THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

AN ENSEMBLE OF CONVOLUTIONAL NEURAL NETWORKS FOR THE RECOGNITION OF HANDWRITTEN DIGITS

By

MATHEW MONFORT

Major Professor: Xiuwen Liu
Committee Member: Zhenhai Duan
Committee Member: Daniel G. Schwartz

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2011

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This paper describes a neural network ensemble that is made up of 8 convolutional networks that are based on the Lenet-5 architecture, with a denser network, the addition of a personal bias for each neuron, and the move from a final Euclidean Radial Basis Function layer, (RBF), into a regular convolutional layer. A transitional training program was created in order to train the networks using a stochastic levenberg-marquadt gradient descent algorithm with the addition of an adaptive momentum term and the integration of a neuronal ranking system. The networks are then trained in parallel on the MNIST training set that is elastically distorted and then combined together in order to classify the images in the MNIST testing set. There are two different methods used to classify the images. The first is the regularly used average output method in which the average output vector of all of the networks is used to recognize a pattern. The other method used is a confidence based method that uses the network that has the highest degree of confidence in its decision to recognize an image. A controlled unsupervised learning system was then created to allow the networks to continue to learn from experience efficiently after the initial training phase. In addition to all of this, a system was included where the weights of the connections, as well as the outputs of the neurons, were slightly mutated during training in order to simulate real world neuron failure, and noise, to allow for improved generality. The results section will show that this method is able to achieve a very accurate error rate of 0.38% on the MNIST database.

CHAPTER 1

INTRODUCTION

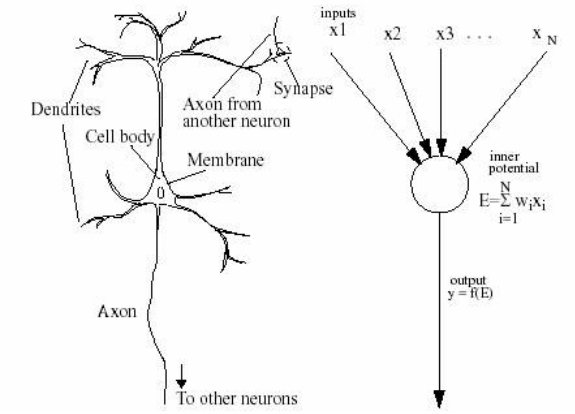

Figure 1.1: A biological and an artificial neuron.

Artificial neural networks have been around for quite a while. However, an increase in computing technology and innovation by various researchers has allowed for these networks to approach a new level of usability. This paper will describe a method that combines many different techniques into a single neural network in an attempt to maximize it's efficiency and accuracy.

This paper describes a neural network ensemble that is made up of 8 convolutional networks that are based on the Lenet-5 architecture introduced in [9], with a denser network, the addition of a personal bias for each neuron, and the move from a final Euclidean Radial Basis Function layer, (RBF), into a regular convolutional layer. A stochastic levenberg-marquadt gradient descent algorithm[9] is used to train the network with the addition of an adaptive momentum term[10]. The networks are then trained on the MNIST training set that is elastically distorted using the technique described in [11] and the values listed in [4]. The networks are trained in parallel and then combined together in order to classify

the images in the MNIST testing set. There are two different methods used to classify the images. The first is the regularly used average output method in which the average output vector of all of the networks is used to recognize a pattern. The other method used is a confidence based method that uses the network that has the highest degree of confidence in it's decision to recognize an image. The results section will show that this method is able to achieve a very accurate error rate of 0.4%.

Chapter 2 of this paper will introduce and explain the different aspects of how a feedforward artificial neural network works. It will then explain the characteristics of convolutional networks and describe the specific architecture used in this project. Chapter 3 will describe the different techniques that are used to train the networks to recognize images. Chapter 4 will then display the results obtained from training and testing the networks on the MNIST database. Finally, chapter 5 will conclude the report with an overview of what was done.

CHAPTER 2

NEURAL NETWORKS

2.1 Biological Model

When designing artificial neural networks it is important to remember that they are based upon the biological neurological system that resides in every person. Therefore, in order to fully understand how an artificial neural network works, it is necessary to first study how a biological neural network functions.

In 1943 McCulloc and Pitts introduced a set of simplified neurons that represented models of biological networks that were formed into circuits that could perform computational tasks. The core function of a neuron is to receive signals as input, process the input, and then send signals as output. This procedure is repeated by all of the neurons in a network. The combination of such a large number of simple processors is what allows the brain to be so efficient and flexible.

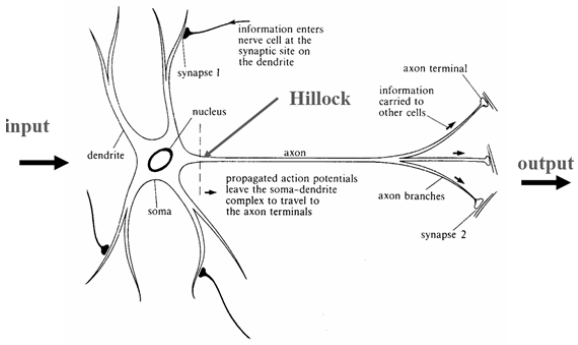

Figure 2.1: A Biological Neuron.

Figure 2.1 shows the structure of a biological neuron. The neuron has input connections from other neurons through it's dendrites. The input signals then move into the cell body

where the input signals are processed into the propagated action potentials that are sent for output. The output signal is then sent along the axon of the neuron and out to the axon branches that form output connections with other neurons. The synapses are the gaps between the axons/dendrites of one neuron and the axons/dendrites of another. The strength of the synapse affects how well the output signal of a neuron will be passed to another. This allows a network to adjust the strength of it's synapses in order to fit process it's signals efficiently. This is how a network learns.

2.2 Mathematical/Artificial Model

2.2.1 Artificial Neuron

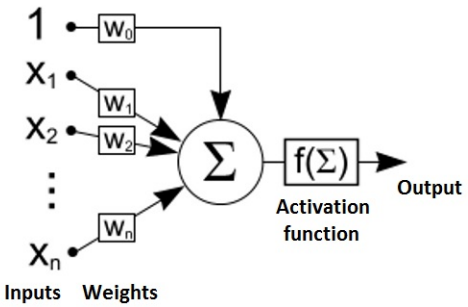

Figure 2.2: An Artificial Neuron.

The mathematical model of a neuron works as a simple function that simulates the process of a biological neuron. The neuron receives input from other neurons, the inputs are then altered by connection weights that represent the synapses of the neurons connections. Since the synapse of a neuron gives strength to the connections, the synapses are modeled as weights that can be altered in order to adjust the power of an output connection. These weights can be either positive or negative values which respectively represent excitatory and inhibitory connections. The magnitude of the weight would determine the strength of the connection.

The output signal of the neuron is achieved by summing together all of the input values, after they are modified by the input weights, with a trainable bias. This is refereed to as linear combination. An activation function is then applied to this input signal to determine the output signal. The activation function is usually chosen so that the output of the neuron lies either between 0 and 1, or -1 and 1. The purpose of the activation function is to make

sure that the output of each neuron is within a specified range of values. This prevents any 'blowing up' of values within the network.

With this model in place, the process of a neuron can be represented by the following functions:

$$y_i = b_i + \sum_{j=1}^{k} w_{ij} x_j \tag{2.1}$$

$$x_i = \phi(y_i) \tag{2.2}$$

Here x_i is the output value of the ith neuron, ϕ is the activation function, k is the total number of input connections, w_{ij} is the weight parameter applied to the output value x_j, and x_j is the output value of the jth neuron that neuron i has an incoming connection from. b_i is the trainable bias that is applied to the total input sum of neuron x_i.

Once this function is applied to an entire network of neurons with one input layer and one output layer, the result is a functional artificial neural network.

Biases

Adding trainable biases into the input signals of each neuron is very important in adjusting the network. A bias acts as a neuron whose output value is always 1 and has a connection with a trainable weight to each neuron.

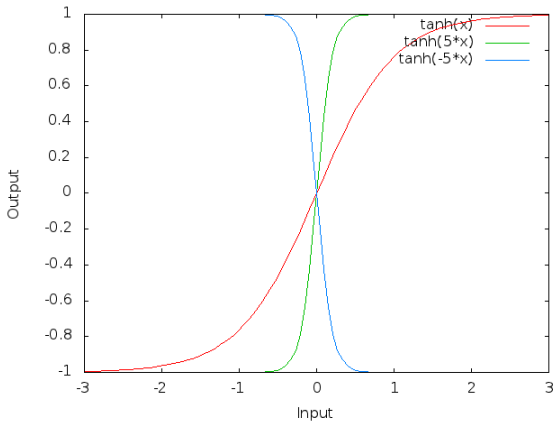

Figure 2.3: Activation function output with no bias.

The function of the bias is to shift the output of the activation function to a more desirable value. Consider the graph in figure 2.3. There are three plots of the hyperbolic

tangent sigmoid function that will be described in the section on activation functions. Each plot has the same input, but a different weight value. This shows that adjusting the weight value only increases the steepness of the sigmoid function.

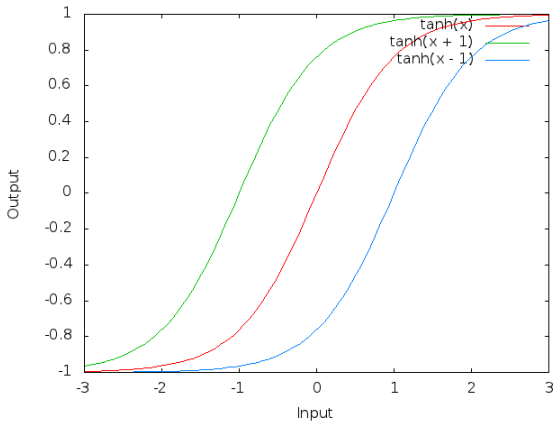

Figure 2.4: Activation function output with bias.

While adjusting the weights in the network does have a positive result, consider the case where you would like to change the output of the function in figure 2.3 with input 1 to 0. This is where incorporating a trainable bias becomes useful. The bias value can be added to the input of the sigmoid function in order to shift the output of the function to the right or left in the graph. Figure 2.4 demonstrates the shift that can be achieved by adding a bias of varying values. It is clear that including trainable bias values in a network is important in achieving a high level of accuracy.

2.2.2 Artificial Network

An artificial neural network consists of an input layer, one or more hidden layers, and an output layer. The input layer takes input from a pattern and then forward propagates it through the neurons in the network layer by layer. The output layer consists of one neuron for each category that the pattern can be classified as. The neuron in the output layer with the highest output value represents the category that the input pattern is classified as. The process that propagates the signals of the neurons through the network is called forward propagation.

The next section explains the process of choosing an appropriate activation function.

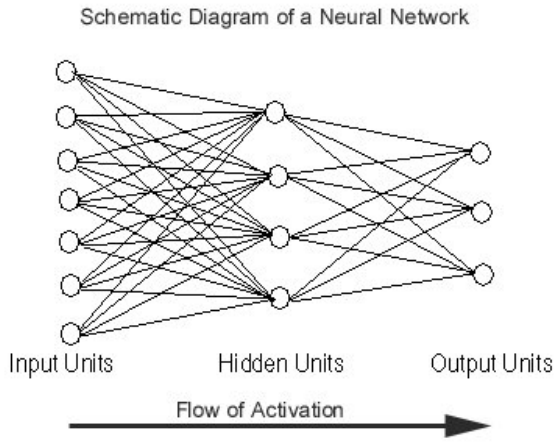

Figure 2.5: An Artificial Neural Network Model.

2.3 The Activation Function

The activation function of an artificial neural network acts as a squashing function that places the output of a neuron within a certain range of values. There are generally three types of activation functions that are used on artificial neural networks. These are the Threshold function, the Piecewise-Linear function, and the Sigmoid function.

2.3.1 The Threshold Function

The Threshold function is used to assign an output value to a neuron based on a range of input values to the function. An example would be a function that outputs 0 if the input is below 0, and 1 if it is not.

$$\phi(y_i) = \begin{cases} 1 & y_i \geq 0 \\ 0 & y_i < 0 \end{cases} \tag{2.3}$$

One of the main issues here is that the threshold function only returns a binary value of 1 or 0. This places a heavy restriction on the network and is not an accurate model of how a biological neuron functions. A biological neuron is an analog function that can returns a range of different values. This allows for a much more complex relationship to exist between the neurons in a network. Therefore, it is necessary for an artificial network to contain analog neurons in order to correctly model the complex operations that are undertaken by a biological network.

The Piecewise-Linear Function

The Piecewise-Linear function is similar to the Threshold function as it can take on values depending on certain thresholds. However, it also has the ability to take on intermediate values depending on an amplification factor in a certain region of linear operation. This helps to solve the problem of the digital threshold function by returning an analog value.

$$\phi(y_i) = \begin{cases} 1 & y_i \geq 2 \\ y_i & -2 < y_i < 2 \\ 0 & y_i \leq -2 \end{cases} \tag{2.4}$$

The issue here is that the function is not continuously differentiable. This is a necessary requirement that is explained in detail in the chapter on learning.

2.3.2 The Sigmoid Function

The sigmoid function is the most commonly used activation function in artificial neural networks. The reason for this is that it more accurately represents the actual functions of a biological neuron. This is due to its analog nature which allows a neuron to have an output that is within a range of values. This is in contrast to the digital on/off behavior of the Threshold function. The Piecewise-Linear function is a combination of both the Threshold function and the Sigmoid function.

The fact that the Sigmoid function is analog allows a network to be carefully tuned to an optimal solution. The range of values that the Sigmoid will output depends on which Sigmoid function is used.

The next three sections will discuss three of the more common Sigmoid functions that are found in artificial neural networks. These are the traditional logarithmic Sigmoid function, the hyperbolic tangent Sigmoid function, and the softsign Sigmoid function.

The Logarithmic Sigmoid Function

$$\phi(y_i) = \frac{1}{1 + e^{-y_i}} \tag{2.5}$$

The logarithmic Sigmoid function was once the most commonly used Sigmoid function. This function places the output of a neuron on the range between 0 and 1 allowing the function to return an analog value. Another positive aspect of the logarithmic sigmoid function is that it is continuously differentiable. This allow it to be used in the neural network learning algorithm described in the next chapter.

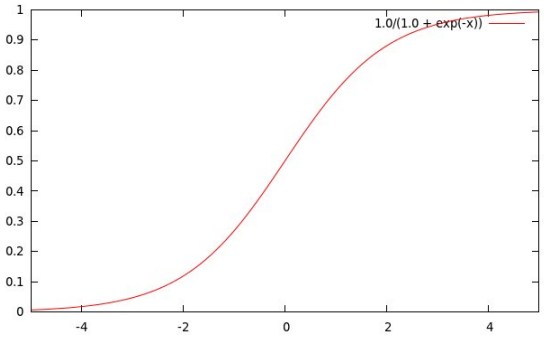

Figure 2.6: The Logarithmic Sigmoid Function.

While logarithmic sigmoid function does work well with artificial neural networks, it is not ideal. First of all, the function is not symmetric about the x-axis. This forces the outputs of a neuron to lie in the range of 0 and 1.

This means that the function only returns excitatory values and not inhibitory values. This does not allow a neurons output to represent the negative strength of it's input. Instead, when the neuron receives a negative value as it's total input, it will only output 0 regardless of the value of the negative input. This is not good model to use. It is more appropriate to have a neurons output represent it's negative input on a the same range as it does it's positive input. Therefore, it is necessary to use a sigmoid function that is symmetrical about the x-axis.

The Hyperbolic Tangent Sigmoid Function

$$\phi(y_i) = \tanh(y_i) = \frac{e^{2y_i} - 1}{e^{2y_i} + 1} \tag{2.6}$$

The hyperbolic tangent Sigmoid function, introduced in [9] fixes the issue of the non-symmetrical traditional logarithmic Sigmoid function by returning an output on the range from -1 to 1.

This allows a neuron to represent negative input in the same way that it represents positive input. This method, and variations of it are the most used today for this reason.

The function can be rewritten to:

$$\phi(y_i) = A \tanh(By_i) \tag{2.7}$$

Here A and B are constant values that can be used to alter the sigmoid function to fit specific needs.

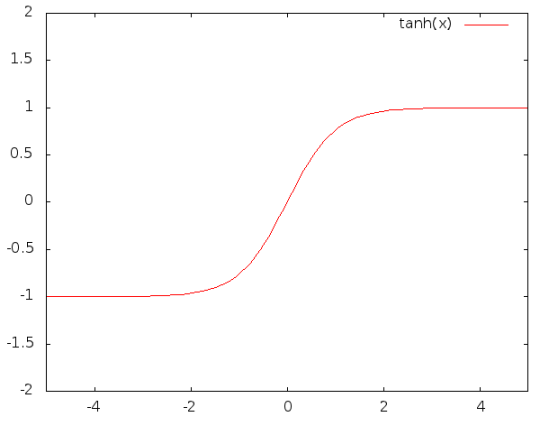

Figure 2.7: The Hyperbolic Tangent Sigmoid Function.

Using this idea, an improvement of the hyperbolic tangent sigmoid function was described in [9]. The function is multiplied by 1.7159 and the input is multiplied by $\frac{2}{3}$. This allows for $f(1) = 1$ and $f(-1) = -1$, while keeping the input within the optimal operating domain of the function. It also allows for the absolute value of the second derivative of the sigmoid function to reach maximum values at 1 and -1. According to [9], this improves the convergence of the network towards the end of the learning process.

$$\phi(y_i) = 1.7159 \tanh(\frac{2}{3}y_i) \tag{2.8}$$

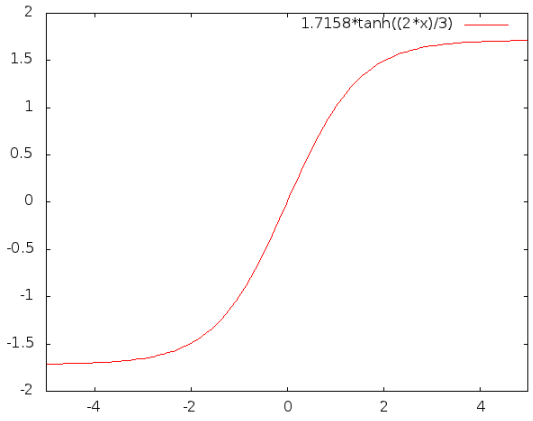

Figure 2.8: Dr. Lecun's Hyperbolic Tangent Sigmoid Function.

One issue here, as explained in [2] is that this sigmoid function experiences a sequential

saturation phenomenon which is not yet understood.

Another negative aspect of the hyperbolic tangent sigmoid is that it's exponential structure causes it to have very steep asymptotes. This means that there is not a slow transition of output values along the domain of the function. This causes issues when the input values of the function lie close to edges of it's domain.

The Softsign Sigmoid Function

$$\phi(y_i) = \frac{y_i}{1 + |y_i|} \tag{2.9}$$

The softsign sigmoid function is a sigmoid function that is symmetric about the x-axis and avoids the issue of sequential saturation. When using the softsign sigmoid the layers of the network saturate simultaneously rather than sequentially[2].

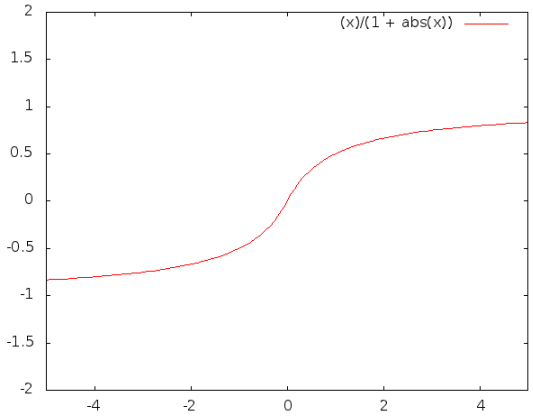

Figure 2.9: The Softsign Sigmoid Function.

The asymptotes of the softsign sigmoid are also smoother than those of the hyperbolic tangent sigmoid. Therefore, the function experiences a slower transition between the two edges of it's range. This is due to it's polynomial, rather than exponential, structure. The smoothness of the function results in a slower transition toward it's extremities and a more analog representation of it's input. The histogram of the activation values also lie in an area that has "substantial non-linearity but where the gradients would flow well"[2].

$$\phi(y_i) = \frac{ay_i}{b + |y_i|} \tag{2.10}$$

The function can be rewritten using constant values in the same way that the hyperbolic tangent sigmoid function did. There does not need to be a third constant value that is

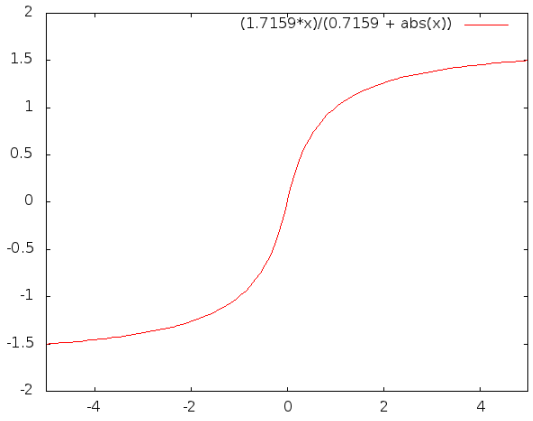

Figure 2.10: The Modified Softsign Sigmoid Function.

multiplied with $|y_i|$ since it can be represented by the other two constant values. For example, if $A = 6$, $B = 4$, and $C = 2$, where C is the other constant value, then the function can be rewritten so that $A = 3$, $B = 2$, and $C = 1$. This can be applied with any value of C, so C can be assumed to equal 1.

Here it is helpful to apply all of the insights from [9] into this new sigmoid function. In order for the function to equal 1 when the input is 1, and -1 when the input is -1, the equation $A = B + 1$ must be true. Also, A must equal the maximum approachable value of the function. This is due to the fact that the limit of the function as y_i approaches infinite is equal to A and the limit of the function as y_i approaches negative infinite is equal to $-A$. Therefore, $A = MAX$ and $B = MAX - 1$ where MAX is the maximum value of the absolute value of the function.

Since the network used in this project is based on one that uses the hyperbolic tangent sigmoid function described in [9], it seems appropriate to keep the maximum value constant between the two sigmoid functions. Therefore, the new values will be $A = 1.7159$ and $B = 0.7159$. This allows for $f(1) = 1$ and $f(-1) = -1$ while keeping the maximum of the absolute value of the function to be 1.7159. Figure 2.10 shows the graph of this function.

One limitation to the softsign sigmoid function is that the absolute value of the second derivative of the sigmoid function does not reach maximum values at 1 and -1. Since [9] states that this improves the convergence of the network towards the end of the learning process, the softsign sigmoid function may not perform as well as the hyperbolic tangent sigmoid function on high accuracy networks.

2.4 Network Architectures

While the activation function determines how an individual neuron will act, the structure of the network will determine how the entire network will function. One of the important aspects in the structure of a network is how the connections in the network will be organized. Some of these organization techniques will be presented in this section.

2.4.1 Shallow vs. Deep

Shallow architectures are limited by their lack of depth. [1] states that functions that can be compactly represented by an architecture of depth k, would require an exponential number of computational elements to be represented by an architecture with a depth of $k-1$. Because of this, poor generalization may be experienced when using an architecture that is too shallow for the problem. [3] also states that deep architectures yield a greater expressive power than their shallow peers. Therefore, increasing the depth will not only reduce the necessary complexity of a network, it will also improve generalization.

Deep architectures are able to represent a wide family of functions in a more compact form because they trade breadth for depth[3]. Each succeeding layer combines the features of the previous layer in order to form a higher level of abstraction than that of the layer before it. This increasing level of abstraction from layer to layer allows deep networks to produce strong generalizations for highly varying functions.

2.4.2 Convolutional Networks

Convolutional networks are special types of deep networks that were inspired by the structure of the biological visual cortex. Convolutional networks do not suffer from the typical convergence issues of other deep architectures[3]. There is currently no definitive explanation for why this is true, however, it may be due to the heavily constrained parameterization and asymmetry of the convolutional architecture. Each layer of a Convolutional network represents sets of progressively higher features, and the last layer represents categories for classification. The typical convolutional network contains a feature detection layer followed by a feature pooling layer. The network can have many pairs of detection/pooling sets, which are then followed by one or more classification layers.

The main differentiation between Convolutional networks and most other deep architectures is the use of local receptive fields, shared weights, and spatial/temporal subsampling[9]. For example, the Convolutional network Lenet-5 is shown in figure 2.11. The input layer receives images of characters that are size normalized and centered. Each neuron in a layer receives as input the output of a small neighborhood of neurons in the previous layer. These local receptive fields can extract small features such as edges, corners, and

end-points. These features are then combined together in the next layer in order to extract higher level features. This process is repeated in each layer so that a network with a large enough depth will be able to detect very high level features from the input images. Distortions and shifts in the input of the network can result in varying positions of the features that are to be detected. This problem is solved by sharing weight vectors among receptive fields in different areas of the input. Therefore, neurons are organized in feature maps that contain identical weight vectors. Each layer of the network contains a set of feature maps that allow for the detection of a range of features within the same area of the input map. The sub-sampling layers of a Convolutional network reduce the resolution of the features that were extracted in the preceding layers. This reduces the sensitivity of the network to shifts and distortions.

Convolutional Layers

One of the most important characteristics of a convolutional network is the use of convolutional layers. In a convolutional layer, the feature maps of the previous layer are convolved using a trainable kernel and then combined with a trainable bias. The result of this combination is then run through an activation function in order to produce the output for the feature map on the convolutional layer. Each output map may contain multiple feature maps as input from the preceding layer. The equation below shows how this works:

$$x_n^i = F(b_i + \sum w_n^{ij} x_{n-1}^j) \tag{2.11}$$

F is the activation function, x_n^i is output of the ith neuron on the nth layer, w_n^{ij} is the weight shared by the neurons i and j, and b_i is the bias weight that is applied to neuron i.

Sub-Sampling Layers

Sub-sampling is a technique that is used in some convolutional networks that can reduce computation time and increase the spatial and configurable invariance of the network. The sub-sampling layers used in this paper always follow a convolutional layer. The sub-sampling layers have the same number of feature maps as the previous convolutional layer has. This is so that the maps in the two layers match up. Each map in the sub-sampling layer has one quarter the number of neurons that are the preceding layers feature maps. Each neuron in a feature map of a sub-sampling layer takes as input a 2x2 neighborhood of neurons from the matching feature map in the preceding convolutional layer. These 4 neurons are then each multiplied by a trainable bias and divided by 4. This essentially finds the average of the outputs of the 4 neurons once they are combined with their trainable parameters. This average is then run through the activation function in order to create the output for the

neuron in the sub-sampling layer. This process is repeated in each sub-sampling layer in the network. The equation below shows how this works:

$$x_n^i = F(\frac{\sum w_n^{ij} x_{n-1}^j}{4}) \tag{2.12}$$

F is the activation function, x_n^i is the output of the ith neuron on the nth layer, w_n^{ij} is the weight shared by the neurons i and j.

Output Layer

The output layer of a convolutional network is normally a convolutional layer that has as many neurons as there are categories to classify. In the case of digit recognition, this would be ten with each neuron corresponding to a single digit. Once a pattern has been run through the entire network, the final output values of the neurons in this layer are analyzed and a classification is chosen based on which neuron in the output layer has the highest output value. This is how a convolutional network recognizes an image. The next section will describe an example of a convolutional network and the modifications that were made to it for the purposes of this paper.

Lenet-5

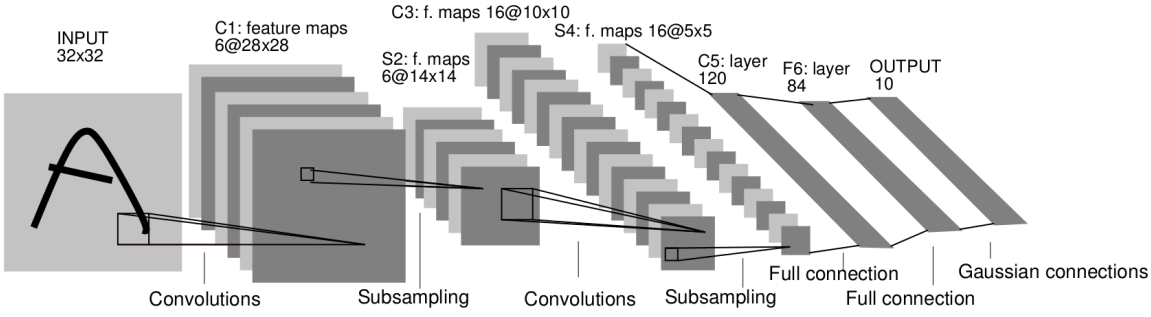

Figure 2.11: Lenet-5

Lenet-5, Figure 2.11, is a Convolutional network with 8 layers that is described in [9]. It was created to recognize handwritten numerical digits. The input layer takes in a 28x28 image that is then padded to 32x32. This allows any features that may reside on the edges of an image to lie in the center of the receptive field of the feature maps.

The first Convolutional layer has 6 feature maps that are 28x28 neurons. Each neuron has a receptive field that is a 5x5 neighborhood of the input image. These neighborhoods

overlap to allow for multiple features to be detected in the same locations. The outputs of the receptive field are convolved with the weight vector of the feature map and then added to a trainable bias. The biases in lenet-5 are shared among members of a feature map. This sum is then put into the sigmoid function to produce the output of the neuron. The equation below shows how this is done. This is same equation described in the section on convolutional layers.

$$x_n^i = F(b_i + \sum w_n^{ij} x_{n-1}^j) \tag{2.13}$$

F is the sigmoid function, x_n^i is output of the ith neuron on the nth layer, w_n^{ij} is the weight shared by the neurons i and j, and b_i is the bias weight that is applied to neuron i.

The second layer is a sub-sampling layer with the same number of feature maps as the previous layer. Each feature map in this layer has half of the neurons that the feature maps in the previous layer have. The neurons in this layer take as input a 2x2 non-overlapping area of the corresponding feature map in the previous layer. The input is then averaged and added to a trainable bias before being passed through the sigmoid function to produce the output of the neuron.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Figure 2.12: Lenet-5 Connection Table

This process is repeated in the next two layers, this time using 16 feature maps. The difference here is that while the first Convolutional layer had only one input map, the second one has 6. The feature maps of the second Convolutional layer are connected to the feature maps of the first sub-sampling layer according to the table in Figure 2.12. The columns represent feature maps in the second convolutional layer, the rows represent feature maps in the first sub-sampling layer, and an X denotes a connection between the two feature maps. This allows for the connections to remain within reasonable bounds.

After the second sub sampling layer, the network has two fully connected layers. One has 120 neurons, and the next has 84 neurons. The layer with 84 neurons is used to compute the Euclidean Radial Basis Function units (RBF), each unit representing a different class

of number. The output of each such RBF unit is computed by:

$$y_i = \sum_j (x_j - w_{ij})^2 \tag{2.14}$$

A Variation on Lenet-5

The network created for this project is almost identical to lenet-5 with some important distinctions. First, every feature map is connected to allow the features detected in each map to be considered in each layer. The second difference is the use of a unique personal bias for each neuron. The biases are not shared and each neuron has a unique trainable biases in [11]. The last difference in this network is the removal of the RBF layer. The output layer is replaced with a regular fully connected layer. This, again, is similar to the network described in [11]. This allows for simpler construction, propagation, and training throughout the network.

2.5 Artificial Neural Network Ensemble

One method of improving the accuracy of recognizing images is to use a network ensemble. In a network ensemble, multiple networks are used to recognize the same image. This increases the variability and generality in the training of the networks. Network ensembles have also become more attractive due to the increase in parallel computing. This allows multiple networks to be trained in a much shorter amount of time.

The main drawback to this method is the extra time that needs to be taken to train each of the networks. This can be alleviated, however, since the networks can be set up to run as separate entities, they could be set up to run in parallel. This would allow for each network to be trained simultaneously. They would only need to be combined when recognizing patterns.

2.5.1 Confidence Based Classification

The usual method of choosing how to classify an image is to use the average output. This is calculated by calculating the average output vector for all of the networks in the ensemble and then using that to identify the image.

Another method that can be used to identify an image is to use the different levels of confidence each network has in it's personal classification of the image. The network with the highest confidence in it's choice is used to identify the image.

The confidence of a network is determined by first having it classify an image. Now the mean squared error for the recognition of the image is calculated with the assumption that

the classification is correct. This is done by using the equation below:

$$E_n^p = \frac{1}{2}\sum(x_n^i - D_n^i)^2 \tag{2.15}$$

Here x_n^i represents the output of the ith neuron in the output layer. Each of these neurons corresponds to a category. A value of 1 for x_n^i means that that the image fits that category, a -1 means that it does not. D_n^i represents the desired output for x_n^i. In this case, if the network classifies an image to lie in category k, then D_n^i will be 1 for $i = k$ and -1 for $i \neq k$.

This confidence factor allows for a network to be chosen to decide the classification of a pattern depending on how confidant it is in it's decision. With this in place it is possible to use two, or more, differing networks to decide on what to recognize an image as based on how sure they are in their recognition.

CHAPTER 3

LEARNING

3.1 The MNIST Database

The MNIST is a database of handwritten digits with 60,000 training samples and 10,000 test samples. The images have been size-normalized and centered into 28×28 pixel images. This allows for the images to vary only slightly from sample to sample, freeing the network to concentrate on feature extraction. The MNIST database has become a standard by which to train and test the strength of an artificial neural network. The results on this database are also well documented, allowing it to be a suitable benchmark by which to test the network[3].

3.2 Network Initialization

3.2.1 Input Normalization

The values of the input pixels of each image are normalized to lie between the values of -0.1 and 1.175. The background will be -0.1 and the absolute foreground would be 1.175. All other values will fall between these two values. This allows for the mean input to be about 0.0, and the variance to be about 1.0. This is done to speed up the learning process[9].

3.2.2 Weight Initialization

When creating the network, the weights are initialized randomly initialized using a uniform distribution between -2.4/Fi and 2.4/Fi where Fi is the number of connections associated with that weight. This is done to keep the initial standard deviation of the weighted sums within the same range for all neurons while having a small enough magnitude to be applied to the sigmoid function properly[9].

3.3 Gradient Descent

Gradient descent is a method that has been used for many years to accurately train convolutional networks. The next few sections introduce some adaptive forms of gradient descent that allow the network to learn each training pattern at a rate based upon the error of the network.

3.3.1 Stochastic Diagonal Levenberg-Marquadt Gradient Descent

The stochastic Levenberg-Marquadt algorithm is an adaptive learning method that creates a unique learning rate that is used to update the weights in a neural network. It is explained in depth in [9] and [8]. The algorithm is very similar to the standard stochastic gradient descent algorithm. The main difference here is that second order derivatives are used to adapt the learning rate to the problem.

The mean squared error of the output map is used to determine the error of the network for each input. Therefore, the mean squared error function is first calculated for each neuron in the last layer.

$$E_n^p = \frac{1}{2}\sum(x_n^i - D_n^i)^2 \tag{3.1}$$

Here E_n^p is the mean squared error for the output of the network on training pattern p. x_n^i represents the output value of neuron i on layer n. D_n^i is the desired output of this neuron for the pattern p. Once this error is calculated, the first and second order partial derivatives of the error function can be calculated for each neuron in the output layer.

$$\frac{\partial E_n^p}{\partial x_n^i} = x_n^i - D_n^i \tag{3.2}$$

$$\frac{\partial^2 E_n^p}{\partial {x_n^i}^2} = 1 \tag{3.3}$$

These values will be back propagated through the network in order to determine how the parameters in the network will be updated. In order to do this, it is necessary to create a few more functions that will help with the calculation of these weight changes.

First, remember that the sigmoid function that is used in this network is the soft sign sigmoid. This function determines the output values of each neuron in the network. Consider the ith neuron in the nth layer, denoted x_n^i, with a total input value of y_n^i. The value of this neuron is calculated using the sigmoid function. In this case that is the hyperbolic

tangent function introduced in [9]:

$$x_n^i = F(y_n^i) = 1.7159 \tanh(\frac{2}{3} y_n^i) \tag{3.4}$$

The derivative of this function will then be:

$$F'(y_n^i) = \frac{3.4318}{3}(1 - \tanh(\frac{2}{3} y_n^i)^2) \tag{3.5}$$

Now the next first and second order partial derivatives of the error function can be computed:

$$\frac{\partial E_n^p}{\partial y_n^i} = F'(a_n^i) \frac{\partial E_n^p}{\partial x_n^i} \tag{3.6}$$

$$\frac{\partial^2 E_n^p}{\partial {y_n^i}^2} = F'(a_n^i)^2 \frac{\partial^2 E_n^p}{\partial {x_n^i}^2} \tag{3.7}$$

These values can then be used to calculate the partial derivatives of the error function with respect to the weight that is going to be updated, w_n^{ij}. This is the weight shared between neuron i on the nth layer and neuron j on the $n-1$ layer. Since the network used has a weight sharing architecture, the output values of each neuron that shares the weight w_n^{ij} with neuron i must be summed up in order to properly calculate the proper derivatives.

$$\frac{\partial E_n^p}{\partial w_n^{ij}} = \frac{\partial E_n^p}{\partial y_n^i} \sum x_{n-1}^j \tag{3.8}$$

$$\frac{\partial^2 E_n^p}{\partial {w_n^{ij}}^2} = \frac{\partial^2 E_n^p}{\partial {y_n^i}^2} \sum (x_{n-1}^j)^2 \tag{3.9}$$

Now, since the error values need to be back propagated through the network, it is necessary to send the error values through to each neuron sharing a connection with neuron i in the preceding layer. Note the weight sharing summations.

$$\frac{\partial E_n^p}{\partial x_{n-1}^j} = \frac{\partial E_n^p}{\partial y_n^i} \sum w_n^{ij} \tag{3.10}$$

$$\frac{\partial^2 E_n^p}{\partial {x_{n-1}^j}^2} = \frac{\partial^2 E_n^p}{\partial {y_n^i}^2} \sum (w_n^{ij})^2 \tag{3.11}$$

The learning rate is then calculated using the second derivative of the error function with respect to the weight being updated. ϵ is the global learning rate that is preset in the network and μ is a constant that is used to keep the learning rate from blowing up in the

case that $\frac{\partial^2 E_n^p}{\partial {w_n^{ij}}^2}$ is very small.

$$\epsilon_n^{ij} = \frac{\eta}{\mu + \frac{\partial^2 E_n^p}{\partial {w_n^{ij}}^2}} \tag{3.12}$$

Once this learning rate is computed, the weight update can finally be updated.

$$\delta w_n^{ij} = -\epsilon_n^{ij} \frac{\partial E_n^p}{\partial w_n^{ij}} \tag{3.13}$$

This process is repeated for every layer in the network that has input connections, starting at the last layer, and working the way back to the first layer.

3.3.2 Adaptive Momentum

One other improvement can be further added to the gradient descent algorithm presented in the previous section. This is the adaptive momentum method described in [10]. This technique speeds the convergence of the weights and allows for a greater accuracy to be achieved by the network in a shorter amount of training epochs.

Here, $\delta^{k-1} w_n^{ij}$ is the most recent update that was applied to the weight w_n^{ij}. α_n^{ij} is the adaptive momentum rate that determines the strength that the previous update to the weight will play in the current update.

$$\delta^k w_n^{ij} = -\epsilon_n^{ij} \frac{\partial E_n^p}{\partial w_n^{ij}} + \alpha_n^{ij} \delta^{k-1} w_n^{ij} \tag{3.14}$$

The adaptive momentum rate is set so that when the inner product of $\frac{\partial E_n^p}{\partial w_n^{ij}}$ and $\delta^{k-1} w_n^{ij}$ is less than zero and the weight update will be in the same direction as the previous update, the momentum rate is set using the equation below. If the inner product is not less than 0 and the weight updates are not in the same direction, then no momentum is applied.

$$\alpha_n^{ij} = \begin{cases} \alpha \frac{-\epsilon_n^{ij} \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij}}{||\delta^{k-1} w_n^{ij}||^2} & \text{if } \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij} < 0 \\ 0 & \text{else} \end{cases} \tag{3.15}$$

A variation to this algorithm has been made so that when the inner product of $\frac{\partial E_n^p}{\partial w_n^{ij}}$ and $\delta^{k-1} w_n^{ij}$ is greater than 0 and the weight updates are in different directions, a negative momentum rate is applied to the update. This allows for updates to speed up when they are similar to previous ones and slow down when they are not. In order to ensure that the slow down is not too large, the negative momentum is divided by 2. If the inner product is

0, then no momentum is applied.

$$\alpha_n^{ij} = \begin{cases} \alpha \frac{-\epsilon_n^{ij} \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij}}{||\delta^{k-1} w_n^{ij}||^2} & \text{if } \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij} < 0 \\ \alpha \frac{\epsilon_n^{ij} \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij}}{2||\delta^{k-1} w_n^{ij}||^2} & \text{if } \frac{\partial E_n^p}{\partial w_n^{ij}} \delta^{k-1} w_n^{ij} > 0 \\ 0 & \text{else} \end{cases} \tag{3.16}$$

3.3.3 Stochastic vs. Batch

There are two methods of updating the weights of a network in a gradient descent learning algorithm, batch and stochastic. Batch gradient descent is when the gradients are collected over the whole training set and then updated after each epoch. This method allows for faster computation and makes it simpler to parallelize the learning algorithm. The disadvantage to batch training is that the gradients are averaged over the whole training set before being applied to the network. This method eliminates the advantage of learning similar patterns in the same training set. For example, imagine that the network was trained on two similar images. Batch training would average the gradient from both training patterns and then apply the average. This would be similar to training on just one of the images. The advantage of both images being similar is lost[9].

Stochastic training solves this problem by updating the network after each training pattern. This way the network is improved with each and every pattern that is shown to it. The disadvantage here is that it requires much more computation to execute. However, overall, stochastic gradient descent is usually much faster than batch learning[8].

3.3.4 Neuronal Ranking

One method that was added to the algorithm was that of neuronal ranking. Values are assigned to each neuron to determine their specific impact on the network. The values for the input neurons were generated by averaging the normalized input values at each input neuron for the entire training set. The average was then added to 1 in order to generate a ranking value that circled around 1. Since the input is normalized, this allows the ranking values to be normalized as well. Once the values for the input layer were determined, the values for the neurons in each succeeding layer were calculated by taking the average of all of the ranking values of the neurons that it receives input from. This is done for each neuron in the network. Biases receive a ranking value of 1 so that the ranking value will have no effect.

During training, the ranking value of an output neuron is multiplied into the $\frac{\partial E_n^p}{\partial w_n^{ij}}$ calculation so that the weight updates will be increased or decreased depending upon the

overall importance of that neuron in the network.

$$\frac{\partial E_n^p}{\partial w_n^{ij}} = \frac{\partial E_n^p}{\partial y_n^i} \sum (imp_{n-1}^j)(x_{n-1}^j) \tag{3.17}$$

3.3.5 Mutated Values

In addition to the neuronal ranking system, a method of mutating the outputs of neurons and the strengths of the weights was added to the learning phase of the networks. This was done in an attempt to simulate real world noise and signal failure. The idea is that by allowing the signals to be slightly inaccurate, the network would gain greater generality in recognizing images. The signals were only mutated during the initial learning phase in order to ensure consistent performance once fully trained. In order to simulate the signal failure, the equation for the input of a neuron, equation 2.11, was altered using the following equations:

$$x_n^i = F(b_i + \sum (w_n^{ij} + (r_w|w_n^{ij}|m - \frac{|w_n^{ij}|m}{2}))(x_{n-1}^j + (r_x|x_{n-1}^j|m - \frac{|x_{n-1}^j|m}{2}))) \tag{3.18}$$

Here, r_w and r_x are uniformly generated random values from 0 to 1 created independently for the weight value and the neuron output value respectively. m is a mutation factor that allows for the scaling of the strength of the mutations. In this paper, $m = 0.05$. The mutations were kept small so that the neuron signals were not distorted to a point of negatively affecting the rate of learning.

3.4 Distortions

One of the issues encountered when training a network with a gradient descent algorithm is that it requires labeled training data in order for the network to learn. The more varied the data, the more general the network can be. However, there is a limited supply of labeled training data available. A method has been proposed for expanding the training set using elastic distortions to distort the training images in [11]. This allows the network to learn from a more varied set of training data without the need of more labeled images.

The method used in this project employs the use of elastic distortions as well as translations, rotations, and scaling. All of the distortions are created randomly before training on each image. This makes it very unlikely that the network would ever see the exact same image more than once.

In order to perform randomized elastic distortions, two different random displacement fields are created for each image, one for horizontal distortions and one for vertical dis-

tortions. The values in these displacement fields are uniform random numbers between -1 and 1. The values are then convolved with a Gaussian of standard deviation σ pixels. The values are then normalized to a value of 1 and multiplied by a constant that is used to control the strength of the distortion. These displacement fields now hold the new horizontal and vertical positions of the pixels in the input image. These values are then applied using interpolation in order to create the elastically distorted input image. The values used for the elastic distortion are based on the values used in [4].

The elastically distorted images are then scaled by a random percentage between -15% and 15%, rotated by a random angle between -15° and 15°, and then translated by a random step size between -1.5 and 1.5. These transformations are then applied using interpolation to create a brand new training image.

3.5 Fine-Tuning

One of the issues with constantly distorting the training images is that the network does not get a chance to settle it's weights to some constant values. The ever changing input images keep the weights circling around their ideal locations without being able to hit them. One way to alleviate this issue is to train the network on the non-distorted training set for a few epochs after the training has completed. This allows for the network to be trained on the distorted training set, while also giving it time at the end to settle into its desired state.

3.6 A Transitional Training Program

This project used the basic idea behind fine-tuning to create a training program designed to reduce the learning rate at a faster pace as the network trained. The learning rate is first initialized to 0.001. The learning rate is then degraded by 10% for the first two epochs and then dropped by 10% every two epochs following that. After 20 epochs, the learning rate is set to 0.0001 and then degraded by 10% every two epochs. This is done until the network has been trained for 40 epochs. After this, the network is then trained for ten epochs on the undistorted training set. This method of gradual decay of the learning rate allows the training of the network to smoothly transition from a strong initial learning phase into a soft fine-tuning. This transition helps situate the weights of the network into the desired values for accurate predictions.

3.7 Unsupervised Experience Learning

Since a network can be trained to such a high level of efficiency, it can be assumed that the network will recognize unseen patterns accurately. This allows an unsupervised gradient descent algorithm to be run while testing the network. This test/learning epoch will first attempt to categorize a testing pattern and then learn the pattern based on how the network categorized it. For example, if the testing pattern was recognized by the network to be a 2. Then the output neuron for 2 would be trained to a value of 1 while the other output neurons would be trained to a value of -1. In other words, the network would be trained on an image assuming that the networks original classification of the image were true. This allows for the network to never stop learning patterns that it encounters. This is similar to how a person learns how to read different types of handwriting. When a slightly unfamiliar letter is read, the person will first figure out what they think the letter is, which in most cases will be correct, then they store that image in their memory so that when another similar letter is read it can be recognized quickly. This method helps to incorporate the idea of learning from life experience into the network rather than just learning through a specified training phase.

There are two drawbacks to this method. First, the network must first be trained to a high enough level of accuracy so that a very large percentage of patterns encountered will be recognized correctly. Second, if the network does come across a lot of ambiguous patterns in a row and mis-recognizes them, it can negatively affect the performance of the network. However, since the learning rate is kept low for unsupervised experience learning, the likelihood that the network would come across such a situation is unlikely. Just in case, the learning rate is altered depending on how confident the network is in it's classification of the pattern that it is learning. The more confident the network, the higher the learning rate, the less confident, the lower the learning rate.

This process can be further controlled when done using network ensembles. In this case a pattern would be classified by the same method that the ensemble would usually classify it. Then the unsupervised experience learning will commence for each network in the ensemble based on how the ensemble as a whole classified the image.

CHAPTER 4

RESULTS

This section will display the results obtained from training an ensemble of 8 convolutional networks on the MNIST training set. First, the benefit of using distorted training images will be shown, Then, the advantage that can be gained by adding momentum to the training algorithm will be discussed. And finally, the results of training the network ensemble will be displayed.

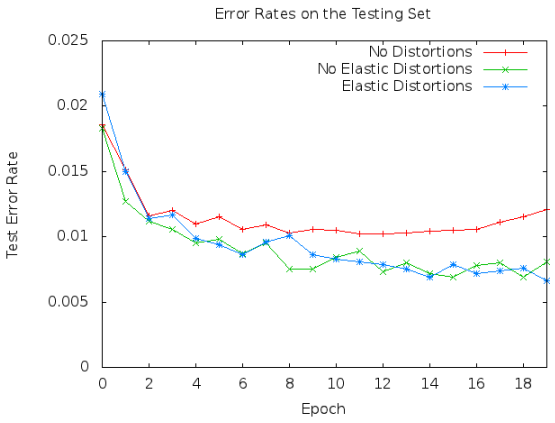

Figure 4.1: Error Rate of Testing Set with Different Levels of Distortions.

It has been stated in a previous section that a large improvement can be made by distorting the training images so that the network learns on an ever changing array of data. Figure 4.1 shows that this is very true. After 20 epochs, the network trained with no distortions ended with a error rate of 1.21% on the testing set with a best error rate of 1.02% achieved after 12 epochs. The network trained using rotation, translation, and scaling distortions finished with an error rate of 0.81% with a best error rate of 0.69% achieved after 16 epochs. Once elastic distortions were added to the distorted training set,

the network finished with a best error rate of 0.66%. This clearly shows the benefit of using elastic distortions to train the network.

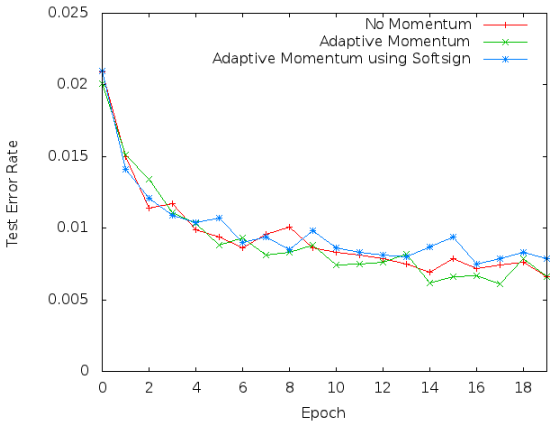

Figure 4.2: Error Rate of Testing Set using Adaptive Momentum.

Adding adaptive momentum to the gradient descent algorithm used to train the network has a positive effect as well. This is shown in Figure 4.2. The results here show a network trained without momentum, a network trained with adaptive momentum, and a network trained with adaptive momentum using the softsign sigmoid function. The graph clearly sows the benefit of using the adaptive momentum technique.

The graph also shows, as hypothesized, that the softsign sigmoid function does not perform as well as the hyperbolic tangent function in high accuracy networks. This is likely due to the previously mentioned issue of the absolute value of the second derivative not having maximum values at 1 and -1. This characteristic would contribute to a slowed convergence of the outputs to 1 and -1.

After 20 epochs, the network trained with no momentum ended with a best error rate of 0.66%. The network trained with adaptive momentum using the softsign sigmoid function ended with an error rate of 0.79% with a best error rate of 0.75% achieved after 17 epochs. The network trained with adaptive momentum using the hyperbolic tangent sigmoid function ended with an error rate of 0.66% with a best error rate of 0.61% achieved after 18 epochs. The graph clearly shows the benefit of using adaptive momentum to train the network.

Now that the best methods for training a singular network have been deduced, they can be applied to an ensemble of many neural networks. Figure 4.3 shows a graph displaying the results obtained by using an ensemble of 8 artificial neural networks. In this case, each network is identical to the next except that they have different initial weight values. The

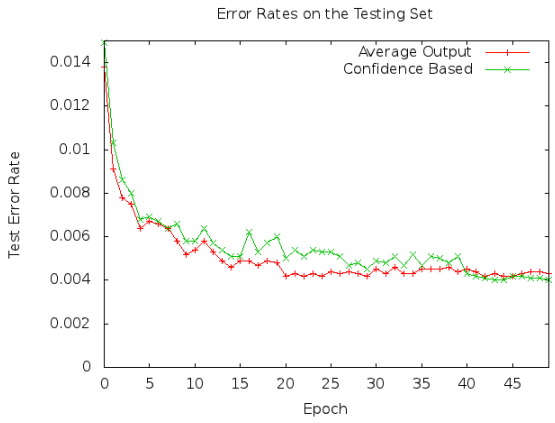

Figure 4.3: Error Rate of Testing Set using an Ensemble of 8 Networks.

training images are distorted before being passed into each network so that each network is trained on a slightly different set of images. This allows for more variability in the training of the networks and leads to greater generality when classifying the images on the testing set. For each epoch, the networks were trained in parallel on the distorted training sets and then run sequentially in order to classify the testing set. The training method is described in detail in the section on fine-tuning.

The two methods used to identify the testing set were average output and confidence based classification. These are the methods that were discussed in a previous section. From the graph, it is clear that the average output method outperforms the confidence based method on most of the epochs. However, The confidence based technique responded extremely well to the fine-tuning phase that used the undistorted training set. Prior to this phase the confidence based method yielded results that were more erratic and less accurate than the average output method, but once the final phase of fine-tuning commenced, the results of the confidence based method leveled out and actually outperformed the average output technique. This shows that the confidence of the networks in the ensemble increased once they were trained on the undistorted training set.

After 50 epochs, the average output technique ended with an error rate of 0.43% and a best error rate of 0.42% achieved after 21 epochs. The confidence based technique finished with a best error rate of 0.4% that it first reached after 44 epochs. This is a very strong result, especially considering the fact that the networks are all virtually identical and trained on similar data as discussed previously. None of the networks were specialized for any type of data, and the images passed into the networks were trained using the same techniques.

The integration of the mutated signals into the networks, as described in section 3.3.5,

improved the overall accuracy of the ensemble slightly. The mutated values were only used during the initial training phase with a mutation factor of 0.05. The values were not mutated for the ensemble on the testing set. This improved the final error rate to a value of 0.38%. This improvement is likely due to the increased generality that was gained due to the networks not being trained with complete efficiency on the training set.

Paper	**Error Rate on the MNIST Testing Set**
Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition[4]	0.35%
This paper	*0.38%*
Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis[11]	0.40%
What is the Best Multi-Stage Architecture for Object Recognition?[5]	0.53
Deformation Models for Image Recognition[6]	0.54%
A Trainable Feature Extractor for Handwritten Digit Recognition[7]	0.54%
Gradient-based Learning Applied to Document Recognition[9]	0.70%

The table above shows the results of various different methods of training artificial neural networks on the MNIST training set. It can be seen from the table that the method used in this paper has achieved an error rate on the MNIST testing set that is in line with some of the top performing networks out there. The network used in [4] did perform better, however that network was extremely dense with close to 10 million trainable parameters and took hundreds/thousands of epochs to train. The networks used in this paper each have 61,492 trainable parameters and took only 50 epochs to train. Granted there are 8 networks, but that is still only 491,936 trainable parameters when combined.

Once the training of the network was complete, the experience learning technique mentioned in a previous section was applied on the testing set. The network was then tested on the testing set in the normal way. The experience learning epoch yielded an error rate of 0.38%, and the subsequent confidence based test yielded an error rate of 0.38%. While this is not an improvement on the previous listed error rate, an improvement can be seen when looking at the average mean squared error of the testing set. The average mean squared error on the testing set before the experience learning epoch was 0.0143703, and it was

0.0143199 after the experience learning. This shows that there is an advantage to having a network learn from experience. Although the improvement is small, it should also be greater when used to classify larger amounts of data as the testing set only contains 10,000 images.

CHAPTER 5

CONCLUSION

This paper has presented a method for using an ensemble of artificial neural networks to recognize handwritten digits. The networks were created based on the design presented in [9], with some modifications. The networks were then trained with elastically distorted training images using the values presented in [4] and the technique presented in [11]. The technique for learning the images is a combination of the stochastic levenberg-marquadt gradient descent algorithm presented in [9], and the adaptive momentum technique presented in [10], again with some modifications.

The combination of these various techniques into a single convolutional network results in a very accurate network. However, this accuracy has been improved by combing 8 different versions of the network and using the network that is the most confident in it's classification of a test image to decide on how to recognize the image. This network ensemble performs very well when recognizing the testing images on the MNIST database. It was able to achieve an error rate of 0.0040 for the image set. This is a vast improvement over even the best error rate earned by the single network system, 0.0066.

It is clear that a large improvement in the accuracy of an artificial neural network system can be achieved by using ensembles of multiple neural networks.

BIBLIOGRAPHY

[1] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. Also published as a book. Now Publishers, 2009.

[2] Yoshua Bengio and Xavier Glorot. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of AISTATS 2010*, volume 9, pages 249–256, May 2010.

[3] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In Léon Bottou, Olivier Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*. MIT Press, 2007.

[4] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. Technical Report arXiv:1003.0358, Mar 2010. Comments: 14 pages, 2 figures, 4 listings.

[5] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE, 2009.

[6] Daniel Keysers, Thomas Deselaers, Christian Gollan, and Hermann Ney. Deformation models for image recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29:1422–1435, August 2007.

[7] Fabien Lauer, Ching Y. Suen, and Gérard Bloch. A trainable feature extractor for handwritten digit recognition. *Pattern Recogn.*, 40:1816–1824, June 2007.

[8] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[9] Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[10] Hongmei Shao and Gaofeng Zheng. Convergence analysis of a back-propagation algorithm with adaptive momentum. *Neurocomputing*, 74:749 – 752, Feb 2011.

[11] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practice for convolutional neural networks applied to visual document analysis. In *In International Conference on Document Analysis and Recogntion (ICDAR), IEEE Computer Society, Los Alamitos*, pages 958–962, 2003.