Hybrid Evolution
of Convolutional Neural Networks

by

Brian W. Cheung

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Engineering

May 6, 2011

Advisor

Prof. Carl Sable
This thesis was prepared under the direction of the Candidate’s Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dr. Simon Ben-Avi
Acting Dean, School of Engineering

Prof. Carl Sable
Candidate’s Thesis Advisor
Abstract

With the increasing trend of artificial neural networks toward larger structures with more layers, the training characteristics of these structures has become an important area of research. Many of the initial parameters of these networks have an enormous impact on the convergence characteristics during training. Poorly tuned parameters can often lead to orders of magnitude longer training. In this work, a general framework has been created to define the training and structural parameters of convolutional neural networks using a combination of the stochastic diagonal Levenberg-Marquardt method to accelerate training and evolutionary search for structural and training parameter optimization. Our results show that for the task of handwritten digit classification, the networks created by such a system outperform manually defined networks in speed of convergence.
Acknowledgments

I would like to thank Professor Carl Sable for his guidance and support of this thesis and my family for their encouragement. I would also like to thank Deian Stefan and the SProCom lab for generously providing computational resources for this research and Sherry Young for her assistance in the final revisions.
Contents

Table of Contents v
List of Figures vii

1 Introduction 1

2 Neural Networks 3
  2.1 Structure .................................................. 3
     2.1.1 Perceptron ......................................... 3
     2.1.2 Multilayer Perceptrons .............................. 4
     2.1.3 Activation Function .................................. 6
  2.2 Training ..................................................... 9
     2.2.1 Maximum Likelihood ................................ 9
     2.2.2 Backpropagation ..................................... 11
     2.2.3 Inputs for Training .................................. 17
     2.2.4 Early Stopping ....................................... 18
     2.2.5 Second Order Methods ............................... 19

3 Convolutional Neural Networks 30
  3.1 Overview .................................................. 30
  3.2 Neocognitron .............................................. 30
  3.3 Structure .................................................. 32
  3.4 Feedforward Propagation ................................. 35
  3.5 Backpropagation .......................................... 37

4 Evolutionary Algorithms 39
  4.1 Overview .................................................. 39
     4.1.1 Population ........................................... 40
     4.1.2 Fitness ............................................... 41
     4.1.3 Crossover/Mutations ................................ 41
  4.2 Self-Adaptive Evolution ................................ 42
  4.3 Comparison to Gradient Descent .......................... 43
  4.4 Application to Neural Networks .......................... 45
## CONTENTS

5  **Implementation**  
   5.1  MNIST Dataset  
   5.2  EBLearn  
   5.3  Hybrid Method for Evolving Weights  
   5.4  Hybrid Method for Evolving Network Architecture  

6  **Results**  
   6.1  Analysis of Weight Evolution  
   6.2  Analysis of Architecture Evolution  

7  **Conclusion**  

8  **Future Work**  

Bibliography
# List of Figures

2.1 The perceptron is outlined by the dotted ellipse. ........................................ 4
2.2 A multilayer network of perceptrons with three inputs and two outputs. The hidden and output layer is considered fully connected while the input and hidden layers are not. ................................................................. 5
2.3 The propagation of input signal to the output of neuron $j$ ................................. 6
2.4 Comparison of activations functions: logistic function (blue), hyperbolic tangent (green) and LeCun’s hyperbolic tangent (red) ................................................................. 8
2.5 Two radial basis functions centered at $-1$ (green) and $+1$ (blue) ......... 9
2.6 A geometric interpretation of the error surface $E$ in weight space. Note the gradient $\nabla E$ points exactly opposite of the minimum point $w_{\text{min}}$. 13
2.7 Surface plot of the Rosenbrock function on a logarithmic scale ................. 20
2.8 Countours of a quadratic $E$ .......................................................................... 28
3.1 First layer of a Convolutional Neural Network .............................................. 34
3.2 Diagram of LeNet-5 ....................................................................................... 35
3.3 As the feature maps become smaller in layers closer to the output, the portion of the image which contributes to its input becomes larger .. 36
4.1 Diagram of the stages of a genetic algorithm ................................................. 40
5.1 Samples of the training set of the MNIST database. ................................. 49
5.2 Class distribution of the entire MNIST training set and test set. ............. 50
5.3 Class distribution of the various subsets of the MNIST training set used. The first column shows the class distribution of each subset of MNIST. The second and third column show the class distribution of these subsets divided into two disjoint subsets. ......................... 51
5.4 Block diagram of the hybrid architecture evolution process. ................... 58
6.1 Total Energy on 200 sample MNIST training subset for 6000 generations of weight evolution. ................................................................. 60
6.2 Total Energy on 10000 sample MNIST test set sampled at every 100 generations of weight evolution. ................................................................. 61
6.3 Performance of different training and validation strategies on the MNIST test set with 4000 MNIST training samples ............................. 64
6.4 Performance starting at generation 14 of different training and validation methods on the MNIST test set ............................................. 65
6.5 Error on the training set at the 20th (last) epoch on the 4000 sample MNIST training subset. ................................................................. 66
6.6 Comparison of the training error after every epoch on the 4000 sample MNIST training subset. ................................................................. 67
Chapter 1

Introduction

Demonstrating the performance of a neural network is often difficult because of the variability in reproducing results. This can often be attributed to the many hand-tuned parameters involved when specifying the neural network. This often includes the number of nodes of each layer and the connections that exist between these layers. Much of these parameters are heuristically tuned and often a trial and error based approach is used to find the parameters which work well for a particular network given a particular task. When making comparisons between other novel networks or machine learning methods where the performance margin is relatively small, these parameters become much more significant.

With the emergence of deeper networks with more hidden layers, the number of hand-tuned parameters that specify the network increases even more. This leads to many more possible parameter permutations making the trial and error process even more difficult. Because these parameters are often coupled with one another in an unknown manner, manually searching for optimal values of these parameters becomes unfeasible. At the time of this writing, there is no widely accepted method for searching for these important parameters. The heuristic-based methods used vary
greatly and are often based solely on user preference.

Evolutionary algorithms encompass a set of stochastic optimization methods which tend to work well when little a priori knowledge of the parameters is known. Therefore this can be seen as a natural choice when the optimization parameters are non-differentiable and complex. Although the search is based on the heuristic of natural selection, it conducts a much more global search and is less biased than many other optimization techniques due to its non-deterministic nature. Evolutionary algorithms tend to excel in applications where more formally defined hill-climbing techniques are not possible.

But the greatest weakness in evolutionary algorithms is their computational complexity. Because the search relies on so little a priori knowledge, it can often take orders of magnitude longer to converge than more analytic methods. Thus in many non-linear optimization tasks, hybrid methods are proposed where the global search of evolutionary algorithms are combined with the much faster analytic methods to find better optima [18]. With this in mind, a method of combining analytic and evolutionary search is proposed to create and train high performance convolutional neural networks without hand-tuning many of the network parameters.
Chapter 2

Neural Networks

2.1 Structure

2.1.1 Perceptron

The smallest individual component of a neural network is the perceptron. More commonly referred to as a node, this component is considered to be a basic binary classifier. A single perceptron is only capable of discriminating an input feature vector $x$ between two linearly separable classes $\{0, 1\}$. The following steps describe its operation which is also illustrated in Figure 2.1:

1. Feed input feature vector $x$

2. Computed $v = x \cdot w$, where $w$ is a weight vector

3. Pass the result through a nonlinear activation function $\varphi(v)$ to produce output $y$, where $y = \varphi(v)$

This is often called a feedforward process as the inputs are fed forward from the input of the perceptron to its output. In Figure 2.1, multiple directed edges leaving the
node, denoted by the ellipse, correspond to the outputs $y$ being passed to other nodes creating a network. The first perceptrons used a non-differentiable step function as an activation function $\varphi$. Today, the activation functions used are both continuous and differentiable for most neural networks [4]. As will be discussed in later sections, this difference plays a key role in the training neural networks.

Figure 2.1 The perceptron is outlined by the dotted ellipse.

2.1.2 Multilayer Perceptrons

Multilayer perceptrons are composed of a network of connected perceptrons. As shown in Figure 2.2, it is represented as a directed graph with one or more inputs and outputs. The typical structure is organized into multiple layers of nodes. In a fully connected architecture, each node in one layer has incoming connection from all nodes in its preceding layer. Nodes within the same layer are not connected with one another. Therefore the output of a single node in one layer is a function of all
2.1 Structure

Figure 2.2 A multilayer network of perceptrons with three inputs and two outputs. The hidden and output layer is considered fully connected while the input and hidden layers are not.

connected nodes in the preceding layer. Layers which are not at the input or output of the network are referred to as *hidden*.

Figure 2.3 shows the connectivity for one node. All connected nodes in a preceding layer, $x_0$ to $x_L$, are shown at the input of node $j$. For hidden and output layers, $x_0$ is a node with a fixed output of +1 and no inputs. This node is known as the *bias* or *threshold*. Similar to linear classifiers, the bias is used to compensate for the differences between the average input and the average output of a node [4]. This enables each hidden and output node the ability to shift its respective activation function to account for constant offsets in its inputs. Note in Figure 2.3 that node $j$ is in the hidden layer of the network, the multiple arrows pointing outwards from node $j$ indicate its output will serve as inputs to nodes in the next layer.

Sparsely connected topologies and skip-layer connections also exist. Skip-layers
connection are edges which connect nodes two or more successive layers away. A fully connected network can create functionally equivalent topologies. Theoretically, a fully connected two-layer neural network, similar to that shown in Figure 2.2, with linear outputs is capable of approximating to arbitrary precision any finite continuous function on a compact input domain [4]. Even so, constraining the structure artificially may enforce certain desireable characteristics which will not likely occur from training.

### 2.1.3 Activation Function

The activation function is also known as the basis or squashing function. In most cases, this is a differentiable function which adds a non-linear component to the multilayer feedfoward neural network. Without this non-linearity, the network would be reduceable to a single layer of perceptrons [10]. This non-linearity enables the network to learn complex non-linear tasks which the single-layer perceptron cannot.

A popular example is the XOR function.

A commonly used activation function is the logistic function:

\[
y_j = \frac{1}{1 + e^{x_j}}
\] (2.1)
The term $v_j$ is known as the *induced local field* for node $j$. Formally, it is defined as:

$$v_j = \sum_{i \in M} w_{ji} x_i$$  \hspace{1cm} (2.2)

$M$ denotes the indices of the set of nodes in the preceding layer (assuming a fully connected network). As shown in Figure 2.3, the result of the activation function of a preceding layer is linearly combined to become inputs to nodes in the current layer. The network can be viewed as a series of non-linear transformations from one layer to the next [4].

Another widely used activation function is the hyperbolic tangent. This activation function is defined as:

$$y_j = \tanh(v_j)$$  \hspace{1cm} (2.3)

Unlike the logistic function, the hyperbolic tangent function is symmetric about the origin. This symmetry is preferred because it can accelerate convergence [13] and is more likely to produce outputs that have a mean close to zero. But one potential problem with symmetric activations is that the error surface can be very flat near the origin. Flat regions can significantly increase convergence time during training because the saddle point at the origin created by this function is attractive in almost all directions [13]. Therefore, initializing the network with very small weights should be avoided when using symmetric activation functions. Furthermore, saturation of activation functions in general (ie. being very far from the origin) can also result in flat regions. [14] suggests adding a small linear term to the activation to help avoid such regions.

[11] recommends a scaled hyperbolic tangent defined as:

$$y_j = 1.7159 \tanh\left(\frac{2}{3} v_j\right)$$  \hspace{1cm} (2.4)

The resulting gain of (2.4) is approximately one for $v_j$ ranging between $-1$ and $1$. In addition to simplifying the interpretation of the state of the network, the absolute
value of the second derivative of (2.4) is maximal at $-1$ and $1$ which further improves convergence [13]. The plot of each activation function is shown in Figure 2.4.

![Figure 2.4](image)

**Figure 2.4** Comparison of activations functions: logistic function (blue), hyperbolic tangent (green) and Lecun’s hyperbolic tangent (red)

Radial basis function (RBF) networks use an alternative activation function for nodes. The output of a node is computed as

$$y_j = \sum_{i \in M} w_i \exp\left(-\frac{1}{2\sigma_i^2}||x_i - \mu_i||^2\right)$$

(2.5)

where $\sigma_i$ is the standard deviation and $\mu_i$ is the mean of the $ith$ node in the preceding layer [14]. These two parameters are usually found through unsupervised clustering. Figure 2.5 shows two RBFs centered at different points. Unlike sigmoid type units, RBF nodes cover a specific locality of the input space, centered on $\mu_i$. The locality can lead to faster learning but may also require many more nodes for high dimensional spaces. RBF nodes can coexist in the same network with the more common sigmoid type units. RBF nodes are often used at the output layers [14].
2.2 Training

2.2.1 Maximum Likelihood

The maximum likelihood approach estimates the parameters $\mathbf{w}$ of a model which maximizes the probability of the observed data $X$. For example, given a set of ob-
servations $X = \{x_1, \ldots, x_N\}$ and system parameters $w$, we can use the method of maximum likelihood to maximize the conditional probability $p(X|w)$.

\begin{align*}
\text{Likelihood} : p(X|w) \\
\text{Prior} : p(w) \\
\text{Posterior} : p(w|X)
\end{align*} (2.8) (2.9) (2.10)

**Supervised Maximum Likelihood**

In supervised maximum likelihood, we assume that the set of labels $D = \{t_1, \ldots, t_N\}$, corresponding to observations $X$, are Gaussian distributed. The conditional probability of a label $t$ given system parameters $w$ and observation $x$ is defined as:

$$p(t|x, w) = N(t|y(x, w), \beta^{-1})$$ (2.11)

$N$ is defined in the notation of the Gaussian distribution $N(\mu, \sigma^2)$. $\beta$ is the inverse variance of the labels (precision). Here our observation $x$ is fixed while our system parameters $w$ are adjustable. The function $y(x, w)$ is the system output. Assume the set of observations $X$ are independent and identically distributed, the goal is to maximize the likelihood of $D$ which can be written as:

$$p(D|X, w, \beta) = \prod_{n=1}^{N} p(t_n|x_n, w, \beta)$$ (2.12)

$$= \prod_{n=1}^{N} N(t_n|y(x_n, w), \beta^{-1})$$ (2.13)

In other words, we wish to find the system parameters which most likely produces the distribution of the labels. Since the logarithm function is monotonically increasing, maximizing the logarithm of the likelihood will also maximize the likelihood. Therefore, taking the logarithm of (2.13), the equation becomes:

$$\ln p(D|X, w, \beta) = -\frac{\beta}{2} \sum_{n=1}^{N} (y(x_n, w) - t_n)^2 - \frac{N}{2} \ln(\beta^{-1}) - \frac{N}{2} \ln(2\pi)$$ (2.14)
2.2 Training

Negating (2.14) leads to:

$$- \ln p(D|X, w, \beta) = \frac{\beta}{2} \sum_{n=1}^{N} (y(x, w) - t_n)^2 - \frac{N}{2} \ln(\beta) + \frac{N}{2} \ln(2\pi)$$  \hspace{1cm} (2.15)$$

Minimizing (2.15) is equivalent to maximizing the likelihood equation in (2.13). Removing constant terms, the function to minimize becomes the sum of squares error.

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} (y(x, w) - t_n)^2$$  \hspace{1cm} (2.16)$$

An alternative approach is maximum a posteriori (MAP) estimation which is used in Bayesian Neural Networks. It can be considered a regularized version of maximum likelihood in which the error function (2.16) also includes a regularization term. Regularization prevents the problem of overfitting by imposing a prior. The Bayesian approach to estimation takes into account the prior distribution $p(w)$ and views the system parameters $w$ as uncertain. The uncertainty of the parameters is evaluated after the data $X$ is observed. It is represented as the posterior distribution $p(w|X)$ [4].

2.2.2 Backpropagation

Backpropagation is the most common method of training a neural network. Generalizing the least mean square algorithm used in adaptive filtering, backpropagation utilizes the optimization method known as gradient descent to adjust the weights of each connection in the network to minimize the average error [10]. In the simplest form, error is defined as the difference between the desired output $d_j$ of the network and its actual output $y_j$ when presented with a training example $n$. Therefore, the error $e_j$, of a single output neuron $j$, is defined as:

$$e_j(n) = d_j(n) - y_j(n)$$  \hspace{1cm} (2.17)$$
The instantaneous error, $E(n)$, of a given example is known as the *sum of squares error function* over the set of output neurons $C$ [4]. This can be viewed as a more specific form of (2.16).

$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n)$$

(2.18)

The goal of backpropagation is minimize the average error $E_{av}$ of all training examples in the training set of size $N$.

$$E_{av} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

(2.19)

With these definitions of error in mind, the general approach which backpropagation uses to minimize these values is described.

The negative gradient of $E(n)$, $-\nabla E(n)$, represents a vector which points in the direction of the greatest decrease of $E(n)$ with respect to the current weights $\mathbf{w}$ of the network. Figure 2.6 visually depicts the direction of the gradient of a network with only two weight parameters. Using a method known as *gradient descent*, these weights are updated by a value proportional to $-\nabla E(n)$ as shown in (2.20).

$$\mathbf{w}(t) = \mathbf{w}(t - 1) - \eta \nabla E(n)$$

(2.20)

In *on-line* or *stochastic* learning, these weights are adjusted in this fashion after each iteration of a training example in the training set and $\nabla E(n)$ is considered to be an approximation of the true gradient $\nabla E_{av}$.

This process iteratively searches the weight space, adjusting the old weights of the network $\mathbf{w}(t - 1)$ to move towards the direction which minimizes $E(n)$ (and presumptively $E_{av}$). $\eta$ is a learning rate factor which controls the ‘step size’ of each iteration. This value is often adaptively adjusted during training to control the speed of convergence of the network towards a minimum error.
Figure 2.6 A geometric interpretation of the error surface $E$ in weight space. Note the gradient $\nabla E$ points exactly opposite of the minimum point $w_{\text{min}}$.

Focusing on the individual components of $\nabla E(n)$, the partial derivative $\frac{\partial E(n)}{\partial w_{ji}(n)}$ is a measure of the relationship between the ‘total error energy’, $E(n)$, and a weight $w_{ji}$. For each training example $n$ presented to the network, the weight is adjusted by $\Delta w_{ji}(n)$ which represents the correcting factor for the connection weight between neuron $j$ and neuron $i$ in $j$’s preceding layer. The delta rule expresses the correcting factor for an output neuron $j$ as:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$  \hspace{1cm} (2.21)

Similar to the gradient descent formula in (2.20), the weight update is negated with a learning rate factor $\eta$. This ensures the weight update will move in the direction to minimize the error $E(n)$. According to the chain rule, $\frac{\partial E(n)}{\partial w_{ji}(n)}$ can be redefined as:
2.2 Training

\[
\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{2.22}
\]

Differentiating (2.18) yields:

\[
\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{2.23}
\]

Differentiating (2.17) leads to:

\[
\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{2.24}
\]

Using \(y = \varphi_j(v_j(n))\) where \(v_j(n)\) is the activation function:

\[
\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{2.25}
\]

Differentiating (2.2) with respect to \(w_{ji}\):

\[
\frac{\partial v_j(n)}{\partial w_{ji}} = x_i(n) \tag{2.26}
\]

where \(x_i(n)\) is the output from neuron \(i\) in the preceding layer of the network.

Therefore substituting (2.23), (2.24), (2.25) and (2.26) into (2.22) yields:

\[
\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))x_i(n) \tag{2.27}
\]

Further, substituting (2.27) into (2.21) leads to:

\[
\Delta w_{ji}(n) = \eta e_j(n)\varphi_j'(v_j(n))x_i(n) \tag{2.28}
\]

At the output layer, \(e_j(n), v_j(n)\) and \(x_i(n)\) are known when an input is fed forward through the network. However, when \(j\) is a hidden node the expression of \(e_j(n)\) is unknown because the desired response \(d_j(n)\) is only known at the output layer. The term \(e_j(n)\) must be redefined for hidden nodes.
Using (2.23) and (2.24), we can redefine $e_j(n)$ as:

$$e_j(n) = -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)}$$  \hspace{1cm} (2.29)

To prevent confusion, at this point we will redefine $j$ as a hidden node in the layer directly preceding the output layer. We will also define $k$ as a node in the output layer. For example, equations (2.2), (2.17) and (2.18) will be redefined as:

$$v_k(n) = \sum_{j \in M} w_{kj}(n) y_j(n)$$  \hspace{1cm} (2.30)

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n))$$  \hspace{1cm} (2.31)

$$E(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$$  \hspace{1cm} (2.32)

Differentiating (2.32) with respect to $y_j(n)$ leads to:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$  \hspace{1cm} (2.33)

Differentiating (2.31) with respect to $v_k(n)$ yields:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'_k(v_k(n))$$  \hspace{1cm} (2.34)

Similarly, differentiating (2.30) with respect to $y_j(n)$:
\[
\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{2.35}
\]

With the expressions obtained in (2.34) and (2.35), (2.33) can be rewritten as:

\[
\frac{\partial E(n)}{\partial y_j(n)} = -\sum_{k \in C} e_k(n)\varphi'_k(v_k(n))w_{kj}(n) \tag{2.36}
\]

Subsequently with (2.36), (2.29) can be rewritten as:

\[
e_j(n) = \sum_{k \in C} e_k(n)\varphi'_k(v_k(n))w_{kj}(n) \tag{2.37}
\]

The error \(e_j(n)\) for hidden node \(j\) is now expressed in terms of known quantities. Equation (2.37) uses the error \(e_k(n)\) of all nodes in the output layer to calculate error for a node \(j\) in the preceding layer.

Therefore the correcting factor of weight \(w_{ji}(n)\) is:

\[
\Delta w_{ji}(n) = \eta e_j(n) \left( \varphi'_j(v_j(n))y_i(n) \sum_{k \in C} e_k(n)\varphi'_k(v_k(n))w_{kj}(n) \right) \tag{2.38}
\]

This formula can be further extended for nodes in the preceding layer of \(j\) and so on. We can define a node \(i\) as any node in the layer preceding node \(j\). The error for node \(i\) can be found once the errors for node \(j\)'s layer have been found which likewise is dependent on the errors in node \(k\)'s layer.

The recursive nature of this formula can be seen where the errors of the last layer (output layer) must be known before the errors of hidden and input nodes can be found. Therefore using (2.28), weights in the network are updated from output layer towards the input layer as the formula 'propagates' 'backwards'.
2.2.3 Inputs for Training

To generalize, the backpropagation algorithm pushes the output error of the network towards the input layer. There is no particular restriction on interpreting the meaning of this error and in more recent work, researchers have propagated reconstruction error for unsupervised training [6]. Though more commonly, this error is from the labels of one or multiple training examples fed forward through the network. The number of training examples used to calculate the error differentiates the two training techniques known as batch training and stochastic learning.

Batch Learning

In batch learning, the error is calculated for multiple training samples before the weights in the network are updated. Often this is calculated for the entire dataset before backpropagation is performed. This means the average or ‘true’ error gradient of the dataset is used to update the learning parameters. The ‘true’ error gradient refers to the fact that the error of the entire dataset is used rather than the error of a single sample of the dataset. Theoretically, this error should better reflect the error of the actual classification problem. Batch learning is also substantially easier to parallelize because the forward propagation algorithm plays a much larger role in training.

But batch learning also has many caveats. It becomes wasteful when there are redundant samples in the training set because it must iterate through the entire dataset before weight updates are performed. Calculating the error on the same sample multiple times does not yield any more error of the actual problem to the training process. Therefore it becomes wasteful to calculate the error multiple times before backpropagating [14].
2.2 Training

Stochastic Learning

In stochastic learning, the error is calculated per sample of the training set and the weights are adjusted after each sample. This training process is stochastic in nature because it does not always descend down the 'true' error surface. There is a degree of noise associated with error of a single sample because it does not reflect the error of the full problem. On the otherhand, this noise can be advantageous because it allows the descent some leniency in moving in potential directions where there could be better minima. In addition, redundant data in the training set is not an issue because redundant samples affect the training process the same way additional epochs would [14].

2.2.4 Early Stopping

Early stopping is a method of limiting the complexity of a network [4]. Often in practice, two mutually exclusive subsets of the full training set are created. One is considered a validation set while the other is a smaller training subset. The network is trained on the smaller training subset and between iterations tested on the validation set. The assumption is that the error of the validation set is a better representation of the networks performance on the test set. Overtraining can be detected when there is a degradation in classification performance on the validation set. But it must be noted that the same overtraining point may not be the same as for the actual test set. This is especially true when the size of the training set changes which occurs when the network is trained on the full training set after performing validation.
2.2.5 Second Order Methods

The learning rate parameter used in backpropagation is often adjusted dynamically to accelerate convergence in multilayer networks. Several empirical arguments have been made to support this:

- The learning rate parameter suitable for one weight may not be suitable for other weights in the network. Having one learning rate parameter may cause overadjustments in some weights and/or miniscule adjustments in others.

- Different regions of the objective function have different surface characteristics. Therefore while iterating, different learning parameters are appropriate for different regions.

- In regions where the objective function is relatively flat and unidirectional, it can be advantageous to increase the learning rate parameter to accelerate training [10].

To illustrate these points, Figure 2.7 is a surface plot of the Rosenbrock function. The z-axis has been scaled logarithmically to emphasize surface characteristics. The narrow valley shown in blue is surrounded by steep walls. With standard gradient descent, the narrow valley would be problematic because the high slopes of the walls would cause the weight vector to zig-zag between one side to the other. In addition, because the narrow valley itself has a relatively gentle slope, very little progress would be made in following the valley itself to the global minimum at (1, 1). Surface characteristics like these have been termed pathological curvature [16].

Furthermore, literature seems to suggest that gradient descent with manually defined learning rate parameters perform poorly in deeper networks, underfitting the training data. Second order methods attempt to model the local curvature of the error
2.2 Training

surface to dynamically adjust the learning rate parameters, substantially reducing the training time on large datasets. In addition, some second order methods proposed have been shown to substantially improve training quality of deep networks when compared to traditional gradient descent [16].

![Rosenbrock Function Surface Plot](image.png)

**Figure 2.7** Surface plot of the Rosenbrock function on a logarithmic scale

**Newton Raphson Method**

The Newton-Raphson method is similar to gradient descent where the minimum of a function is found through iterated descent of the objective function. Unlike gradient descent, this method takes advantage of curvature information to take more efficient steps at each iteration.

First, the objective function is approximated with a second order Taylor series expansion centered at the location of the current iteration $n$, with weights of $w(n)$. $w(0)$
2.2 Training

will usually be randomly selected during network initialization. To better show the Taylor series approximation, the differentiation is shown explicitly and the following substitution for \( w(n) \) and \( w(n + 1) \) is made:

\[
\begin{align*}
  w(n) &= w_c \\
  w(n + 1) &= w
\end{align*}
\]

\[
E(w) \approx E(w_c) + \left( \frac{\partial E(w)}{\partial w} \right)_{w=w_c}^T \Delta w + \frac{1}{2} \Delta w^T H(n) \Delta w
\]

(2.41)

where \( \Delta w \) is defined as

\[
\Delta w = w - w_c
\]

(2.42)

and

\[
\left( \frac{\partial E(w)}{\partial w} \right)_{w=w_c} = \nabla E(w_c)
\]

(2.43)

is the gradient of \( E(w) \) evaluated at \( w_c \) and the Hessian matrix \( H(n) \) is

\[
H(n) = \frac{\partial^2 E(w)}{\partial w^2} |_{w=w_c}
\]

(2.44)

also evaluated at \( w = w_c \). The Hessian matrix is a \( m \times m \) sized matrix where \( m \) is the number of trainable weights of the network. Expanding the Hessian matrix yields:

\[
H = \begin{pmatrix}
  \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\
  \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\
  \vdots & \vdots & \ddots & \vdots \\
  \frac{\partial^2 E}{\partial w_m \partial w_1} & \frac{\partial^2 E}{\partial w_m \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_m \partial w_m}
\end{pmatrix}
\]

(2.45)
Differentiating (2.41) with respect to \( w \) leads to:

\[
\frac{\partial E(w)}{\partial w} = \left( \frac{\partial E(w)}{\partial w} \bigg|_{w=w_c} \right) + H(n)\Delta w \tag{2.46}
\]

(2.41) is minimized when \( \frac{\partial E(w)}{\partial w} = 0 \). Thus setting (2.46):

\[
\left( \frac{\partial E(w)}{\partial w} \bigg|_{w=w_c} \right) + H(n)\Delta w = 0 \tag{2.47}
\]

Solving for \( \Delta w \) the weight update becomes:

\[
\Delta w = -H^{-1}(n)\nabla E(w) \tag{2.48}
\]

and substituting (2.48) into (2.42) results:

\[
w = w_c - H^{-1}(n)\nabla E(w) \tag{2.49}
\]

where \( H^{-1}(n) \) is the inverse of the Hessian matrix.

Compare (2.48) to the original weight update in (2.21). The original weight update only made use of first-order information of the error surface. This method usually results in a much slower convergence rate because the drastic simplication of the characteristics of the error surface causes zig zagging descent [10]. In (2.21), \( \eta \) is used as a very broad tuning parameter to alleviate some of these issues.

By minimizing the second order taylor approximation of the objective function, each weight in the network is provided its own tuning parameter based on the curvature of the region which the the network occupies at iteration \( n \).

The Hessian matrix has been applied in many forms to neural network training and machine learning techniques in general. LeCun used the Hessian matrix to prune neural network connections in his optimal brain damage technique [15]. In his work,
he was able to prune weights of a trained neural network based on the \textit{saliency} of that connection. This was derived from an approximated cost function with regularization of network complexity. His results showed little network degradation despite pruning a large portion of the weights in a network [15].

However despite the usefulness of the Hessian matrix in various applications, there are many caveats. The inversion of the Hessian, $H^{-1}$, can become computationally expensive for networks with many weight parameters because the size grows with a complexity of $O(m^2)$ where $m$ is the number of network weights. Another more fundamental problem is that this second order method makes certain assumptions about the curvature of the error surface around the point $w_c$. The Hessian matrix, $H(n)$, must be positive definite for the values its inverse produces to be useful for descent. For the region around $w_c$ to be convex in shape, positive definiteness is required. However, there is no guarantee the error surface has this characteristic. Furthermore there is no guarantee of convergence because the error surface may not be quadratic. Finally, the Hessian matrix may be rank deficient and therefore not even invertible [10].

\textbf{Approximations of the Hessian Matrix}

Given the caveats of computing the Hessian matrix mentioned in the previous section, researchers have found methods to approximate the Hessian matrix to yield acceptable computational costs and ensure invertability. These methods are widely used to improve the training of neural networks beyond the capabilities of stochastic gradient descent. These equations are written in a more generic form because they are considered numerical optimization techniques which can be applied to neural networks in multiple ways.
2.2 Training

Conjugate Gradient Descent

The average cost function $E_{av}$ shown in (2.19) is approximated to the quadratic form which is defined as:

$$f(x) = \frac{1}{2}x^TAx - b^Tx + c$$

(2.50)

where $A$ is a symmetric, positive definite $M \times M$ matrix, $x$ and $b$ are $M \times 1$ vectors and $c$ is a scalar constant. In this context, $A$ corresponds to the Hessian matrix and $x$ corresponds to the weight vector $w$. Functions in the quadratic form can be iteratively minimized using the method of conjugate gradient descent. Each movement on the error surface is conjugate to the direction taken in the previous iteration. Two vectors are considered conjugate by the relationship:

$$d_i^TA d_j = 0, \ i \neq j$$

(2.51)

It can be shown that for quadratic functions this method will converge in $O(M)$ time where $M$ is defined as the number of trainable parameters [14]. This type of descent amounts to a line search where descent is only along the direction of one vector $d_i$ from a set of $M$ mutually conjugate vectors $d_1, \ldots, d_M$ at each iteration [19]. After $n$ iterations, the method will have minimized (2.50) over a linear vector space spanned by $d_1, \ldots, d_n$. Each successive conjugate vector $d_n$ is not known before hand and is computed in a sequential manner [10]. Therefore, the update formula is defined as:

$$x_{n+1} = x_n + \eta_n d_n$$

(2.52)

where $x_0$ is an arbitrary starting vector and $\eta_n$ is a scalar defined by:

$$\min_{\eta} f(x_n + \eta d_n)$$

(2.53)
2.2 Training

Details about computing $\eta$ and the successive conjugate vectors $d_n$ can be found in [10].

One clear advantage of conjugate gradient is the linear time complexity because there is no direct computation of the Hessian matrix. This makes this suitable for large networks with many weight parameters. However, the main disadvantage is that this method of descent uses $E_{av}$ and therefore can only be applied to batch learning which makes it unsuitable for large redundant datasets. These types of datasets often appear in classification tasks [14].

**Levenberg-Marquardt Algorithm**

The *Levenberg-Marquardt algorithm* is a method to approximate the Hessian matrix of the error function $E(w)$ under the constraint that it must be in a *residual* (sum-of-squares) form as shown below:

$$f(x) = \frac{1}{2} \sum_{p=1}^{P} r_p^2(x)$$  \hfill (2.54)

The residuals $r_p(x)$ is defined as:

$$r_p(x) = d_p - y_p(x)$$  \hfill (2.55)

where $x = (x_1, x_2, \ldots, x_n)$. In this context, the residual $r_p(x)$ represents the error of the network at input sample $p$ from a training set of size $P$. For clarity, the generic vector $x$ is used to represent the weights and inputs of the network at sample $p$.

Taking the gradient of $f(x)$ yields:

$$\nabla f(x) = \sum_{p=1}^{P} r_p(x) \nabla r_p(x) = J(x)^T r(x)$$  \hfill (2.56)

where $J(x)$ is the Jacobian matrix with entries $\frac{\partial r_p}{\partial x_i}$. 
Taking the gradient once more and applying the product rule leads to:

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{p=1}^{P} r_p(x) \nabla^2 r_p(x)$$

(2.57)

Assuming the second term is insignificant, the following approximation can be made:

$$\nabla^2 f(x) \approx J(x)^T J(x)$$

(2.58)

$$H(x) \approx J(x)^T J(x)$$

(2.59)

This approximation of the Hessian by a squared Jacobian implies that the residual function \(r(x)\) can be approximated as a linear function with respect to its parameters \(x\) [17]. A benefit is that this ensures the Hessian matrix will be positive semi-definite.

$$\nabla^2 r_p(x) \approx 0$$

(2.60)

Because the inverse of the Hessian, \(H^{-1}\), is often the quantity desired, a diagonal approximation of the Hessian in (2.59) is used to reduce the complexity of inversion operation to \(O(W)\) instead of \(O(W^2)\) [4].

**Regularization of Levenberg Marquardt**

Levenberg added an additional regularization term to the Newton Raphson weight update in (2.49). The new weight update becomes:

$$x_{i+1} = x_i - (H + \lambda I)^{-1}(n) \nabla f(x_i)$$

(2.61)

where \(I\) is the identity matrix and \(\lambda\) is a regularization parameter. The addition of \(\lambda I\) balances the potential inaccuracies of the quadratic approximations used in second order method with some influence from standard gradient descent.
The value of $\lambda$ follows an update rule based on whether an iteration was successful (reduction in the error function). If the iteration was successful, the quadratic approximation is considered to be working and $\lambda$ is reduced to decrease the influence of standard gradient descent. But on unsuccessful iterations (increase in the error function), $\lambda$ is increased and the weights are reverted to the previous iteration as the approximations made in $H$ were not accurate [17]. Marquardt further modified this formula:

$$x_{i+1} = x_i - (H + \lambda \text{diag}[H])^{-1}(n) \nabla f(x_i)$$ (2.62)

which scales each component of $\nabla f(x_i)$ according the inverse of the curvature which is measured using $H$. LeCun uses a very similar method to update the individual weights of a neural network [13].

**Interpretation of the Hessian**

In Figure 2.8(a), the eigenvectors of the Hessian matrix represent the major and minor axis of the oval. The contours in the figure represent the surface of the quadratic approximation of the error function $E$ in (2.41). The lengths of the eigenvectors are determined by their corresponding eigenvalues and indicate the steepness of $E$ in those directions.

In [14], LeCun analyzed the distribution of the size of the eigenvalues in a multilayer network and noticed that there was a very wide spread. In his test, he found $E$ to be very steep in the direction of some eigenvectors while very flat in others creating a surface similar to the one shown in Figure 2.7. In addition, he found the eigenvalues tended to be much larger in the last layers than the first layers which caused the last layers to converge much faster than the first [14].

In a diagonal matrix, the eigenvalues are the diagonal entries with associated
2.2 Training

Figure 2.8 Countours of a quadratic E. (a) shows the eigenvectors of the Hessian in weight space $w_1, w_2$. (b) shows the eigenvectors of a diagonlized Hessian in a another space $v_1, v_2$

eigenvectors along the coordinate axes as shown in Figure 2.8(b). LeCun uses a diagonalization of the Hessian matrix to uncouple the learning rates of individual weights in a network. This enables the learning rates of each weight to be assigned independently with its corresponding eigenvalue.

In [13], LeCun uses the following update equation:

$$w_k = w_k - \epsilon_k \frac{\partial E^p}{\partial w_k}$$  \hspace{1cm} (2.63)

where $k$ is an index of weight $w_k$ in the network and step size $\epsilon_k$ is defined as:

$$\epsilon_k = \frac{\eta}{\mu + h_{kk}}$$  \hspace{1cm} (2.64)

where $h_{kk}$ is the $k^{th}$ diagonal entry of the Hessian matrix. $\eta$ denotes the global learning rate and $\mu$ is a 'hand-picked' constant which is used in a very similar fashion as $\lambda$ in
2.2 Training

(2.61) \[14\]. It prevents the step size $\epsilon_k$ from becoming too large with small values of $h_{kk}$ much in the same way $\lambda$ prevents the effects of the second order approximations from dominating the movements of the weights.

**Diagonal Approximation of the Hessian**

LeCun diagonalizes the Hessian matrix simply by setting the off-diagonal entries to zero. With this approximation, $h_{kk}$ becomes:

$$h_{kk} = \frac{\partial E^p}{\partial w_k^2}$$  \hspace{1cm} (2.65)

Furthermore, to reduce computational costs, he used ‘mini-batches’ of the training set to calculate the Hessian with the formula:

$$h_{kk} = \frac{1}{P} \sum_i^P \frac{\partial^2 E^p}{\partial w_k^2}$$  \hspace{1cm} (2.66)

Interestingly in \[13\], LeCun notes that the choice of the subset of training examples chosen did not greatly effect the Hessian estimates implying that the error surface was mainly determined by the structure of the network rather than the training examples.

It is also possible to perform the Hessian calculations in stochastic mode and therefore LeCun called this process **stochastic diagonal Levenberg-Marquardt method** \[13\].
Chapter 3

Convolutional Neural Networks

3.1 Overview

Convolutional neural networks (CNNs) are a specific type of neural network widely used in the field of image recognition. The success of this architecture has led to research in many new convolution based neural network structures which share many of the same characteristics as the original developed by Yann LeCun.

3.2 Neocognitron

Like many of the other advances in artificial neural networks, the convolutional neural network models many of the qualities found in biological neural networks. As such, it has many similarities with an earlier biologically-inspired structure known as the neocognitron. The neocognitron was developed by Kunihiko Fukushima in 1980. Its foundations were based upon the research of Hubert and Weisel who discovered locally-sensitive, orientation-selective neurons of the visual system in cats [12]. From their work, Fukushima developed the concept of a hierarchy of alternating layers of
3.2 Neocognitron

simple and complex cells (S-Cells and C-Cells) which makeup a neocognitron [7].

**S-Cells** S-Cells are considered feature extracting cells resembling the simple cells found in the primary visual cortex. These cells would respond selectively to features in its receptive field. Each cell is restricted to a local spatial regions of an image. Local features such as edges and lines with specific orientation are learned in the lower layers of the network while more global features are learned at layers closer to the output [7].

**C-Cells** C-cells represent the complex cells found in the visual cortex. The inputs of the C-cells would come from the outputs of a preceding layer of S-cells. These cells are more invariant to shifts and deformations of its input. A C-cell would respond if any of the S-cells connected to its inputs, which extract the same feature at slightly different spatial locations, are excited. This is equivalent to a spatial blurring operation [7].

A neocognitron network of these cells is created with layers of alternating S-Cells and C-Cells. S-Cells feed the inputs of a successive layer of C-Cells. Each of these layers are further divided into sub-layers called ”cell-planes” corresponding to the features to which a group of cells responds to. These groups of cells have identical input connections and only differ in the locations of their receptive fields. As will be shown layer, these cell-planes are much like the feature maps with their corresponding kernels that make up a convolutional neural network.

As input propagates through successive layers of cells, the feature extraction of the S-Cells is given positional tolerance by the next layer of C-Cells. Simple local features are extracted in the early layers and given positional tolerance. Further into the network, the deeper layers extract more complex global features which are again given some positional tolerance [7].
3.3 Structure

A convolutional neural network is made up of several parts, many of which have functionally equivalent analogues in regular fully connected neural networks. Because of the large number of variation in convolutional network structure which heavily depends on the application, the following terms will be defined in terms of the well known LeNet-5 architecture used in handwritten digit classification [13].

**Feature Map**  Feature maps represent the neurons of a convolutional neural network. The dimensionality of the feature maps is defined by the convolution operation. For image classification using two-dimensional convolution, the feature maps are two-dimensional planes of pixels where each pixel represents a neuron. All neurons in a single feature map have identical weights. This similar to the cells in “cell-planes” of the neocognitron. From a implementation perspective, the pixels of a feature map stores the output of a neuron. The location of each neuron on a feature map is defined by the region convolved on a feature map in the preceding layer as shown in Figure 3.1.

**Kernel/Receptive Field**  The kernel (or receptive field) represent the weights of the neurons in a convolutional neural network. In Figure 3.1, each kernel is shown as a different color and corresponds to a different feature map. For example, the red kernel corresponds to the second feature map. Each feature map has one corresponding kernel map. In other words, every neuron in a feature map shares the same weights. As shown in Figure 3.1, the neurons only differ in the location of the image the kernel operates on.
**Convolutional Layer**  A convolutional layer is made up of feature maps in which the convolution operation is performed on the output of the previous layer. The convolution operation is performed over the boundaries defined by the size of feature maps in the previous layer. Therefore the feature map sizes of successive layers will decrease in size. The amount of decrease is a function of the size of the kernel. A trainable bias (also shared over a single feature map) is added and the result is passed through an activation function.

For an input of size $14 \times 14$ and a kernel size of $5 \times 5$, the feature maps will be size $10 \times 10$. In LeNet-5, the image at the input layer is padded to allow the center of the kernel to reach the edges of the image. But in [20], it was shown that this did not improve performance significantly.

**Subsampling Layer**  A subsampling layer is made up of feature maps in which the subsampling operation is performed on its input from the previous layer. The subsampling operation is defined as a summation over non-overlapping blocks from the previous feature maps that are multiplied by a trainable coefficient and added to a trainable bias. This result is then passed through an activation function.

**Connection Table**  The connection tables map the existing connections from one layer of feature maps to the next. Unlike fully connected neural networks, convolutional networks often have specially defined connections between layers. These connections exist at the feature map level. The output of all neurons in one feature map will act as input to the feature map at the next layer if a connection is defined between them.

Aside from avoiding the computational cost of having fully connected layers, LeCun also reasons that this breaks the symmetry in the network. Different feature maps are forced to extract different features because they have inputs from different
3.3 Structure

The connection table can be represented by a matrix $B$ where each row corresponds to a feature map of one layer and each column corresponds to a feature map in the next. If the $i^{th}$ feature map has a connection to the $j^{th}$ feature map in the next layer, the $b_{ij}$ entry of $B$ is 1 and 0 otherwise.

$$A = \begin{pmatrix} O & B \\ B^T & O \end{pmatrix}$$ (3.1)

$B$ represents the meaningful part of an adjacency matrix $A$ of a bipartite graph. $B$ is a $M \times N$ matrix where $M$ is the number of feature maps in one layer and $N$ is the number of feature maps in the next. $O$ represents an all zero matrix.

**Figure 3.1** First layer of a Convolutional Neural Network
3.4 Feedforward Propagation

The feedforward process for LeNet-5 is illustrated in Figure 3.2. There is a hierarchy of alternating convolutional and subsampling layers which operate over the entire image. The output of this hierarchy is then fed into a fully connected neural network which begins at layer C5. The final $5 \times 5$ convolution kernel maps the features maps of S4 to a $1 \times 1$ pixel.

As shown in Figure 3.2, while moving from the input towards the output in the feature maps layers, the feature maps not only become smaller but also more numerous. This results in more convolutional kernels and less weight sharing. Therefore the weights in the early layers of the network can have a very large impact on the rest of the network because they are shared among more neurons [13].

Figure 3.3 illustrates the region, shown as the largest solid square, of the input image, shown by the dashed square, which contributes to the input of a single neuron located at the top left corner of a feature map in layer S4. This region becomes larger
still in the layer after S4 as the network is expected to learn more global features of the input image. The $1 \times 1$ pixel in C5 covers the entire input image. Because of the subsampling and convolution of the previous layers, these global features are constrained to be invariant to small deformations such as shift and scale.

Figure 3.3 also illustrates that spatial locality is still preserved as the input is propagated through the network. The top left corner of a pixel in S4 'sees' a much larger region but this region is still located in the top left corner of the input. This constrains neurons in the first layers of the network to learn more localized features while neurons in later layers learn features which are globally present in the input image.

![Figure 3.3](image)

**Figure 3.3** As the feature maps become smaller in layers closer to the output, the portion of the image which contributes to its input becomes larger.

One of the aspects which make convolutional neural networks so powerful is its ability to learn from nearly unprocessed image data. This removes the manual task of choosing which features to present to the network. The feature map layers can be
considered the feature extraction stage where the network is able to automatically extract features from an image during training. These extracted features are then fed into a fully connected neural network which can be viewed as a classification stage.

The connection between the last hidden layer and the output layer of LeNet-5 is labeled as Gaussian connections because the output neurons use a euclidean radial basis function whose output can be interpreted as the unnormalized negative log-likelihood of a Gaussian distribution [13].

### 3.5 Backpropagation

The shared weight architecture of CNNs makes the process of backpropagation distinct from that of regular neural networks. But this difference is mostly from an implementation perspective as the concepts described in section 2.2.2 are still applicable.

Each neuron in a CNN is initially treated as though it possessed its own weight parameters. Therefore, the error gradient is calculated separately for each individual neuron. The weight sharing constraint is applied once these gradients are calculated and the gradients of the neurons which share the same weight parameters are combined. This is shown in (3.3).

Let $V_k$ be the set of index pairs $(i, j)$ where the connection between neuron $i$ and another neuron $j$ share the same weight $w_k$. The constraint can then be defined as:

$$u_{ij} = w_k \quad \forall (i, j) \in V_k \quad (3.2)$$

$$\frac{\partial E^p}{\partial w_k} = \sum_{(i,j)\in V_k} \frac{\partial E^p}{\partial u_{ij}} \quad (3.3)$$
where $u_{ij}$ is the weight between neuron $i$ and neuron $j$. The weight update equation for weight sharing becomes:

$$w_k = w_k - \epsilon_k \frac{\partial E^p}{\partial w_k}$$  \hspace{1cm} (3.4)

where $\frac{\partial E^p}{\partial w_k}$ is defined as the summation of error gradients $\frac{\partial E^p}{\partial u_{ij}}$ in (3.3). The same approach is done for propagating the Hessian defined in (2.66).

$$\frac{\partial^2 E^p}{\partial w^2_{kk}} = \sum_{(i,j) \in V_k} \frac{\partial^2 E^p}{\partial u^2_{ij}}$$  \hspace{1cm} (3.5)

LeCun describes this sharing constraint as an 'ill-conditioning' of the error surface because the high amount of weight sharing in the input layers amplifies their influence on the output [13]. Furthermore, this derivation illustrates the importance of the weights in the first layers because these summations become larger when more weights are shared. This can become problematic when the second derivative becomes very large when compared to other less shared weights in the network. Therefore, LeCun uses $\mu$ in (2.64) as a 'safety factor' to prevent weights with small second derivative estimates from taking excessively large steps [13].

The shared weight architecture of CNNs may help alleviate the vanishing gradient problem described in [3]. The vanishing gradients problem occurs in deep networks when the movements of weights in the earlier layers of the network become extremely small relative to the movements of weights in the layers closer to the output. Because the gradients of shared weights are summed in (3.3), the weight movements could be made more substantial where they would normally diminish as backpropagation moves toward the input of the network.
Chapter 4

Evolutionary Algorithms

4.1 Overview

Unlike the analytic optimization methods discussed in section 2.2.2, evolutionary algorithms are biologically inspired. They imitate the principles found in nature. Conceptually, these algorithms are much simpler and more straightforward to understand than gradient descent for large complex systems. The scope of their application is very wide because they are not as mathematically constrained as the other methods which tend to rely on certain conditions such as differentiability [9]. In addition such algorithms also tend to be easier to parallelize because they utilize multiple independent searches.

*Genetic algorithms* represents a subset of evolutionary algorithms. They were originally developed by John Holland in 1975 and were applied to function optimization by De Jong in the same year [9]. Due to their success, genetic algorithms started gaining popularity for solving many problems which had complicated cost surfaces. Figure 4.1 shows the typical stages of a genetic algorithm.
4.1 Overview

Figure 4.1 Diagram of the stages of a genetic algorithm

4.1.1 Population

First, a population of potential candidate solutions is generated. Each candidate is represented by a chromosome. A chromosome is made up of the parameters $x_i$ in the system which are to be optimized. These parameters describe the genotype of the system while the actual traits exhibited by the system are considered its phenotype [9]. The chromosome parameters are often represented as an array of size $M$ optimization parameters. Therefore, this size defines the dimensionality of the optimization problem ($M$-dimensional).

The size of the population is application dependent. Larger populations allow the algorithm to become more explorative in search space than smaller ones. But this comes at the cost of increased computational complexity. Every candidate in a population must have its fitness evaluated and very often this operation is not computationally efficient. Therefore the population size is often limited by the resources available.
4.1 Overview

4.1.2 Fitness

Fitness forms the basis of the natural selection aspect of a genetic algorithm. The fitness function determines a chromosome’s chances of survival to the next generation. For optimization problems, fitness can be considered the opposite of a cost or error function. More ‘fit’ chromosomes have a lower error. The most fit chromosomes are non-deterministically chosen for mutation and/or crossover while the least fit are probabilistically eliminated from the population. This non-deterministic selection makes evolutionary search a stochastic process where chromosomes with high fitness are likely but not guaranteed to chosen to continue to the next generation.

Unlike the error functions mentioned in section 2.2.2, a fitness function need not be differentiable or even continuous. Both the cost function and the parameters to optimize it can take on discrete values. Furthermore, the fitness function can be multiobjective, enabling the algorithm to find a balance between two potentially conflicting optimization goals [9].

4.1.3 Crossover/Mutations

The chromosomes which survive the fitness selection process are used to create new chromosomes for the next generation. This is done with a combination of crossover and mutation operations. Crossover, also known as mating, is specific to genetic algorithms. It refers to the creation of a new chromosome (offspring) with inherited characteristics from parent chromosomes. The parents are considered to have ‘fit’ characteristics which could potentially be combined to create an even more ‘fit’ offspring.

Mutations are used to prevent genetic algorithms from converging too quickly to local optima. It enables the genetic algorithm to search beyond the possible chro-
mosomes in the population pool. For continuous genetic algorithms with real valued chromosomes \( x_i \in \mathbb{R} \), mutations are probabilistic changes in the parameters which makeup the chromosome. When a chromosome is chosen for mutation, each parameter which makes up the chromosome is probabilistically changed or ‘mutated’. The mutation rate defines the probability that a parameter will change.

### 4.2 Self-Adaptive Evolution

Parameters in evolution such as mutation rate and probability of mutation are very problem specific. Often in cases where an evolutionary algorithm is being used, very little a priori knowledge about the problem is known and finding appropriate values for these parameters is difficult. Such parameters can introduce an additional bias in the evolutionary search and an improper bias can lead to poor convergence. A common strategy solve this issue is to evolve the evolution parameters along with the original optimization parameters. This method is known as self-adaptive evolution. Types of self-adaptive evolution are classified by their scope of affect with respect to the population of candidates. Population-level adaptation evolve parameters which affect the mutation operators of the entire population. In contrast, individual-level adaptation affect the mutation operators for each individual. Component-level adaptation goes even further, adapting the mutation operators of each component in the chromosome [2].

In the context of self-adaptive evolution, these parameters are called strategy parameters. An example of the mutation rule for a strategy parameter in component-level adaption is shown in (4.2).

\[
x_i' = x_i + \eta_i N(0, 1) \tag{4.1}
\]
\[
\eta_i' = \eta_i \exp(\tau N(0, 1) + \tau' N_i(0, 1)) \tag{4.2}
\]
In (4.1), $x_i$ is a component which undergoes a Gaussian mutation with zero-mean. The variance is defined by the adaptive strategy parameter $\eta_i$. $\tau$ and $\tau'$ are user defined parameters. For network weight evolution, [22] defined these values as:

\[
\tau = \left( \sqrt{2\sqrt{n}} \right)^{-1} \tag{4.3}
\]
\[
\tau' = (\sqrt{2n})^{-1} \tag{4.4}
\]

where $n$ is the number of offspring. Such adaptive evolution distributes new candidates in the population in the direction which had provided improvements for the candidates’ ancestors [2].

### 4.3 Comparison to Gradient Descent

The global minimum is the smallest of the local minima. But in almost all situations referring to local minima, the global minimum is usually not included. In fact local minima usually refers to the set of minima that is not the global minimum. [18] Genetic algorithms are considered a global optimization strategy. Even though they do not guarantee convergence on a global minimum, the non-deterministic search strategy it employs explores a much larger area of the cost surface than many other algorithms.

When comparing the qualities of optimization methods it is useful to describe the differences between "exploratative" and "exploitative". Methods which use local information and aggressively exploit this knowledge to reach convergence are considered exploitative. Gradient descent is considered exploitative because it uses gradient information to descend down the cost surface. Even more exploitative are the second order methods such as Newton-Raphson which use second order curvature information to optimize these descents. In another example, stochastic learning is considered
4.3 Comparison to Gradient Descent

more explorative than batch training because stochastic learning accepts the discrepancy of representing the error of a single training example as the error for the 'true' cost function. LeCun reasons that this may be one of the reasons stochastic learning tends to converge better minima than batch training. [14]

On the other end, genetic algorithms tend to use very little local information and instead takes a more heuristic approach to search. Genetic algorithms are much more explorative than the hill-climbing strategies because they explore a population of potential solutions rather than attempting to improve upon a single solution (which is often generated randomly at initialization). Adding the non-deterministic aspect of probabilistically eliminating and crossing/mutating candidates enables genetic algorithms to accept a certain degree of 'mistake'. There is no guarantee in genetic algorithms that the new candidates of the next generation will perform better than previously. However the motivation is that in the long term, the genetic algorithm becomes more likely to improve as it continues non-deterministically searching.

The obvious tradeoff to the explorative methods is computational resources and time. The same benefits genetic algorithms provide in its explorative strategy is also its largest caveat. Having such a broad search space makes genetic algorithms often infeasible for many applications, especially when the cost of the evaluating fitness is high. Therefore hybrid approaches of managing the benefits of both hill-climbing strategy with genetic algorithm strategies have been proposed. Many of these techniques have been shown to give very promising results without the large increase in computational resources [18].
4.4 Application to Neural Networks

The application of genetic algorithms to neural network training seems viable because the error surface of these systems are not necessarily ‘well-behaved’. In fact, the characteristics of these surfaces are not even fully understood. The lack of reliability in convergence has been a very large issue in the application of neural networks to many problems. The unpredictable nature of their performance has led much research to shift to learning machines with much more well defined cost surfaces such as support vector machines.

Furthermore, hill climbing methods only optimize the weights of a network with a given topology. Network topology is often manually defined based on a trial and error basis even though it plays an extremely important role in the performance of the network and can drastically affect the convergence characteristics of hill climbing. There has been a great amount of research in the application of genetic algorithms in designing neural network architectures [22].

Permutation Problem

There are many issues in applying genetic algorithms to neural networks. For fully connected networks, an issue known as the permutation problem exists. A fully connected neural network can be represented in many functionally equivalent ways. Because each node is fully connected to the next layer, swapping any number of nodes with other nodes in the same hidden layer still gives an identical network structure. The number of possible functionally equivalent permutations of a fully connected network with $L$ hidden layers is given by:

$$\prod_{i=1}^{L} (n_i!)$$  \hspace{1cm} (4.5)

where $n_i$ is the number of nodes in hidden layer $i$. 
4.4 Application to Neural Networks

With respect to genetic algorithms, this issue can theoretically cause problems with crossover. Because chromosomes apply absolute indices to weights in a neural network, two networks with functionally equivalent weights in different locations could be crossed. This would produce an offspring which would have the same inherited quality in more than one location in its chromosome. Such an offspring is not very likely be an improvement over its parents. This is especially true for neural networks because weights in a network tend to be very correlated with one another and share many interdependencies. This had led some researchers to argue towards evolutionary algorithms for neural networks which only use mutations and does not attempt any crossovers. On the other hand, there has been some research which seems to suggest that the issues caused by the permutation problem may be exaggerated [21].

Permutation Problem in Convolutional Neural Networks

At the neuron level, convolutional neural networks are not as susceptible to the permutation problem as fully connected networks. This is because neurons (which represent pixels in a feature map) cannot be swapped to give a functionally equivalent network\(^1\). The locality of the neurons defines the connections it will have with neurons in the next layer. This can be seen in Figure 3.3 where the location of a neuron in S4 defined the local region of the input it operated on. However, given a full connection table (a matrix where all entries are 1), convolutional neural networks can suffer from the permutation problem at the feature map level.

However, it is difficult to judge whether crossing at the neuron (pixel) level or

\(^1\)Note that implementing crossover would actually involve crossing the kernel maps and not the feature maps. In this discussion, pixels in a feature map have their own weights, but these weights are constrained to be equal with those of other connections to enforce the 'sharing' effect. These connections are mathematically defined in (3.2).
crossing at the feature map level would yield better results. One can argue that that crossing at the neuron level would destroy the correlated effect weights in a given locale may have with each other. For example, if two parent CNNs were crossed at the neuron level to yield an offspring, that offspring would have weights of one parent in the neighbourhood of weights of the other. These weights could come from kernels which extract completely different features and therefore it would not yield any benefit to have these weights mixed. With this reasoning, it can be argued that crossing at the feature map level would preserve the functionality of the kernels when used by the offspring while also accepting the risk of the permutation problem associated with it.
Chapter 5

Implementation

5.1 MNIST Dataset

Modified NIST (MNIST) is a handwritten digit database derived from the National Institute of Standards and Technology’s (NIST) Special Database 3 (SD-3) and Special Database 1 (SD-1). NIST designated SD-3 as the training set and SD-1 as the test set. LeCun created the MNIST database because of a noticeable disparity in the handwriting quality between the two NIST sets. This can be attributed to the fact that SD-3 was collected from Census Bureau employees while SD-1 was sampled from high school students. To ensure results would be independent of the choice in training and test sets, he created new training and test sets which were a mix of both SD-3 and SD-1 [13]. The MNIST dataset has become a common benchmark for finding the performance of many machine learning methods including support vector machines and deep belief networks.

The images in MNIST are $28 \times 28$ grayscale samples of the handwritten digits 0-9. A sample of the training set is shown in Figure 5.1. The training set is composed of 60,000 examples while the test set contains 10,000 examples. The identities of the
5.1 MNIST Dataset

500 writers from SD-1 was determined and the database was split into two disjoint sets of 250 writers each. The first set of 250 writers were placed in the training set. In addition, because the writers in SD-3 appeared in sequential blocks, images from SD-3 were simply added to the training set to complete the full set of 60,000. The first 5,000 examples of the test set were taken from SD-3 starting at pattern #35,000 and are cleaner than the last 5,000 which were taken from SD-1 [13]. The class distributions of the training and test sets is shown in Figure 5.2.

Additional subsets of the training set were taken to observe the affects of scaling between small and large sets. Subsets of 200, 500, 2000, and 4000 examples were taken from the training set. Each subset was take at the beginning of the MNIST training dataset. To ensure validity of results, the full MNIST test set was still used. The class distributions of each training subset is shown in Figure 5.3. Notice that the class distributions are relatively unbalanced in the smaller sets and become more uniformly distributed as the subsets become larger.

Each subset created was further divided into two disjoint sets of equal size. For
Figure 5.2 Class distribution of the entire MNIST training set and test set.

example, the MNIST 500 sample subset was divided into two sets of 250 examples. These smaller sets were used as both cross validation and training sets. The distribution of these cross validation/training sets is also shown in Figure 5.3. More detail of the purpose of these subsets is given in the section 5.4.

5.2 EBLearn

EBLearn (Energy Based Learning) is a machine learning library currently in development at the Computational & Biological Learning Laboratory (CBLL) [1]. CBLL is run by Yann LeCun at the Courant Institute of Mathematical Sciences at New York University. The library is written in C++ and modeled after Lush, another object oriented machine learning library primarily based on the Lisp language. EBLearn is capable of implementing many machine learning models including standard neural networks and CNNs. Although still in the pre-alpha stages of development, this library was chosen for the following reasons:
5.2 EB\textit{L}earn

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.3}
\caption{Class distribution of the various subsets of the MNIST training set used. The first column shows the class distribution of each subset of MNIST. The second and third column show the class distribution of these subsets divided into two disjoint subsets.}
\end{figure}

**Highly Modularized** Following the principles of object oriented-design, the components of EB\textit{L}earn are highly modularized to enable the construction of many different types of networks.

**Implementation of Stochastic Diagonal Levenberg-Marquardt Method** The stochastic diagonal Levenberg-Marquardt method, discussed in section 2.2.5, is used by EB\textit{L}earn to improve convergence when training neural networks.
5.3 Hybrid Method for Evolving Weights

**Speed**  EBLearn was found to be orders of magnitude faster than other implementations based on interpreted languages such as MATLAB.

**Strong Training**  EBLearn exhibits very good training results and appeared to have relatively consistent training characteristics. Results for many tests have been easily reproducible.

5.3 Hybrid Method for Evolving Weights

**CNN Structure**  A modified LeNet-5 structure has been used for testing. The modified structure excludes the RBF layer located at the output. The results achieved on the full MNIST training set with this modified structure are still comparable to the results given in [13] which used the standard LeNet-5 structure including the RBF layer. With this change, the 84 nodes in layer F6 have been reduced to 10 nodes to serve as the output of the network. Additionally, the final convolutional layer, C5, has been reduced to 11 nodes (one-pixel feature maps) which appears to have better classification performance than the original network with 120 nodes for much smaller training sets of 200 samples or less.

**Chromosome Representation**  Chromosomes have been represented as a list of all trainable weight parameters of a CNN. These trainable parameters are composed of the weights and biases of all layers. This includes the convolutional, subsampling and fully connected layers. In addition to the trainable parameters of the network, the mutation factor and probability of mutation have also been encoded into the chromosome as a form of self-adaptive evolution.
5.3 Hybrid Method for Evolving Weights

**Mutation**  The mutation factor $\alpha$ is a multiplier in the formula:

$$x_i' = x_i + \alpha U(-1,1)x_i$$

(5.1)

where $U(-1,1)$ is a number sampled from the uniform distribution from $-1$ to $1$. The mutation factor controls the amount of mutation each parameter can undergo when it is selected to mutate. The probability of mutation approximates the fraction of the chromosome which will be mutated. Each value in the chromosome has been represented by a real number in double precision format. Given a static network topology, chromosome lengths are identical across all individuals of the population.

**Fitness Function**  The total mean squared error of the network over the entire training set has been used to evaluate the fitness of each chromosome. Since the goal is to minimize these error values, more 'fit' individuals would have smaller error values.

**Selection Function**  A tournament style selection algorithm was used to rank the chromosomes after fitness evaluation. Each chromosome is matched with $q$ randomly and uniformly chosen chromosomes from the population. For each match, the fitness value between the two are compared and the chromosome deemed more 'fit' receives a 'win'. This is performed on each individual in the population and chromosomes are ranked in descending order of most wins. With this type of selection, one can adjust the greediness of the evolutionary algorithm to more deterministically select the most fit individuals to move on to the next generation simply by increasing the number of matches. Alternatively, the selection can be made less deterministic with less matches. A more formal description of the tournament procedure is given below [22]:

1. Select $q$ opponents randomly and uniformly from the population.
2. Compare the fitness of the individual with each opponent. If an individual’s fitness is no smaller than that of its opponents, award the individual a ‘win’.

3. Repeat this process for every individual in the population.

4. Sort the population in descending order by number of ‘wins’.

The top non-deterministically ranked individual will have the most wins. If there is a tie for the most wins, one of the individuals is arbitrarily chosen.

**Crossover** In the context of neural networks, random crossover is unlikely to produce successful offspring. Yao notes that crossover may contradict the very nature of how neural networks function [22]. Crossover relies on ”building blocks” of traits to exist such that fitness is improved when these traits are combined. On the other hand, neural networks rely on a ‘distributed’ representation, making these traits possibly inseparable at the weight level [22].

To demonstrate the effects of crossover, one-point crossover has been used to combine two parent chromosomes. Only one crossover point has been chosen to help preserve many of the possibly inseparable traits encoded into the chromosome. Two parents are randomly chosen from the top half of the sorted population to be crossed. The crossing point is chosen from a discrete uniform random distribution, $U(0, n)$, where $n$ is the length of the chromosome. A blending method is used as the crossover formula:

$$x_c = \beta x_{p1} + (1 - \beta) x_{p2}$$

The index $c$ is the offspring of parents $p1$ and $p2$. The blending parameter $\beta$ is a uniform random number $U(0, 1)$. This blending formula is applied for each chromosome parameter $x_i$ for $i < CP$. For $i \geq CP$, the parameters of $p1$ are encoded into the rest of the offspring’s chromosome.
Initially, the network is trained to convergence with backpropagation using stochastic diagonal Levenberg-Marquardt as a second order method. After convergence, the network is mutated $n - 1$ times to create population of $n$ individuals where one chromosome contains the original non-mutated state of the network. The fitness of each individual is evaluated and the tournament selection process ranks them non-deterministically. The top $n$ chromosomes are chosen to move on to the next generation and an additional $n$ chromosomes are created by mutating/crossing from these chosen individuals. Therefore the population pool to select from contains $2n$ chromosomes. In the first generation, all $n$ chromosomes will survive to the next generation. In future generations, the number of newly created individuals which survives varies depending on their fitness relative to those which survived in the previous generation. Keeping fit individuals from previous generations prevents generations of no observed improvement from destroying any progress made in the past.

## 5.4 Hybrid Method for Evolving Network Architecture

A relatively indirect coding scheme has been used for the chromosome where the number of feature maps, nodes and connection tables between select layers are specified by the chromosome. In contrast, a very direct encoding scheme would specify every possible connection between every node in the network. This scheme could encode feedforward as well as recurrent networks [22]. However, it has been claimed that an indirect coding scheme is more biologically plausible because studies in neuroscience have shown it is impossible for chromosomes to specify the entire nervous system [22]. Only the connection tables between layers S2 to C3 and between S4 to C5 are encoded
in the chromosome while connections between all other layers are left fully connected. The encoded connection tables are forced to have the following structure:

\[
\begin{pmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & X & X & \ldots & X \\
1 & X & X & \ldots & X \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & X & X & \ldots & X
\end{pmatrix}
\]

X represents an entry in the connection table which was encoded into the chromosome while 1 represents a permanent connection. This constraint removes any chance of orphaned nodes which would create a degenerate network that would be equivalent to a structure structure with fewer nodes and a different chromosome representation. A caveat is that this constraint also limits some of the search space of possible network architectures.

When additional nodes are added, they are initialized as fully connected to the layers preceding and succeeding it. Therefore, the new rows and/or columns of the connection tables corresponding to the new nodes are initially set to 1. Only the following generations will be able to evolve these connections. When nodes are removed, the last rows and/or columns are also removed from the connection table.

Note that there are two possible issues arising from this evolution process. First, there is an arbitrary coupling created between the evolution of the number of feature maps and the initialization of the connection tables. Second, the columns and/or rows chosen for removal should be random rather than at the ends of the connection table. A more generalized implementation is discussed in Chapter 8.

One major issue in evolving network topologies separately from the weight vector is the non-deterministic fitness evaluation. This is described as a noisy fitness function in [22]. The noise is caused by the one-to-many mapping of one chromosome to several
different fitness values. This occurs because the weight vector is not defined by the chromosome but still remains an important parameter to fitness evaluation. Therefore to partially remedy this issue, parameters of the random initialization formula, $v$ and $e$, of the weight vector are evolved. The random initialization formula is defined as:

$$w_{\text{init}} = U\left(-\frac{v}{f_e}, \frac{v}{f_e}\right)$$  \hspace{1cm} (5.3)

The parameter $f$ is defined as the 'fan in' parameter which is the number of incoming connections a node possesses. For example, a node in a feature map with a $5 \times 5$ kernel would have a fan in of 25. This allows the chromosome to indirectly define some weight information into the network structure. Additionally, it enables customized weight initialization boundaries for a particular network architecture. [3] proposed a specific layer by layer initialization formula for networks using hyperbolic tangent activations.

The $\eta$ parameter in (2.64) is also evolved. Evolving the learning parameters such as $\eta$ with network architecture has been done in the past [8]. This approach takes advantage of possible relationships between the backpropagation parameters and the network architecture [22].

The ranges of these parameters has been constrained for obvious computational reasons. Having an unbounded range of feature maps creates the potential for extremely large and complex networks with very minimal fitness gain. In addition, there would be far fewer generations progressed in the same amount of time.

The evolution process is described by the diagram in Figure 5.4. First the population of candidate networks are randomly initialized within user set boundaries. These parameters include the number of feature maps in layers C1, C3 and C5. Since each subsampling map is directly connected with its respective convolution map as shown in Figure 3.2, the number of subsampling maps in each layer is equal to the number of convolution maps preceding it. The $\eta$, $v$, and $e$ parameters are randomly
5.4 Hybrid Method for Evolving Network Architecture

Figure 5.4 Block diagram of the hybrid architecture evolution process.

initialized but were not bound during evolution because their values did not impose any additional computational complexity. The connection tables between layers S2 to C3 and S4 to C5 are initialized as fully connected.

After initialization, each individual in the population is trained with backpropagation for a user-defined number of epochs on a subset of the training set. These subsets are defined in Figure 5.3. The Stochastic Diagonal Levenberg-Marquardt Method has been used as a second order method to help accelerate training. 100 samples from the training subset has been used to approximate the Hessian before each epoch of training. The 'safety parameter' $\mu$ in (2.64) is fixed at 0.02.

The fitness is determined by the total error on the cross validation subset of the
training set. Between generations, the training subset and cross validation subset are switched. This enables candidates to be evaluated on the full training set every two generations, but at the expense of an increase in fitness evaluation noise. Tournament selection described in the previous section is used to non-deterministically rank the population. The top half of the population is chosen to survive to the next generation and produce offspring for that generation.

The mutation operator in (5.1) is used with self-adapting $\alpha$ at the component-level for each parameter except the connection table. The binary format of the connection tables eliminates the need for a mutation factor. Each connection is switched from 1 to 0 or vice versa when it is mutated. The mutation of each entry in the connection tables is treated independently for mutation. Gaussian mutations are also tested but appear to perform poorly and would not converge successfully during poor initializations.
Chapter 6

Results

6.1 Analysis of Weight Evolution

The crossover operator led to early convergence and provided little benefit in improving fitness. Because it led to early convergence on local minima, the crossover was not used during evolution. Instead, a similar approach to EPNet described in [22] has been implemented where mutation was the primary search operator used to adjust the chromosomes between each generation.

![Figure 6.1](image.png)

**Figure 6.1** Total Energy on 200 sample MNIST training subset for 6000 generations of weight evolution.
6.1 Analysis of Weight Evolution

Figure 6.2 Total Energy on 10000 sample MNIST test set sampled at every 100 generations of weight evolution.

<table>
<thead>
<tr>
<th>Training</th>
<th>Classification Error</th>
<th>Total Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GD + SDLM</td>
<td>13.19%</td>
<td>0.653656</td>
</tr>
<tr>
<td>GD + SDLM + EA</td>
<td>13.5%</td>
<td>0.66367</td>
</tr>
</tbody>
</table>

Table 6.1 Gradient Descent with Stochastic Diagonal Levenberg-Marquardt method (GD + SDLM) compared with an additional 6000 generations of weight evolution (GD + SDLM + EA).

As shown in Figure 6.2, changes in the error on the training set do not appear to improve the error on the test set. In fact, it appears that overtraining has occurred as the energy of the test set increased during evolution. While the evolution algorithm is able to decrease the error on the training set, Table 6.1 shows this did not improve the network’s classification ability or error on the test set. This reflects the diminishing returns affect when the training error becomes less reflective of performance on the test data.
6.2 Analysis of Architecture Evolution

The stopping criterion for training is usually chosen at a specific epoch. But the
decision to stop after iterating through an entire training set or stopping at the a
particular iteration within the training set is often based on preference. When using
stochastic gradient descent, the difference is a matter of granularity in choosing the
point to end training. One can view epochs following the first as simply a set of
iterations without the introduction of any new training data.

In many cases, the improvements seen between epochs degrades rather quickly
after the first epoch. Without the introduction of new training data, the following
epochs serve to fine tune the parameters of the network. Experiments on MNIST
with validation sets have shown that the most substantial improvements in network
performance occur at the first epoch. This observation can be further extended to
the iteration level, especially if the training set is highly redundant as explained in
section 2.2.3.

However, it must be noted that the first epoch does not necessarily provide a good
indication of final network performance. Instances can be shown where slower con-
verging networks often have better absolute classification performance when allowed
to fully converge. But this evidence does not eliminate the possibility of an unknown
network which can reach the same classification performance in fewer iterations.

Many of the parameter choices have been intended to limit computational com-
plexity in a nonparallelized environment. Results are shown with one epoch performed
at the training stage in Figure 5.4. For each epoch, the network has been trained
on an equal number of instances of each class. In the case of unbalanced datasets,
the same samples would be reused in a single epoch if it had fewer examples than
the most common class. This prevents potential bias of the network based on the
training set distribution which for small sets may not represent the true class distribution. The limits of the allowable number of feature maps for layers C1, C3 and C5 has been set to be between 2 to 40, 2 to 40, and 2 to 120 respectively. For each candidate, the number of feature maps was uniformly randomly initialized between those boundaries. In addition, $\eta, f, v$ have been randomly initialized based on the distributions $U(0, .1), U(0, 1)$ and $U(0, 1)$ respectively. The probability to mutate and mutation factor have been both set to 70%. The mutation factor is allowed to self-adaptively adjust during evolution while the probability of mutation remains fixed for the duration of the tests. The population pool contained 20 candidates with the top 10 ranked candidates surviving to the next generation. The tournament selection process uses $q = 3$ opponents for each candidate.

Results are tested at the end of every generation after the selection process. Because convergence is relatively quick, only the first 100 generations are shown. In Figure 5.4 between the tournament selection and mutation stages, the top nondeterministically ranked network is selected to be evaluated on the full MNIST test set. In each experiment, the network is trained on full subsets shown in 5.3 for 20 epochs before testing on the full 10,000 sample MNIST test set.

Figure 6.4 shows the MNIST test set performance for different methods of dividing the training set into a smaller training and validation sets. Swapping refers to switching the smaller subsets in 5.3 during each generation. Shuffle refers to uniformly randomly shuffling the order of the initial training set before the beginning of each generation and dividing the shuffled set into a training and cross validation set. The ratio of training/cross validation sizes were 2000/2000 and 500/3500.

As seen in Figure 6.3, convergence to trainable networks occurs relatively quickly. Faster convergence is seen with the swapping 2000/2000 strategy than the shuffle 2000/2000 or shuffle 500/3500 methods. This is likely because shuffling introduces
6.2 Analysis of Architecture Evolution

Figure 6.3 Performance of different training and validation strategies on the MNIST test set with 4000 MNIST training samples

more noise in fitness evaluation leading to difficulty in differentiating networks that are somewhat trainable and networks which were considered untrainable. An untrainable network is a network which makes very little or no progress in improving the classification error beyond random guessing on the validation set. Furthermore, these networks often show little to no progress in converging on the training set.

The presence of these untrainable networks in the search space of evolution did pose problems. At the initial generations of exploration, it is difficult to discern the difference in fitness between a trainable network and an untrainable network. The total error energy of a trainable network with unoptimized parameters is much closer to the fitness of an untrainable network whose outputs tend to be very small. These small outputs prevent the error energy from becoming large despite no classification ability. When population size is relatively small while candidate diversity is relatively large, untrainable networks which reduce the error to a values comparable or even lower than more promising networks tend to survive and reproduce in future generations. Therefore at initialization, the potential for very deep local minima exist. But this issue can resolved by relying on the percentage classification error during initial generations rather than error energy. In addition, some a priori knowledge of parame-
ter boundaries and seeding initialization does help greatly in increasing the likelihood of convergence. But this will be at the expense of introducing more exploration bias.

![Figure 6.4](image)

**Figure 6.4** Performance starting at generation 14 of different training and validation methods on the MNIST test set

During the first few generations, the drastic difference in total error energy between an unoptimized network and one much closer to convergence, as illustrated in Figure 6.3, makes comparison between the different training/cross validation methods difficult. For clarity, Figure 6.4 shows the error energy on the test set after generation 14. Generation 14 is the point when all 3 methods converge on trainable networks. Although relatively noisy because of the random weight reinitialization, Figure 6.4 shows swapping 2000/2000 having lower error energy than both shuffle 2000/2000 and shuffle 500/3500. Table 6.2 shows the mean classification error and error energy from generations 14 to 100 on the test set for the 3 training/cross validation methods. The mean values for 100 random reinitializations of the LeNet-5 architecture is also shown in the table. The parameters to initialize LeNet-5 are shown in Table 6.3.

As shown in Table 6.2, the total energy error of the three evolved networks is substantially lower than LeNet-5 after 20 epochs of training. But interestingly, the classification error of LeNet-5 was better than shuffle 2000/2000 and shuffle 500/3500 strategies. Despite having a higher energy, it is possible that the LeNet-5 architecture
6.2 Analysis of Architecture Evolution

<table>
<thead>
<tr>
<th>Training/Validation Type</th>
<th>Classification Error</th>
<th>Total Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swapping 2000/2000</td>
<td>2.14%</td>
<td>0.0869</td>
</tr>
<tr>
<td>Shuffle 2000/2000</td>
<td>2.35%</td>
<td>0.0948</td>
</tr>
<tr>
<td>Shuffle 500/3500</td>
<td>2.33%</td>
<td>0.0946</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>2.23%</td>
<td>0.1217</td>
</tr>
</tbody>
</table>

Table 6.2 Mean values of Classification Error and Total Energy of MNIST test set after 20 epochs of training on the 4000 sample MNIST subset.

is able to generalize better at the 20th epoch. The possibility still exists that the evolved networks may be overtraining.

The difference in performance can also be seen in the error on the training set at the 20th epoch as illustrated in Figure 6.5. Here, the difference is more pronounced and shows swapping the training and validation sets leads to parameters with better convergence on the training set than other methods.

![Figure 6.5 Error on the training set at the 20th (last) epoch on the 4000 sample MNIST training subset.](image)

Interestingly, the evolutionary algorithm created networks which are able to converge on the training set with less epochs and often less trainable parameters than LeNet-5 as shown in Figure 6.6. Aside from the boundaries of the possible number
of feature maps and hidden nodes, there is no regularization of network complexity enforced in the fitness or selection functions. The parameters of the final structures for each training/cross validation method is shown in Table 6.3. The connection tables are not shown because their size and fairly random appearance provides little information.

As a test to show the importance of the parameter \( \eta \), the pre-defined \( \eta \) of LeNet-5 has been changed to the same value as the \( \eta \) evolved in the swapping method. As Figure 6.6 shows, this improved convergence speed of LeNet-5 to values much closer to the networks evolved, but total error energy on the training set was still slightly higher than the evolved networks.

As shown in Table 6.3, the evolved networks also possessed many more feature maps at layers C1 and C3 than LeNet-5. But because these parameters are shared, the number of trainable parameters does not increase dramatically with an increase in the number of feature maps. The natural evolution of sparse connection tables further reduces the number of trainable parameters. This is counterintuitive to past
6.2 Analysis of Architecture Evolution

<table>
<thead>
<tr>
<th>Training/Validation Type</th>
<th>C1</th>
<th>C3</th>
<th>C5</th>
<th>(\eta)</th>
<th>(v)</th>
<th>(c)</th>
<th># Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swapping 2000/2000</td>
<td>37</td>
<td>33</td>
<td>85</td>
<td>0.000344318</td>
<td>3.14241</td>
<td>0.751579</td>
<td>66355</td>
</tr>
<tr>
<td>Shuffle 2000/2000</td>
<td>37</td>
<td>26</td>
<td>60</td>
<td>0.00053939</td>
<td>0.835079</td>
<td>0.42311</td>
<td>38709</td>
</tr>
<tr>
<td>Shuffle 500/3500</td>
<td>22</td>
<td>40</td>
<td>51</td>
<td>0.000641646</td>
<td>0.729287</td>
<td>0.409034</td>
<td>40382</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>6</td>
<td>16</td>
<td>120</td>
<td>0.0001</td>
<td>1</td>
<td>0.5</td>
<td>51046</td>
</tr>
<tr>
<td>LeNet-5 Modified</td>
<td>6</td>
<td>16</td>
<td>120</td>
<td>0.000344318</td>
<td>1</td>
<td>0.5</td>
<td>51046</td>
</tr>
</tbody>
</table>

**Table 6.3** Details of the structure and parameters of the evolved networks at Generation 100.

experiments where networks with more trainable parameters usually converged better on training sets with gradient descent methods. In [13], it was reasoned that networks with more parameters were able to overdimensionalize the problem space and avoid local minima.
Chapter 7

Conclusion

These results illustrate that convolutional neural networks with good performance can be created with little a priori knowledge using a combination of evolutionary search and backpropagation using stochastic diagonal Levenberg-Marquardt method. For a fixed number of epochs, the evolved networks outperform LeNet-5 when trained on a subset of the MNIST dataset. Experiments of training these evolved networks to convergence on the full MNIST training set have not yet been performed.

The swapping method of training/cross validation outperforms the alternative shuffling methods. It is likely that the introduced noise due to shuffling between generations hinders the accuracy of fitness evaluation. In addition, it is possible that the evolved network is dependent on the order of the training data which the swapping method preserves.

Such an evolution method also provides a great deal of information about the characteristics of CNNs at this particular task. Interestingly, the architecture evolution is able to reach a steady state rather quickly as all methods appeared to reach stable networks on or before the 14th generation. This indicates that there are very deep minima in the evolutionary search space and the difference between such minima
defines the difference between a trainable and an untrainable network. By tracing the lineage of the candidates, it has been found that the $\epsilon$ parameter in (2.64), which guides the learning rate, plays an important role in defining a trainable network.

In addition, Table 6.3 shows something that may be uncharacteristic of deep networks, the evolved networks which converged faster had substantially more feature maps in the early layers. At the same time, the number of hidden nodes remains relatively low when compared to LeNet-5. This seems to imply that convergence rates are not diminished when the weights are more evenly distributed in towards layers C1 and C3 of the network. This is in contrast to LeNet-5 where a majority of the weight parameters are located in layers C5.
Chapter 8

Future Work

Evolutionary algorithms are considerably easier to parallelize than backpropagation and can therefore prove especially advantageous in environments where parallel computing is an available resource. For example, it has been shown that CNNs can be parallelized on a GPU to give substantial improvements as shown in [5]. To the best of our knowledge, no published works dealing with extending this parallelization to multiple computers or multiple GPUs exist at the time of this writing. This is likely due to the bandwidth limitations and extremely large latency penalties when data dependencies and communications exist outside the GPU. On the other hand, evolutionary algorithms are naturally scalable because individuals in a population are completely independent from one another. In addition, any gains from parallelizing backpropagation would also extend to this hybrid method.

With the emergence of cloud computing, scaling this method to larger populations and datasets appears very promising. Resources such as Amazon EC2 and frameworks like MapReduce are readily available today. Parallel implementation overhead would be low because of the method’s natural scalability. In addition, the opportunity to train to near convergence for moderately sized datasets may also provide more
visibility on the performance of a particular network making the evolutionary search more global.

In addition, it is possible to apply more recent advances in deep network training such as Hessian-Free optimization in [16] to train deep auto-encoders which are gaining popularity. [16] discusses a new method of second order training which substantially improves the overall convergence of deep networks on training sets. This addresses the serious problem of underfitting with these types of networks.

It has been noted that the hybrid method often created networks with much fewer training parameters than LeNet-5. Therefore it may be possible to include the number of trainable parameters as a regularization parameter for fitness evaluation in order to evolve smaller networks with less trainable parameters, which might improve computational performance and generalizability. Because the hybrid method already creates smaller networks, the additional regularization parameter may not substantially impact network error performance.
Bibliography


