FLEXIBLE IMAGE RECOGNITION SOFTWARE TOOLBOX (FIRST)

By

PRANITA PATIL

Bachelor of Science in Electrical Engineering
Pune University
Maharashtra, India
2010

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2013

FLEXIBLE IMAGE RECOGNITION SOFTWARE

TOOLBOX (FIRST)

Thesis Approved:

ACKNOWLEDGMENTS

I would begin this acknowledgement by thanking my advisor, Dr. Martin Hagan for the opportunity to research under his guidance. I immensely appreciate his endless support, caring, patience and efforts in providing me excellent ambience in each step of my research. This work would not have been possible without his advise.

I would also like to express my gratitude to my committee members Dr. Damon Chandler and Dr. George Scheets for being a part of my committee as well as their help and support.

Most importantly, I would like to thank my mother Surekha Patil, my father Pramod Patil, sister Siddhi Patil and all my relatives for their moral support, unconditional love, patience and encouragement even though they have been miles away from me. Also, I appreciate the help from all of my friends from OSU and back in India for motivating me in my research. This is a truly a great opportunity for me to publicly thank those who have been influential during my thesis and without them I would never have been able to complete my research. Specially I am grateful to God for his blessings.

Name: PRANITA PATIL

Date of Degree: July, 2013

Title of Study: FLEXIBLE IMAGE RECOGNITION SOFTWARE TOOLBOX (FIRST)

Major Field: Electrical Engineering

The deep convolutional neural network is a new concept in the neural network field, and research is still going on to improve network performance. These networks are used for recognizing patterns in data, as they provide shift invariance, automatic extraction of local features, by using local receptive fields, and improved generalization, by using weight sharing techniques. The main purpose of this thesis is to create a Flexible Image Recognition Software Toolbox (FIRST), which is a software package that allows users to build custom deep networks, while also having ready-made versions of popular deep networks, such as Lecun's LeNet-1, LeNet-5 and LeNet-7. This software package is created for designing, training and simulating deep networks. The goal is to reduce the amount of time required by users to implement any particular network. To design this software package, a general modular framework is introduced, in which simulation and gradient calculations are derived. Due to this modularity and generality, FIRST provides flexibility to users in easily designing specific complex or deep networks. FIRST includes several training algorithms, such as Resilient Backpropagation, Scaled Conjugate Gradient and Steepest Descent. This thesis also describes the usage of the FIRST software and the design of functions used in the software. It also provides information about how to create custom networks. The thesis includes two sample training sessions that demonstrate how to use the FIRST software. One example is phoneme recognition in 1D speech data. The second example is handwritten digit recognition in 2D images.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Pattern Recognition and Neural Networks

Pattern recognition is related to the automatic discovery of regularities in data by using computer algorithms. These regularities help to take actions, such as classifying the data into different categories. Pattern recognition is one of the traditional uses of neural networks. From the pattern recognition point of view, neural networks are extensions of conventional methods that have been used over decades. These neural networks are adaptive systems that change their parameters based on the difference between the output of the network and the desired output. The desired outputs are target values that are predefined during the training phase. The process of finding the optimal network parameters is called training, and it is generally done with gradient-based methods, which will be described in Chapter 4. A neural network is a parameterized non-linear mapping between sets of input and output variables. During network training we want to adjust the network parameters until the correct mapping is made. For pattern recognition, the network input is an image, and the target output is a code that indicates the corresponding category that the image belongs to. The following section describes the basic network operation for two types of neural network.

1.2 Multilayer Perceptron

To begin the discussion of neural networks, we start with a single neuron, as shown in Figure 1.1. For a single input neuron, a scalar input p is multiplied by scalar weight w, and then it is added to a bias *b* to form the net input *n*, which is then applied to a transfer function *f*() to produce the neuron output *a*.

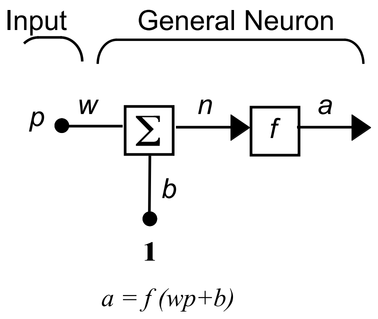


Figure 1.1: Single Input Neuron

The multiple input neuron perceptron has R inputs and a single output. For each input there will be one weight, $w_{1,1}, ...w_{1,R}$, as shown in Figure 1.2.

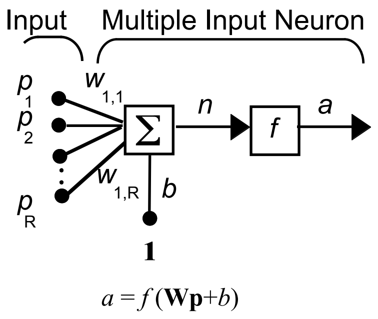


Figure 1.2: Multiple Input Neuron

Multiple neurons can be stacked together to create a layer of neurons, as shown in Figure 1.3. This layer consists of weight matrix **W**, bias vector **b**, summing junction, transfer function and output vector **a**. Each input is connected to each neuron. (This is considered a “fully connected” layer.) This network is called a single layer perceptron. It is capable of discriminating between two linearly separable classes.

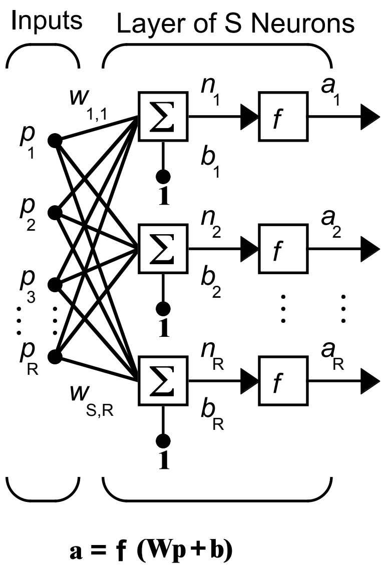


Figure 1.3: Layer of S Neurons

Multilayer perceptrons are composed of multiple perceptrons connected in series, as shown in Figure 1.4. They are capable of creating arbitrary decision boundaries. Here superscripts are used to identify the layer number. This type of network is also called a feed forward network, because calculations flow forward from input to output layer. Also, as each neuron in the previous layer is connected to all neurons in the present layer, it is called a fully connected network.

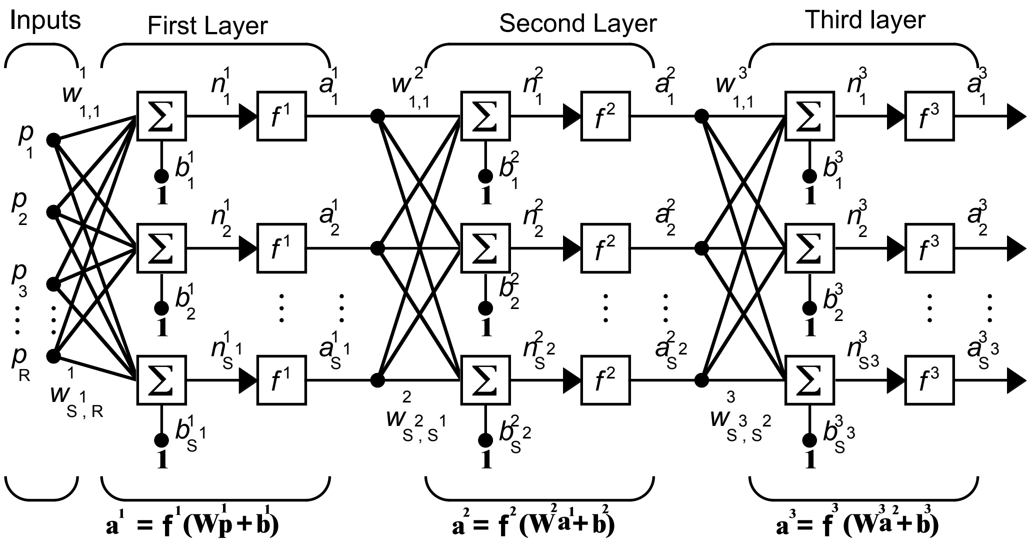

Inputs
First Layer
Second Layer
Third layer
$w^1_{1,1}$
$w^1_{S^1,R}$
$w^2_{1,1}$
$w^2_{S^2,S^1}$
$w^3_{1,1}$
$w^3_{S^3,S^2}$
$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)$
$\mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2)$
$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$

Figure 1.4: Three-Layer Network

1.3 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are deep (more than three layers) neural network architectures that have been proposed for classification or recognition tasks. The CNN was inspired by a description of the cat's visual system, and it is an efficient architecture, because it uses shared weights and fewer connections than fully connected multilayer networks. These constraints reduce the computational burden, so it is easier to train than fully connected deep networks. These shared weights play the role of a filtering (i.e., convolution) kernel. Lecun [6] developed the CNN, which is reminiscent of the Neocognitron. The CNN is capable of extracting local features by using a local receptive field. A certain degree of shift and distortion invariance is obtained by reducing the spatial resolution of the feature map. Also, the number of free parameters is significantly decreased by using the same set of weights for all features in the feature map (weight sharing technique). The Neocognitron and the CNN will be discussed in more detail in Chapter 2.

1.4 Previous Work

The multilayer perceptron has several drawbacks when it comes to real-world applications such as pattern recognition. First, the number of trainable parameters becomes extremely large for large inputs, like images. Secondly, it offers little or no automatic invariance to shifting, scaling, and other forms of distortion. Third, the topology of the input data is completely ignored, yielding similar training results for all permutations of the input vector. To overcome these problems, a classifier is often split up into a hand designed feature extractor and the actual trainable classifier module. Designing the feature extractor by hand requires a lot of heuristics and a great deal of time. To tackle these problems, the CNN was introduced. In order to verify the performance of the CNN, it is compared with different classifiers based on accuracy, run time, memory requirement and training time. By comparing the CNN with linear classifiers, tangent distance classifiers, large fully connected multilayer neural networks, principal component analysis and polynomial classifiers, optimum margin classifiers, it was found that the CNN is superior [7]. In [7], the error rate score was compared, and they found that the CNN (boosted LeNet-4) was best. Training time was also measured. The CNN often takes more time than TDC or other classifiers, but this is dependent on the designer's requirements. When comparing memory requirements, it is found that the CNN is best (LeNet-4). Research is still going on to improve the performance of CNNs, and extensions continue to be invented. It has been found that boosted LeNet-4 is giving high accuracies with less memory requirements and a smaller computational burden. Thus, convolutional networks have been found to be well suited for recognizing or rejecting shapes with widely varying size, position, and orientation. Because of all these properties, CNNs have been used in speed sign recognition, handwritten digit recognition, text recognition, face recognition, and in an operational bank check reading system developed at AT & T in the early 1990 [4]. It was first used commercially in 1993 for check-reading ATM machines in Europe and the US, and was deployed in large bank check reading machines in 1996. It was reading over 10% of all the checks in the US by the late 90s. Microsoft used CNN techniques in handwriting recognition systems, including for Arabic [8] and Chinese characters [9]. Google recently used this to detect faces and license plates in Street View images, so as to protect privacy [10]. France Telecom used this technique for face detection systems for video-conferencing [11]. Due to all its abilities, CNNs are widely used in many applications, and research is still going on. There is a need to have a software package so that researchers will be able to experiment with different deep architectures without having to derive and program their own gradient calculations and training functions. With the modular structure of the FIRST design, users can invent new layer types and network architectures with minimal programming.

1.5 Objectives

There have been many deep networks proposed for pattern recognition, such as the Neocognitron and several variations of the CNN. We do not yet know what the best architecture is. Whenever a new architecture is considered, new training algorithms must be developed and new computer code must be written. This delays the progress of new developments. In this thesis, we will construct a Flexible Image Recognition Software Toolbox (FIRST). With FIRST, a user will be able to quickly implement new network architectures for pattern recognition in images. In order to create a flexible software package, it will first be necessary to generate the deep network concept and to develop a modular framework that can capture previously proposed architectures while enabling the creation of totally new networks.

1.6 Outline of Thesis

In the following chapter, we will discuss the Convolutional Neural Network (CNN), which is the foundation of this work. We will describe some of the most important extensions of the CNN proposed till now and their applications. In Chapter 3, the main focus is on a general modular network framework that can include all of the networks that we discuss in Chapter 2. This general network framework can be used to create any type of deep CNN, as well as new networks. Also, we will discuss the type of training data needed for such a general network framework. In Chapter 4, we will discuss training algorithms for the general network framework. In addition to steepest descent, we will also discuss more efficient optimization strategies based on the gradient. We will derive gradient equations that will work for all networks. Issues associated with training algorithms are also described in this chapter. Chapter 5 describes how FIRST can be used to implement deep convolution neural networks for practical pattern recognition problems. We will also present experimental results for recognizing handwritten digits and speech (phonemes). We will focus on the processing of two-dimensional gray-level images and one-dimensional frequency domain speech data. Finally, Chapter 6 will conclude this thesis work and provide some perspectives for future work.

CHAPTER 2

CONVOLUTIONAL NEURAL NETWORK

2.1 Introduction

The multi-layer feed-forward neural network is a popular network for pattern classification. It is a very powerful machine learning technique, because it can be trained to approximate complex non-linear functions. The most common type of this network is the Multi-Layer Perceptron (MLP). However, the MLP often performs best when a separate feature extraction algorithm is used to reduce the dimension of the input data. However, selecting the right feature extraction method is often difficult [12].

The problem faced by the MLP is that many times the input dimensions for pattern recognition systems are high, as in images. This requires a large number of connections in the network, thus a large number of trainable parameters, because in the MLP the inputs are fully connected to each hidden unit. The number of connections can be several 10000 or several 100000, depending on the application. This leads to highly complex neural network architectures that require very large training data sets, since overfitting can occur when the number of training points is less than the number of network parameters. Also, such a large network will require a large amount of memory, which is not practical. Another drawback of fully connected architectures is that they do not conveniently take into account correlations between neighboring inputs (i.e., neighboring pixels in images). For pattern recognition, the topology of the inputs is important, because in images or speech there is a high amount of correlation. Finally, slight variations or distortions in input images can cause changes in the position of distinctive features. Standard MLPs have no built-in invariance w.r.t. local distortions and small translations. [12]

In order to solve pattern recognition problems, we need neural networks that allow direct application of raw input data, and whose training algorithms will find the best feature extractors by adjusting trainable parameters. They should implement weight sharing to reduce the number of free network parameters and thus to increase generalization. Also, it would be preferable to extract local features by restricting the receptive field to be local. Lastly we would like to have neural networks that can automatically achieve shift invariance. All of these features can be found in an approach called the Convolutional Neural Network(CNN).

2.2 Background History

This section presents the history behind the development of the CNN.

2.2.1 Neocognitron Model

The original idea for the CNN is based on the neocognitron. The neocognitron is a biologically inspired structure. It was developed by Fukushima in 1980 and is based upon the research of Hubel and Weisel, who discovered locally-sensitive, orientation-selective neurons in the visual system of cats [13]. From their work, Fukushima invented the concept of hierarchy in alternating layers of S cells and C cells, which make up the neocognitron.

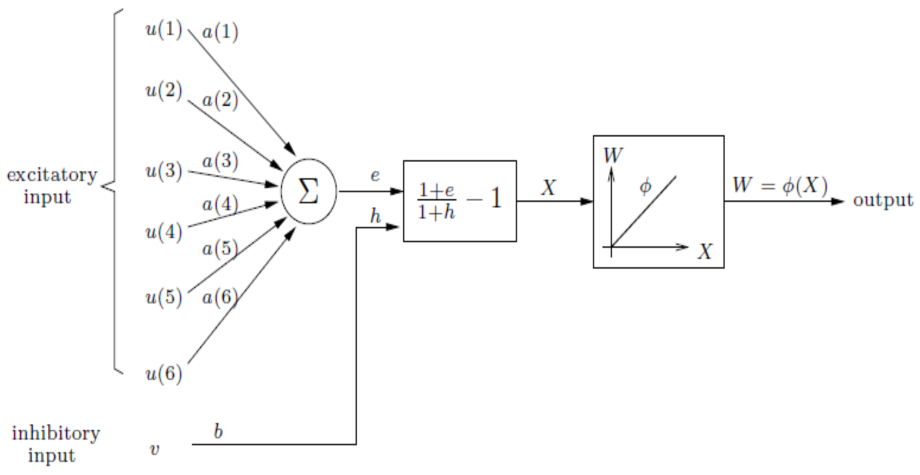


Figure 2.1: The model of an S cell, taken from [1]

S cells are feature extracting cells that resemble the simple cells found in the primary visual cortex, and are therefore called simple cells. The basic model of the S cell is shown in Fig. 2.1. This model shows how these cells would respond to features in its receptive field. Since features are local, each unit can only extract the attributes of one local neighborhood of the input. Therefore, the parameters of features are shared among a grid of units, so that they can extract the same feature from different locations. This strategy of sharing parameters among subsets of units is called weight sharing.[1]

Fukushima describes weight sharing as follows:

“ It is assumed that all the cells in a single cell-plane have input synapses of the same spatial distribution, and only the positions of the presynaptic cells are shifted in parallel from cell to cell. Hence, all the cells in a single cell-plane have receptive fields of the same function, but at different positions ”.[2]

Essentially, weight sharing is a way of implementing a filtering operation within locally connected neural networks. The outputs of S cells approximate convolution, normalized by length of weight vector and input vector. S cells extract local visual features, such as edges or corners. Several S planes contain different sets of weights to extract different features at the same locations, so these groups of cells and these S planes are called an S layer.[14]

C cells represent complex cells found in the visual cortex. The input to the C cells comes from the output of the preceding S cells. C cells respond if any of the S cells are excited. C cells reduce the resolution of S cells. They reduce the sensitivity to small shifts and distortions in input patterns. This is equivalent to a blurring operation.[2]

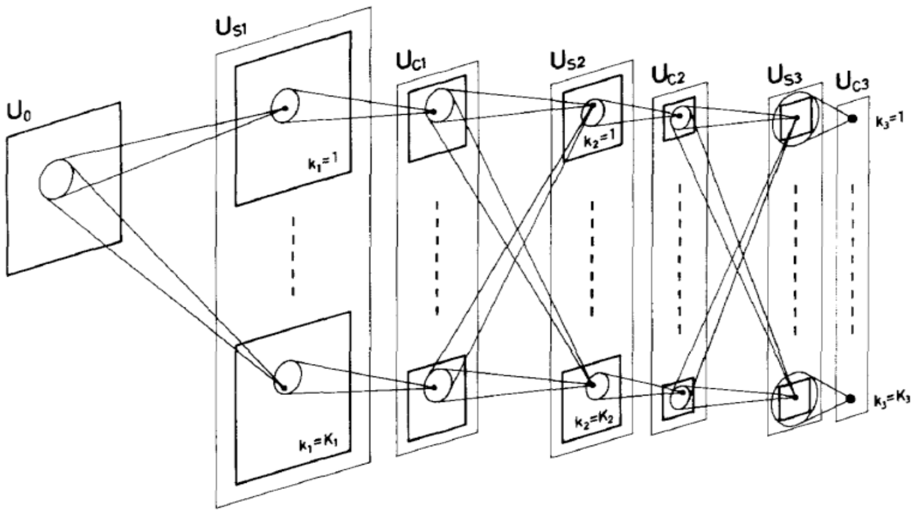


Figure 2.2: Neocognitron, taken from [2]

Fig.2.2 shows the structure of the neocognitron. This is a three stage local feature detector hierarchy, where each stage consists of two layers: filtering or non-linearity and pooling. In Fig. 2.2 the filtering layers are visualized as the links between U0 and US1, UC1 and US2, and UC2 and US3, and the pooling layers are the subsequent connections of US1 and UC1, US2 and UC2, and US3 and UC3. Motivated by biology, units that perform local feature computation were called simple cells (denoted by US) and pooling units were referred to as complex cells (denoted by UC). Here in this figure, inhibitory inputs v are not depicted. The inhibitory inputs of the S planes are connected to V planes. The v neurons of each V plane receive their inputs from corresponding sub-regions in each preceding C plane. By alternating S and C layers, complex feature extractors can be constructed. In early layers, simple local features are extracted (e.g., line segments), and the following layers combine local features to extract more complex features (e.g., combinations of line segments).[2]

The neocognitron was introduced in a decade when computational models were at a preliminary stage of development. A rather ad-hoc unsupervised learning algorithm was used to adapt the weights of simple cells, without any exact criterion for learning (e.g., minimizing energy or maximizing likelihood). This unsupervised learning is based on self- organization [2]. Later studies showed that the performance of neocognitron with supervised training is better than the original unsupervised training. Also, the model is a bit complicated, as it was designed to match existing biological understanding.

2.2.2 Lecun's CNN model

Lecun introduced the first CNN in [6]. It was trained by backpropagation and was applied to handwritten digit recognition (the numerals 0 through 9). This first CNN model is also called LeNet-1.

The CNN is simpler than the neocognitron and its extensions. Nevertheless, the neocognitron and the CNN architecture are similar in many aspects. In the neocognitron, S layers and C layers alternate. In the CNN, instead of S and C layers, convolution and sub-sampling layers are alternated. The S and C layers consist of planes, which contain cells. These planes correspond to feature maps of the convolution and sub-sampling layers. In the CNN model, the inhibitory input and the V planes are not present, which simplifies the model. The CNN also uses weight sharing and implements the concept of receptive fields. In the CNN, local features are not extracted manually; they are updated automatically by learning the convolution kernels.

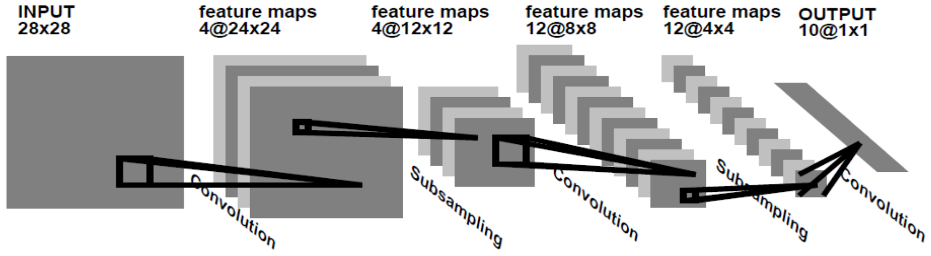


Figure 2.3: Architecture of LeNet-1. Figure taken from [3]

The CNN LeNet-1 [6] architecture consists of 5 layers, which are denoted by C1, S2, C3, S4, and F5. The architecture is shown in Fig. 2.3. The output of the first layer serves as input to the next layer. C1 and C3 are convolution layers, and S2 and S4 are sub-sampling layers. The last layer is a fully connected layer, as in the MLP. The input is a gray scale 28×28 image. The pixel intensities are normalized. The C1 layer consists of 4 Feature Maps (FMs), each having 25 trainable weights. This means that each unit in an FM takes its receptive field from a 5×5 neighborhood area on the input plane. The convolution FMs are obtained by convolving the input images with a convolution kernel (the layer weights), adding a bias and applying an activation function. The output of C1 is 24×24. The dimension is reduced to avoid the boundary effect of the convolution calculation, as blank pixels are added around the border of the original image to increase its dimensions from 16×16 to 28×28.

S2 is a sub-sampling layer that is composed of 4 FMs of size 12×12. The S2 layer performs down-sampling and reduces the dimension of each convolution FM by a factor of two. The units in a plane share the same weight and bias. This layer has a total of 4 weights and 4 biases. The output of this layer is obtained by averaging the previous layer output and then multiplying by a scalar weight, adding a bias and then applying an activation function.

Table 2.1: Connection table between FMs S2 and C3

	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X	X		X	X						
2		X	X	X	X	X						
3							X	X	X		X	X
4								X	X	X	X	X

Layers C3 and S4 contain 12 FMs of dimension 8×8 and 4×4, respectively. They perform the same functions as C1 and S2. The only difference is the connection between

S2 and C3. The input to C3 is a combination of the FMs in S2. The connection between S2 and C3 is shown in Table 2.1. In this table, the columns indicate the number of the FM in C3, and the rows indicate the number of the FM in S2. One convolution map in layer C3 can have several kernels operating on different FMs in layer S2. The output of the convolution is simply the addition of all convolution results, and then the bias is added and the activation function is applied. Therefore, C3 extracts complex features by combining all selected local features of layer S2.

The last layer is a full connection layer. In LeNet-1, the output of S4 is fully connected to next layer F5, which contains 10 neurons (representing the numerals 0 through 9). For this full connection, the output of S4 is transformed from a matrix to a column vector before being input to the last layer, F5. When the input pattern represents a certain digit, then the desired output is 1 for the corresponding neuron and -1 for other neurons.

In summary, the network has 4635 units, 98442 connections and 2578 independent trainable parameters. Lecun trained this network by using online backpropagation. The training of CNNs is similar to training of other types of neural networks.

2.3 Extensions of CNN

This section describes enhancements to the CNN model of Lecun. In the CNN, the number of layers, dimensions, connections, and parameters are based on the structure of the problem. There is no algorithm that can automatically determine optimal values for these parameters according to the chosen task, so we need to apply trial and error. That's why many enhancements and extensions of the CNN have been developed. However, the basic ideas used in the CNN remain the same: 1) local receptive field, 2) weight sharing and 3) sub-sampling [4] .

2.3.1 Boosted LeNet-4

For more complex problems, there is need for larger networks, so LeNet-4 was introduced. It is similar to LeNet-1, but it has more FMs, as well as an additional hidden layer that is fully connected to the last layer. The input to the network is 32×32. It has about 260000 connections and 17000 free trainable parameters, but the performance achieved by this network was not significantly improved, so the concept of boosting was developed by R. Schapire, Drucker et al.[15]. They proposed using a combination of multiple classifiers to improve performance. Drucker used this idea and constructed a combination of three classifiers.[3]

The architecture of boosted LeNet-4 is as follows. It contains three LeNet-4 networks. The first LeNet-4 is trained in the usual way. The second one is trained on patterns that are filtered by the first network, so that the second network has a mix of patterns in which half of the patterns were correctly classified by the first LeNet-4 and the second half were not. Finally, patterns on which the first and the second networks give different results, are given as input to the last (third) LeNet-4. During testing, the outputs of the three networks are combined.[3]

In order to train these three networks, a large training set is necessary. To produce such a large data is not practical, so it is produced artificially by distorting a smaller training set. The performance of the boosted LeNet-4 is better than LeNet-1 and LeNet-4.[3]

The computation time required for this combination of three versions of LeNet-4 is not three times that of a single network. It only requires 1.75 times as much computation, because when the first network produces an answer with high confidence then there is no need to call the other two networks.[3]

2.3.2 LeNet-5

In order to improve recognition rate and generalization capacity, a more complex network, LeNet-5, was developed by Lecun [4]. This network is typically used for isolated handwritten character recognition. LeNet-5 is composed of 7 layers, with an input of pixel size 32×32.

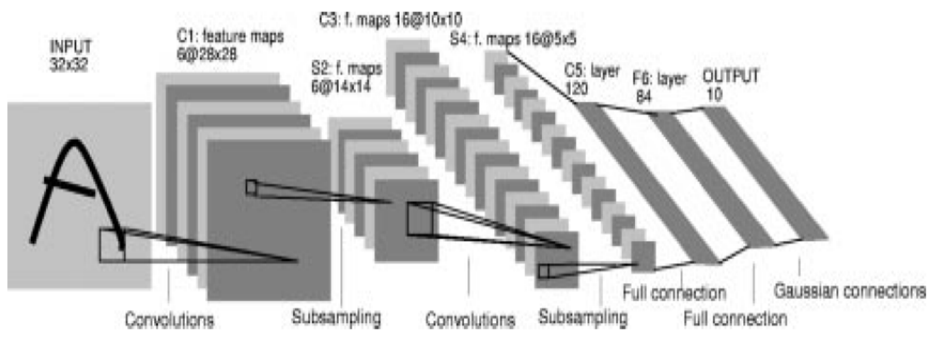


Figure 2.4: Architecture of LeNet-5, taken from [4]

The architecture of LeNet-5 is shown in Fig. 2.4 [4]. There are 3 convolution layers. The first convolution layer is composed of 6 FMs, so C1 contains 156 trainable parameters in six 5×5 kernels plus 6 bias values to create 122304 connections. The size of the FM in the C1 layer is 28×28, due to the boundary conditions, whereas the second convolution layer, C3, contains 1500 weights and 16 biases, which are shared between 151600 connections over 16 FMs. The connections between S2 and C3 are shown in Table 2.2. Lecun designed these connections to maximize the number of features extracted by C3, while reducing the number of weights. The last convolution layer, C5, contains 120 FMs, and the output dimension is 1×1, so there are 120 output elements.

There are two sub-sampling layers, S2 and S4. S2 contains 6 FMs, and S4 has 16 FMs. This is a non-overlapping layer. FMs of this layer are always half the size of the preceding FMs of the convolution layer. Layer S2 has 12 trainable parameters with 5880 connections, whereas S4 has 32 trainable parameters with 156000 connections.

Table 2.2: Connection table between FMs S2 and C3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
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
6				X	X	X			X	X	X	X		X	X	X

F6 is an additional hidden layer, which is fully connected to layer C5 and contains 86 units. This layer has 10164 trainable parameters. The output layer consists of Euclidean Radial Basis Function (RBF) units, one for each character, so the output is computed as follows:

$$y_i = \sum_j (x_j - w_{i,j})^2 \tag{2.1}$$

The RBF unit calculates the Euclidean distance between its input vector and the weight vectors. It calculates the fit between the input vector and the target that is associated with that class. These weight vectors are fixed, and their elements are set to -1 and +1 to form 7×12 matrices representing stylized images of the characters. This can be help to eliminate confusion between similar characters, like uppercase O, lowercase o and zero, because this network produces similar outputs for similar characters[4].

The error function used in this network is designed to get the F6 output as close as possible to the stylized images. The equation is as follows:

$$E(W) = 1/P \sum_{p=1}^{P} (y_{D_p}(Z^p, W) + log(e^{-j} + \sum_i e^{y_i(Z^p,W)})) \tag{2.2}$$

Here P is number of training patterns, y_{D_p} is output of the D^pth RBF unit and D^p is the desired class for input p. The first term in Equation 2.2 is the Euclidean distance between the desired output and the RBF centers. However, if the RBF parameters are learned, using only this term could lead to all RBF centers becoming equal and all RBF outputs equal to zero. So to avoid this, the second term is added. It keeps RBF centers apart. Also, the positive j inhibits the error function from increasing for classes that have large outputs [4].

2.3.3 Convolutional Deep Belief Networks

In many problems, labeled data are scarce. The Convolutional Deep Belief network provides a mechanism for using unlabeled data in learning features. The convolutional deep belief network is a stacked max-pooling Convolutional Restricted Boltzmann Machine (CRBM).

For this network, a layer-wise learning algorithm is developed for locally connected feature hierarchies. Layers are trained separately and from the bottom layer upward. When this type of training is performed, a multiplicative reduction in gradients does not occur. After training each layer, its weights are frozen and it is stacked. This model can be used to extract features from images, but it does not explicitly capture the spatial structure of images. In order to capture spatial structure, weight sharing and filtering are used. CRBM is introduced to incorporate weight sharing. It also consists of a visible layer, a detection layer and a pooling layer, as shown in Fig. 2.5. By stacking this CRBM into a multilayered hierarchy to extract large scale features, we obtain the Convolutional Deep Belief network. The CRBM consists of a detection layer to extract features and a max-pooling layer to make it translational invariant and to reduce the computational burden. [16]

The first layer in the CRBM is the visible layer. This layer is trained first, then the parameters for this layer are frozen and the probability of hidden units being activated are computed. These values are in between 0 and 1. They are obtained by applying a logistic sigmoid to the convolution output. Then this probabilistic output is down-sampled. The sub-sampled probabilities are input to the next visible layer, followed by the max pooling layer. [16]

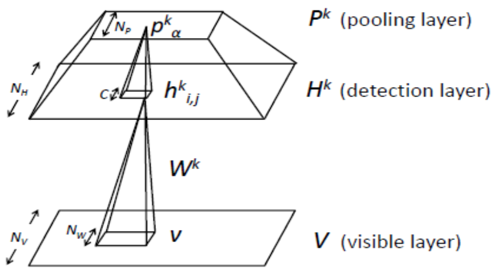


Figure 2.5: Convolution RBM with probabilistic max-pooling, from [4]

2.4 Some Applications

CNNs have been used for many visual pattern recognition systems. Some of these application are listed below.

Fukushima introduced the neocognitron, and he used it for handwritten digit recognition [2]. Then he introduced exensions of the neocognitron and used them for more complicated tasks, such as handwritten connected and isolated character recognition [17].

Lecun proposed his first CNN, LeNet-1, for handwritten digit recognition [6]. Later he introduced LeNet-5 to recognize isolated characters. He also proposed integrating the network into a graph framework to recognize handwritten amounts on bank checks. He also used CNNs based systems to recognize objects like cars or planes on noisy backgrounds [4]. For a camera based steering system for an autonomous mobile robot, he used a CNN architecture to avoid obstacles in an off-road environment [18].

CNNs have been used for facial recognition and robust face localization [19]. Fasel [20] used CNNs for face and facial expression detection. CNNs were also used for eye and face detection by Tivive and Bouzerdoum [21]. They used shunting inhibitory CNNs.

Saidane and Garcia used CNNs for scene text detection, where characters had considerable variations in color, shape and background [22]. CNNs were also used for text detection [23], hand detection and tracking [24] and for crack detection in sewer pipes [25], among many other applications.

2.5 Conclusions

In this chapter, the development of CNNs was discussed. It is an excellent network for pattern classification with automatic feature extraction. The CNN is based on an earlier neural network architecture, the neocognitron. Simple cells and complex cells in the neocognitron are replaced with convolution and sub-sampling layers in the CNN. The use of weight sharing in the CNN makes the network computationally strong. It reduces the number of free parameters, which improves convergence and generalization capacity. Also, the use of sub-sampling makes the network invariant to small distortions as well as to translations. The CNN is able to extract complex features from high dimensional input data. The main difference between the original neocognitron and the CNN is the method of training. The CNN is trained by using backpropagation to minimize a global error function, so there is no need to extract features manually.

The performance of any recognition system depends on accuracy, memory requirements and training time. Extensions of LeNet continue to be introduced and research is still going on to improve CNN performance. LeNet-4 and LeNet-5 were developed to create larger capacity recognizers for large, high dimensional data sets. Boosted networks also show significant improvements in accuracy, but at the expense of memory and computation speed. Also, for boosted networks, there is no need for unlimited data, since data can be created using a distortion strategy.

In the next chapter, we will focus on a general network architecture that can include all of the networks we have discussed so far as special cases. This general network architecture can be used to create any type of CNN. Also, we will discuss the type of training data needed for such a general network architecture.

CHAPTER 3

GENERAL NETWORK ARCHITECTURE AND SIMULATION

3.1 Introduction

As we have seen in chapter 2, different classification tasks require different numbers of layers, numbers of FMs, error functions, weight functions, etc. So there is a need to build a general network framework that can help users to create any kind of network, depending on the classification task. The main goal here is to develop a general purpose software package that can be used to simulate and train arbitrary layered neural networks for recognizing patterns in images and to enable new types of networks to be easily implemented. The goal is to develop user friendly software in which one can easily modify existing functions or add new functions, in order to reduce the amount of time required by users to implement any particular network. We will call this software package the Flexible Image Recognition Software Toolbox, or FIRST.

3.2 General Layered Structure

This section describes the general layered structure of the proposed network object. The number of layers, the number of FMs and the weight functions are dependent on the classification task. Therefore, there is a need to first design a general layered network structure, and then we will describe the details of each layer. A diagram of the general layered structure is shown in Fig. 3.1. We can see in the figure that there are M layers that are connected one after the another. The input is an image in matrix form.

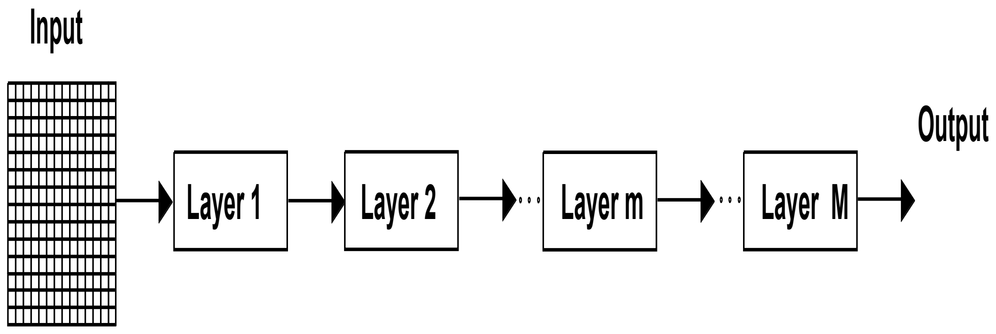


Figure 3.1: General Layered Structure

In this network, each layer consists of H^m cell (FM) components and a connection matrix, as shown in Fig. 3.2.

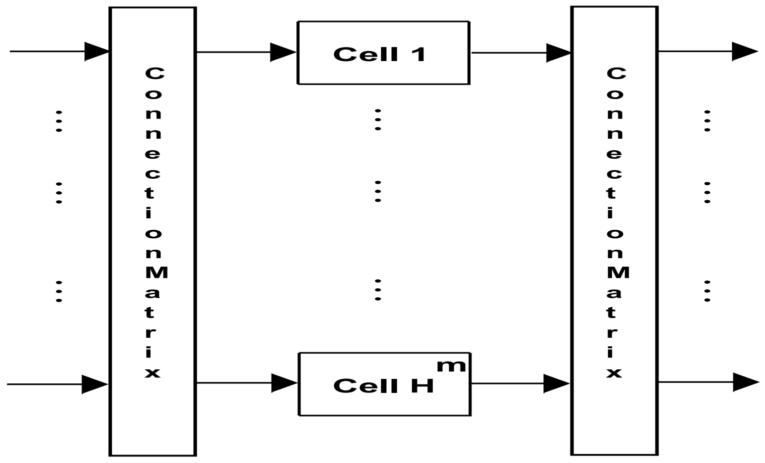


Figure 3.2: General Structure for a Single Layer

Each cell component consists of five parts: connection matrix, transformation function, weight function, net input function and transfer function. A detailed diagram showing the structure of a general cell is shown in Figure 3.3. Descriptions of the five parts of a layer follow.

1) Connection Matrix - a matrix that defines the connections between the FMs of two adjacent layers. It is denoted by $\mathbf{C}^m$ where m is the layer number.

2) Transformation Function - a function that comes after the connection matrix. It performs conversions from matrix to vector or vector to matrix. For some layers, no conversion

will take place. This transformation depends on the input requirement of next layer. It is denoted by vf, and a hexagon symbol is used to represent it graphically. The output of this transformation function is represented by $\mathbf{V}^{m,h}$. Here m is the layer number and h is the FM number.

3) Weight Function - an adjustable function that operates on the output of the transformation function. The adjustable parameters of this function are called weights. Examples of weight functions are: convolution, sub-sampling and dot product (standard matrix multiplication). The weight function is wf, and is denoted by a diamond shape in the network diagram. The weight is denoted $\mathbf{W}^{m,h}$. The output of this function is $\mathbf{Z}^{m,h}$.

4) Net Input Function - a function that combines the output of a weight function with a vector bias. The most common net input function is a summation between the bias and the output of the weight function. It is represented by nf and denoted by a circle in the network diagram. The bias is $\mathbf{b}^{m,h}$, and the output of the net input function is $\mathbf{N}^{m,h}$. It is called the net input.

5) Transfer Function - a fixed static function that converts the net input to the layer output. There are different types of transfer functions, such as linear, tan-sigmoid function, logistic sigmoid, etc. It is denoted as tf, and is represented by a square in the network diagram. The output of this function is $\mathbf{A}^{m,h}$, which is the layer output. We will provide more detailed explanations for each cell component in the following sections.

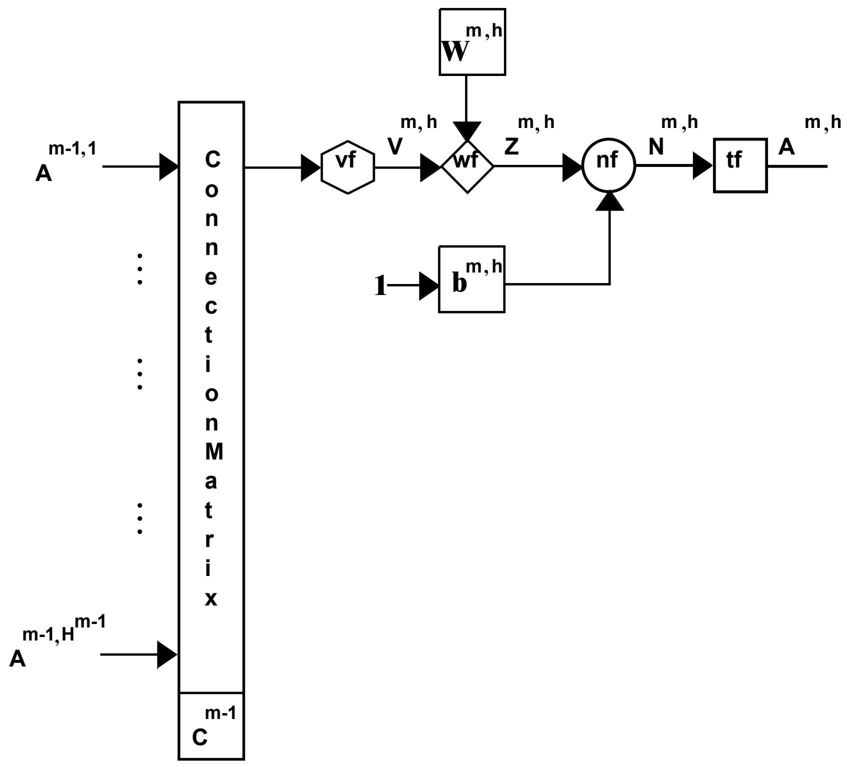


Figure 3.3: Diagram showing the structure of a general cell

3.3 Input Image Patterns

Digital images are composed of a finite number of pixels arranged in a grid pattern. The general network structures addressed in this thesis are designed to have inputs that are digital images. In most cases these images are two dimensional, although three dimensional or even one dimensional images could be used. In addition, we can think of the output of each layer as being a filtered image (or a set of filtered images), which is called an FM. These various types of images are discussed in more detail in the following sections.

3.3.1 2D Images

The basic two-dimensional image is a monochrome (grey scale) image that has been digitised. A 2D Image is a two-dimensional light intensity function $f(x, y)$ where x and y are spatial coordinates and the value of f at any point (x, y) is proportional to the brightness or grey value of the image at that point. A typical digitised image is one where 1) spatial and grey scale values have been made discrete, 2) intensity is measured across a regularly spaced grid in x and y directions, 3) intensities are sampled to 8 bits (256 values). Each element in the image array is called a pixel (picture element). Intensity values range between 0 to 255, where 0 represents black and 255 represents white. This type of 2D imaging is used in X-ray, MRI, etc. [26] (For the FIRST software, intensity values do not have to be 8 bit. Any value can be used.)

2D images can also be in color, but they require more storage space than gray scale images. Pixels in gray scale images need just one number to indicate the intensity of gray needed to render the pixel on screen. Any color can be created using the correct combination of red, green, and blue. Thus, pixels in color images are represented by three values (r,g,b). The values indicate the intensity of red, green, and blue needed to render the pixel on screen. The range of intensities is exactly the same as gray scale images, where 0 means none of the color appears in the pixel and 255 indicates the highest level of the color present in the pixel. For example, the triple (128, 0, 128) represents a medium purple while (255, 215, 0) represents gold [26]. In a sense, we can think of color images as having a third dimension, since a numerical value is provided at each x and y location in addition to each color.

3.3.2 3D Images

3D images are three dimensional. Like 2D images, 3D images also contain x and y coordinates but also contain one additional z coordinate for depth values (or color). 2D images have only height and width information, but 3D images have depth information as well. Therefore, 2D images give only limited information about the physical shape and size of an object in a scene. But this is not case with 3D, as 3D images express the geometry in terms of three-dimensional coordinates. [26]

3.3.3 Feature Maps

Generally, for images, feature extraction is used to reduce the dimensionality of data. Features contain relevant information, so irrelevant information is suppressed (e.g., image backgrounds). Features are meaningful and detectable. Features are associated with interesting scene elements, and they are invariant to some variations in the image formation process (e.g., invariant to viewpoint and illumination). There are two types of features. 1) Local Features: They are computed at multiple points in the images, so they are robust to occlusions and clutter; they describe image patches, e.g., edges, corners, etc. 2) Global Features: They have the ability to generalize an entire object with a single vector. They describe images as a whole, e.g., contour representations, shape descriptors, and texture features [26].

In the general network structure, the network input will be an image (usually a 2D image, but it could also be 3D or 1D). The output of a layer will consist of sets of images. (We use the term image in an abstract sense.) Neurons in a layer are arranged in a plane, where all neurons share the same set of weights. This set of planar neurons is referred to as a feature map (FM). A layer may consist of multiple FMs, each one corresponding to a different network cell (see Fig. 3.2).

An initial layer in a general network may extract local features in its FMs. Later layers can combine local features to extract global features. FMs in convolutional neural networks act like neurons in fully connected neural networks. The location of each pixel in an FM is formed by the region convolved on the FM of the preceding layer. The dimensions of these FMs depend on the dimensions of the inputs presented to the cells and on the operation performed on the inputs of those cells. Every cell output is an FM. Each pixel of an FM stores the output of a neuron. All the pixels or neurons in one FM share identical weights. This is similar to the cells in the cell-planes of the neocognitron. The total number of FMs (or cells) for each layer depends on the application. Each FM extracts a different feature, because they each have inputs from different sources, as well as different weights. For a general network, the index of an FM is denoted by h, and the total number of FMs for layer m is denoted by H^m .

3.4 Connection Matrix

The connection matrix defines the connections between the cells (FMs) in adjacent layers. It is similar to the connection table in the neocognitron (see Table 2.1). If element i, j of the connection matrix is equal to 1, then cell j of the current layer is connected to cell i in the previous layer. If that element is 0, there is no connection between those cells. Using these connection matrices, there can be any number of possible combinations of connections between FMs of two layers, which allows the structure to be used for many different applications. There is no need to always have full connections between layers. This can avoid high computational costs and overfitting. Also, irregular connections can break network symmetry.

The connection matrix columns correspond to the FM number in the current layer, and the rows represent the FM number in the previous layer. There is a connection matrix for each layer. If there is a connection between FM j of the current layer m to FM i of the previous layer, then $C^m(i, j) = 1$.

3.5 Transformation Function

The transformation function comes after the connection matrix. The output of this function is denoted $\mathbf{V}^{m,h}$. This is represented in variable form as follows:

$$\mathbf{A^{m-1}} \rightarrow \mathbf{V^m} \tag{3.1}$$

3.5.1 General Transformation Idea

In this network, along with convolution and sub-sampling layers, full connection layers are also present. For a full connection layer, the input and output are in vector form, whereas, for other types of layers, the input and output are in matrix (image) form. So sometimes there is a need for conversion from matrix to vector or vector to matrix. Therefore, before passing the output of one layer as input to the next, it is passed through a transformation function. The types of transformations are discussed in detail in the next subsections.

3.5.2 Vector Transformation

The output of this operation is a vector. There can be a matrix to vector transformation, or, if the input is already a vector, no transformation will take place. If the input is a matrix, then the rows of the matrix are stacked together to form the vector output, as shown in Equation 3.2 below.

Matrix to Vector conversion

Scalar form (v):

$$v_i^{m,h} = a^{m-1,h}_{ceil(i/S_r^{m-1}),S_r^{m-1}-mod(S_r^{m-1}-i,S_r^{m-1})} \tag{3.2}$$

Vector form ($\mathbf{v}$):

$$\mathbf{v}^{m,h} = vec(\mathbf{A}^{m-1,h}) \tag{3.3}$$

Vector to Vector conversion

Scalar form (V):

$$v_i^{m,h} = a_i^{m-1,h} \tag{3.4}$$

Vector form ($\mathbf{v}$):

$$\mathbf{v}^{m,h} = \mathbf{a}^{m-1,h} \tag{3.5}$$

Here S_r^{m-1} represents the number of rows in layer $m - 1$, and the maximum value of i is the total number of elements in the matrix. Here vec denotes the vector transformation.

3.5.3 Matrix Transformation

The output of this transformation is a matrix form. There is a vector to matrix conversion, or there is no conversion if the input is a matrix.

Vector to Matrix conversion

Scalar form (v):

$$v^{m,h}_{ceil(i/S^m_r),S^m_r-mod(S^m_r-i,S^m_r)} = a^{m-1,h}_i \tag{3.6}$$

Matrix form ($\mathbf{V}$*):*

$$\mathbf{V}^{m,h} = mat(\mathbf{a}^{m-1,h}) \tag{3.7}$$

Matrix to Matrix conversion

Scalar form (V):

$$v^{m,h}_{i,j} = a^{m-1,h}_{i,j} \tag{3.8}$$

Matrix form ($\mathbf{V}$*):*

$$\mathbf{V}^{m,h} = \mathbf{A}^{m-1,h} \tag{3.9}$$

Here mat denotes the matrix transformation.

3.6 Weight Function

3.6.1 General Concept of Weight Function

In the standard multilayer perceptron, a weight matrix multiplies the input to a layer. For the network structure used in this thesis, we generalize this concept. The input to a layer is operated on by a set of adjustable parameters that we call weights. This operation could be a standard matrix multiplication (dot product), or it could be a convolution operation. There are many different types of weight operations, a few of which are defined in following subsections.

A weight function is defined for each cell of each layer. The output of a weight function is denoted by $\mathbf{Z}^{m,h}$ for cell h of layer m. The output of the transformation function is the input to the weight function, so it is written as follows:

$$\mathbf{V^m} \to \mathbf{Z^m} \tag{3.10}$$

3.6.2 Convolution Weight Function

A convolution weight function performs a two-dimensional convolution between an input image and a weight matrix (convolution kernel). It is a kind of spatial filtering, and because the weights are adjustable, it is an adaptive local filter. In this filter, each output pixel is unresponsive to variations outside of its receptive field with respect to the input. So convolution is able to detect local features, such as edges, corners, etc. Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part. This means that the kernel that we learn in one part of the image can also be applied to other parts of the image. We can use the same kernel at all locations.

A convolutional layer is made up of FMs in which the convolution operation is performed on the previous layer's output. There is one convolution kernel or receptive field (weight matrix) for each connection between an FM in the previous layer and an FM in the current layer. A convolution layer acts like feature extractor, where one or more output images (FMs) from previous layers are convolved with one or more kernels or weights to produce an output image (or multiple images).

The convolution operation is performed over boundaries that are defined by the size of the FMs in the preceding layer. The sizes of FMs in successive convolutional layers decrease. The amount of decrease depends on the size of the kernel. If a convolutional layer takes an input of size $R_r \times R_c$ and performs a convolution with H^m kernels of size $r^m \times c^m$, this will produce H^m outputs of size $(R_r - r^m + 1, R_c - c^m + 1)$. So H^m specific features are detected by scanning across the entire input image. By convolving with the same kernel across an image produces a more complex network without exponentially increasing the number of network parameters. The input image is padded with zeros around its boundaries to allow the center of the kernel to reach the edge of the image. This was suggested by Lecun to increase the accuracy of the pattern recognition [6].

The general equation for convolution is

Scalar form (z):

$$z_{i,j}^{m,h} = \sum_{l \epsilon C^{m,h}} \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} w_{u,v}^{m,h,l} v_{u+i-1,v+j-1}^{m,l} \tag{3.11}$$

(For the first layer, $v^{0,l} = p_l$ where p_l is the input image. In most cases $H^0 = 1$, so l is only equal to 1.)

Matrix form ($\mathbf{Z}$):

$$\mathbf{Z}^{m,h} = \sum_{l \epsilon C^{m,h}} \mathbf{W}^{m,h,l} \star \mathbf{V}^{m,l} \tag{3.12}$$

The symbol $\star$ is used to denote convolution. $\mathbf{W}^{m,h,l}$ is the weight matrix (convolution kernel) that connects the l^{th} FM in layer $m - 1$ to FM h in layer m. The corresponding element of this matrix in row u and column v is $w_{u,v}^{m,h,l}$. $\mathbf{Z}^{m,h}$ is the FM h output of the weight function in layer m, and $z_{i,j}^{m,h}$ is the corresponding element in row i, column j. $C^{m,h}$ is the set of indices of FMs in layer $m - 1$ that connect to FM h in layer m. $v_{i,j}^{m,h}$ and $\mathbf{V}^{m,l}$ are transformation function outputs in scalar and vector (or matrix) form, respectively.

3.6.3 Subsampling Weight Function

A subsampling layer, like a convolution layer, takes FMs (or images) as inputs and produces FMs as outputs. Subsampling layers generally follow convolution layers. The subsampling operation consists of a summation over non-overlapping blocks from a previous layer's FMs, followed by multiplication by a scalar weight. This operation reduces the size of the FM by some factor. This factor can be two or more, depending on the size of the input image. In practice, this factor would not be too large, or important information might be lost. The reduction is done in such a way that a feature detected at the output of the previous layer is equally detected at the output of the subsampling layer. The only change is in the resolution. The subsampling operation adds robustness against spatial dependence (e.g., shifts, distortions, translations, scaling), and it also reduces the number of network parameters.

The subsampling operation is defined as

Scalar form (z):

$$z_{i,j}^{m,h} = \left\{ \sum_{k=1}^{r^m} \sum_{l=1}^{c^m} v_{r^m(i-1)+k, c^m(j-1)+l}^{m,h} \right\} w^{m,h} \tag{3.13}$$

*Matrix form (**Z**):*

$$\mathbf{Z}^{m,h} = w^{m,h} \boxplus_{S_r^m, S_c^m} \mathbf{V}^{m,h} \tag{3.14}$$

The summation operation is denoted $\boxplus_{S_r^m, S_c^m}$, where S_r^m and S_c^m are the dimensions of the output matrix $\mathbf{Z}^{m,h}$. The scalar weight is denoted $w^{m,h}$. Other variables are defined in the same manner as in the convolution weight function.

3.6.4 Dot Product Weight Function

A dot product layer performs a standard matrix multiplication to produce the layer output. This operation is a multiplication between the weight matrix and the input vector, which is the output of the preceding layer. The input to this function is always in vector form, and the output is also in vector form.

The equation for the dot product function is

Scalar form (z):

$$z_k^m = \sum_{h \epsilon C^{m,1}} \sum_{i=1}^{S_r^{m-1} \times S_c^{m-1}} w_{k,i}^{m,h} v_i^{m,h} \tag{3.15}$$

*Vector form (**z**):*

$$\mathbf{z}^m = \sum_{h \epsilon C^{m,1}} \mathbf{W}^{m,h} \mathbf{v}^{m,h} \tag{3.16}$$

3.7 Net Input Function

The net input function combines the bias with the output of the weight function to produce the net input. It performs the following operation.

$$\mathbf{Z^m} \rightarrow \mathbf{N^m} \tag{3.17}$$

This function consists of operations like summation, multiplication, etc., depending on the application. If a radial basis layer is used at last layer, for example, then the operation is multiplication. These operations are performed on the bias and the output of the weight function. The bias enables neurons to shift their transfer functions. This is necessary to give the network full pattern recognition capability.

3.7.1 Summation

The equation for the summation net input function is

Scalar form (n, feature map):

$$n_{i,j}^{m,h} = z_{i,j}^{m,h} + b^{m,h} \tag{3.18}$$

Matrix form ($\mathbf{N}$*, feature map):*

$$\mathbf{N}^{m,h} = \mathbf{Z}^{m,h} + b^{m,h} \times \mathbf{1}_{S_r^m \times S_c^m} \tag{3.19}$$

Scalar form (n, vector):

$$n_k^m = z_k^m + b_k^m \tag{3.20}$$

Vector form ($\mathbf{n}$*, vector):*

$$\mathbf{n}^m = \mathbf{z}^m + \mathbf{b}^m \tag{3.21}$$

The dimension of the output of the net input function is always the same as the dimension of the input. $n_{i,j}^{m,h}$ and $\mathbf{N}^{m,h}$ represent the output of the net input function in scalar and matrix form, respectively.

3.8 Transfer Function

The transfer function, also called the activation function, is a fixed, static, differentiable function that converts the net input to the layer output. This can be written as

$$\mathbf{N^m} \rightarrow \mathbf{A^m} \tag{3.22}$$

The transfer functions add non-linearities to the network. Because of these non-linearities, the network can learn complex non-linear tasks. There are different types of transfer functions. The most commonly used non-linear transfer functions are the logistic function and the hyperbolic tangent function. These sigmoid functions are linear near zero, but they limit the output of a neuron to positive or negative asymptotes for larger magnitude inputs, acting like a smooth step function. A neuron is said to be saturated when the output is approaching one of the asymptotes. The letter f is used to denote the transfer function. For example, if the transfer function is the logistic sigmoid, then f can be written as:

$$f(n) = 1/(1 + exp(-n)) \tag{3.23}$$

3.8.1 General Transfer Function

The generale transfer function equation is shown below:

Scalar form (a, feature map):

$$a_{i,j}^{m,h} = f^{m,h}(n_{i,j}^{m,h}) \tag{3.24}$$

Matrix form ($\mathbf{A}$*, feature map):*

$$\mathbf{A}^{m,h} = \mathbf{F}^{m,h}(\mathbf{N}^{m,h}) \tag{3.25}$$

Scalar form (a, vector):

$$a_k^m = f^m(n_k^m) \tag{3.26}$$

Vector form ($\mathbf{a}$*, vector):*

$$\mathbf{a}^m = \mathbf{F}^m(\mathbf{n}^m) \tag{3.27}$$

3.9 Conclusions

In this chapter, we introduced a general framework for representing pattern recognition neural networks that are designed to work on images. This framework forms the basis for a software package for building, training, and simulating such networks. The software has ready-made versions of some classical network architectures as Lecun's LeNet-1, LeNet-5 and LeNet-7. However, it is also easy for users to create custom networks with arbitrary numbers of layers. Users can define their own weight functions, net input functions and transfer functions.

The basic idea is that the general framework should be very modular. Users can define their own architectures and even insert custom modules, while the software engine does the training and simulation without the need of any modification by users. It is the generality and modularity of the network framework that was introduced in this chapter that makes this possible. (See Figure 3.4 for an example of the types of networks that can be created.)

In the next chapter, we will discuss training algorithms for the general network framework introduced in this chapter. In addition to steepest descent, other gradient-based methods are also discussed and compared. The accuracy of networks trained with steepest descent is good, but computational speed is major drawback, so more efficient optimization strategies are discussed in the next chapter. The key to all of the training algorithms is the computation of the gradient of the performance index with respect to the weights and biases of the network. This gradient computation will be the major focus of the next chapter.

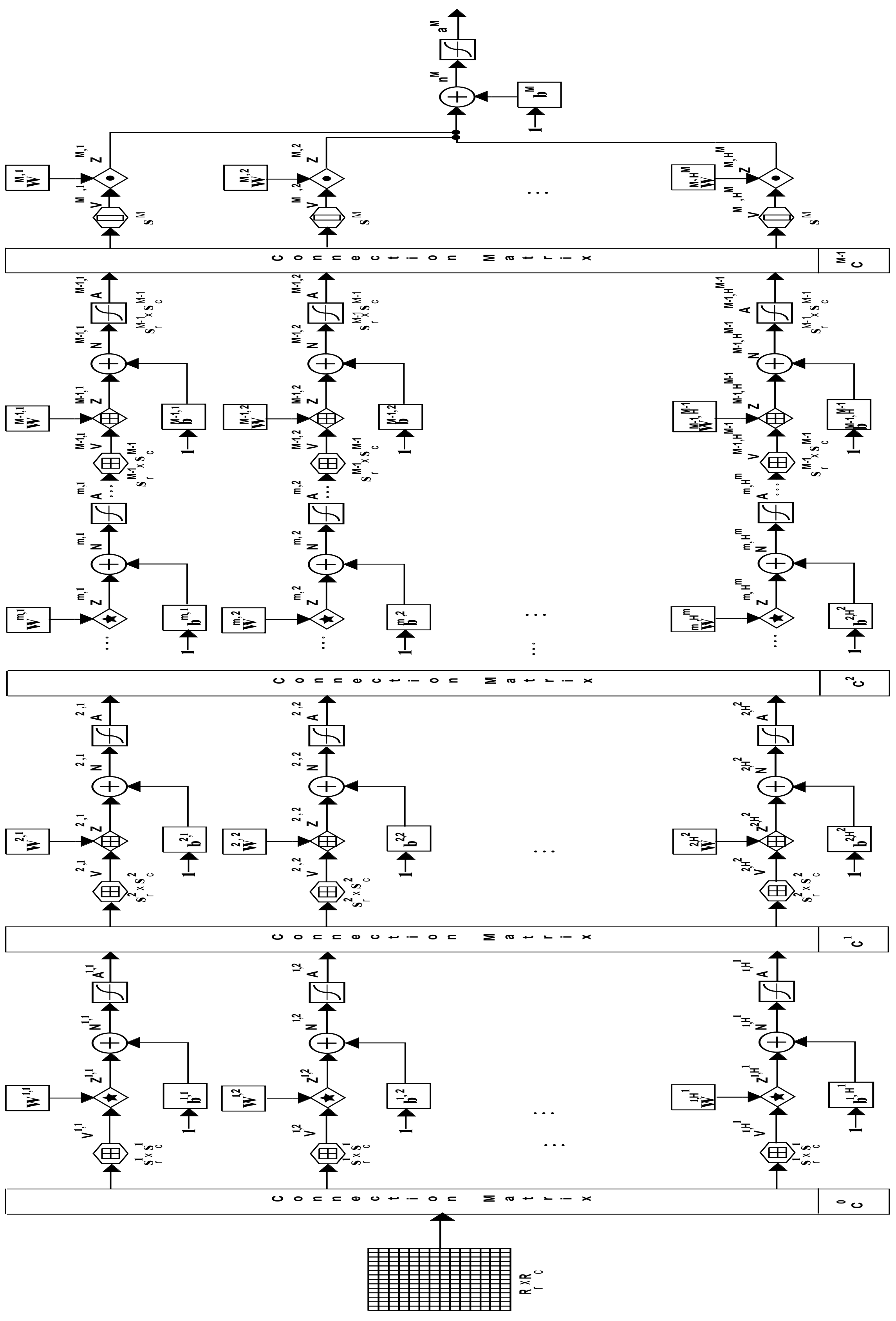


Figure 3.4: Overall General Block-Diagram

CHAPTER 4

General Network Training

4.1 Introduction

Network training refers to the process of adjusting the weights and biases of the network. Most training techniques are types of optimization algorithms. A performance index is chosen (e.g., mean square error), and then the weights and biases are adjusted in an iterative way until the performance index is optimized. The optimization algorithms considered in this thesis are all based on the gradient of the performance index with respect to the network weights and biases. The simplest optimization algorithm is steepest descent, in which the weights are moved in the negative gradient direction. Conjugate gradient and quasi-Newton algorithms also use the gradient and are more efficient than steepest descent. A detailed discussion of several training algorithms is given in the following sections. The most suitable training algorithm for a particular problem is dependent on the nature of training data, the neural network architecture and the complexity of the desired decision function. There is no specific method to decide which training algorithm is best for a given problem. FIRST has incorporated several training algorithms, and others could be easily added. The framework is modular.

The key to all of our training algorithms is the computation of the gradient, so this chapter will spend significant space describing how the gradient can be computed for the general network structure. The key concept is the chain rule of calculus. Because the network output is computed in stages (layers), and because the performance index is computed at the last layer of the network, we can compute the derivative of the performance with respect to the weights of any layer by using the chain rule on each layer, starting from the last layer and moving backward through the layers. This backpropagation of the derivatives through the network to compute the gradient will be described in the following sections.

4.2 Performance Index

The performance index (PI) is defined to measure how well the network performs. Generally it is a normed difference between the network's output and the correct or target outputs over a specified training set. During training, the network parameters (weights and biases) are updated to minimize the PI. The training algorithm forces the neural network to implement the optimal decision function [5].

For a general network, the choice of PI will depend on the classification task, the nature of the training data, and the nature of the target output, e.g., the PI used in LeNet-5 is different than in LeNet-1. In our software, the PI is defined in such a way that it can be changed easily. The description of two commonly used PIs are given next. We use F to represent the PI.

1) Mean square error: It is the most commonly used PI, and is defined in Equation 4.1 [5].

$$F = \frac{1}{Q \times S^M} \sum_{q=1}^{Q} \sum_{i=1}^{S^M} (t_{i,q} - a_{i,q})^2 \tag{4.1}$$

Here q is the pattern number, i is the neuron number in the output layer, t is the target value, and a is the network output.

2) Cross Entropy: For classification tasks, the cross entropy PI is often used, since the normal mean square PI does not provide the proper distinction between classes. The multi-class cross-entropy loss is given in Equation 4.2. It is often used when the output of the neural net uses a softmax transfer function. This equation is non-negative and equals zero at its minimum value, when $a_{i,q} = t_{i,q}$ for all i and q. Here a 1-of-c class coding scheme is used where c denotes the number of classes. [27]

$$F = \sum_{q=1}^{Q} \sum_{i=1}^{S^M} t_{i,q} ln(a_{i,q}/t_{i,q}) \tag{4.2}$$

4.3 Gradient Calculation

All of the training methods described in this thesis are based on the gradient, which is the vector derivative of the PI with respect to the weights and biases of the network. This section describes how the gradient can be computed for the general network architecture.

4.3.1 Backpropagation Algorithm Introduction

The term "backpropagation" refers to the manner in which the gradient is computed. The general network architecture is composed of layers that are connected in series from the network input to the output of the network (see Figure 3.1). Within each layer, the network response is also computed in stages, $v - z - n - a$ (see Figure 3.3). When computing the derivative of the PI with respect to the weights and biases of a given layer, we can use the chain rule backward through the layers of the network $(M - (M-1) - ...... - 2 - 1)$ and through the stages of each layer ($da - dn - dz - dv$).

Equation 4.3 is a simple scalar description of the backpropagation process through a layer, and Equation 4.4 shows the corresponding gradient calculation. Figure 4.1 illustrates the process.

$$\frac{\partial F}{\partial v} = \frac{\partial F}{\partial a} \times \frac{\partial a}{\partial n} \times \frac{\partial n}{\partial z} \times \frac{\partial z}{\partial v} \tag{4.3}$$

$$\frac{\partial F}{\partial w} = \frac{\partial F}{\partial a} \times \frac{\partial a}{\partial n} \times \frac{\partial n}{\partial z} \times \frac{\partial z}{\partial w} \tag{4.4}$$

A detailed description of the complete general backpropagation calculation of the gradient is given in the following subsection. The notation used for the gradient calculations are shown in the following tables.

Table 4.1: Scalar Element at feature map h of layer m

Derivative of PI w.r.t.	Symbol
Transform function output	$dV_{i,j}^{m,h}$
Transfer function output	$dA_{i,j}^{m,h}$
Net function output	$dN_{i,j}^{m,h}$
Weight function output	$dZ_{i,j}^{m,h}$
Weight	$dW_{i,j}^{m,h}$
Bias	$db^{m,h}$

Table 4.2: Matrix at feature map h of layer m

Derivative of PI w.r.t.	Symbol
Transform function output	$\mathbf{dV}^{m,h}$
Transfer function output	$\mathbf{dA}^{m,h}$
Net function output(sensitivity)	$\mathbf{dN}^{m,h}$
Weight function output	$\mathbf{dZ}^{m,h}$
Weight	$\mathbf{dW}^{m,h}$
Bias(vector form)	$\mathbf{db}^{m}$

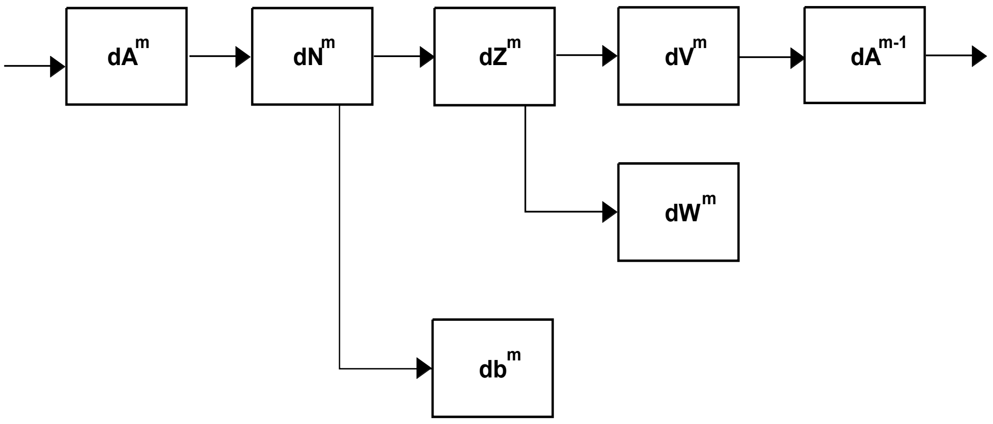


Figure 4.1: Overall General Block-Diagram

In the remaining subsections we will step through the backpropagation process one stage at a time, beginning with an initialization at the output of the network. For simplicity of development, we will demonstrate the gradient calculation for a partial PI involving one input pattern (a stochastic gradient). To find the complete (batch) gradient, we can just add the partial gradients over all training patterns.

4.3.2 Initialization

The backpropagation process begins at the output layer by taking the gradient of the PI with respect to the network output. The following equations demonstrate the initialization for the case where the PI is mean square error.

Scalar form (dA):

$$dA_k^M = \frac{\partial \hat{F}}{\partial a_k^M} = \frac{\partial \sum_{k=1}^{S^M} (1/S^M) \times (a_k - t_k)^2}{\partial a_k^M} = \sum_{k=1}^{S^M} \frac{2}{S^M}(a_k - t_k) \tag{4.5}$$

Vector form ($\mathbf{da}$):

$$\mathbf{da}^M = \frac{2}{S^M}(\mathbf{a}^M - \mathbf{t}) \tag{4.6}$$

4.3.3 Transfer Function

The first step in backpropagation through a layer is to backpropagate through the transfer function. The transfer function output is a. The derivative of the PI with respect to the net input is

$$\frac{\partial F}{\partial n} = \frac{\partial F}{\partial a} \times \frac{\partial a}{\partial n} \tag{4.7}$$

Here,

$$\frac{\partial a}{\partial n} = \frac{\partial f(n)}{\partial n} = \dot{f}(n) \tag{4.8}$$

The term $\frac{\partial F}{\partial a}$ is dA, which was computed in the initialization stage (or, if this is not the last layer, it was backpropagated from the next layer), so the process is

$$\mathbf{dA^m} \rightarrow \mathbf{dN^m} \tag{4.9}$$

The general backpropagation equation through the transfer function is given as follows:

Scalar form (dN):

$$dN_{i,j}^{m,h} = \frac{\partial \hat{F}}{\partial n_{i,j}^{m,h}} = \frac{\partial a_{i,j}^{m,h}}{\partial n_{i,j}^{m,h}} \times \frac{\partial \hat{F}}{\partial a_{i,j}^{m,h}} \tag{4.10}$$

here,

$$\frac{\partial \hat{F}}{\partial a_{i,j}^{m,h}} = dA_{i,j}^{m,h} \tag{4.11}$$

$$dN_{i,j}^{m,h} = \dot{f}^{m,h}(n_{i,j}^{m,h}) \times dA_{i,j}^{m,h} \tag{4.12}$$

Matrix form ($\mathbf{dN}$*):*

$$\mathbf{dN}^{m,h} = \dot{\mathbf{F}}^{m,h} \circ \mathbf{dA}^{m,h} \tag{4.13}$$

Here $\circ$ is the Hadamard product, which is an element by element matrix multiplication. There are different types of transfer functions, and the derivatives for each transfer function will be different [5].

Tan sigmoid transfer function

This transfer function squashes the input into the range from -1 to +1, according to

$$a = \frac{e^n - e^{-n}}{e^n + e^{-n}} \tag{4.14}$$

The derivative for this function is

$$\begin{aligned}
\dot{f}(n) &= \frac{\partial a}{\partial n} \\
&= \frac{\partial}{\partial n}(e^n - e^{-n}/e^n + e^{-n}) \\
&= -(e^n - e^{-n}/(e^n + e^{-n})^2)(e^n - e^{-n}) + (e^n + e^{-n}/e^n + e^{-n}) \\
&= 1 - ((e^n - e^{-n})^2/(e^n + e^{-n})^2) \\
&= 1 - a^2
\end{aligned} \tag{4.15}$$

Log sigmoid transfer function

This is a transfer function which takes the input and squashes it into range the from 0 to 1, according to

$$a = \frac{1}{1 + e^{-n}} \tag{4.16}$$

The derivative for this function is given as

$$\begin{aligned}\dot{f}(n) &= \frac{\partial a}{\partial n} \\ &= \frac{\partial}{\partial n}(1/1+e^{n}) \\ &= -(e^{-n}/(1+e^{-n})^2) \\ &= (1-(1/(1+e^{-n})))(1/(1+e^{-n})) \\ &= (1-a)a\end{aligned} \tag{4.17}$$

Linear transfer function

This transfer function maps the input to the output in a linear fashion, as in

$$a = n \tag{4.18}$$

Its derivative can be easily calculated.

$$\dot{f}(n) = \frac{\partial a}{\partial n} = \frac{\partial}{\partial n}(n) = 1 \tag{4.19}$$

4.3.4 Net Input Function

The next stage of backpropagation is to backpropagate through the net input function. There are two paths for this stage. One is to compute the derivative of the PI with respect to the output of the weight function, z, to continue the backpropagation. The other path is to compute the derivative of the PI with respect to the bias in the current layer. This can be written

$$\frac{\partial F}{\partial z} = \frac{\partial F}{\partial n} \times \frac{\partial n}{\partial z} \tag{4.20}$$

$$\frac{\partial F}{\partial b} = \frac{\partial F}{\partial n} \times \frac{\partial n}{\partial b} \tag{4.21}$$

The term $\frac{\partial F}{\partial n}$ is dN, which was computed at the previous layer, so the process can be presented

$$\mathbf{dN^m} \rightarrow \mathbf{dZ^m} \tag{4.22}$$

$$\mathbf{dN^m} \rightarrow db^m \tag{4.23}$$

The following subsections describe the details of this operation for two common net input functions.

Summation

The summation net input function is $n = z + b$, where z is the weight function output (see Equation 3.20). The partial derivative of the net input n with respect to the weight function output z is one, so backpropagation through the net input function is given by

Scalar form (dZ):

$$dZ_{i,j}^{m,h} = dN_{i,j}^{m,h} \tag{4.24}$$

Matrix form ($\mathbf{dZ}$*):*

$$\mathbf{dZ}^{m,h} = \mathbf{dN}^{m,h} \tag{4.25}$$

The bias gradient for feature maps is given by

Scalar form (db):

$$db^{m,h} = \frac{\partial \hat{F}}{\partial b^{m,h}} = \frac{\partial \hat{F}}{\partial n_{i,j}^{m,h}} \times \frac{\partial n_{i,j}^{m,h}}{\partial b^{m,h}} \tag{4.26}$$

$$db^{m,h} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dN_{i,j}^{m,h} \tag{4.27}$$

Matrix form (db*):*

$$db^{m,h} = (\boxplus_{1,1} \mathbf{dN}^{m,h})) \tag{4.28}$$

where $\boxplus_{1,1}$ is the same operator used for the sub-sampling reduction operation, and after reduction the size of matrix will be 1×1.

The bias gradient for the vector form (dot product layers) is given by

Scalar form (db):

$$db_k^m = \frac{\partial \hat{F}}{\partial b_k^m} = \frac{\partial \hat{F}}{\partial n_k^m} \times \frac{\partial n_k^m}{\partial b_k^m} \tag{4.29}$$

$$db_k^m = dN_k^m \tag{4.30}$$

Vector form ($\mathbf{db}$*):*

$$\mathbf{db}^m = \mathbf{dn}^m \tag{4.31}$$

Product

Here product means $n = z \times b$. Therefore, the derivative can be calculated as follows:

Scalar form (dZ):

$$dZ_{i,j}^{m,h} = b^{m,h} dN_{i,j}^{m,h} \tag{4.32}$$

Matrix form ($\mathbf{dZ}$):

$$\mathbf{dZ}^{m,h} = \mathbf{b}^{m,h}\mathbf{dN}^{m,h} \tag{4.33}$$

Bias gradient for feature maps is given by

Scalar form (db):

$$db^{m,h} = \frac{\partial \hat{F}}{\partial b^{m,h}} = \frac{\partial \hat{F}}{\partial n_{i,j}^{m,h}} \times \frac{\partial n_{i,j}^{m,h}}{\partial b^{m,h}} \tag{4.34}$$

$$db^{m,h} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dN_{i,j}^{m,h} z_{i,j}^{m,h} \tag{4.35}$$

Matrix form (db):

$$db^{m,h} = (\boxplus_{1,1}\mathbf{dN}^{m,h} \circ \mathbf{Z}^{m,h}) \tag{4.36}$$

Bias gradient in standard form is

Scalar form (db):

$$db_k^m = \frac{\partial \hat{F}}{\partial b_k^m} = \frac{\partial \hat{F}}{\partial n_k^m} \times \frac{\partial n_k^m}{\partial b_k^m} \tag{4.37}$$

$$db_k^m = dN_k^m z_k^m \tag{4.38}$$

Vector form ($\mathbf{db}$):

$$\mathbf{db}^m = \mathbf{dn}^m \mathbf{z}^m \tag{4.39}$$

4.3.5 Weight Function

The next stage of the backpropagation process is through the weight function. This stage has two paths. First, we need to find the derivative of the PI with respect to the output of the transformation function, v, to continue the backpropagation process. Second, we need to find the derivative of the PI with respect to the weights in the current layer. This can be written in mathematical form as follows:

$$\frac{\partial F}{\partial v} = \frac{\partial F}{\partial z} \times \frac{\partial z}{\partial v} \tag{4.40}$$

$$\frac{\partial F}{\partial w} = \frac{\partial F}{\partial z} \times \frac{\partial z}{\partial w} \tag{4.41}$$

The term $\frac{\partial F}{\partial z}$ is dZ, which was computed at the previous backpropagation stage.

$$\mathbf{dZ^m} \rightarrow \mathbf{dV^m}, \mathbf{dZ^m} \rightarrow \mathbf{dW^m} \tag{4.42}$$

Convolution Weight Function

Taking the derivative of Equation 3.11, we can find the following forms.

Scalar form (dV):

$$dV_{i,j}^{m,l} = \frac{\partial \hat{F}}{\partial v_{i,j}^{m,l}} = \frac{\partial \hat{F}}{\partial z_{u,v}^{m,h}} \times \frac{\partial z_{u,v}^{m,h}}{\partial v_{i,j}^{m,l}} = \sum_{h \epsilon C_b^{m,l}} \sum_{u=1}^{S_r^m} \sum_{v=1}^{S_c^m} dZ_{i,j}^{m,h} w_{i-u+1,j-v+1}^{m,h,l} \tag{4.43}$$

Where $C_b^{m,l}$ is the set of FMs in layer m that the lth FM in layer $m-1$ connects to.

Matrix form ($\mathbf{dV}$):

$$\mathbf{dV}^{m,l} = \sum_{h \epsilon C_b^{m,l}} (rot180(\mathbf{W}^{m,h,l})) \star (\mathbf{dZ}^{m,h}) \tag{4.44}$$

Here $\star$ is used to denote convolution operation and rot180 means matrix is rotated by 180 angle. Weight update is given in following equation.

Scalar form (dW):

$$dW_{u,v}^{m,h,l} = \frac{\partial \hat{F}}{\partial w_{u,v}^{m,h,l}} = \frac{\partial \hat{F}}{\partial z_{i,j}^{m,h}} \times \frac{\partial z_{i,j}^{m,h}}{\partial w_{u,v}^{m,h,l}} \tag{4.45}$$

$$dW_{u,v}^{m,h,l} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dZ_{i,j}^{m,h} v_{u+i-1,v+j-1}^{m,l} \tag{4.46}$$

here l is feature map in layer $m-1$.

Matrix form ($\mathbf{dW}$):

$$\mathbf{dW}^{m,h,l} = \mathbf{dZ}^{m,h} \star \mathbf{V}^{m,l} \tag{4.47}$$

Sub-sampling Weight Function

Taking the derivative of Equation 3.13, we can write the following.

Scalar form (dV):

$$dV^{m,h}_{r^m(i-1)+k,c^m(j-1)+l} = \frac{\partial \hat{F}}{\partial v^{m,h}_{i,j}} = \frac{\partial \hat{F}}{\partial z^{m,h}_{i,j}} \times \frac{\partial z^{m,h}_{i,j}}{\partial v^{m,h}_{i,j}} = dZ^{m,h}_{i,j} \times w^{m,h} \tag{4.48}$$

For k=1 to r^m and l=1 to c^m.

Matrix form ($\mathbf{dV}$):

$$\mathbf{dV}^{m,h} = (w^{m,h}) \boxminus_{S_r^{m-1},S_c^{m-1}} (\mathbf{dZ}^{m,h}) \tag{4.49}$$

where $\boxminus_{j,k} A$ takes each element of the matrix A and expands it so that the resulting matrix has j rows and k columns.

Scalar form (dW):

$$dW^{m,h} = \frac{\partial \hat{F}}{\partial w^{m,h}} = \frac{\partial \hat{F}}{\partial z^{m,h}_{i,j}} \times \frac{\partial z^{m,h}_{i,j}}{\partial w^{m,h}} \tag{4.50}$$

$$dW^{m,h} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dZ^{m,h}_{i,j} \left\{ \sum_{k=1}^{r^m} \sum_{l=1}^{c^m} v^{m,h}_{r^m(i-1)+k,c^m(j-1)+l} \right\} \tag{4.51}$$

Matrix form (dW):

$$dW^{m,h} = \boxplus_{1,1}(\mathbf{dZ}^{m,h} \circ (\boxplus_{S_r^m,S_c^m} \mathbf{V}^{m,h})) \tag{4.52}$$

First it performs a reduction operation that produces a matrix of size S_r^m by S_c^m, and then, after a Hadamard matrix multiplication, it performs another reduction to produce the scalar gradient.

Dot product Weight Function

Taking the derivative of Equation 3.15, we can derive the following.

Scalar form (dV):

$$dV^{m,h}_i = \frac{\partial \hat{F}}{\partial v^{m,h}_i} = \sum_{k=1}^{S^M} \frac{\partial \hat{F}}{\partial z^m_k} \times \frac{\partial z^m_k}{\partial v^{m,h}_i} = \sum_{k=1}^{S^M} dZ^m_k \times w^{m,h}_{k,i} \tag{4.53}$$

Matrix form ($\mathbf{dV}$ *):*

$$\mathbf{dV}^{m,h} = (\mathbf{W}^{m,h})^T * (\mathbf{dZ}^m) \tag{4.54}$$

Here $*$ is used to denote normal matrix multiplication. The gradient is as follows.

Scalar form (dW):

$$dW_{k,i}^{m,h} = \frac{\partial \hat{F}}{\partial w_{k,i}^{m,h}} = \frac{\partial \hat{F}}{\partial z_k^m} \times \frac{\partial z_k^m}{\partial w_{k,i}^{m,h}} \tag{4.55}$$

$$dW_{k,i}^{m,h} = dZ_k^m v_i^{m,h} \tag{4.56}$$

Matrix form ($\mathbf{dW}$ *):*

$$\mathbf{dW}^{m,h} = \mathbf{dz}^m * (\mathbf{v}^{m,h})^T \tag{4.57}$$

4.3.6 Transformation Function

The final backpropagation stage for each layer is the backpropagation through the transformation function. The derivative of the PI with respect to the transformation function input, a, is given as follows.

$$\frac{\partial F}{\partial a^{m-1}} = \frac{\partial F}{\partial v^m} \times \frac{\partial v^m}{\partial a^{m-1}} \tag{4.58}$$

The term $\frac{\partial F}{\partial v^m}$ is dV, which was computed at the previous stage.

$$\mathbf{dV^m} \rightarrow \mathbf{dA^{m-1}} \tag{4.59}$$

The transformation function simply converts a vector to a matrix or a matrix to a vector (or makes no change), so the backpropagation step simply performs the reverse operation. If the forward transformation converts a vector to a matrix ($V = mat(a)$), then the backpropagation calculation will convert a matrix to a vector ($da = vec(dV)$), and vice-a-versa. If no conversion is required on the forward step, then no conversion is required for the backpropagation. The following headings refer to the forward calculation, as in previous sections.

Vector to Matrix transformation

Scalar form (dA):

$$dA^{m-1,h}_{ceil(i/S^{m-1}_r),S^{m-1}_r-mod(S^{m-1}_r-i,S^{m-1}_r)} = dV^{m,h}_i \tag{4.60}$$

Matrix form ($\mathbf{dA}$*):*

$$\mathbf{dA}^{m-1,h} = mat(\mathbf{dv}^{m,h}) \tag{4.61}$$

Here mat is used to denote the matrix transformation.

Matrix to Matrix transformation

Scalar form (dA):

$$dA^{m-1,h}_{i,j} = dV^{m,h}_{i,j} \tag{4.62}$$

Matrix form ($\mathbf{dA}$ *):*

$$\mathbf{dA}^{m-1,h} = \mathbf{dV}^{m,h} \tag{4.63}$$

Matrix to Vector conversion

Scalar form (dA):

$$dA^{m-1,h}_i = dV^{m,h}_{mod(i,S^m_r),1+fix(i/S^m_r)} \tag{4.64}$$

Vector form ($\mathbf{da}$ *):*

$$\mathbf{da}^{m-1,h} = vec(\mathbf{dV}^{m,h}) \tag{4.65}$$

Here vec performs the vector transformation, which stacks the rows of a matrix on top of each other.

Vector to Vector transformation

Scalar form (da):

$$dA^{m-1,h}_i = dV^{m,h}_i \tag{4.66}$$

Vector form ($\mathbf{da}$*):*

$$\mathbf{da}^{m-1,h} = \mathbf{dv}^{m,h} \tag{4.67}$$

4.3.7 Summary

The previous subsections have shown how the gradient of the PI with respect to the weights and biases of the network can be computed with a backpropagation process. This process begins at the last layer of the network (layer M) and continues backward to the first layer. The gradient described in these subsections has been for a partial PI, which involves only one input pattern. This is sometimes referred to as a stochastic gradient. The total gradient, also referred to as the batch gradient, can be found by summing the individual gradients over the entire training set, since the gradient of the sum is equal to the sum of the gradients.

4.4 Training Algorithms

The neural network training algorithms that we consider in this thesis are optimization algorithms that minimize the PI. All of the algorithms that we consider are based on the gradient. There are many such algorithms that could be considered, such as variations of steepest descent, conjugate gradient and quasi-Newton algorithms. FIRST is very modular, and any of these optimization algorithms could be easily added. For the preliminary version of FIRST, we have implemented three training algorithms. The first is steepest descent, because it is the most straightforward. The other two, Resilient Backpropagation and Scaled Conjugate Gradient, are among the most efficient algorithms for training neural networks, especially for pattern recognition problems. The following subsections describe these three algorithms.

4.4.1 Steepest Descent

Steepest descent is also called gradient descent. This is because the negative of the gradient (of the PI with respect to the weights and biases of the network) is the direction in which the PI decreases most rapidly. The steepest descent algorithm can be written as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \mathbf{g}_k \tag{4.68}$$

Where **w** is a vector containing all of the weights and biases of the network, k represents the iteration number, α_k is the learning rate and g_k is gradient. Although the steepest descent algorithm is simple, it often has slow convergence. This is because α must be kept small in order to guarantee the stability of the algorithm. The next two sections describe algorithms that use the gradient more efficiently.

4.4.2 Resilient Back-propagation

To overcome shortcomings of the gradient descent algorithm, an adaptive learning algorithm, Resilient Back-propagation (RPROP) was developed by Riedmiller and Braun [28]. In this technique, the size of the weight update is not dependent on the magnitude of the gradient. Only the sign of the gradient is used. The basic idea of RPROP is as follows. The size of the weight update depends on the previous update and the sign of the gradient. When the sign of the element of the gradient corresponding to a specific weight changes, that means the previous update may have been too large, and overshot the minimum point, so the update size is decreased by a factor of η^-, which is typically chosen to be 0.5. If the sign of the element of the gradient does not change, that means the update is going in the right direction, so the size is increased by a factor of η^+, in order to increase convergence speed. When the element of the gradient is zero, the update maintains the same value. In all cases, the sign of the update depends on the sign of the gradient element. [28]

The details of the RPROP algorithm follow. If

$$\frac{\partial E}{\partial w_{i,j}}(k-1) \times \frac{\partial E}{\partial w_{i,j}}(k) > 0 \tag{4.69}$$

then

$$\Delta_{i,j}(k) = \eta^+ * \Delta_{i,j}(k-1) \tag{4.70}$$

This value is compared with $\Delta_{max} = 50$, and the minimum is taken as the update value.

$$\Delta w_{i,j}(k) = -sign(\frac{\partial E}{\partial w_{i,j}}(k)) * \Delta_{i,j}(k) \tag{4.71}$$

$$w_{i,j}(k+1) = w_{i,j}(k) + \Delta w_{i,j}(k) \tag{4.72}$$

On the other hand, if

$$\frac{\partial E}{\partial w_{i,j}}(k-1) \times \frac{\partial E}{\partial w_{i,j}}(k) = 0 \tag{4.73}$$

then

$$\Delta w_{i,j}(k) = -sign(\frac{\partial E}{\partial w_{i,j}}(k)) * \Delta_{i,j}(k) \tag{4.74}$$

$$w_{i,j}(k+1) = w_{i,j}(k) + \Delta w_{i,j}(k) \tag{4.75}$$

Finally, if

$$\frac{\partial E}{\partial w_{i,j}}(k-1) \times \frac{\partial E}{\partial w_{i,j}}(k) < 0 \tag{4.76}$$

then

$$\Delta_{i,j}(k) = \eta^{-} * \Delta_{i,j}(k-1) \tag{4.77}$$

This value is compared with $\Delta_{min} = 1e^{-6}$, and the maximum is taken as the update value.

$$w_{i,j}(k+1) = w_{i,j}(k) - \Delta w_{i,j}(k) \tag{4.78}$$

The parameter η^{+} is used to increase convergence speed in flat regions, and it is typically chosen to be 1.2. Notice that all algorithm parameters are pre-decided. There is no need to change them for most problems. This is an important advantage of this technique. [28]

4.4.3 Scaled Conjugate Gradient

It is well known that optimization algorithms can be improved by using second order information (i.e., the Hessian matrix) in addition to the first order information contained in the gradient. The most direct way of doing this is Newton's method. However, the Hessian matrix requires a significant amount of computation. Conjugate gradient algorithms were introduced to utilize second order information, obtained through successive gradient computations, without requiring the Hessian computation. Instead of searching for the minimum along the negative gradient directions, the searches are made along conjugate directions. Most conjugate gradient algorithms require that a line search be carried out along the conjugate directions, which imposes a large computational burden. Moller introduced the Scaled Conjugate Gradient (SCG) algorithm [29] to eliminate the line search requirement.

The SCG weight update is similar to the steepest descent algorithm of Equation 4.68.

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k \tag{4.79}$$

where p_k is a conjugate direction, which is computed from successive gradients. For the SCG algorithm, the learning rate α_k is computed at each iteration, but it does not require a line search, so the computational burden is quite small. Like RPROP, SCG does not require fine-tuning of any adjustable parameters.

4.5 Other Training Issues

4.5.1 Convergence Plot

The convergence plot shows the PI versus the iteration number. It provides insights into the progression of the network training. For example, if the slope of the convergence plot is negative at the final iteration, this may indicate that further training might improve performance. Whereas a flat region at the end of training would indicate complete convergence. FIRST displays a convergence plot during the training process. If the data has been divided into training, validation and test sets, then all three convergence plots will be displayed. An example convergence plot is shown in Fig. 4.2.

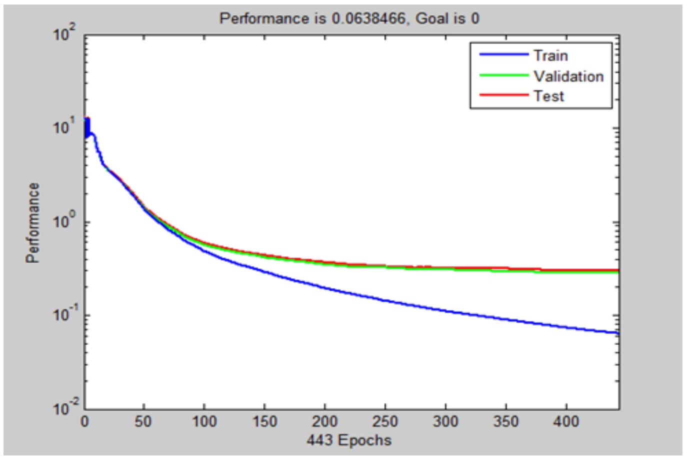


Figure 4.2: Example Convergence Plot

4.5.2 Overfitting and Early Stopping

Overfitting occurs when the network has many more parameters than points in the training data set. If care is not taken during training, the network will memorize the training data and will not generalize well. A technique that is used to improve generalization capacity is called early stopping. In this technique, the data is divided into two mutually exclusive sets: 1) training data and 2) validation data. The network is trained on the training data, but the PI for the validation data is monitored during training. If the validation PI goes up for several iterations, training is stopped, and the weights at the minimum of the validation PI are returned. In most cases, it is observed that, at the beginning of training, the PI for both training and validation sets decrease, but after some point the validation PI increases. At this point, it is assumed that network starts to overfit, or memorize the training data.

In addition to the training and validation data sets, a test set is also selected. The test set is not used as part of the training process, but the error of the trained network on the test data is a measure of future performance.

4.5.3 Network Validation

After the network has been trained, it is important to validate the network performance. FIRST provides two tools for network validation- the confusion matrix and the Receiver Operating Characteristic (ROC) curve. They are described below.

1) Confusion Matrix: The confusion matrix [30] is a matrix in which each column represents a predicted class and each row represents an actual class. The matrix gives us information about the quality of the classification done by the network. For example, see Fig. 4.3. This represents a network that was trained to classify the handwritten digits 0 and 1. The diagonal cells show the number of correctly classified patterns for each class, whereas the remaining cells show incorrectly classified patterns.

		Predicted Class	
		0	1
Actual Class	0	**151**	**13**
	1	**19**	**211**

Figure 4.3: Example Confusion Matrix

2) Receiver Operating Characteristic (ROC) curve: The ROC curve is a plot of the true positive rate (TPR) versus the false positive rate (FPR) as the decision threshold is varied. The TPR is the ratio of the number of correctly classified positive patterns in the data set to the total number of positive patterns. The FPR is the ratio of the number of incorrectly classified negative patterns to the total number of negative patterns. The decision threshold is the value of the final layer neuron output beyond which we say the classification is positive. For example, if we have a network that classifies 0's and 1's, and we use a hyperbolic tangent sigmoid in the last layer, an output of 1 could represent the digit 1, and an output of -1 could represent the digit 0. In that case, we would normally say that the predicted class was 1 for any network output greater than 0, but this decision threshold could be set to any

value between -1 and 1. As we adjust this threshold over that range, while computing the TPR and FPR, we can create the ROC curve. An example ROC curve is shown in Fig. 4.4. The quality of the classification is sometimes measured by the area under the ROC curve. The larger this area is, the better the performance of the classifier. The worst case is the $45°$ line in Fig. 4.4 , which represents a random guess.

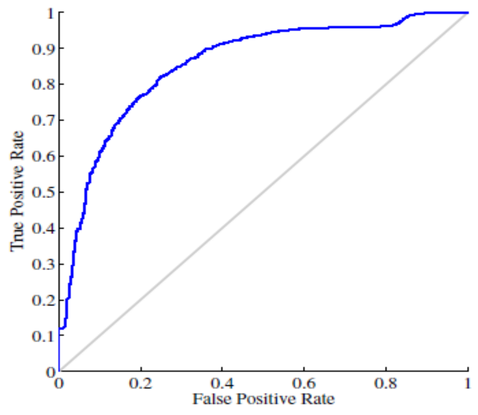


Figure 4.4: Example ROC curve, taken from [5]

4.6 Summary

In this chapter, we have discussed two important topics: gradient calculations and training algorithms. The training algorithms are optimization procedures that are designed to minimize a performance index, like mean square error, by adjusting the weights and biases of the network. All of the training algorithms presented here use the gradient of the performance index with respect to the network weights and biases to determine their search directions. The steepest descent algorithm moves in the negative gradient direction. The resilient backpropagation algorithm uses changes in signs of the elements of the gradient to determine the search direction, and has been found to be much more efficient than steepest descent. The scaled conjugate gradient algorithm uses successive gradients to deduce second order information in order to adjust the search directions. This algorithm is also much more efficient than steepest descent.

The gradient computation procedures described here are very general and modular. They can be applied to a network with an arbitrary number of layers and with arbitrary weight functions, net input functions and transfer functions in each layer. The gradient is computed using the chain rule, starting from the network output, and proceeding backward through the network.

In next chapter, we will focus on how FIRST can be used to implement deep convolution neural networks for practical pattern recognition problems.

CHAPTER 5

FIRST Software Design and Usage

5.1 Introduction

This chapter focuses on software implementation and how to use the software. It also explains how the concepts and equations introduced in Chapter 3 and 4 are implemented in the FIRST software. This chapter serves as a guide for the FIRST user.

The FIRST software contains many modular functions, such as transfer functions, weight functions, transform functions, net input functions, etc. All of these functions are implemented in Matlab. FIRST can run on basic Matlab installations, as the modular functions do not use any Matlab toolboxes. The modular functions, network structure, functions for calculating gradient, functions for calculating performance functions, etc., are constructed in order to arrange the code in a systematic format and to make it general so that users can easily build new custom networks. In the following sections, we explain how the functions in the FIRST software are designed and how FIRST can be used to build networks.

5.2 Network Object

Network objects are used in FIRST to define network architectures and initialize network parameters. The network architecture is defined by the number of layers, number of feature maps used for each layer, initialization of weights and biases of each layer and their dimensions, weight functions of each layer (e.g., convolution, subsampling, etc.), transfer functions of each layer, etc., so all these are included in the network object. The network architecture can vary according to the application so users can modify these network objects to build different network with different numbers of layers or different weight functions, etc.

A detailed explanation for each component of the network object follows. The total number of layers is denoted `net.numLayers`. The input is considered a layer, to make it easier to define the connection matrix between the input and the first regular layer. For each layer, to define the total number of feature maps, `net.featureMap(`m`)` is used, where m is the layer number. The structure used for a feature map is a simple array. A connection matrix is defined for each layer (see Section 3.4). The dimensions of the connection matrix depend on the number of feature maps in the previous and present layers, so the size of the connection matrix is different for each layer. The structure used to contain the connection matrices is a cell array and is denoted `net.ConnectionMatrix`$\{m\}$ (C^m), where m is the layer number, and the curly parenthesis is used to define the cell.

The row and column reduction factors for the subsampling layer are defined for the layer number m as `net.subfactor_r(`m`)` (r^m), `net.subfactor_c(`m`)` (c^m).

The weights and biases are defined as `net.w` and `net.b`. A cell structure is used for both of these parameters, because their sizes in each layer might be different. The weights and biases are randomly initialized in the network creation function. For the weight matrices, a three dimensional cell structure is used, because each layer has multiple feature maps. The weight is defined as `net.w`$\{i, j, k\}$ ($W^{i,j,k}$) where i is the layer number, j is the feature map of that layer and k is the feature map of the previous layer. For a full connection layer, there is one feature map, so we can define it as `net.w`$\{i, 1, k\}$. There is one bias for each feature map, so it is defined by `net.b`$\{i, j\}$ ($b^{i,j}$) where i is the layer number, j is the feature map of that layer.

The weight function for each layer is defined by using a cell structure and is denoted `net.wf`$\{m\}$`='weight function name'`. In similar ways, we can assign the net input function using `net.nf`$\{m\}$`='net input function name'`, and the transfer function using `net.tf`$\{m\}$`='transfer function name'`.

To select the performance function, use `net.performFcn ='performance function name'`. Training algorithms that are based on batch training need the gradients and the weights and biases to be stored in vector form in order to conveniently update the weights and biases. To convert sets of matrices into one large vector, we need to assign indices. These indices are stored in `net.wind`$\{i, j, k\}$ and `net.bind`$\{i, j\}$, where i is the layer number, j is the feature map of that layer and k is the feature map of the previous layer. FIRST contains three batch training algorithms: `trainrp` (resilient backpropagation), `trainscg` (scaled conjugate gradient) and `trainsd` (steepest descent backpropagation). To select the training algorithm, use `net.trainFcn ='training function name'`.

Network creation functions define the above mentioned objects. FIRST includes network creation functions for Lecun's networks: LeNet-1, LeNet-5 and LeNet-7. Also, this software can be easily modified to accommodate custom networks.

5.2.1 Existing FIRST Network Creation Functions

FIRST includes a LeNet-1 creation function, which is called `newcnn`. This function does not contain any argument list, since the LeNet-1 architecture is completely specified. It returns the network object, with all necessary parameters defined. The syntax used to call this function is shown below:

```
net=newcnn
```

FIRST also includes a LeNet-5 creation function, which is called `newcnn5le`. The `newcnn5le` command also returns a network object. The syntax used to call this function is shown below:

```
net=newcnn5le
```

FIRST also includes a `newcnn5`, which is similar to LeNet-5, except for the last layer. It also contains two full connection layers, but the error is calculated as the difference between the output of the last layer and the target, which is formed by using the 1 of k encoding method. The syntax used to call this function is shown below:

```
net=newcnn5
```

The FIRST implementation of LeNet-7 is called `newcnn7`. The syntax used to call this function is shown below:

```
net=newcnn7
```

In all of the above network creation functions, network objects are defined as follows:

`net.featureMap`(m)	(number of feature maps)
`net.numLayers`	(number of layers)
`net.trainFcn=''`	(training algorithm)
`net.performFcn=''`	(performance function)
`net.ConnectionMatrix`$\{m\}$	(connection between feature maps of adjacent layers)
`net.w`$\{i,j,k\}$	(weight for layer i between feature map j of layer i and feature map of previous layer k)
`net.b`$\{i,j\}$	(bias for feature map j of layer i)
`net.subfactor_r`(m)	(subsampling factor for row wise reduction)
`net.subfactor_c`(m)	(subsampling factor for column wise reduction)
`net.wf`$\{m\}$`=''`	(weight function for layer m)
`net.nf`$\{m\}$`=''`	(net input function for layer m)
`net.tf`$\{m\}$`=''`	(transfer function for layer m)
`net.vf`$\{m\}$`=''`	(transform function for layer m)
`net.wind`$\{i,j,k\}$	(weight indices between feature map j of layer i and feature map k of the previous layer)
`net.bind`$\{i,j\}$	(bias indices for feature map j of layer i)

5.2.2 Adding Custom Network Creation Functions and Network Modules

In addition to the network creation functions described in the previous section, users can create custom networks. Users can change the number of layers or dimensions of weights and biases, number of feature maps, training algorithms or connection matrix. These changes can be easily done, and a new network can be formed. Also, any weight function or transfer function can be used for any layer.

In addition to custom network architectures, users can also create custom weight functions, custom net input functions or custom transfer functions,. The process of customization will be described in more detail in Section 5.7.

5.3 Simulation Flow

Simulation flow is the flow of computations when calculating the network output. In the simulation flow, first the transform function output v is calculated from the network input p, then the weight function output z is calculated, which is given to the net input function, which produces n. Then the transfer function is applied to n to produce a. This simulation is performed in the `calcgx` function in the FIRST software. This flow is followed for each layer and for every pattern. The flowchart is given in Figure 5.1.

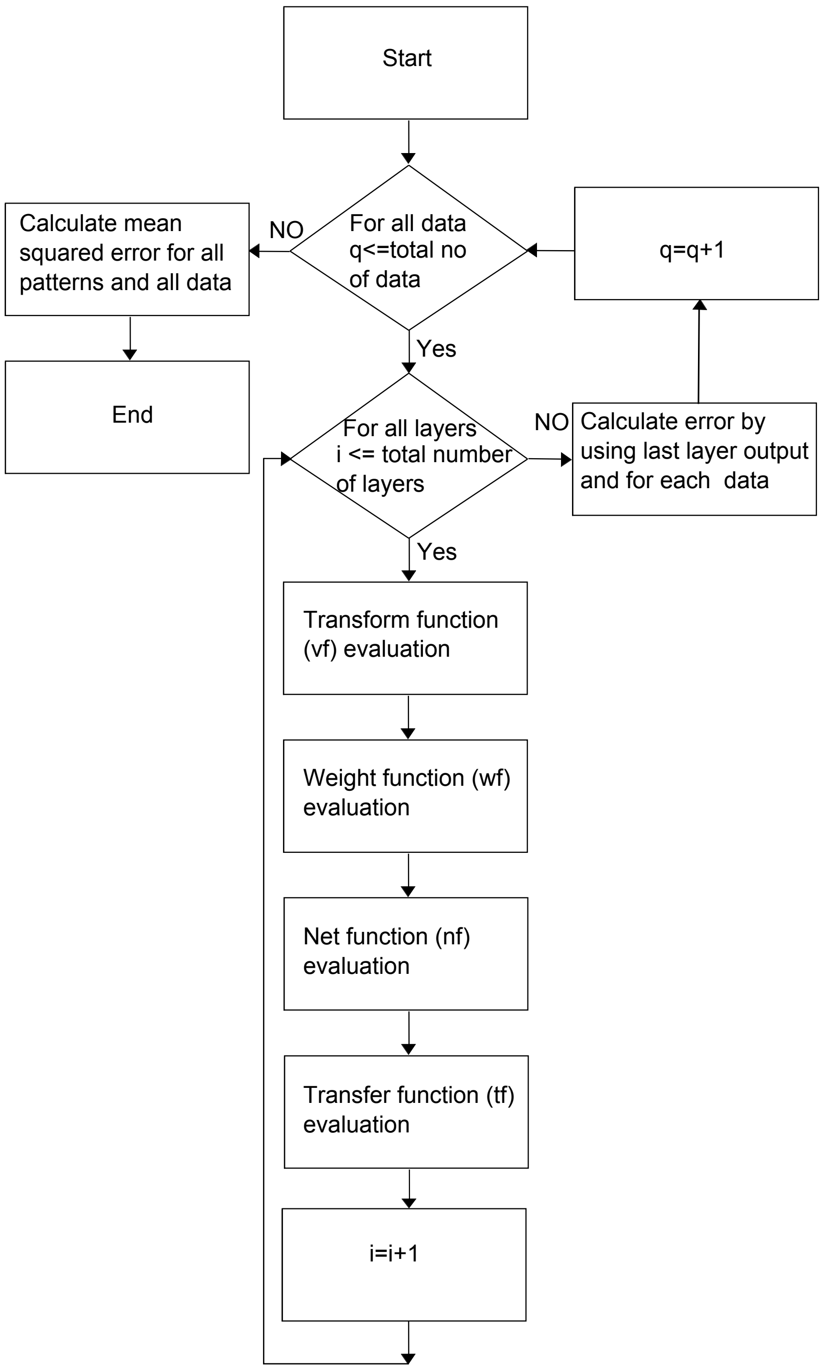

Start
For all data q<=total no of data
NO
Calculate mean squared error for all patterns and all data
q=q+1
End
Yes
For all layers i <= total number of layers
NO
Calculate error by using last layer output and for each data
Yes
Transform function (vf) evaluation
Weight function (wf) evaluation
Net function (nf) evaluation
Transfer function (tf) evaluation
i=i+1

Figure 5.1: Flowchart for simulation

In the FIRST software, in order to perform the forward calculation and to calculate the performance, the `calcgx` function is used. In this function, the error is calculated for all patterns and is summed together to find total performance. The syntax for this function is given below:

```
[perf,gX]=calcgx(net,p,t,perg)
```

Here `net` is a network, `p` is the network input, `t` is the network target, `perg` is a flag to denote whether only a PI calculation is done or both gradient and PI calculations are done and `perf` is the performance. When `perg` is zero, it returns only the performance `perf`. In this function, first the transform function is invoked and the output is stored in v. Then v is given as input to the weight function. The output of the weight function z is given to the net input function to calculate n, which is given to the transfer function to calculate a. This is repeated for all layers, and then the output of the last layer is compared with the target value to calculate the error. This is repeated for all input data, and then the total PI is calculated.

To evaluate transform functions, the following syntax is used:

```
v=feval(net.vf{i},a,ba,i,net)
```

Here `a` is the transfer function output, `ba` is a flag to indicate whether it is a backpropagation calculation or a forward calculation (here `ba`=0), `i` is the layer number, `net` is a network and `v` is the output of the transform function (a cell structure).

To evaluate the weight functions, the following syntax is used:

```
z=feval(net.wf{i},w,v,ba,i,net)
```

Here `w` is the weight, `v` is the transform function output, `ba` is a flag to indicate whether it is a backpropagation calculation or a forward calculation (here `ba`=0), `i` is the layer number, `net` is a network and `z` is the output of the weight function (a cell structure containing the results for all FMs).

To evaluate net input functions, the following syntax is used:

```
n=feval(net.nf{i},b,z,ba,i,net)
```

Here `b` is the bias, `z` is the weight function output, `ba` is a flag to indicate whether it is a backpropagation calculation or a forward calculation (here `ba`=0), `i` is the layer number, `net` is a network and `n` is the output of the net input functions (a cell structure containing the results for all FMs).

To evaluate transfer functions, the following syntax is used:

`a=feval(net.tf{i},n,ba,i,net)`

Here `n` is the net input, `ba` is a flag to indicate whether it is a backpropagation calculation or a forward calculation (here `ba`=0), `i` is the layer number, `net` is a network and `a` is the output of the transfer function (a cell structure).

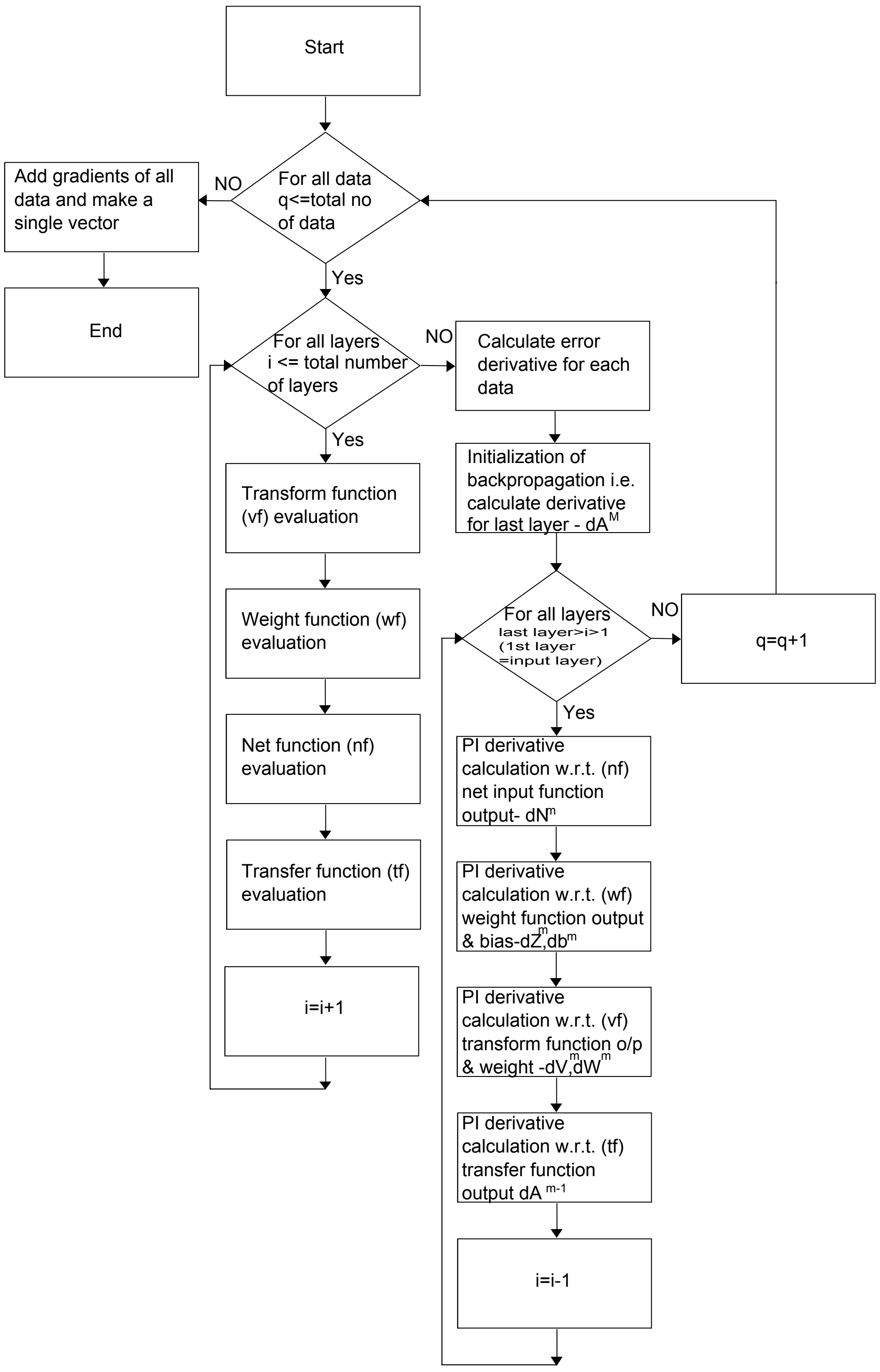
The final step in the network simulation is the calculation of the PI. This is done after the last layer output has been computed. The syntax of the PI calculation is

`perf=feval(net.performFcn,a,t,ba)`

where `net.performFcn` is the name of the PI function, `a` is the output of the last layer of the network, `t` is the target output and `ba` is a flag to indicate whether a forward calculation or a backpropagation calculation is needed (here `ba`=0).

5.4 Gradient Flow

Gradient flow is the flow of computations during the gradient calculation. The gradient is calculated in a backward direction in the `calcgx` function. The flow begins by calculating the gradient of the PI with respect to the output of the last layer dA. This is used to calculate the gradient of the PI with respect to the net input dN, and then this is used to calculate the gradient of the PI with respect to the weight function output dZ. This gradient is used to calculate the gradient of the PI with respect to the transform function dV. This process is illustrated in Figure 5.2. The remainder of this section will step through the details of this backpropagation process. In addition to Figure 5.2, it may be helpful to review Figure 4.1 before continuing in this section.



Start
For all data q<=total no of data
NO
Add gradients of all data and make a single vector
End
Yes
For all layers i <= total number of layers
NO
Calculate error derivative for each data
Initialization of backpropagation i.e. calculate derivative for last layer - dA^M
Yes
Transform function (vf) evaluation
Weight function (wf) evaluation
Net function (nf) evaluation
Transfer function (tf) evaluation
i=i+1
For all layers last layer>i>1 (1st layer =input layer)
NO
q=q+1
Yes
PI derivative calculation w.r.t. (nf) net input function output- dN^m
PI derivative calculation w.r.t. (wf) weight function output & bias-dZ^m,db^m
PI derivative calculation w.r.t. (vf) transform function o/p & weight -dV^m,dW^m
PI derivative calculation w.r.t. (tf) transfer function output dA^m-1
i=i-1

Figure 5.2: Flowchart for gradient calculation

In the FIRST software, `calcgx` is used to perform the gradient calculation. As this is batch training, the gradient is computed for each pattern, and then the individual gradients are summed. The gradient is calculated in vector form. The syntax for this function is

```
[perf,gX]=calcgx(net,p,t,perg)
```

Here `net` is a network, `p` is the training input, t is the target output, `perg` is a flag to denote whether just the PI calculation is done or both the gradient and the PI calculations are done, `perf` is the performance, and `gX` is the gradient. When `perg` is one it returns gradient `gX` as well as performance `perf`.

The gradient calculation begins with the gradient of the PI with respect to the layer output. The syntax is

```
dA{M,1}=feval(net.performFcn,a,t,ba)
```

where `a` is the final layer output, `t` is the target, `ba` is the backpropagation flag (here `ba`=1). `dA` is calculated at the output layer M.

The next step is to compute the gradient of the PI with respect to the net input n. The syntax is

```
dN(i,:)=feval(net.tf{i},a,ba,i,net,dA(i,:))
```

For this backpropagation calculation, `ba` is 1.

We now compute the gradient with respect to the weight function output, z, and also with respect to the bias. The syntax is

```
[db(i,:),dZ(i,:)]=feval(net.nf{i},dN(i,:),z,ba,i,net)
```

For this backpropagation calculation, `ba` is 1 and `z` is 0.

Then the gradient with respect to the transform function output, v, and with respect to the weights is calculated.

```
[dw(i,:,:),dV(i,:)]=feval(net.wf{i},dZ(i,:),a(i-1,:),ba,i,net)
```

For this backpropagation calculation, `ba` is 1.

To calculate the gradient of the PI with respect to the previous layer output (completing the backpropagation through one layer) the syntax is

```
dA(i-1,:)=feval(net.vf{i},dV(i,:),ba,i,net,a{i-1})
```

For this backpropagation calculation, `ba` is 1, and $a\{i-1\}$ is the network output.

5.5 Training Flow

The general flow of the training functions is shown in 5.3. The algorithms begin by setting the training parameters. Next, the training, testing and validation performance is computed. Then the gradient is calculated for the training data and it is used to update weights and biases. And if the maximum number of epochs is reached, the maximum amount of time is exceeded, the performance gradient falls below min_grad or validation performance has increased more than max_fail times, then training stops. To update the weights and biases, different optimization techniques are used. A good selection of optimization techniques are included in the FIRST software. Detailed descriptions of the implementations of these algorithms are included in next subsection. It is also possible for the user to add custom training algorithms by using existing training algorithms as templates. This is also explained in a later subsection.

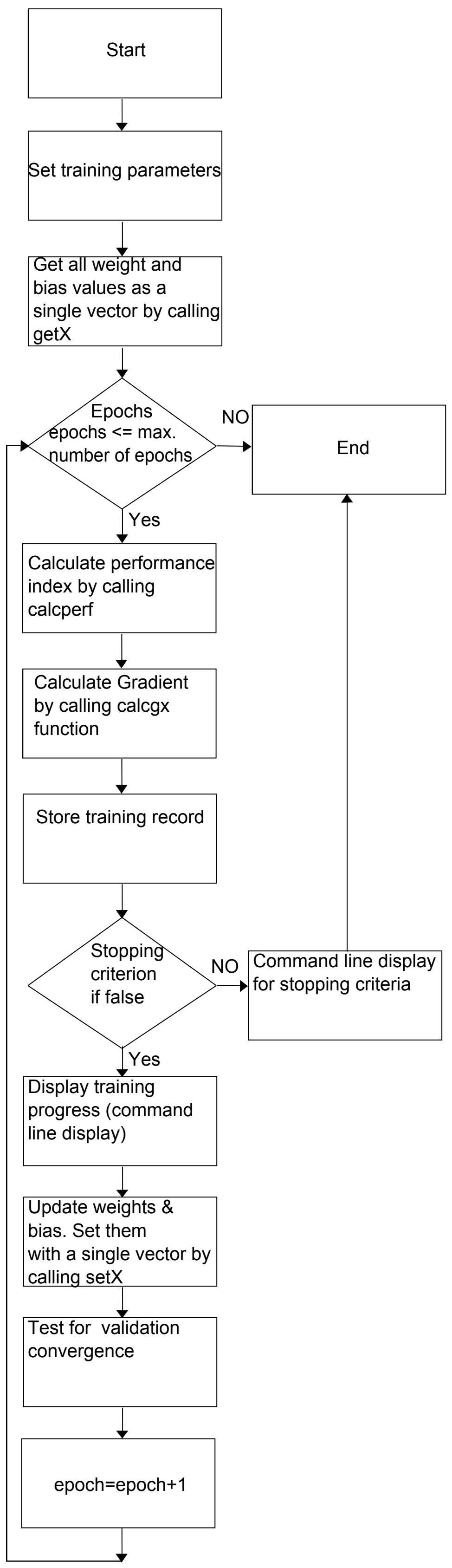

Start
Set training parameters
Get all weight and bias values as a single vector by calling getX
Epochs epochs <= max. number of epochs
NO
End
Yes
Calculate performance index by calling calcperf
Calculate Gradient by calling calcgx function
Store training record
Stopping criterion if false
NO
Command line display for stopping criteria
Yes
Display training progress (command line display)
Update weights & bias. Set them with a single vector by calling setX
Test for validation convergence
epoch=epoch+1

Figure 5.3: Flowchart for training

5.5.1 Existing FIRST Training Algorithms

The FIRST software includes three batch training algorithms: Resilient Backpropagation, Scaled Conjugate Gradient and Steepest Descent.

For resilient backpropagation (RPROP), the `trainrp` function is used (see Section 4.5.2 for a description of the RPROP algorithm). The syntax of `trainrp` is shown below.

```
[net,tr]=imtrain(net,p,t,p1,t1,p2,t2)
```

The inputs to the function are the initialized network, `net`, the training inputs and targets `p` and `t`, the validation inputs and targets, `p2` and `t2`, and the testing inputs and targets, `p1` and `t1`. The training record `tr` contains information about the progress of training and `net` is the trained network. The parameters that are used in this algorithm are shown in Table 5.1, with their default values:

Table 5.1: Default Parameters for `trainrp`

Parameter	Description	Default value
`net.trainParam.epochs`	Maximum number of epochs to train	1000
`net.trainParam.show`	Epochs between displays	20
`net.trainParam.goal`	Performance goal	0
`net.trainParam.time`	Maximum time to train in seconds	inf
`net.trainParam.min_grad`	Minimum performance gradient	1e-10
`net.trainParam.max_fail`	Maximum validation failures	20
`net.trainParam.delt_inc`	Increment to weight change	1.2
`net.trainParam.delt_dec`	Decrement to weight change	0.5
`net.trainParam.delta0`	Initial weight change	0.07
`net.trainParam.deltamax`	Maximum weight change	50.0

The scaled conjugate gradient algorithm is implemented in the FIRST software using `trainscg` (see Section 4.4.3 for a description of the scaled conjugate gradient algorithm).

The syntax used for `trainscg` is

```
[net,tr]=imtrain(net,p,t,p1,t1,p2,t2)
```

The parameters that are used by the SCG algorithm are given in Table 5.2, along with their default values.

Table 5.2: Default Parameters for `trainscg`

Parameter	Description	Default value
`net.trainParam.epochs`	Maximum number of epochs to train	1000
`net.trainParam.show`	Epochs between displays	20
`net.trainParam.goal`	Performance goal	0
`net.trainParam.time`	Maximum time to train in seconds	inf
`net.trainParam.min_grad`	Minimum performance gradient	1e-10
`net.trainParam.max_fail`	Maximum validation failures	20
`net.trainParam.sigma`	Determine change in weight for second derivative approximation	5.0e-5
`net.trainParam.lambda`	Parameter for regulating the indefiniteness of the Hessian	5.0e-7

The `trainsd` function is used to implement the steepest descent batch training algorithm in the FIRST software. The following syntax is used

```
[net,tr]=imtrain(net,p,t,p1,t1,p2,t2)
```

Seven parameters are associated with this function. The parameters, along with their default values, are given Table 5.3.

Table 5.3: Default Parameters for `trainsd`

Parameter	Description	Default value
`net.trainParam.epochs`	Maximum number of epochs to train	1000
`net.trainParam.show`	Epochs between displays	20
`net.trainParam.goal`	Performance goal	0
`net.trainParam.time`	Maximum time to train in seconds	inf
`net.trainParam.min_grad`	Minimum performance gradient	1e-10
`net.trainParam.max_fail`	Maximum validation failures	20
`net.trainParam.lr`	Learning rate	0.01

5.5.2 Adding Custom Training Algorithms

The easiest way to create a custom training function is to use one of the three existing training functions as a template. For algorithms that use the gradient, very few changes would need to be made. The calculation of the gradient would remain the same. The setting of the default parameters and the calculation of the weight update from the gradient would be the main parts of the code that would need to change.

5.6 Sample Training Sessions

In order to illustrate how FIRST is used, we will demonstrate the software on two different problems. One problem involves recognizing 2D images of handwritten digits. The second problem involves recognizing speech phonemes from 1D frequency domain data.

5.6.1 Sample Training Session for Image Data

The image data are handwritten digits. The data are present in the file usps_resampled.mat [31]. The data are divided into training and test sets, containing 4649 cases each. The mat file contains 4 variables:

test_labels : 10x4649

test_patterns : 256x4649

train_labels : 10x4649

train_patterns : 256x4649

The pattern variables contain a raster scan of the 16 by 16 grey level pixel intensities, which have been scaled to the range [-1; 1]. The label variables contain a one-of-k encoding, with values -1 and +1 of the classification; there is one +1 per column, which indicates the digit. The USPS digits data were gathered at the Center of Excellence in Document Analysis and Recognition (CEDAR) at SUNY Buffalo, as part of a project sponsored by the US Postal Service. The data set is described in [32]. In order to reduce boundary effects due to the convolution operation, background pixels are added around the boundary of each image. Also, to avoid saturating the transfer functions, the target values are multiplied by 0.76. This loading and preprocessing of data is done in the `datag` function. Also, data are divided into training, testing and validation sets. For this example, 60 % of the data is used for training, 20% is used for testing, and the remaining 20% is used for validation (to stop training before overfitting occurs). This is also done in the `datag` function.

The digit recognition example uses the LeNet-5 network and also a smaller modified LeNet-5 structure. The steps followed during execution of this example follow.

1) Load the data, which consists of input and target values. Also, the loaded data are preprocessed by adding extra pixels around the boundary of each image and the data are randomly divided into training, testing and validation sets.

```
[p,t,p1,t1,p2,t2]=datag5
```

This loaded data gives `p` and `t` as training inputs and targets, `p1` and `t1` as testing inputs and targets, and `p2` and `t2` as validation inputs and targets.

2) Create the network. We use `newcnn5` to create a LeNet-5 network.

```
net=newcnn5
```

3) Train the network. The network uses the resilient backpropagation algorithm for training. Here `net.trainFcn='trainrp'` is already defined in `newcnn5`. So to train the network, enter the following command

```
[net1,tr]=imtrain(net,p,t,p1,t1,p2,t2)
```

During training, the following results start displaying in the command window. This window displays training progress: training algorithm name, epoch value, performance value for that epoch and calculated gradient value.

```
TRAINRP,Epoch 0/10000,performance 2.79359/0,
Gradient 0.000184095/1e-010
TRAINRP, Epoch 20/10000, performance 0.211999/0,
Gradient 0.452726/1e-010
TRAINRP, Epoch 40/10000, performance 0.191529/0,
Gradient 0.24936/1e-010
TRAINRP, Epoch 60/10000, performance 0.152278/0,
Gradient 0.329577/1e-010
TRAINRP, Epoch 80/10000, performance 0.117953/0,
Gradient 0.120948/1e-010
................................................
........................
TRAINRP, Epoch 300/10000, performance 0.010985/0,
Gradient 0.0528472/1e-010
TRAINRP, Epoch 318/10000, performance 0.0105404/0,
Gradient 0.017447/1e-010
TRAINRP, Validation stop.
Elapsed time is 619861.349836 seconds.
```

The training stopped when the validation error increased for twenty iterations, which occurred at iteration 318. During training, it also shows a plot of the training errors, validation errors, and test errors as, shown in the Figure 5.4.

In this example, the result is reasonable because of the following considerations: The final performance value for training is small. The test set error and the validation set error have similar characteristics. No significant overfitting has occurred by iteration 298 (where the best validation performance occurs).

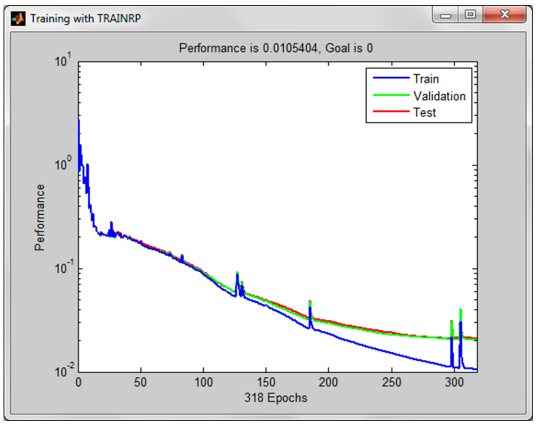


Figure 5.4: Performance Plot for LeNet-5

4) If you want to calculate the confusion matrix and the accuracy for the training results, as well as store images of the convolution kernels in tif format, enter the following command.

```
[confusionmatrix_train,accuracy_train]=ker(net1,p,t)
confusionmatrix_train =
```

941	0	1	0	1	0	0	0	0	0
0	757	1	0	2	0	0	0	0	0
1	0	537	0	1	0	0	0	2	0
2	0	0	455	0	5	0	1	2	0
0	1	3	0	480	0	1	0	0	4
1	0	0	4	0	391	0	0	0	1
1	0	0	0	1	1	483	0	0	0
0	0	1	0	2	0	0	480	1	6
1	0	0	0	0	2	0	0	404	0
0	0	0	1	0	0	0	2	0	483

```
accuracy_train =

97.0061
```

For all convolution kernels, images are formed with tif extensions and stored in the current folder. A few kernels are shown in Figure 5.5. These tif images are saved with name $kernel - x - y - z$, where x is the layer number, y is the FM number of that layer, and z is the FM number of the previous layer. By looking at the images, one can understand the working of the network. Using the convolution kernels, features are extracted from images, and then the dimensions of the features are reduced at the subsampling layer to reduce resolution.

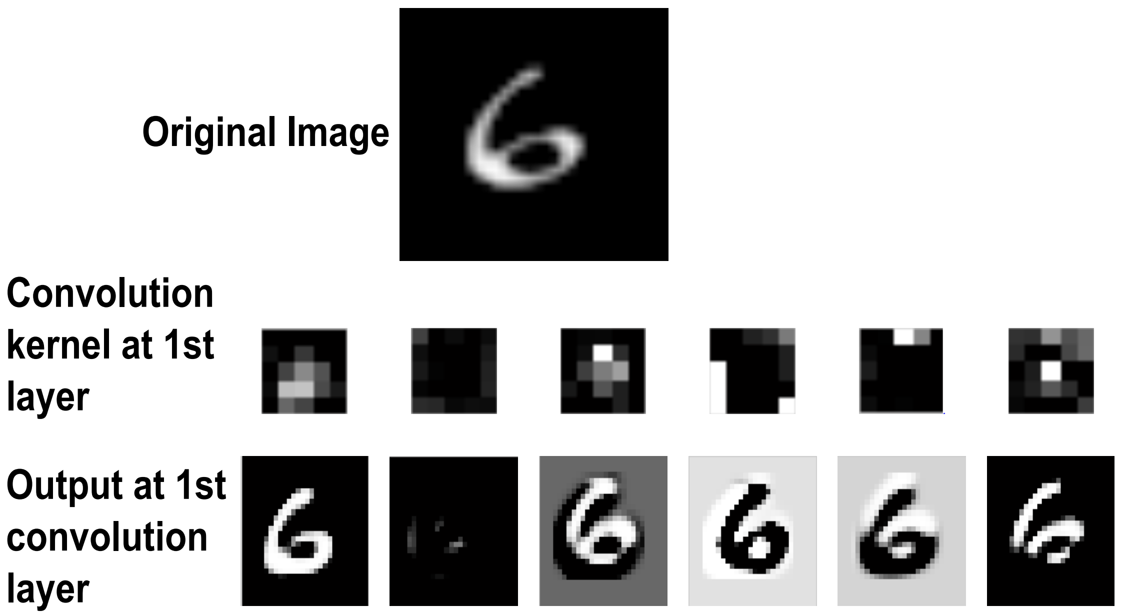

Original Image
Convolution kernel at 1st layer
Output at 1st convolution layer

Figure 5.5: Convolution kernel for LeNet-5

5) To calculate confusion matrix and accuracy for testing, the following command is used.

```
[confusionmatrix_test,accuracy_test]=testcnn(p1,t1,net1)
confusionmatrix_test =
```

270	0	0	0	1	1	4	0	2	0
0	231	0	0	1	0	0	0	0	0
2	0	174	2	1	0	0	0	3	1
0	1	4	156	0	4	0	0	1	0
1	2	0	0	154	0	0	0	0	3
3	0	0	4	1	140	3	0	1	0
1	0	1	0	2	0	172	0	1	0
0	0	0	0	1	0	0	143	4	0
0	1	1	1	0	1	0	0	141	0
0	0	0	0	1	0	0	1	1	154

```
accuracy_test =
93.2796
```

If even more accurate results were required, you could try any of these approaches: Reset the initial network weights and biases to new values by calling the network creation function again. Increase the number of layers or number of feature maps. Increase the number of training data. Try a different training algorithm, such as `trainscg` or `trainsd`.

5.6.2 Sample Training Session for Speech Data

The speech data were extracted from the TIMIT database (TIMIT Acoustic-Phonetic Continuous Speech Corpus, NTIS, US Dept of Commerce) which is a widely used resource for research in speech recognition [33]. The data set is described in [34]. A data set was composed of five phonemes such as “sh” as in “she”, “dcl” as in “dark”, “iy” as the vowel in “she”, “aa” as the vowel in “dark”, and “ao” as the first vowel in “water”. From continuous speech of 50 male speakers, 4509 speech frames of 32 msec duration were selected, approximately 2 examples of each phoneme from each speaker. Each speech frame is represented by 512 samples at a 16kHz sampling rate, and each frame represents one of the above five phonemes.

From each speech frame, a log-periodogram is calculated, because it is a suitable form for speech recognition. Thus the data consist of 4509 log-periodograms of length 256, with known class (phoneme) memberships. The data contain 256 columns labelled "x.1" - "x.256" and a column labelled "speaker" identifying the different speakers. To avoid saturation of the transfer functions, the target values are multiplied by 0.76. This loading and preprocessing of data is done in the `datas` function. The data are divided into training, testing and validation sets. For this experiment, 60 % of the data is used for training, 20% is used for testing, and the remaining 20% is used for validation. This is also done in the `datas` function.

For the speech data, LeNet-1 is used. The following steps demonstrate how to train a network from the command line,

1) Load the data. Also, the data are randomly divided into training, testing and validation data sets.

```
[p,t,p1,t1,p2,t2]=datas
```

This loaded data gives `p` and `t` as training inputs and targets, `p1` and `t1` as testing inputs and targets, `p2` and `t2` as validation inputs and targets.

2) Create the network. We use `newcnns` to create a LeNet-1 network.

```
net=newcnns
```

3) Train the network. The network uses the resilient backpropagation algorithm for training. Here `net.trainFcn='trainrp'` is already defined in `newcnns`. So to train the network, enter the following command

```
[net1,tr]=imtrain(net,p,t,p1,t1,p2,t2)
```

During training, the following results display in the command window. This window displays training progress: training algorithm name, epoch value, performance value for that epoch and calculated gradient value.

```
TRAINRP, Epoch 0/10000, performance 0.391371/0,
Gradient 1.65438/1e-012
TRAINRP, Epoch 20/10000, performance 0.183133/0,
Gradient 0.327583/1e-012
TRAINRP, Epoch 40/10000, performance 0.179088/0,
Gradient 0.445899/1e-012
TRAINRP, Epoch 60/10000, performance 0.170226/0,
Gradient 0.246669/1e-012
TRAINRP, Epoch 80/10000, performance 0.130636/0,
Gradient 0.715547/1e-012
..................................................
.........................
TRAINRP, Epoch 760/10000, performance 0.0257026/0,
Gradient 0.044878/1e-012
TRAINRP, Epoch 780/10000, performance 0.0255818/0,
Gradient 0.0767968/1e-012
TRAINRP, Epoch 794/10000, performance 0.0254977/0,
Gradient 0.0497147/1e-012
TRAINRP, Validation stop.
Elapsed time is 115923.296911 seconds.
```

This training stopped when the validation error increased for twenty iterations, which occurred at iteration 794. During training, it also shows a plot of the training errors, validation errors, and test errors, as shown in the Figure5.6.

In this example, the result is reasonable because of the following considerations: The final performance value for training is small. The test set error and the validation set error have similar characteristics. No significant overfitting has occurred by iteration 773 (where the best validation performance occurs).

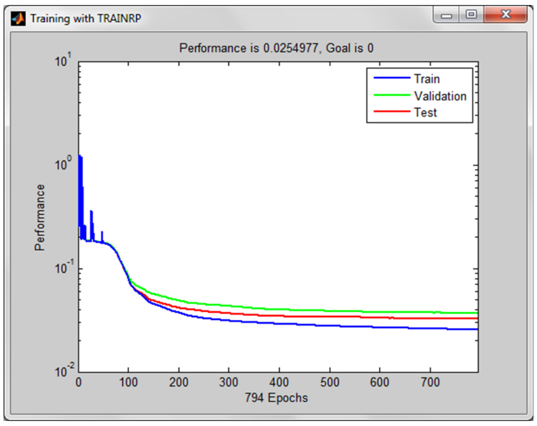


Figure 5.6: Performance Plot for LeNet-1

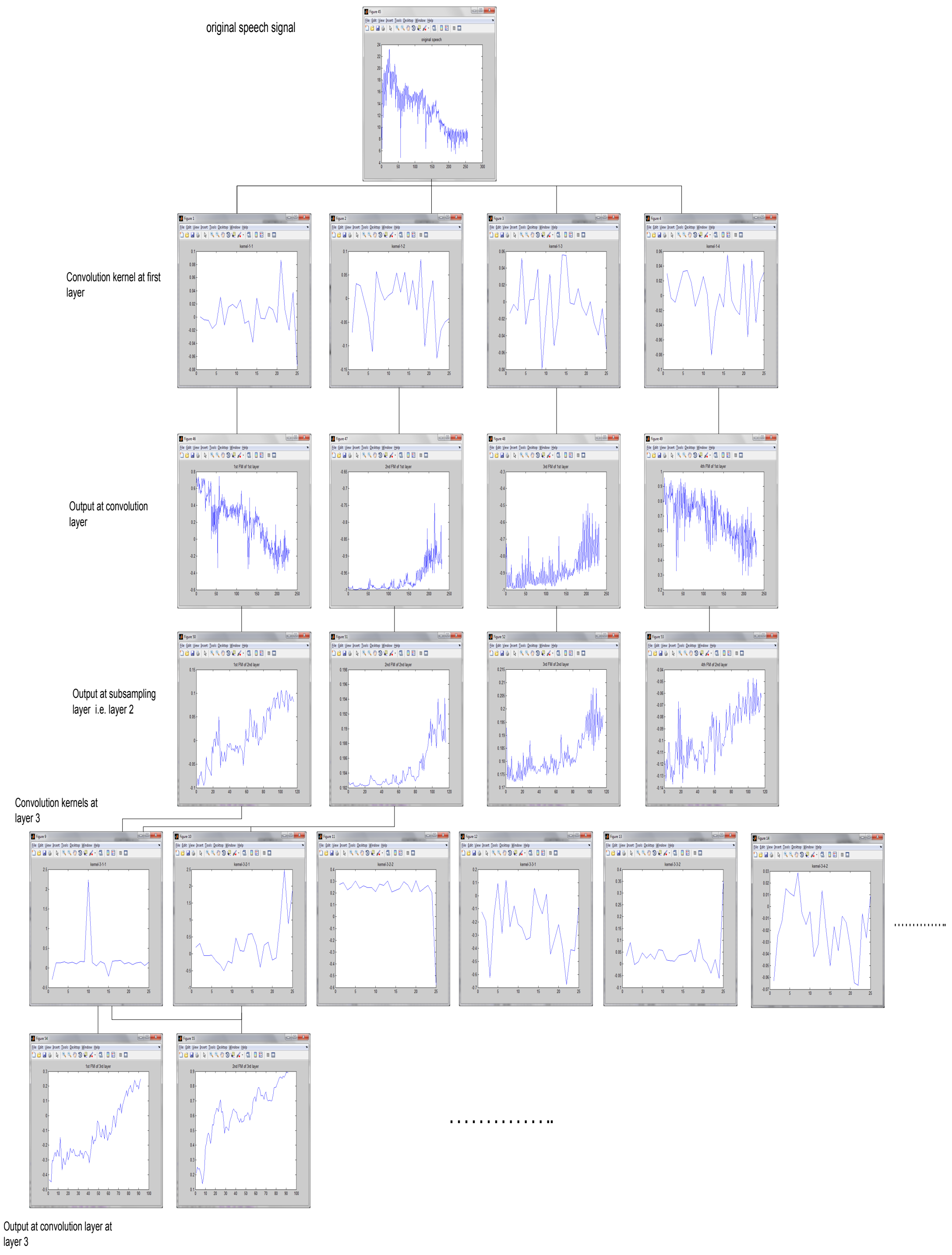
4) If you want to calculate the confusion matrix and the accuracy for the training results, as well as to plot convolution kernels, enter the following command.

```
[confusionmatrix_train,accuracy_train]=ker(net1,p,t)
confusionmatrix_train =
```

301	79	0	0	0
37	560	0	0	0
0	0	436	2	3
0	0	0	700	1
0	0	0	0	517

```
accuracy_train =
92.9390
```

Here the convolution kernels are one dimensional data (rather than images), therefore they are plotted. A few kernels are shown in Figure 5.7. The title of each figure takes the form $kernel - x - y - z$, where x is the layer number, y is the FM number of that layer and y is the FM number of the previous layer. By looking at these plots, one can understand the working of the network. Using convolution kernels, features are extracted, and then the dimensions of the features are reduced at the subsampling layer to reduce resolution.



original speech signal
Convolution kernel at first layer
Output at convolution layer
Output at subsampling layer i.e. layer 2
Convolution kernels at layer 3
Output at convolution layer at layer 3

Figure 5.7: Convolution kernels for LeNet-1

5) To calculate the confusion matrix and the accuracy for testing, the following com-

mand is used.

```
[confusionmatrix_test,accuracy_test]=testcnn(p1,t1,net1)
confusionmatrix_test =
```

83	37	0	0	0
17	176	0	0	0
0	0	142	3	0
0	0	1	240	0
0	0	0	0	171

```
accuracy_test =
90.0222
```

If you want to run LeNet-1 for handwritten digit images, then instead of `datas` use `datag`, and instead of `newcnns`, use `newcnn`. Also, one can try different training algorithms, such as `trainscg` or `trainsd`.

5.7 Customization

In addition to the built-in networks like LeNet-1 and LeNet-5, FIRST allows the user to create custom networks. The FIRST software is designed in a such a way that a user can easily modify the number of layers in a network or the number of FMs in a layer, simply by modifying a few parameters. The user can also create new weight functions or transfer functions. This customization is possible because networks have an object-oriented representation, which allows you to define various architectures and assign various algorithms to those architectures. There are three types of customization: 1) you can change the number and types of layers in the network, 2) you can create new network modules (like weight functions, transfer functions, and net input functions), and 3) you can create new training functions. This can be done without changing the gradient function (`calcgx` with `perg=1`), the performance function (`calgx` with `perg=0`) or other elemental functions.

The built-in network creation functions in the FIRST software are `newcnn`, `newcnn5` and `newcnn7`. These network creation functions can be modified to create new functions by changing the number or types of layers, number of FMs for each layer, or by changing the connection matrix for each layer. In order to change number of layers, `net.numLayers` is changed. To change number of FMs, `net.featureMap(`m`)` is changed. To change connection matrix, `net.ConnectionMatrix`$\{m\}$ matrix is changed. One can also create new network functions by giving any name to the functions and storing all network objects into a net structure. The existing network creation functions can be used as templates. According to application requirements, the user can define or initialize parameters such as `net.w`, `net.b`, `net.numLayers`, etc. The user can also pass an argument list, if the number of network modules is small. This could provide a flexible network creation function that allow many kinds of networks to be created.

The weight function, net input function, transform function and transfer function for each layer are assigned in the network creation functions. These functions can be modified for each layer. For example, to assign the convolution weight function to the second layer, you would use `net.wf{2}='convol'`. Here `convol` is an existing function in FIRST. Similarly, other functions, such as net input functions, transfer functions and transform functions can be modified for each layer. The user can also create new weight functions. The best way to do this is to use an existing weight function (e.g., `convol.m`) as a template. Note that all of the modular functions have two modes of operation: forward (simulation) and backward (gradient calculation). The backward operation requires that the derivative of the function be computed.

In a similar way, users can modify net input functions with `net.nf`$\{m\}$, transfer functions with `net.tf`$\{m\}$ and transform functions with `net.vf`$\{m\}$. New training functions can be assigned in the network creation function by setting set `net.trainFcn` to `'training function name'`. New training functions can be created by using one of the existing functions as a template (`trainrp`, `trainscg` or `trainsd`). New training functions can use the existing gradient and performance calculation function `calgx`.

5.8 Summary

This chapter gave a detailed explanation of the FIRST software. This modular software is built to create deep convolutional neural networks. FIRST includes all of Lecun's networks (LeNet-1, LeNet-5 and Lenet-7), and is flexible enough to enable users to easily create custom networks with arbitrary architectures. FIRST contains a computational engine that will calculate network performance and performance gradients. This engine does not have to be modified, even if new network architectures or new network modules are used. FIRST contains three different network training functions, and new training functions can be easily added. FIRST also has network validation tools, to assess the performance of trained networks.

A list of functions included in the FIRST software follow.

Network Creation Functions:

`newcnn`	LeNet-1
`newcnn5`	LeNet-5 but with last layer output used to calculate the performance function
`newcnn5le`	LeNet-5 as per Lecun's
`newcnn7`	LeNet-7

Weight Functions:

`convol`	Convolution weight function
`subsampling`	Subsampling weight function
`fullconnection`	Dot product weight function

Net input Functions:

`mplus`	plus operation

Transfer Functions:

`imagetansig`	Hyperbolic tangent sigmoid transfer function
`imagelogsig`	Logarithmic sigmoid transfer function

Transform Functions:

`vec`	to change matrix into vector

Training Functions:

`trainsd`	Gradient(Steepest) descent backpropagation
`trainrp`	Resilient Backpropagation
`trainscg`	Scaled conjugate gradient backpropagation

Utility Functions:

`calcgx`	calculate weight and bias performance gradient as single vector, also calculate network outputs, signals, and performance
`getx`	all network weight and bias values as single vector
`setx`	set all network weight and bias values with single vector

Analysis Functions:

`ker`	to calculate confusion matrix, accuracy for training data and to plot (speech data), convolution kernel or to create image of convolution kernel (image data)
`testcnn`	to calculate confusion matrix, accuracy for testing data

Processing Functions:

`datag`	for loading data, preprocessing data and dividing data into training , validation and testing.(for images-2D)
`datas`	for loading data, preprocessing data and dividing data into training , validation and testing.(for speech-1D)

Users can create their own version of functions in the software as well as modify or add any functions to the software. The above mentioned functions can be easily modified if the user wants to change network objects, network parameters or training parameters. New functions can be created inside any of the functions which are mentioned above. The user needs to follow the steps which are mentioned in this chapter to customize the network.

The next chapter will conclude the work done in the thesis and provide a brief summary. Proposed future work will also be included.

CHAPTER 6

CONCLUSIONS

The main objective of this thesis has been to create a general purpose user friendly software package that can be used to simulate and train arbitrary deep neural networks for recognizing patterns in data of any dimension. In order to build such a general neural network framework, we began by analyzing previously proposed deep network architectures. We discovered that these networks could be represented by a standard set of modules, connected in various ways. This lead us to propose a modular framework, which could include existing networks while allowing new types of networks to be easily constructed. By using this framework, a software package was created for designing, training, and simulating many network types. We call this software package the Flexible Image Recognition Software Toolbox (FIRST).

This software includes ready-made versions of some classical network architectures, such as Lecun's LeNet-1, LeNet-5 and LeNet-7. Also, by using this software, users can create custom networks with arbitrary numbers of layers and arrangements of existing layer modules without writing new code. A library of common layer modules, such as weight functions, net input functions, transfer functions, and transform functions is provided. Users can easily create their own architecture or add new modules without changing the existing training and simulation functions. This addition of new module types to the FIRST library is easy, as it requires minimal programming.

FIRST contains several training algorithms, and the addition of other algorithms is easy, as the framework is modular. The gradient calculation does not need to be changed. FIRST includes three efficient training algorithms: 1) Resilient Backpropagation 2) Scaled Conjugate Gradient Backpropagation and 3) Steepest Descent Backpropagation. The gradient computation procedures derived for this software are very general and modular. Because of this generality and modularity, it can be applied to any network that the user has created.

The FIRST software accommodates signals (1D), images (2D) and higher order objects. In this thesis, we have demonstrated the software using two practical pattern recognition problems. One problem consists of 1D speech data, and the second problem uses 2D images of handwritten digits. FIRST displays training progress and provides plots to assess network performance. Also, to verify results, the software calculates the confusion matrix and the accuracy for training and testing data.

For future work, this FIRST software can be used to train 3D data, multi-spectral data or color images. There are also some improvements that could be made to the weight initialization procedures. For multi-layer networks, layer weights and biases are often initialized with the Nguyen-Widrow [35] method. It may be possible to develop similar procedures for deep convolutional neural networks. This could help speed up network convergence.

REFERENCES

[1] K. Fukushima and S. Miyake, "Neocognitron: A new algorithm for tolerant of deformations and shifts in position," *Pattern Recognition, vol.15, no.6*, pp. 455–469, 1982.

[2] K. Fukushima, "Neocognitron: A self-organizing neural-network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics, vol. 36*, pp. 193–202, 1980.

[3] Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural Networks: The Statistical Mechanics Perspective, World Scientific*, pp. 261–276, 1995.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *PROCEEDINGS OF THE IEEE, vol. 86, no. 11*, 1998.

[5] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*. PWS Publishing Co, 3rd ed., 1996.

[6] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," *Advances in Neural Information Processing Systems (NIPS 1989), 2, Morgan Kaufman, Denver, CO*, 1990.

[7] Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Learning algorithms for clas-

sification: A comparison on handwritten digit recognition," *Neural Networks: The Statistical Mechanics Perspective*, pp. 261–276, 1995.

[8] A. Abdulkader, "A two-tier approach for arabic ofine handwriting recognition," *IWFHR*, 2006.

[9] K. Chellapilla and P. Simard, "A new radical based approach to ofine handwritten east-asian character recognition," *IWFHR*, 2006.

[10] A. Frome, G. Cheung, A. Abdulkader, M. Zennaro, B. Wu, A. Bissacco, H. Adam, H. Neven, and L. Vincent, "Large-scale privacy protection in street-level imagery," *ICCV*, 2009.

[11] C. Garcia and M. Delakis, "Convolutional face finder: A neural architecture for fast and robust face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2004.

[12] Y. Lecun and Y. Bengio, "Convolutional networks for images,speech,and time-series," *in Arbib, M. A. (Eds), The Handbook of Brain Theory and Neural Networks, MIT Press*, 1995.

[13] D. Hubel and T. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *Journal of Physiology*, 1962.

[14] K. Fukushima, "Analysis of the process of visual pattern recognition by the neocognitron," *Neural Networks, vol.2*, pp. 413–421, 1989.

[15] H. Drucker, R. Schapire, and P. Simard, "Boosting performance in neural networks," *Int.J. Pattern Recognition Artif. Intell., vol. 7*, pp. 705–719, 1993.

[16] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," *proceedings of 26th International Conference on MAchine learning, Montreal, Canada*, 2009.

[17] K. Fukushima and N. Wake, “Handwritten alphanumeric character recognition by the neocognitron,” *IEEE Transactions on Neural Networks, vol. 2, no. 3*, pp. 335–365, 1991.

[18] Y. LeCun, U. Muller, J. Ben, E. Cosatto, and B. Flepp, “Off-road obstacle avoidance through end-to-end learning,” *In Advances in Neural Information Processing Systems. MIT Press*, 2005.

[19] M. Matsugu, K. Mori, and Y. Mitari, “Convolutional spiking neural network model for robust face detection,” *In Proceedings of the 9th International Conference on Neural Information Processing(ICONIP), vol. 2*, pp. 660–664, 2002.

[20] B. Fasel, “Multiscale facial expression recognition using convolutional neural networks,” *In Proceedings of the Third Indian Conference on Computer Vision, Graphics and Image Processing, Ahmedabad, India*, 2002.

[21] F. H. C. Tivive and A. Bouzerdoum., “A fast neural-based eye detection system,” *In Proceedings of the International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pp. 641–644, 2005.

[22] Z. Saidane and C. Garcia, “Automatic scene text recognition using a convolutional neural network,” *In Proceedings of the Second International Workshop on Camera-Based Document Analysis and Recognition (CBDAR)*, 2007.

[23] B. Q. Li and B. Li, “Building pattern classifiers using convolutional neural networks,” *International Joint Conference on Neural Networks, vol.5*, pp. 3081–3085, 1999.

[24] S. J. Nowlan and J. C. Platt, “A convolutional neural network hand tractker,” *Advances in Neural Information Processing Systems, vol. 7*, pp. 901–908, 1995.

[25] R. Ouellette, M. Browne, and K. Hirasawa, “Genetic algorithm optimization of a convolutional neural network for autonomous crack detection,” *In Congress on Evolutionary Computation, vol. 1*, pp. 516–521, 2004.

[26] R. Gonzalez and R. Woods, *Digital Image Processing*. Pearson Education, 3rd ed., 2008.

[27] D. M. Kline and V. L. Berardi, “Revisiting squared-error and cross-entropy functions for training neural network classiers,” *Neural Comput and Applic*, vol. 14, pp. 310–318, 2005.

[28] M. Riedmiller and H. Braun, “A direct adaptive method for faster back-propagation learning: The rprop algorithm,” *in Proc. ICNN 93, San Francisco, CA, 1993*, pp. 586–591, 1993.

[29] M. F. Moller, “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks, vol.6*, pp. 525–533, 1993.

[30] R. Kohavi and F. Provost, “Glossary of terms,” *Machine Learning*, vol. 30, pp. 271–274, 1998.

[31] The Center of Excellence in Document Analysis and Recognition (CEDAR) at SUNY Buffalo,, “Data.” `http://www.gaussianprocess.org/gpml/data/`.

[32] J. J. Hull, “Handwritten text recognition research,” *IEEE PAMI*, vol. 16(5), pp. 550–554, 1994.

[33] TIMIT Acoustic-Phonetic Continuous Speech Corpus, NTIS, US Dept of Commerce, “Datasets for the elements of statistical learning.” `http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html`.

[34] T. Hastie, A. Buja, and R. Tibshirani, “Penalized discriminant-analysis,” *Ann. Statist.*, vol. 23, no. 1, pp. 73–102, 1995.

[35] D. Nguyen and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *Neural Networks, IJCNN International Joint Conference on*, vol. 3, pp. 21–26, 1990.

APPENDIX A Notation

Basic Concepts:

Scalars: small italic letters....*a,b,c*

Vectors: small bold non-italic letters...**a,b,c**

Matrices: capital bold letters....**A,B,C**

Vectors:

Vector means a column of numbers.

Transformations:

vec means converting matrix into vector.

mat means converting vector into matrix.

Weight Matrices:

Scalar Element:

$w_{i,j}^{m,h}$

$w_{i,j}^{m,h,l}$

i -row, j -column, m -layer, h -feature map at layer m , l -feature map at layer $m-1$.

Matrix

$\mathbf{W}^{m,h}$

$\mathbf{W}^{m,h,l}$

Bias Matrices:

Scalar Element:

b_k^m

Vector:

$\mathbf{b}^m$

Matrix

$\mathbf{B}^{m,h}$

bias matrix created by multiplying matrix all elements=1 with scalar $b^{m,h}$

Input Matrices:

Scalar Element:

$p_{i,j}^m$

One of a set of input matrices(patterns):

$\mathbf{P}_{i,j,q}^m$

q-number of patterns

Weight Operation Output matrices:

Scalar Element:

$z_{i,j}^{m,h}$

Matrix:

$\mathbf{Z}^{m,h}$

Net Input matrices:

Scalar Element:

$n_{i,j}^{m,h}$

Matrix:

$\mathbf{N}^{m,h}$

Output matrices:

Scalar Element:

$a_{i,j}^{m,h}$

Matrix:

$\mathbf{A}^{m,h}$

Transfer Function:

Scalar Element:

$a_{i,j}^{m,h}$

Vector:

$\mathbf{a}^{m,h}$

Matrix:

$\mathbf{A}^{m,h}$

Transformation Function:

Scalar Element:

$v_{i}^{m,h}$

Vector:

$\mathbf{v}^{m,h}$

Matrix:

$\mathbf{V}^{m,h}$

Connection Matrix Indices:

$C^{m,h}$

set of indices of feature maps in layer $m-1$ that connect to feature map h in layer m .

$C_b^{m,h}$

set of indices of feature maps in layer m that connect to feature map l in layer $m-1$.

Matrix:

$\mathbf{C}^m$

Scalar Element:

$c_{i,j}^m$

if feature map j in layer $m-1$ connects to feature map i in layer m .

Target Matrices:

Scalar Element:

$t_{i,q}^m$

Vector:

$\mathbf{t}_q^m$

Matrix:

$\mathbf{T}_q^m$

Set of Prototype Input/Target Matrices:

($\mathbf{P}_1^m$, $\mathbf{T}_1^m$),($\mathbf{P}_2^m$, $\mathbf{T}_2^m$) ,.....,($\mathbf{P}_Q^m$, $\mathbf{T}_Q^m$)

Error Vector:

Scalar Element:

$e_{i,q}^m = t_{i,q}^m - a_{i,q}^m$

Vector:

$\mathbf{e}_q^m$

Size and Dimensions:

number of rows in sub-sampling kernel for layer m : r^m

number of columns in sub-sampling kernel for layer m : c^m

number of rows in feature map in layer m : S_r^m

number of columns in feature map in layer m : S_c^m

number of Input matrices(target vectors) and dimensions of input : Q, $R_r \times R_c$

number of layers and number of feature maps : M,H

number of neurons in dot product : K

Performance Variable:

$F(x)$

Gradient:

Scalar Element:

partial derivative of error w.r.t. transform function output : $dV_{i,j}^{m,h}$

partial derivative of error w.r.t. transfer function output : $dA_{i,j}^{m,h}$

partial derivative of error w.r.t. net function output : $dN_{i,j}^{m,h}$

partial derivative of error w.r.t. weight function output : $dZ_{i,j}^{m,h}$

partial derivative of error w.r.t. weight : $dW_{i,j}^{m,h}$

partial derivative of error w.r.t. bias : $db^{m,h}$

Matrix:

partial derivative of error w.r.t. transform function output : $\mathbf{dV}^{m,h}$

partial derivative of error w.r.t. transfer function output : $\mathbf{dA}^{m,h}$

partial derivative of error w.r.t. net function output(sensitivity) : $\mathbf{dN}^{m,h}$

partial derivative of error w.r.t. weight function output : $\mathbf{dZ}^{m,h}$

partial derivative of error w.r.t. weight : $\mathbf{dW}^{m,h}$

Vector:

partial derivative of error w.r.t. bias : $\mathbf{db}^m$

Mathematical Operators for functions:

Convolution operator : $\star$

Sub-sampling Operator for reduction with size of output of operation:

$\boxplus_{S_r^m, S_c^m}$

Sub-sampling Operator for expansion: $\boxminus$

Multiplication : $\times$

Matrix element by element multiplication (Hadamard Product): $\circ$

Normal Matrix Multiplication : $*$

Double summation (i.e Sub-sampling Operator for reduction with size of output of operation as 1×1): $\boxplus_{1,1}$

APPENDIX B List of FIRST Functions

New Networks Functions: to create network architectures

`newcnn`- LeNet-1

`net=newcnn`

returns `net` which is the new network objects.

`newcnn5`- LeNet-5 but with last layer output is taken in calculating performance function

`net=newcnn5`

returns `net` which is the new network objects.

`newcnn5le`- LeNet-5 as per Lecun's

`net=newcnn5le`

returns `net` which is the new network objects.

`newcnn7`- LeNet-7

`net=newcnn7`

returns `net` which is the new network objects.

Weight Functions: to perform operation with weights of layer

`convol`- Convolution weight function

`dd=convol(a,b,ba,i,net,q)`

`a` and `b` - cell array inputs on which convolution is performed.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

and returns,

`dd` - a cell array in which output for each FM is stored for layer i during forward calculation.

`[dw,dZ]`-the gradient with respect to the transform function output is calculated and with respect to the weights during backward calculation.

`subsampling`- Subsampling weight function

`c=subsampling(b,a,ba,i,net)`

`a` and `b` - cell array inputs on which subsampling(reduction or expansion) is performed.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

and returns,

`c` - a cell array in which output for each FM is stored for layer i during forward calculation.

`[dw,dZ]`-the gradient with respect to the transform function output is calculated and with respect to the weights during backward calculation.

`fullconnection`- Dot product weight function

`c=fullconnection(a,b,ba,i,net)`

`a` and `b` - cell array inputs on which dot product is performed.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

and returns,

`c` - a cell array in which output for each FM is stored for layer i during forward calculation.

`[dw,dZ]`-the gradient with respect to the transform function output is calculated and with respect to the weights during backward calculation.

Net input Functions: to sum excitations of layers

`mplus`- plus operation

`n=mplus(b,z,ba,i,net)`

`b` - bias and `z` - output of weight function.

`ba` - a flag (future use).

`i` - layer number.

`net` - a network given with parameters.

and returns,

`n` - a cell array which stores net input function output for layer i during forward calculation.

`[db,dZ]`-the gradient with respect to weight function output and also with respect to the bias during backward calculation.

Transfer Functions: to squash output into the range

`imagetansig`- Hyperbolic tangent sigmoid transfer function.

`a=imagetansig(n,ba,i,net,dA)`

`n` - input to the transfer function is performed.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

`dA` - a gradient of the PI with respect to the previous layer output during backward calculation.

and returns,

`a` - a cell array which stores transfer function output for layer i during forward calculation.

`dN` - the gradient of the PI with respect to the net input function output (requires during backward calculation).

`imagelogsig`- Logarithmic sigmoid transfer function.

`a=imagelogsig(n,ba,i,net,dA)`

`n` - input to the transfer function is performed.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

`dA` - gradient w.r.t. transfer function output (requires during backward calculation).

and returns,

`a` - a cell array which stores transfer function output for layer i during forward calculation.

`dN` - the gradient of the PI with respect to the net input function output (requires during backward calculation).

Transform Functions: to transform output of network layer

`vec`- to change matrix into vector.

`v=vec(a,ba,i,net,q)`

`a` - input on which conversion takes place.

`ba` - a flag i.e. when `ba=0` i.e. forward calculation.

when `ba=1` i.e. backward calculation.

`i` - layer number.

`net` - a network given with parameters.

`q` - required during backward calculation.

and returns,

`v` - a cell array which stores transform function output for layer i during forward calculation.

`dA` - the gradient of the PI with respect to the previous layer output (requires during backward calculation).

Training Functions: to train networks

`trainsd`- Gradient(Steepest) descent backpropagation

`[net,tr] = trainsd(net,Pd,Tl,p1,t1,p2,t2)`

This function takes inputs,

`NET` - Neural network.

`Pd` - training input data.

`Tl` - training target data.

`p1` - testing input data.

`t1` - testing target data.

`p2` - validation data set.

`t2` - validation target data.

and returns,

`NET` - Trained network.

`TR` - Training record of various values over each epoch:

`TR.epoch` - Epoch number.

`TR.perf` - Training performance.

`TR.vperf` - Validation performance.

`TR.tperf` - Test performance.

`trainrp`- Resilient Backpropagation

```
[net,tr] = trainrp(net,Pd,Tl,p1,t1,p2,t2)
```

This function takes inputs,

`NET` - Neural network.

`Pd` - training input data.

`Tl` - training target data.

`p1` - testing input data.

`t1` - testing target data.

`p2` - validation data set.

`t2` - validation target data.

and returns,

`NET` - Trained network.

`TR` - Training record of various values over each epoch:

`TR.epoch` - Epoch number.

`TR.perf` - Training performance.

`TR.vperf` - Validation performance.

`TR.tperf` - Test performance.

`trainscg`- Scaled conjugate gradient backpropagation

`[net,tr] = trainscg(net,Pd,Tl,p1,t1,p2,t2)`

This function takes inputs,

`NET` - Neural network.

`Pd` - training input data.

`Tl` - training target data.

`p1` - testing input data.

`t1` - testing target data.

`p2` - validation data set.

`t2` - validation target data.

and returns,

`NET` - Trained network.

`TR` - Training record of various values over each epoch:

`TR.epoch` - Epoch number.

`TR.perf` - Training performance.

`TR.vperf` - Validation performance.

`TR.tperf` - Test performance.

Utility Functions: internal utility functions

`calcgx`- calculate weight and bias performance gradient as single vector, also calculate network outputs, signals, and performance

`[perf,gX]=calcgx(net,p,t,perg)`

`p` - given input.

`t` - target values.

`net` - network.

`perg` - a flag to denote whether to calculate PI or gradient along with PI

`perf` - the performance

`gX` - gradient value which is in a vector form.

`getx`- all network weight and bias values as single vector

`x = getx(net)`

`net` - Neural network.

`x` - Vector of weight and bias values.

`setx`- set all network weight and bias values with single vector

`net = setx(net,x)`

`net` - Neural network.

`x` - Vector of weight and bias values.

`imtrain`- to evaluate training function.

`[net,tr]=imtrain(net,p,t,p1,t1,p2,t2)`

This function takes inputs,

`NET` - Neural network.

`p` - training input data.

`t` - training target data.

`p1` - testing input data.

`t1` - testing target data.

`p2` - validation data set.

`t2` - validation target data.

and returns,

`NET` - Trained network.

`TR` - Training record of various values over each epoch:

`TR.epoch` - Epoch number.

`TR.perf` - Training performance.

`TR.vperf` - Validation performance.

`TR.tperf` - Test performance.

`imagemse`- to calculate mean square value as PI

`[E]=imagemse(a,t,ba)`

`a` - the final layer output

`t` - the target

`ba` - the backpropagation flag

and returns,

`E`- mean squared value during forward calculation.

`dA`-the gradient of the PI with respect to the layer output during backward calculation.

Analysis Functions: to analyze network performance

`ker`- to calculate confusion matrix, accuracy for training data and to plot (speech data) convolution kernel or to create image of convolution kernel (image data)

`[confusionmatrix_train,accuracy_train]=ker(net1,p,t)`

`net1`- network with updated values of weights and biases.

`p` and `t`-training data and target values for training data.

and returns,

`confusionmatrix_train`- confusion matrix.

`accuracy_train`- accuracy calculated from confusion matrix.

`testcnn`- to calculate confusion matrix, accuracy for testing data

`[confusionmatrix_test,accuracy_test]=testcnn(p1,t1,net)`

`net`- network with updated values of weights and biases.

`p1` and `t1`- testing data and target values for testing data.

and returns,

`confusionmatrix_test`- confusion matrix.

`accuracy_test`- accuracy calculated from confusion matrix.

Processing Functions: to load and preprocess input data

`datag`- for loading data for LeNet-1, preprocessing data and dividing data into training , validation and testing.(for images-2D)

`datag5`- for loading data for modified LeNet-5, preprocessing data and dividing data into training , validation and testing.(for images-2D)

```
[p,t,p1,t1,p2,t2]=datag
[p,t,p1,t1,p2,t2]=datag5
```

`p` and `t`-training data and target values for training data.

`p1` and `t1`-testing data and target values for testing data.

`p2` and `t2`-validation data and target values for validation data.

`datas`- for loading data for LeNet-5, preprocessing data and dividing data into training , validation and testing.(for speech-1D)

```
[p,t,p1,t1,p2,t2]=datas
```

`p` and `t`-training data and target values for training data.

`p1` and `t1`-testing data and target values for testing data.

`p2` and `t2`-validation data and target values for validation data.

APPENDIX C List of Abbreviations

List of Abbreviations

CNN	Convolutional Neural Network
FIRST	Flexible Image Recognition Software Toolbox
RPROP	Resilient Backpropagation
SCG	Scaled Conjugate Gradient
MLP	Multilayer Perceptron
MRI	Magnetic resonance Imaging
RBM	Restricted Boltzmann Machine
FM	Feature Map
RBF	Radial Basis Function
CRBM	Convolutional Restricted Boltzmann Machine
2D	Two Dimensional
1D	One Dimensional
3D	Three Dimesional
PI	Performance Index
ROC	Receiver Operating Characteristic
TPR	True Positive Rate
FPR	False Positive Rate

VITA

Pranita Patil

Candidate for the Degree of

Master of Science

Thesis: FLEXIBLE IMAGE RECOGNITION SOFTWARE TOOLBOX (FIRST)

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2013.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Pune University, Maharashtra, India in 2010.

Experience:

Worked as Research Assistant at University of Pune for Unique Identification Authority of India and fingerprint recognition project from 2010 to 2011.