Feedforward Neural Network Adaptability

for Mid-Run Problem Rescaling

Michael Suggs1,\*

1University of North Carolina Wilmington, Department of Computer Science, Undergraduate

\*Contact: mjs3607@uncw.edu

Abstract—This study examines the effects of apply new training patterns and expanding the number of output nodes for an in-progress Feedforward Neural Network. New patterns were tested as being applied by themselves without retraining on the original data as well as in a combined set with all input patterns new and old. The number of patterns, and thus the number of output nodes, was increased from 6 to 7 originally, but was also trained and tested on up to 14 patterns in total.

Key Words— Feedforward Neural Network, backpropagation, scalability, expandability

1. Introduction

This paper addresses the effects of expanding the number of output nodes of an in-progress neural network by incrementally increasing the number of patterns applied, essentially layering training for different sets of patterns on top of each other. After each round of training, the network is tested against all previous subsets of training data to evaluate the effectiveness of the network at remembering previously applied successful training data sets when supplied with additional data.

The training and testing data is progressively input according to two different methods. In both, the original training data is supplied and trained according to standard backpropagation procedures. That is, the patterns are given with correct responses in which they are to learn from, after which modified or *‘fuzzy’* versions of said patterns are applied. For both training and testing cases, the patterns are presented in an entirely random order to prevent any form of ordered learning. Then, the additional training data will either be provided in a combined set alongside the original training data in one combined set, or will be provided as a separate set apart from the original training data.

All patterns are input in a 5x5 matrix representation, representing 25 individual pixels with normalised greyscale values between 0 and 1. Each input matrix represents a specific symbol from a given list of symbols, all of which will be detailed later.

1. Feedforward Neural Networks

Feedforward Neural Networks, a biologically-inspired machine learning algorithm, was first coined in the 1940s, but improved significantly with the addition of backpropagation, introducing gradient descent learning to the model. Neural networks come in a variety of sizes, scopes, and specifications, each containing an input layer and an output layer of nodes. The number of input nodes corresponds to the number of input values in the input vector and the number of outputs corresponds to the number of desired categories. If these two layers are one in the same, the network represents a single-layer perceptron network, which simply feeds the inputs into the outputs via a weighted pathway.

Alternatively, multi-layered neural networks – such as the one presented in this paper – contain both an input layer, an output layer, and a hidden layer in between. Each input-layer neuron is connected to each hidden-layer neuron via a weighted connection, over which the input value is passed and multiplied by said weight. The computed value at the end of this pathway is then fed into an activation function, which results in the hidden neuron’s output signal. This is then fed to either another hidden layer, or into the output layer – both of which are over another weighted pathway. The same activation function is applied again for the final output signal of each output node.

Neural networks implementing backpropagation seek to minimize a provided loss function by means of gradient descent – for the scope of this paper, the loss function will be given by the total sum of squared errors. This function, which is calculated periodically throughout the network’s learning process, compares the computed output with the expected output, subtracting the two and summing all of these squared differences for each output for each pattern applied. Over time, this works down the gradient towards a local minimum, which is the point of lowest total error for this network.

The network employed for this research consists of an input layer, a single hidden layer, and an output layer. All weights were randomly initialized, both between layers and from the biases for each layer. All biases were set to one and were unchanged throughout the experiment. The chosen activation function for the hidden and output layers was a sigmoid function, and the loss function to minimize was the total sum of squared errors. Although the input layer remained unchanging, the output layer was changed after initial testing and training to accommodate new data and the hidden layer was changed between a number of set values to compare performance with different hidden layer sizes.

1. Data Representation

As mentioned previously, the network presented in this paper was given a 5x5 matrix, which was converted to a 25 element column vector with each component of said vector representing the input value for an individual input neuron. All of these values existed in the closed interval between 0 and 1 – for training data, each “pixel” in a symbol was represented by .9 (a high signal for that pixel) whilst the remainder of the image was represented by .1 (a low signal).

For the testing data, the same interval applies, but the matrix was subject to perturbation before application. The original matrix was again given to the network, but random pixel values within the matrix were modified away from the initial 0.1 or 0.9 values. The function perturbing these matrices was applied in different intensities – as intensity increases, the number of pixels modified also increases, as well as the amount of potential perturbation occurring.

1. Testing Procedures

The aforementioned perturbation occurs to test how well the network is able to identify and classify obscured versions of the initial symbols it has already seen. However, before any perturbation occurs, the original symbols are applied as clean, baseline benchmarks.

After the initial training and testing on the original set of symbols, more symbols are applied according to the two methods set forth in the introductory section. These methods were applied separately to networks that had been through the first set of training and testing and compared against each other. For all testing runs, perturbation was applied in increasing levels throughout the testing cycle – beginning with no perturbation and increasing thusly.

The initial symbols and the additional symbols applied all come from the same sets, with one set encompassing each of the two to prevent any potential overlap. The initial set of symbols consists of six elements – a cross, a dash, a backslash, a forward slash, an X, and a vertical bar. The additional symbols applied consists of seven elements in total – an asterisk, a square, a H, an M, an L, an equals sign, a hexagon, and a diamond.

1. Findings and Results

As expected, adding patterns to the network did indeed perturb the solutions it ended up with. After training on the original data, the network was first given the separate data to learn on all by itself, completely removing the original data from the training scenario. This resulted in almost every single original pattern presented when testing after training on the new pattern alone to activate maximally in said new node – in the case, it recognized every pattern as an asterisk. This resulted in an output vector which should have looked like the leftmost vector to look like the rightmost. This is for the vertical bar pattern, whose activation level is the second element of each vector. The last element is the new pattern, an asterisk.

Combining the two data sets together as the number of output nodes increased greatly resolved the issue, as the network was constantly being ‘reminded’ of the original patterns it had seen and all patterns since. This caused the network to learn patterns similarly to how it would learn if given to an entirely fresh network – albeit a fresh network with a large amount of training.

1. Appendix
   1. *Original Training / Testing Data*

[*# Cross* (([.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1]),  
 (.9, .1, .1, .1, .1, .1)),  
  
 *# Dash* (([.1, .1, .1, .1, .1],  
 [.1, .1, .1, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .1, .1, .1],  
 [.1, .1, .1, .1, .1]),  
 (.1, .9, .1, .1, .1, .1)),

*# Backslash* (([.9, .1, .1, .1, .1],  
 [.1, .9, .1, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .1, .9, .1],  
 [.1, .1, .1, .1, .9]),  
 (.1, .1, .9, .1, .1, .1)),  
  
 *# Forward Slash* (([.1, .1, .1, .1, .9],  
 [.1, .1, .1, .9, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .9, .1, .1, .1],  
 [.9, .1, .1, .1, .1]),  
 (.1, .1, .1, .9, .1, .1)),  
  
 *# X* (([.9, .1, .1, .1, .9],  
 [.1, .9, .1, .9, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .9, .1, .9, .1],  
 [.9, .1, .1, .1, .9]),  
 (.1, .1, .1, .1, .9, .1)),  
  
 *# Vertical Line* (([.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1]),  
 (.1, .1, .1, .1, .1, .9))]

* 1. *Expanded Training / Testing Data*

[*# 0. Cross* (([.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1]),  
 (.9, .1, .1, .1, .1, .1, .1)),  
  
 *# 1. Dash* (([.1, .1, .1, .1, .1],  
 [.1, .1, .1, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .1, .1, .1],  
 [.1, .1, .1, .1, .1]),  
 (.1, .9, .1, .1, .1, .1, .1)),  
  
 *# 2. Backslash* (([.9, .1, .1, .1, .1],  
 [.1, .9, .1, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .1, .9, .1],  
 [.1, .1, .1, .1, .9]),  
 (.1, .1, .9, .1, .1, .1, .1)),

*# 3. Forward Slash* (([.1, .1, .1, .1, .9],  
 [.1, .1, .1, .9, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .9, .1, .1, .1],  
 [.9, .1, .1, .1, .1]),  
 (.1, .1, .1, .9, .1, .1, .1)),  
  
 *# 4. X* (([.9, .1, .1, .1, .9],  
 [.1, .9, .1, .9, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .9, .1, .9, .1],  
 [.9, .1, .1, .1, .9]),  
 (.1, .1, .1, .1, .9, .1, .1)),  
  
 *# 5. Vertical Line* (([.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1],  
 [.1, .1, .9, .1, .1]),  
 (.1, .1, .1, .1, .1, .9, .1)),  
  
 *# 6. Asterisk* (([.9, .1, .9, .1, .9],  
 [.1, .9, .9, .9, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .9, .9, .9, .1],  
 [.9, .1, .9, .1, .9]),  
 (.1, .1, .1, .1, .1, .1, .9)),  
  
 *# 7. Square* (([.9, .9, .9, .9, .9],  
 [.9, .1, .1, .1, .9],  
 [.9, .1, .1, .1, .9],  
 [.9, .1, .1, .1, .9],  
 [.9, .9, .9, .9, .9]),  
 (.1, .1, .1, .1, .1, .1, .1, .9)),  
  
 *# 8. H* (([.9, .1, .1, .1, .9],  
 [.9, .1, .1, .1, .9],  
 [.9, .9, .9, .9, .9],  
 [.9, .1, .1, .1, .9],  
 [.9, .1, .1, .1, .9]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .9)),  
  
 *# 9. M* (([.9, .9, .9, .9, .9],  
 [.9, .1, .9, .1, .9],  
 [.9, .1, .9, .1, .9],  
 [.9, .1, .9, .1, .9],  
 [.9, .1, .9, .1, .9]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .1, .9)),

*# 10. L* (([.9, .1, .1, .1, .1],  
 [.9, .1, .1, .1, .1],  
 [.9, .1, .1, .1, .1],  
 [.9, .1, .1, .1, .1],  
 [.9, .9, .9, .9, .9]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .9)),

*# 11. =* (([.1, .1, .1, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .1, .1, .1],  
 [.9, .9, .9, .9, .9],  
 [.1, .1, .1, .1, .1]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .9)),

*# 12. Hexagon* (([.1, .1, .9, .1, .1],  
 [.1, .9, .1, .9, .1],  
 [.1, .9, .1, .9, .1],  
 [.1, .9, .1, .9, .1],  
 [.1, .1, .9, .1, .1]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .9)),

*# 13. Diamond* (([.1, .1, .9, .1, .1],  
 [.1, .9, .1, .9, .1],  
 [.9, .1, .1, .1, .9],  
 [.1, .9, .1, .9, .1],  
 [.1, .1, .9, .1, .1]),  
 (.1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .9))]