such possible variances, it simply takes note of amoeba positions in images it is given and notes the changes over time. Because a possible variable was not properly accounted for, the network gave a false conclusion, and in any case where there are not carefully controlled variables acting upon the network, this could occur.

It can also be noted our networks accuracy is not proportional to the number of epochs. We know with each epoch passing we must become closer to matching the actual values in our training set for our RMSE to decrease as it should; however, after too many epochs this can be taken too far to be generally useful any longer. If trained upon the same set of data to often it would logically follow the network may become too specialized and focus solely on a particular input value matching a particular output value. Say you trained a network to increment a given value by one every time it is input and say the pair 3,4 was one of your input output values. You want the net to learn after a value, the next value should

be a single increment higher regardless of what the input is, 3 goes to 4 because 3 + 1 = 4 as it were. If you train too long, it may become so specialized to your specific training set it simply associates the given values as always being the only logical pair to go together, after 3 is 4 simply because it always does. In this case if you tried to use the network on values outside the training set, like on 5. It would not know what to do as it has no specific value to return for the vase of 5, whereas in the less trained net it would have known to increment by one to get the next value. Essentially, too much training runs the risk of making the network simply an overly complex lookup table.

5 Acknowledgements

I must express my gratitude to Dr. Bruce MacLennan once again for both is very, very detailed slides on the derivation and implementation of a backpropagation neural network, as well as his detailed project outline. I must also express great gratitude to Dr. Van Hornweder as well for her detailed layout of the programming aspect of creating a neural net with backpropagation learning laid out in a very clear, methodical manner. I must give a final thanks to my Graduate Teaching Assistant Zahra Mahoor, for her lecture detailing various methods of visibly demonstrating collected data for the project.

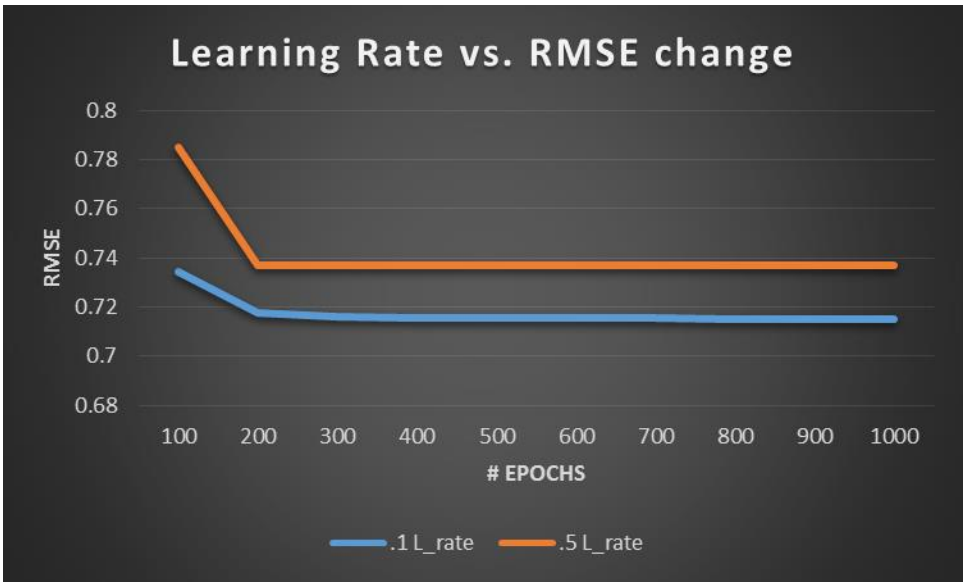

Figure 1: Illustrating the difference of decrease of RMSE in relation to the Learning rate. Done on problem 1 with 3 layers of 15 neurons each.

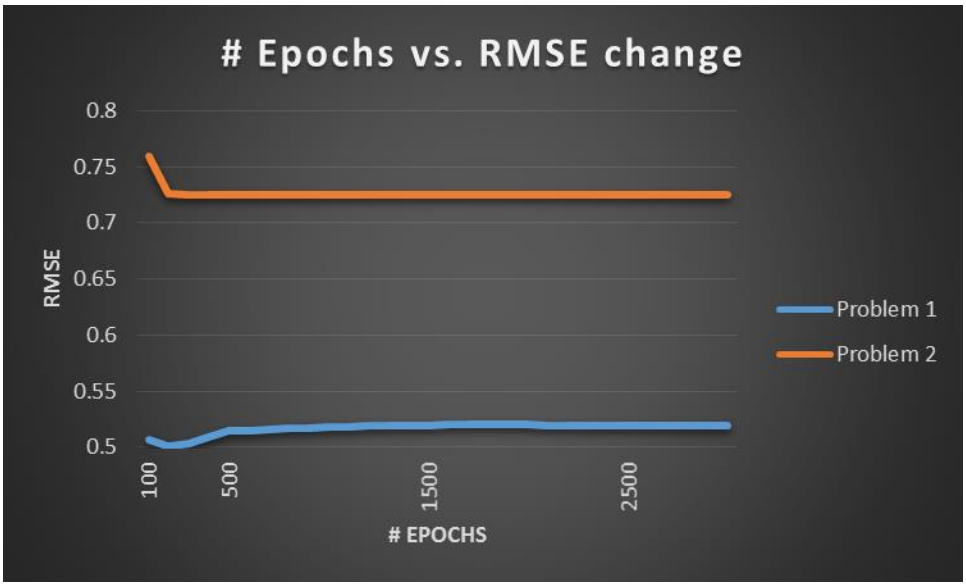

Figure 2: Illustrating the trend of the RMSE as the number of epochs increase. In this case, the positive increase suggests an error in the simulation of problem 2. Conducted on a 3 layer net with 10 neurons each and a learning rate of 0.3.

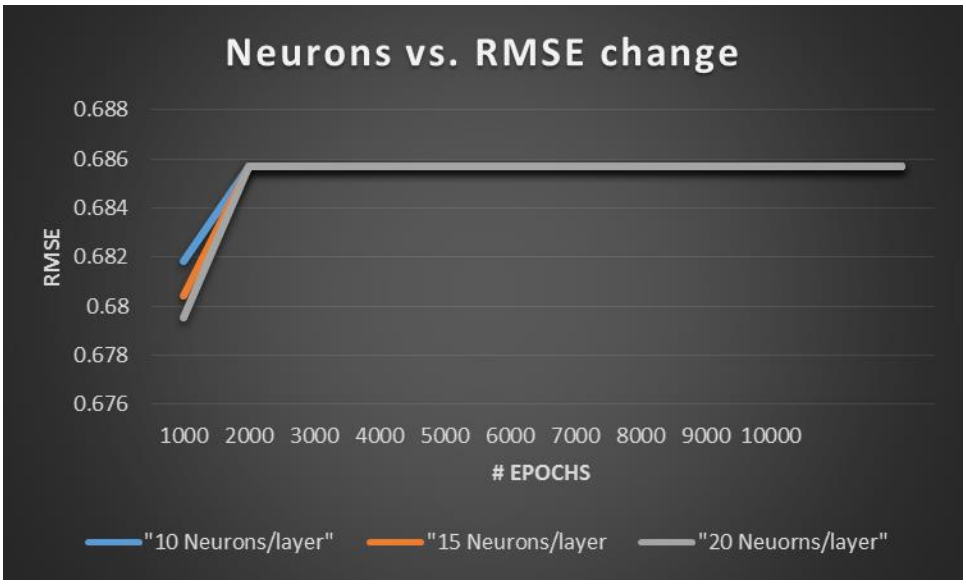

Figure 3: Illustrating the effect of different neuron configurations within the layers. Performed with 3 layers, on problem 1, with a 0.5 learning rate for 10,000 epochs