Artificial neural networks applied to character recognition

Matthew J. Urffer

CS 528 Machine Learning

***Abstract*—Artificial neural networks can be applied for the efficient recognition of optical characters allowing for the efficient conversion of characters into an electronic form. This work discuss the implementation of such a network in the MATLAB computing environment using a back propagation technique. The effects of network performance on the network architecture were explored; specifically the number of nodes in the hidden layer, the hidden layer function, and the learning rate. The developed network was tested against a feedforward neural network in `nntool`. The MATLAB trained network had an accuracy of 81%, where the developed network had an accuracy of 94%. Of the ten letters the network was trained to identify (A,C,D,E,F,G,H,L,P) A is the letter in which the network has the hardest problem classify, while L is the easiest.**

***Index Terms*—Artificial neural networks, character recognition,optical character recognition**

I. Introduction

OPTICAL character recognition is essential for the conversion of printed text into an electronic file. Applications include the sorting of postage, importing large texts for data mining, and translation of medical records [1]. Typically this is accomplished by preprocessing the document, separating out individual characters, and then extracting features from those characters [1], [3]. The extracted features are then used as inputs to a classifier. Artificial neural networks, with the ability to provide an expressive hypothesis space without sacrificing program complexity are an ideal machine learning technique. An artificial neural network typically consist of fully connected graph between an input layer, optional hidden layers, and an output layer. These structures can be concisely represented in matrices, providing a natural formulation of the network evaluation as matrix multiplication and addition for which the MATLAB environment is well suited.

II. Methodology

A. Artificial Neural Network Architecture

The artificial neural network was written in MATLAB attempting to provide a similar function call as MATLAB's Neural Network Toolbox. The developed code has the ability of automatically determining the input and output layer size, in addition to the capability for a user supplied number of nodes in the hidden layer. The network transfer functions were hard-coded as `logsig` and `softmax`, with the ability to provide alternative transfer functions by toggling the performance field of the class (II-A). The weight matrices W_1 and W_2 had their initial values chosen randomly.

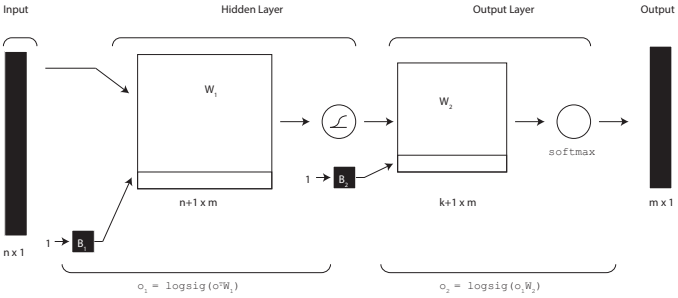

Figure 1. Artificial neural network diagram. The input is assumed to be a vector of size n with k hidden nodes and m outputs.

B. Network Training

Network training was accomplished using supervised stochastic back propagation algorithm with a training set for weight updates and a validation set for calculating the network's performance [2]. Each input $\vec{o}$ is chosen randomly from the set of input patterns P. The input is propagated through the network with transfer functions s_1 and s_2, saving the output of each layer $(\vec{o_1}, \vec{o_2})$. The error is computed and the derivative is calculated. Finally the weights are updated with the back propagated error, scaled by the learning rate. It it is important to note that $\bar{W}_2$ is a sub-matrix of W_2 that has the basis weights removed. The algorithm is outline in 1. Network output errors were computed with the built-in `sse` and `mse`; the sum squared error and the mean squared error, respectively. There were 1,000 training examples provided (divided equally among the characters) and 2,500 examples in the validation and test data sets.

Networks were initially trained out to 500 epochs without noticing a large increase in the class aggregate sum square error that is indicative to over-fitting. However, overfilling was observed in the individual class error (2). demonstrates the performance of individual classes as a function of training epochs for a network with 20 hidden nodes and the `logsig` transfer function. The minimum in the error function is also visible around 75 epochs; this was the reason why 75 epochs was chosen as the maximal number of epochs to train for the best network performance. In the following studies on network architecture the networks were trained to 250 epochs in order to observe any long term trends.

III. Results

Networks where trained for various ranges of the initial weights, varying the number of hidden nodes, varying the hidden node function, and varying the learning rate. After

Algorithm 1 Implemented backpropogation algorithm [2]

while $error > goal$ **do**
  for all $(\vec{o}, \vec{t}) \in P, T]$ **do**
    $\vec{o_1} = s_1\left(\vec{o}^T W_1\right)$
    $\vec{o_2} = s_2\left(\vec{o_1} W_2\right)$
    Compute error and derivatives
    $\vec{e} = \vec{o_2} - \vec{t}$
    $D_1 = \begin{bmatrix} o_1^1(1-o_1^1) & & & \\ & o_1^2(1-o_1^2) & & \\ & & \ddots & \\ & & & o_1^n(1-o_1^n) \end{bmatrix}$
    $D_2 = \begin{bmatrix} o_2^1(1-o_2^1) & & & \\ & o_2^2(1-o_2^2) & & \\ & & \ddots & \\ & & & o_2^m(1-o_2^m) \end{bmatrix}$
    Back-propagate the error up the output of each layer
    $\vec{\delta_2} = D_2 \vec{e}$
    $\vec{\delta_1} = D_1 \bar{W}_2 \vec{\delta_2}$
    Update the weight matrices
    $W_1 = W_1 + \left(-\eta \vec{\delta_1} [\vec{o}_1, 1]\right)^T$
    $W_2 = W_2 + \left(-\eta \vec{\delta_2} [\vec{o}_2, 1]\right)^T$
    compute $error$

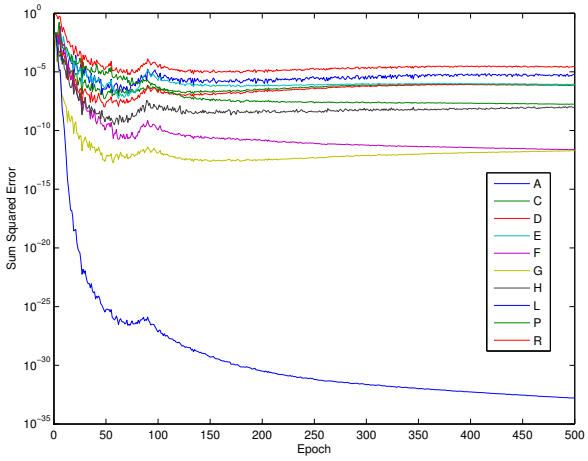

Figure 2. Performance of individual classes for each weight update, plotted versus epoch. There are 1,000 weight updates per epoch.

settling on optimal parameters the network was compared against a similar network from the `nntool`. The results of this network are shown as well as an analysis of the confusion matrix. It should be recalled that training is a random process; a network can be be trained different times with the same architecture and still converge to different local minima in the error surface (3). Therefore, network performance shown is meant to be representative; different training will yield similar but numerically different results.

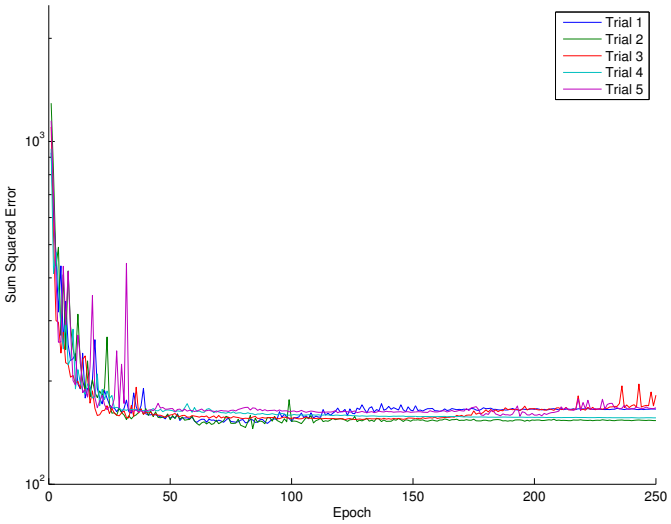

Figure 3. Performance of a network trained multiple times. The randomness in the training is visible in the different values of the asymptotic sum squared error.

Table I
NETWORK ACCURACY (AFTER 250 TRAINING EPOCHS) AND INITIAL WEIGHT RANGE

Initial Weight Range	Accuracy after 250 training epochs
$[-0.005, 0.005]$	0.951
$[-0.125, 0.125]$	0.962
$[-0.25, 0.25]$	0.956
$[-0.375, 0.375]$	0.956
$[-0.5, 0.5]$	0.957
$[-0.625, 0.625]$	0.960
$[-0.75, 0.75]$	0.960
$[-0.875, 0.875]$	0.961
$[-1, 1]$	0.966

A. *Initial Weights*

The initial weights play a fundamental rule in how quickly the networks learn. A study was completed with 20 nodes in the hidden layer and a learning rate of 0.75 in order to determine the optimal weights for the hidden layer. In all cases the weights where initialized randomly between the interval. While it was observed that the final accuracy was not greatly effected by the initial weights, the initial errors were all effected, as well as what local minima the network settled on, as shown by the asymptotic sum squared error in 4. The effect of initial weights can also be seen in 7 where the families of curves follow each other because of the initial weights.

B. *Number of hidden nodes*

Networks were trained with a variable number of nodes in order to determine the optimal number of hidden nodes. It was initially thought that with $n = 96$ binary inputs only $k = \lceil \log_2 n \rceil$hidden nodes would be required to provide an internal representation of the input vector but that proved erroneous. It was determined that anywhere between 20 to 40 would be sufficient (III-B). As the number of hidden nodes increased the network converged faster to weights that minimized the error

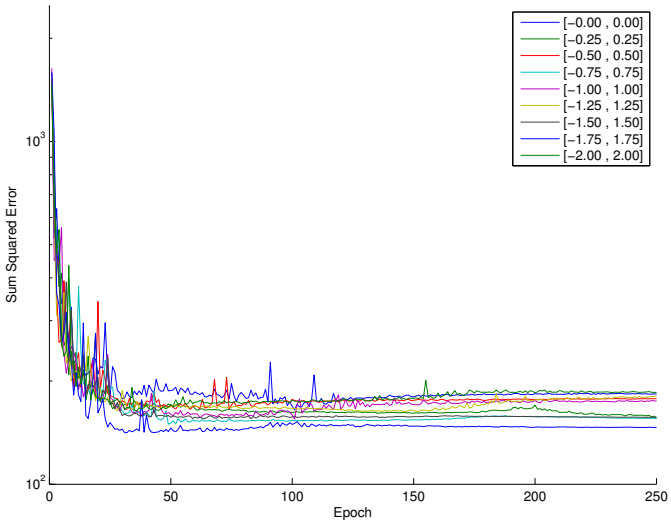

Figure 4. Network performance of initial weight range

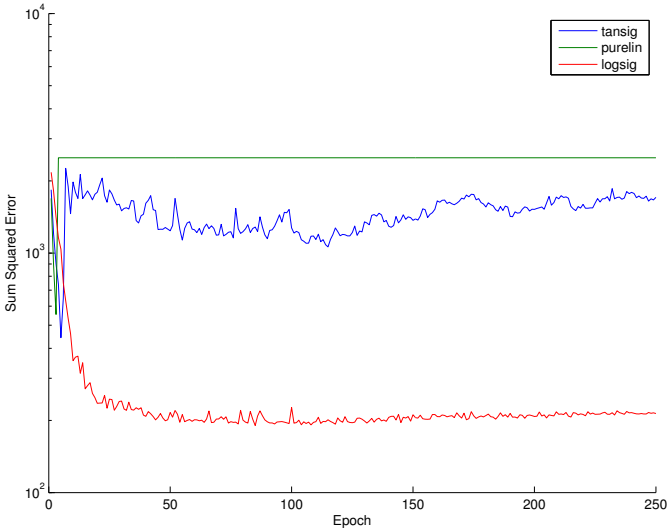

Figure 6. Network training performance for different hidden layer node functions

on the training data set, but past 15 nodes the improvement was minimal.

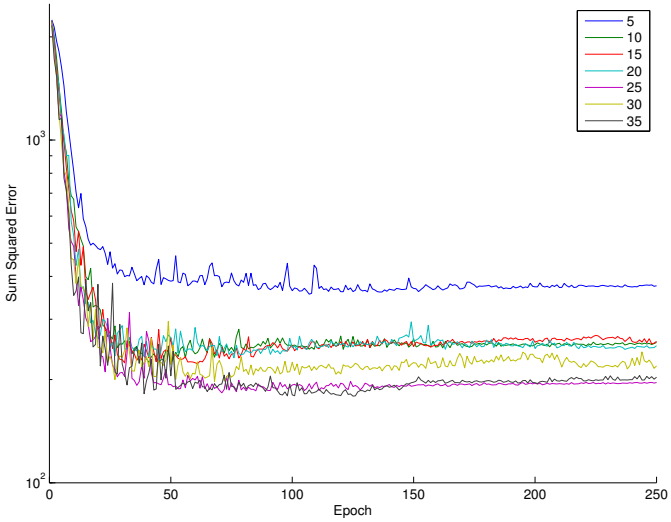

Figure 5. Network performance as the number of hidden nodes

C. Hidden node type

Three functions (`logsig`, `tansig`, and `purelin`) where tested for there impact on the network performance and training (6). In all cases the output layer was set as the softmax function. It was observed that the `tansig` and `purelin` functions did not train, while the logsig reached its training goal in slightly over 50 iterations. The `logsig` ($logsig\,(n) = 1/1+e^{-n}$)function is a natural match for this problem because the output weights should be between zero and one, while the `tansig` ($tansig\,(n) = 2/1+e^{-2n} - 1$)function allows for the outputs to be between negative one and one. The `purelin` function actually implements a preceptron network as the hidden layer, and continually updated weights such that they never came within a range for a meaningful network output.

D. Learning Rate

Networks (composed of 20 hidden nodes with the `logsig` transfer function) were trained with various initial learning rates. It was recognized immediately that having a constant learning rate caused additional training iterations due to the gradients providing too large of weight updates, so an exponential decay term was added to the error. Figure 7 demonstrates the sum squared error for initial learning rates between zero and one. Learning rates below 0.2 caused the network to train very slowly, in most causes having a sum squared error greater than 200 after 250 training epochs. The optimal initial training weight is then close to 0.75. The closeness of the lines suggest that between 50-100 training epochs is all that is required to adequately train the network. It should be noted that the sum squared error for each of the selected training epochs follow each other because they are all on the same network, which has a large dependance on the randomly chosen initial hidden weights.

E. Benchmarking against published software

A feedforward neural network was setup and trained using the `nntool` suite of MATLAB and it's performance was compared against the developed implementation. Both networks were restricted to have the same architecture (20 hidden nodes with the `logsig` transfer function), but the MATLAB's function trained with the Levenberg-Marquardt algorithm while the developed implementation utilized the gradient descent back-propagation outlined above. While the MATLAB network trained much quicker (16 iterations), the accuracy of it was 81%. The developed network was set to stop training after 75 iterations, reaching an accuracy of 94%.

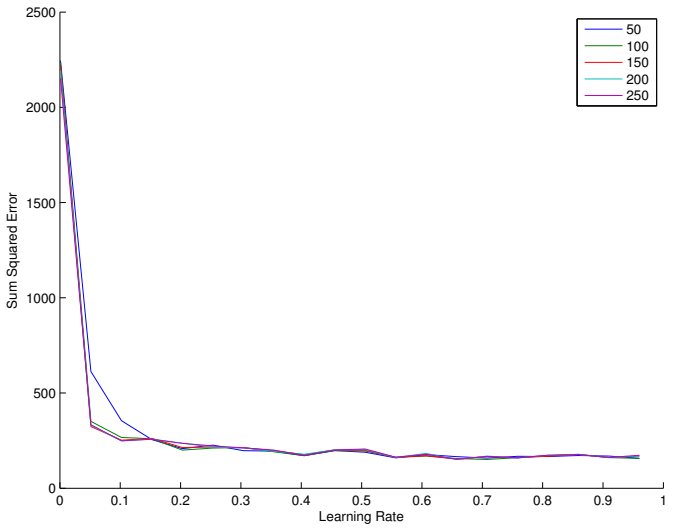

Figure 7. Network error performance for selected learning rates at selected epochs.

Table II
CONFUSION MATRIX OF DEVELOPED AND TRAINED NETWORK

		Predicted Class									
		A	C	D	E	F	G	H	L	P	R
Actual Class	A	229	0	0	1	4	0	11	0	1	4
	C	0	247	0	0	0	3	0	0	0	0
	D	2	9	246	0	0	0	2	0	0	0
	E	3	0	0	238	3	0	2	0	0	4
	F	10	5	0	2	209	0	0	0	29	0
	G	1	0	0	0	0	244	0	0	0	0
	H	12	0	1	2	6	0	225	0	0	4
	L	0	0	1	0	0	0	0	249	0	0
	P	9	0	1	0	8	0	0	0	323	0
	R	9	0	2	2	0	0	0	0	1	236

IV. CONCLUSIONS AND FUTURE WORK

Artificial neural networks have been successfully applied to charter recognition using a pixilated image, reaching an accuracy of 94%. TableII shows the confusion matrix for the generated neural network. It is observed that certain letters, such as L and G, are very distinct. It is also possible to observe which letters have similar structures by noting what they are confused with; for example H and A both have the middle cross-bar and are confused with each other. The same goes for F and H, but F is not confused with H because F does not have two vertical lines extending downwards. P and F are also confused, which is because P and F only differ in that a P is a fully connected F. These results are also noticeable in 2, where the individual class errors can be seen and it observed that the sum squared error for class L dramatically drops per weight update iteration.

Future work could be directed at the preprocessing or ordering of the input files. Improved accuracy might be achieved by reducing the problem space by reducing the number of pixels feed into the network. One such method could be to simply the sum the rows of the input matrix and use the sums as the network input. Another method would be to use the context of the letter as an input; this could be provided by what the preceding letter was in a word, and the a probability score could be assigned to what letter is most probable to come next. This scheme would be helpful in classifying letters that are intrinsically similar (such as the P and F) because few words contain a sequences of P's and F's.

ACKNOWLEDGMENTS

The assistance of Matthew Lish is gratefully acknowledged in the algorithm development.

REFERENCES

[1] LD Jackel. Neural-net applications in character recognition and document analysis. *AT&T Bell Laboratories*.

[2] R Rojas. The backpropagation algorthim. In *Neural Networks, a systematic introduction*.

[3] Michael Sabourin and Amar Mitiche. Original contribution: Optical character recognition by a neural network. *Neural Netw.*, 5(5):843–852, 1992.