UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of master's thesis by

Srinivas Vadrevu

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Richard Maclin

Name of Faculty Adviser

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Acknowledgments

I am grateful to my advisor Dr. Richard Maclin for providing me an opportunity to work with him and the valuable guidance that he provided. I would also like to thank him for his expertise, comments, feedback and encouragement on many of the issues surrounding my work. I wish to offer special thanks to Dr. Taek Kwon and Dr. Tim Colburn for being on the committee and for their useful comments on the thesis.

I would also like to acknowledge the help of the department of computer science at University of Minnesota Duluth. Specifically I would like to thank Lori Lucia, Linda Meek and Jim Luttinen for their help. I am indebted to my fellow grad students and friends Krishna, Nitin, Bano, Amit, Kiranmai, Deepa, Hima and Kirani for their valuable suggestions and comments. Finally I would like to thank my parents and my brother for their support and encouragement.

Thank you everyone.

Contents

List of Figures

List of Tables

Abstract

Neural networks are an efficient and accurate method for classification tasks. Their ability to find patterns in data enables them to generalize well even in the presence of noise. Backpropagation is very popular learning algorithm used to train neural networks. Learning in neural networks employs gradient-descent to repeatedly make small changes to the weights of a neural network until a local minimum is reached. One of the drawbacks of neural networks is that they may require many presentations of a dataset before they converge on an effective model. This makes neural networks impractical in the area of knowledge discovery in databases where the amount of data is large.

In this work we investigate the efficiency of neural networks when the size of the dataset is large and introduce a novel approach to neural network training based on subsets of the data. Our initial experiments suggest that neural networks will converge very quickly in many cases (often in less than 10 epochs), but that it is difficult to reduce the training time by varying the learning parameters and topology without affecting performance. We then introduce a new algorithm for neural learning that makes it possible to train a network in one pass through the data (in the sense that each data point is only read into memory once) . Our approach works by breaking the dataset into groups that are small enough to fit in memory, and then training the network on one group of the data at a time. We performed experiments on a number of larger datasets from the UCI Machine Learning repository. Our empirical results show that the networks produced using the subset approach are often comparable to networks trained on the entire dataset. These results indicate that it should be possible to employ neural networks efficiently for very large datasets.

1 Introduction

In today's world, the amount of data being collected is growing so fast that it is becoming difficult to maintain and manage. One of the challenges in the field of databases is to mine this huge amount of data. Most of these huge databases contain data that can be expressed using simple rules. *Knowledge Discovery in Databases (KDD)*, sometimes also called *data mining*, is the process of extracting useful information from large databases. KDD uses statistical techniques to discover the knowledge in databases. Machine Learning (ML) algorithms are used in KDD to extract vital information from large databases. One class of ML algorithms are *classifiers*, that learn from examples labeled by a teacher in order to be able to classify new, unseen examples. The application of ML techniques to the field of KDD has been proven successful by many researchers [Mitchell, 1997]. Some popular ML methods used in the area of KDD are association rules, decision trees, artificial neural networks (ANNs), and naive Bayes learning. Though ANNs are known for their accurate models, they have received less attention in KDD because they are very slow in building models, requiring multiple presentations of the data, which may not be possible when there is very large amount of data. In this thesis, we explore a novel method to learn from large databases with a reduced training regimen while still being able to produce an accurate classifier. Before discussing the details of our approach, we will first discuss why we focus on neural network models.

1.1 Artificial Neural Networks

Neural networks are a popular approach to classification for a number of reasons. They are one of the most accurate classification methods [Shavlik et al., 1991]. They are called artificial neural networks because they are inspired by the network of neurons in the human brain and operate similar to them. A more complete discussion of neural networks can be found in the Section 2. Neural networks algorithms are able to learn nonlinear models and generalize well in the presence of noise. Neural networks have also been shown to be very effective components in classifier ensembles [Opitz and Maclin, 1999]. The two main drawbacks to neural networks are that: the resulting learned concept is often difficult to express to human users, and a neural network can require significant training time to produce a learned model. A number of authors have suggested mechanisms for addressing the first problem, extracting learned knowledge from a neural network (e.g., [Craven, 1996, Thrun, 1995]). In this work, we focus on the second drawback – that neural networks can require significant training time.

In general, most ANN learning mechanisms require multiple presentations of a set of examples before the learner converges on a model. Research has produced mechanisms for reducing this time such as QuickProp [Fahlman, 1988] and RProp [Riedmiller and Braun, 1993], mechanisms that seek to make much larger changes in weight space. Others have attempted to adapt other aspects of the learning method to achieve speedup [Balakrishnan and Honavar, 1992, Salomon and van Hemmen, 1996, Schmidhuber, 1989, Stager and Agarwal, 1997], but none of these methods focuses on the specific problems associated with large datasets. This leads us to a question: is it possible to adjust the learning mechanisms of a neural network to produce a learned neural network in a reasonable amount of time, especially when applying neural networks to very large datasets?

1.2 Training ANNs for Very Large Datasets

In this work, we propose a solution to the problem of slow learning in neural networks that specifically focuses on the case where the dataset is very large. Our resulting learning method is capable of learning an accurate classifier in only one pass through the data where each data point is read into memory exactly once.

In our initial investigations of the applicability of neural networks to large datasets, we conducted experiments to determine the convergence rate of neural networks with large datasets (with the largest one having a million instances). First we conducted experiments to record the error rate at various points in learning for a number of datasets from the UCI machine learning data repository [Blake and Merz, 1998]. These results indicate that the classifier produced after a small number of presentations of all the data points is almost as good as the one produced after many presentations. For a number of large datasets, we show a neural network learner needs only a small number of epochs (an *epoch* is a presentation of the entire dataset) to converge.

Although the simple approach of using only a small number of training epochs is effective in some cases, it still requires multiple presentations of the data. But when training large datasets, we cannot afford to present the entire dataset multiple times. In the best case, we desire an approach that requires that each data point be read into memory only once. To achieve this, we explore an approach of training the datasets by segregating it into pieces that can fit in the main memory instead of training the whole dataset at once. The dataset is first divided into smaller subsets and each subset of data is used to train the neural network. This approach is illustrated in the Figure 1.

Note that we refer to this as a one pass algorithm because it only requires only one pass over the dataset in terms of reading the data into memory. Within this framework we first investigate the simple approach of training the network on each successive subset of the data. But this approach may suffer from the *moving target problem* defined by Fahlman and Lebiere [1990]. The moving target problem refers to the possibility that the classifier developed on a single subset of data may contain

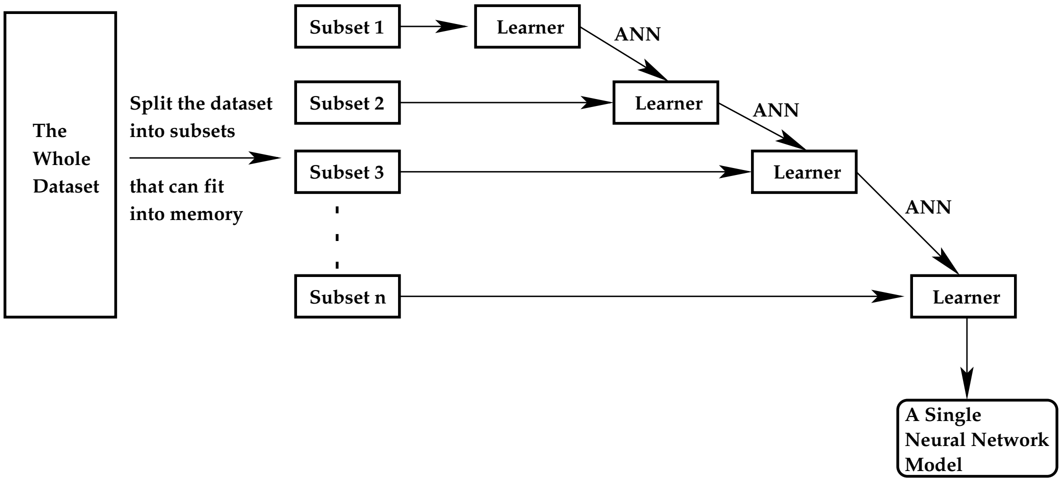


Figure 1: An overview of the subset approach. The dataset is divided into subsets that can fit in the main memory of the computer. Then the network is trained for each subset separately for some number of epochs. The key aspect in this algorithm is that a single neural network is the result of the training.

aspects that are unique to that subset and may override learned aspects of previous subsets. We address this problem by investigating techniques for associating each subset with a number of hidden units that focus on that subset of the data. Empirical results with the techniques we investigate show that our approach is often comparable to standard neural network learning.

1.3 Thesis Outline

This thesis is organized as follows:

- Chapter 2 presents background on neural network learning and some key aspects of convergence.
- The following chapter discusses the details on the datasets we used in our experiments and the methodology of our experiments. It presents the baseline

results with standard ML learning methods for the datasets we chose. It also presents results from our initial experiments showing the effect of convergence rate with training epochs. Then we present experimental results with large numbers of hidden units. In this chapter, we also examine how convergence is affected by some simple adjustments to the parameters for neural learning.

- Chapter 4 describes our new approach for learning a neural network in one pass through a large dataset and presents these results.
- Then we discuss additional related research (Chapter 5) in neural network training using the concept of subsets and other related aspects of this thesis.
- We finish with conclusions and some future directions for our research.

2 Background

In this chapter, we present the background for this work. We first provide a brief introduction to machine learning and classification methods and then discuss various learning mechanisms including neural networks, the approach we use in this thesis. Then we outline important aspects of neural networks such as gradient descent and backpropagation. Later we present some simple techniques that have been developed to speed up learning in neural networks and a brief discussion of training on subsets of data.

2.1 Machine Learning

Learning is a process of improving the knowledge through experience. Humans learn from their experience so that they can do well when a similar situation occurs in future. In Machine Learning (ML), computers often try to learn by trying to simulate the human brain. ML [Mitchell, 1997] is a process of automating the human learning process using computer programs. The goal of an ML algorithm is to learn from experience and build a model that can be used to perform similar tasks in the future. ML algorithms can be broadly classified into two categories: un-supervised learning algorithms and supervised learning algorithms.

Supervised learning algorithms make use of *labeled* or *training data* where each example is labeled with its corresponding class by a human observer with his expertise. These algorithms learn from data and build models of the classes. The goal is to be able to classify new or *test data* that has similar properties as training data. This kind of learning is also called learning with a teacher. Un-supervised algorithms, on the other hand, do not have any labels for the data and they work by developing rules from a set of observations without any prior knowledge of the sample problems that can be solved with them. The goal is to be able to group the data in such a way as to extract knowledge from the groups. Our work mainly focuses on classification problems, which are a kind of problem that is usually solved with supervised methods. We discuss classification methods in detail in the next section.

2.2 Classification Algorithms

Classification algorithms, also called *classifiers*, work by learning a model from a set of training data provided by a teacher, where the model can be used to classify new data into one of the known classes. An example of a classification problem would be to predict a disease based on its symptoms. Here a training example, or *pattern*, consists of a set of symptoms, represented by input values of *input features*, and corresponding disease, formally known as the *output class.* The training data consists of examples of patients, indicating the symptoms of each patient and whether they have the disease. The goal here is to predict for new patients whether they have a disease based on their symptoms.

Some of the popular classification methods that have been developed include decision trees, artificial neural networks, and the naive Bayes method. A decision tree is a tree that contains tests on the input features, ordered in terms of their importance, with each possible value of a feature leading to a separate sub-tree that ends with a leaf containing an output class. That is, the value of a certain input feature decides where to go and what other input features to test. Figure 2 shows an example of a decision tree. Decision tree learning methods learn by constructing a decision tree with the training data and use this tree to classify new data.

The naive Bayes learning method is a probabilistic approach to learning. It works by using Bayes theorem, which provides a way to calculate the probability of a particular classification (hypothesis). Using this theorem, the probabilities of

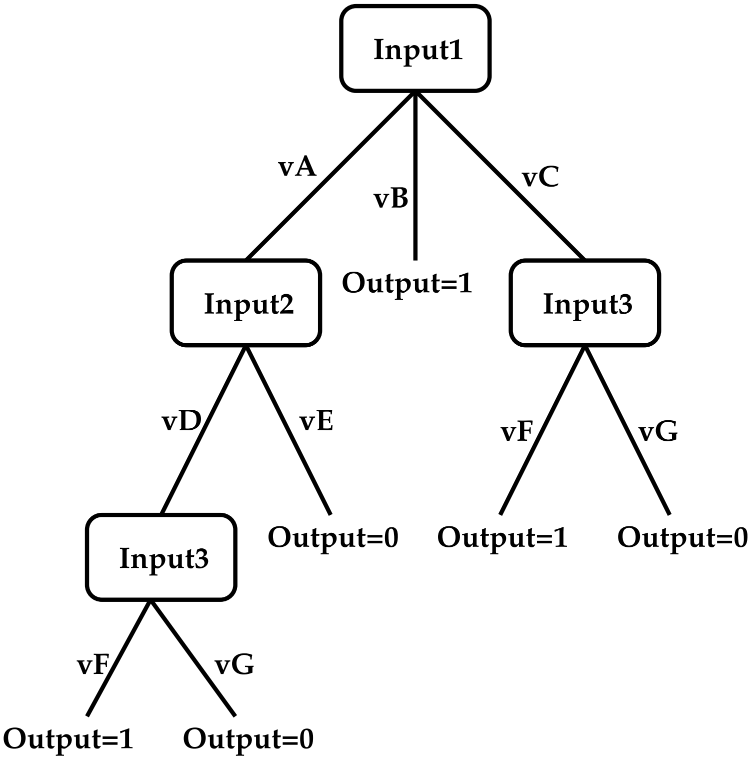


Figure 2: To classify a new example, we first examine the value of feature Input1 of the top of the tree. For example, if its value is vA, then Input2 is examined, expanding the tree accordingly and if its value is vC, then Input3 is examined. If its value is vB, then no more features are tested and tree outputs 1 for that example.

all classifications are estimated and the classification with less conditional risk (i.e., with minimum error rate) is chosen as the final prediction. The naive Bayes method makes this calculation tractable by assuming each input feature is independent of every other input feature. Neural networks are a popular learning mechanism that have been proven to be successful in many problems [Shavlik et al., 1991]. Neural networks algorithms are able to learn nonlinear models and generalize well in the presence of noise. Neural networks have also been shown to be very effective components in classifier ensembles [Opitz and Maclin, 1999]. We focus on this learning method in our work. The rest of this chapter describes neural network learning in detail, outlines the advantages and disadvantages, and discusses techniques to overcome the disadvantages.

2.3 Neural Networks

Neural networks are a non-parametric learning technique that provide an efficient method to solve classification problems. They are called neural networks because they are loosely based on the operations of neurons in the human brain.

A typical feed-forward neural network, when used for classification, consists of a layer of *input units*, which represent the input features of the problem, a layer of *output units* which represent the possible results, and zero or more *hidden layers* of units (see Figure 3).

The neurons in the network are connected to other neurons in the network through weights. A *weight* is a directed link between neurons. The knowledge gained by learning is stored in these weights connecting the units. The goal of a neural network is to find the best possible combination of these weights for a given problem. In this work we will generally employ multi-layer perceptrons where the neural network has an input layer, an output layer, and one hidden layer. In general, we assume that each unit in a layer is connected to all of the units in the next layer by a weight. A network is feed-forward if there are no cycles of weights.

Every unit in the network has a net input signal, which is estimated by summing the weighted input signals coming into it and an activation value, used to limit the output value to some finite value associated with it. The input units are set by the values of the input features of the example. All the other units are activated by computing the net input signal which is computed as

$$n_k = \Big(\sum_{\forall j \in LinkedTo(k)} w_{j \to k} \times a_j \Big), \tag{1}$$

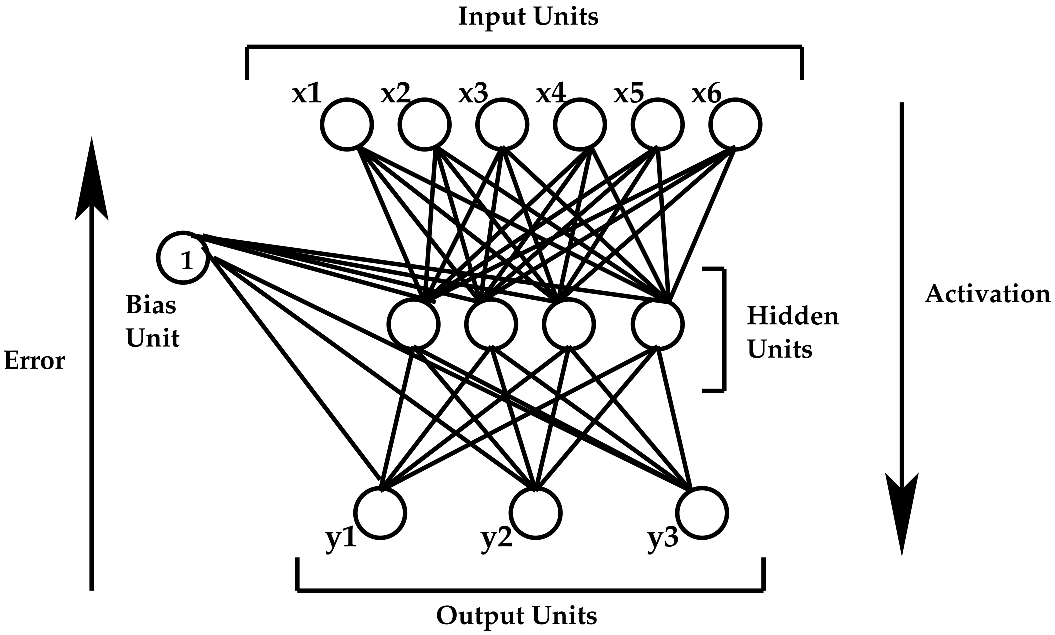


Figure 3: A typical feed-forward neural network. The layer of input units ($x1$ to $x6$) is shown at the top and the output units ($y1$ to $y3$) are at the bottom. The input units are set to describe the features of the example and the network is activated (top to bottom) to predict the class of the example. The predicted output is compared to the expected output and an an error signal is back-propagated through the network and used to alter the weights of the network to more closely predict the correct class. A set of hidden units is included in this model which can be used to capture intermediate concepts. The input units are connected to the hidden units and the hidden units in turn to the output units. The network also includes a bias unit whose activation is always set to one. Weights from these links act as thresholds. This unit has a connection into all of the hidden and output units.

where n_k is the net input signal for unit k, $w_{j \to k}$ is the weight of the link from the unit j to k (including the weight of bias unit), and a_j is the input activation signal of unit j. All the units, except for the input units, also have an external bias which is a weight from a bias unit whose activation is always one. The bias signal is used to offset the total net input signal.

The goal of neural learning is to find an effective combination of weights that accurately predicts the classification of a set of examples. The knowledge obtained by the neural network is captured in the set of weights for that network. A simple way to train a neural network learning algorithm is to use the gradient descent learning rule, described in the next section.

2.3.1 Gradient Descent Learning Rule

Gradient descent derives its name from the fact that it selects a change based on the steepest gradient (derivative) direction. The ultimate goal in neural networks is to find the optimum values for all the weights in the network. Therefore gradient descent is performed on the error with respect to each weight. One method for employing gradient descent is to make very small changes to the weights while searching the error surface, basing the amount of the weight update on the direction and magnitude of the error. Therefore according to gradient descent, the weight are updated according to the following rule

$$\Delta w_{j \to k} = -\eta_{w_{j \to k}} \frac{\partial E}{\partial w_{j \to k}}, \tag{2}$$

where $\Delta w_{j \to k}$ is the amount of update to the weight j from k, $\eta_{w_{j \to k}}$ is the learning rate associated with the weight $w_{j \to k}$, and E is the cost (error) function. The presence of negative sign in the weight update rule implies that we want to go in the direction which decreases the error. Error can be defined in a number of ways. One popular way is the sum of squared differences of actual and estimated outputs at each output unit.

An example of gradient descent learning algorithm is the perceptron training rule. A *perceptron* [Rosenblatt, 1958] consists of only input and output layers with no hidden layers. The idea here is to first set the weights randomly, and then present the patterns repeatedly to the perceptron, modifying the weights appropriately until it classifies all the patterns correctly or a local minimum in error is reached. One complete presentation of all the patterns of a dataset to the network is called an *epoch*. Here the weight update rule is based on a simple error which is just the difference of estimated output and the desired output. Each weight is updated using the following rule,

$$w_{j\to k}(t) = w_{j\to k}(t-1) + \Delta w_{j\to k}(t), \tag{3}$$

and $\Delta w_{j\to k}(t)$ is defined as

$$\Delta w_{j\to k}(t) = \eta_{w_{j\to k}}[y(t-1) - o(t-1)]x(t-1), \tag{4}$$

where t is the time step or epoch number, $w_{j\to k}(t)$ and $w_{j\to k}(t-1)$ are the weights associated with an input vector x at t^{th} and $(t-1)^{th}$ epoch, $y(t-1)$ is the desired output (also called as *target* output) at time $(t-1)$, $o(t-1)$ is the estimated output for the same time generated by the perceptron, and $\eta_{w_{j\to k}}$ is a small positive constant associated with the weight $w_{j\to k}$ called the *learning rate*. This positive constant, which usually ranges between 0 and 1, determines the amount of weight update at each step. The perceptron approach can guarantee a solution to a given problem only when the training data is *linearly separable*, that is when all the examples can be grouped into classes that are clearly separated by a simple linear decision boundary. When the examples are not linearly separable, the convergence of the perceptron training rule is not assured.

The disadvantages of gradient descent are that it is very slow in converging as it updates the weights after one pass through all the examples (after each epoch) and it does not guarantee that the global minimum will be chosen in the presence of many local minima. One common variant of gradient descent used to overcome the slow convergence is *incremental* or *stochastic* gradient descent, where the weights change after each training example. The gradient descent learning rule can also be extended to multi-layer perceptron networks with the backpropagation algorithm discussed in the next section.

2.3.2 Backpropagation

Backpropagation [Rumelhart et al., 1986] is a popular and efficient learning method to train a neural network that contains multiple layers. It works by repeatedly activating a feed-forward ANN and updating the weights for each example, producing an optimum set of weights. The backpropagation algorithm is outlined in Table 1.

The backpropagation algorithm involves two phases of computation, known as the *forward pass* and the *backward pass.* A backpropagation neural network predicts the class of an example by being *activated* for that example. Learning in backpropagation works by repeatedly *presenting* all of the examples in a dataset: activating the network for that example, back-propagating the error, and then updating the weights. To activate the neural network, the input units' activation is set to the values describing that example. This activation then feeds forward (top to bottom in Figure 3) to the next layer of units. This propagation of activation signal through the network is called the forward pass. Each hidden or output unit's activation is determined by summing the incoming signal to that unit. The activation is calculated using an activation function, that is applied to the sum of incoming signal.

One activation function used in backpropagation is the *sigmoidal activation function* which roughly resembles a rounded threshold function. The activation for a unit, k is defined as

$$a_k = \frac{1}{1+e^{-n_k}}, \tag{5}$$

where n_k is the net input coming into the unit k. The net input is computed as shown in Equation 1. The purpose of using the sigmoidal activation function here

- Set the weights to random values, generally in a small range (e.g., -0.5 to 0.5)
- REPEAT for each example
  - *Forward pass: propagate the activation signal forward through the network*
    - Activate the neural network for the example
      - Input units are set by the values of the input features
      - Hidden and output units are activated by summing the incoming signal to that unit and determining the corresponding activation
  - *Backward pass: propagate the error backward through the network*
    - Calculate the error
      - The error for each output unit k is based on the difference between between the estimated and desired output for that unit
      - Calculate the error for each hidden unit, h as

        $$\delta_h = o_h(1 - o_h) \sum_{k \in Outputs} w_{i \to h} \delta_k,$$

        where δ_h is the derivative of the error for hidden unit and δ_i is the derivative of the error for output unit
    - Backpropagate the error
      - Calculate the weight update amount for each weight in the network as $\Delta w_{j \to k} = -\eta \delta_j$, where η is the learning rate, a small positive constant that controls the amount of weight update.
      - Update the weights

Table 1: The backpropagation algorithm for training a feed-forward neural network.

is that it is a smooth function (can be differentiated everywhere). The key aspect of this activation function is that the activation value will be near one when the net input values are large and positive and it will be near zero when the net input values are large and negative. Once the entire network is activated, the output units represent the predicted class for that example.

Next the error is computed and it is back-propagated through the network (bottom to top in Figure 3) and the weights are updated accordingly. The *error* (e_k) of a single output unit k is simply the difference between the predicted class and the expected class. The error (or cost) for an example x, E_x over all the units in the output layer is estimated as the mean squared error, which is given by the following equation

$$E_x = \frac{1}{2} \sum_{j \in Outputs} (t_{x,j} - o_j)^2 \tag{6}$$

where $t_{x,j}$ is the target value of example x for output unit j and o_j is the estimated output for output unit j. The total cost can be computed as the sum over all the examples. Now to compute the update value for a weight, the gradient descent rule is used which involves differentiation of error with respect to that weight. If we simplify the partial derivative of cost for output j, $\frac{\partial E}{\partial w_{j \to k}}$ (or δ_j), we get the following term

$$\delta_j = o_j(1 - o_j)(t_{x,j} - o_j) \tag{7}$$

where $t_{x,j}$ is the target output and o_j is the actual output for the output unit j. For a hidden unit j, the derivative becomes

$$\delta_j = o_j(1 - o_j) \sum_{k \in LinkedFrom(j)} w_{j \to k} \delta_k \tag{8}$$

The weight update rule in backpropagation is defined as

$$\Delta w_{j \to k} = -\eta \delta_j \tag{9}$$

where η is the learning rate, a positive constant which determines the amount of weight update at each step.

The resulting backpropagation algorithm on neural networks works efficiently on many classification problems. However the backpropagation algorithm converges very slowly for many problems and there are several reasons for it. The next section discusses some possible explanations for the slowness in backpropagation and suggests some simple optimization techniques.

2.4 Simple Techniques to Speed Up Backpropagation

The learning speed in backpropagation depends on parameters such as the learning rate (η). The *learning rate* controls the weight updates at every phase of learning. If we assume a high value for the learning rate, then the weights change too widely. We might even miss the global minima in this process of random weight changes. So the smaller this value, the smoother the learning. But this smaller value of learning rate makes the weight updates very small which results in slower learning. A common optimization technique used to speed up learning in neural networks is to introduce a *momentum* term (α) in the weight update rule [Rumelhart et al., 1986]. With the momentum term, the weight update rule now becomes

$$\Delta w_{j\to k}(t) = -\eta_{w_{j\to k}} \frac{\partial E}{\partial w_{j\to k}} + \alpha \Delta w_{j\to k}(t-1) \tag{10}$$

The value of the momentum parameter usually ranges between 0 and 1. The momentum adds a fraction of the previous weight update to the current weight update which often results in faster convergence. This optimization is due to the fact that it speeds up the learning when the weights are moving in a single direction continuously by increasing the size of steps. The closer this value is to one, the more each weight change will not only include the current error, but also the weight changes from previous examples (which often leads to faster convergence). Therefore for the neural network to learn efficiently on a particular classification problem, the learning rate and the momentum values should be carefully chosen. But the selection of appropriate values of learning rate and momentum for a given problem is not easy. One more problem is that the values that seem to be good in the beginning phase of learning may not be so good as the learning progresses.

A simple way to speed up the learning is to make large changes to the weights so that a minimum is reached quickly. A more detailed discussion on simple techniques to speed up learning is discussed in Section 3. Researchers have proposed using different learning rates at various stages of learning. For example, a high learning rate is used in the early phase of learning and as the learning progresses it is decreased gradually. The reason for this is because in the early learning phase where we are far away from the minimum we can afford to make random big steps. But when we are close to the minimum, we avoid large weight changes as there is a possibility that the minimum can get skipped. Though these simple techniques will help converge faster, they will not guarantee the quick convergence in all cases. A detailed discussion on various techniques to speed up backpropagation can be found in the additional related work section. Another reason for slow learning in backpropagation is that it must repeatedly examine each example, thus it is only efficient if each example is in memory. But this may not be possible especially when the size of the dataset is very large. Some researchers have examined using subsets of data in training. In the next section, we present the background details on training on subsets of data.

2.5 Related Work in Training Subsets of Data

The concept of training on subsets of data is not new. Catlett's research [1991a, 1991b] explored interesting details on training subsets of data. When there is a large amount of data, he examined whether training on all the data is necessary. He used decision trees in his experiments and created separate classifiers using only subsets of data as well as all of the data. His statistical results indicate that the classifiers built using all the data are better than the classifiers built using only part of the data. Often we can improve the accuracy by training on more data and we cannot discard some amount of data because of the complexity in the model developed.

Another interesting aspect of training with subsets of data is the selection of subsets. The most common method used is to divide the dataset randomly into subsets, though the subsets can be selected by assigning some weights to the subsets based upon the accuracy obtained. Catlett looked at a number of more complex subset methods including stratified sampling to address various data issues (stratified sampling is used if the distribution of data is skewed).

In addition to training the subsets of training data, research has also been done in training the subsets of input features [Kohavi and John, 1997, Bommaganti, 2001]. When the amount of available data is very large one of the solutions is to reduce the search space (reduce the size of the problem to look at). One of the possible ways to reduce the search space is to reduce the number of input features, leaving out the ones which do not contribute much to the final solution. The critical idea here is to select those input features to consider and those to remove. Feature subset selection is a complex task and often requires knowledge of the problem in order to determine the priorities.

3 Initial Experiments

In this chapter, we present results from our initial experiments that investigate the convergence of neural networks with large datasets and our experimental results varying the neural network parameters and topology. We performed experiments on a number of datasets. Before presenting the results, we first describe the datasets we used in our experiments and our testing methodology. As an initial investigation of training on large datasets, we compare the performance of various machine learning methods including neural networks. Later we present our initial experiments showing the convergence rate of neural networks with respect to number of epochs for various datasets. We then present results from our experiments performed with larger numbers of hidden units. We also include results showing the convergence of the neural networks by changing key parameters such as learning rate and momentum.

3.1 Datasets Chosen and Testing Methodology

We chose as our testbeds a number of larger datasets from the UCI machine learning repository [Blake and Merz, 1998]. Table 2 shows the characteristics of the datasets we used in our experiments including the number of examples, the number of discrete features, and the number of continuous features. Although some of these datasets are smaller than those we expect to apply our methods to, they are large enough to be reasonably difficult, but small enough to allow us to compute results for training on the entire dataset. Thus we can compare the results using our method to results for training on the entire dataset. To get an estimate of the performance and training time for a larger dataset, we experimented with two very large datasets, forest-cover dataset with over 581,000 instances and a synthetic dataset [Breiman, 1999] with one million instances. These two larger datasets are used in our subset experiments.

Table 2: The datasets used in our experiments. These datasets are drawn from the UCI ML repository. Shown are the number of examples, the number of continuous and discrete input features for the datasets, and the number of output classes for each dataset. Also shown are the parameters defining the neural networks we used to learn these datasets.

		Features			Neural Network Structure		
Dataset	#Examples	Continuous	Discrete	Classes	Inputs	Outputs	Hiddens
adult	48842	6	99	2	105	1	20/40/60
anonymous-msweb	37711	–	322	2	322	1	40/80/120
hypo	3772	7	22	5	55	5	15/30/45
kr-vs-kp	3196	–	36	2	74	1	15/30/45
letter-recognition	20000	16	–	26	16	26	40/80/120
satellite	6435	36	–	6	36	6	15/30/45
segmentation	2310	19	–	7	19	7	15/30/45
shuttle-all	58000	9	–	7	9	7	15/30/45
sick	3772	7	22	2	55	1	10/20/30
splice	3190	–	60	3	240	3	25/50/75
forest-cover	581012	10	44	7	54	7	75
synthetic	1000000	61	–	10	61	10	100

Table 2 also shows the number of hidden units of the neural network used for the respective datasets including the number of input units, output units and hidden units. Note that we did not experiment with varying number hidden units on the two larger datasets.

Each neural network had a single input layer, a single hidden layer and a single output layer. Each input unit is connected to all of the hidden units and each hidden unit is connected to each output unit. The number of hidden units used is based on a previous study of neural network performance [Opitz and Maclin, 1999]. We chose as standard parameter values a learning rate of 0.15 and a momentum value of 0.9, values that have been used in the same study. The weights in the neural network are randomly initialized in the range of -0.5 to 0.5.

Our results are averaged over twenty standard 10-fold cross validation experiments. In 10-fold cross validation, the dataset is partitioned into 10 equal-sized sets (called folds). First, one fold is used as test data and the rest are used as training data, and this process is repeated with all the sets such that every set is used as test data once. We use backpropagation [Rumelhart et al., 1986] as our method for training in neural networks.

3.2 Performance of Various Machine Learning Approaches on Large Datasets

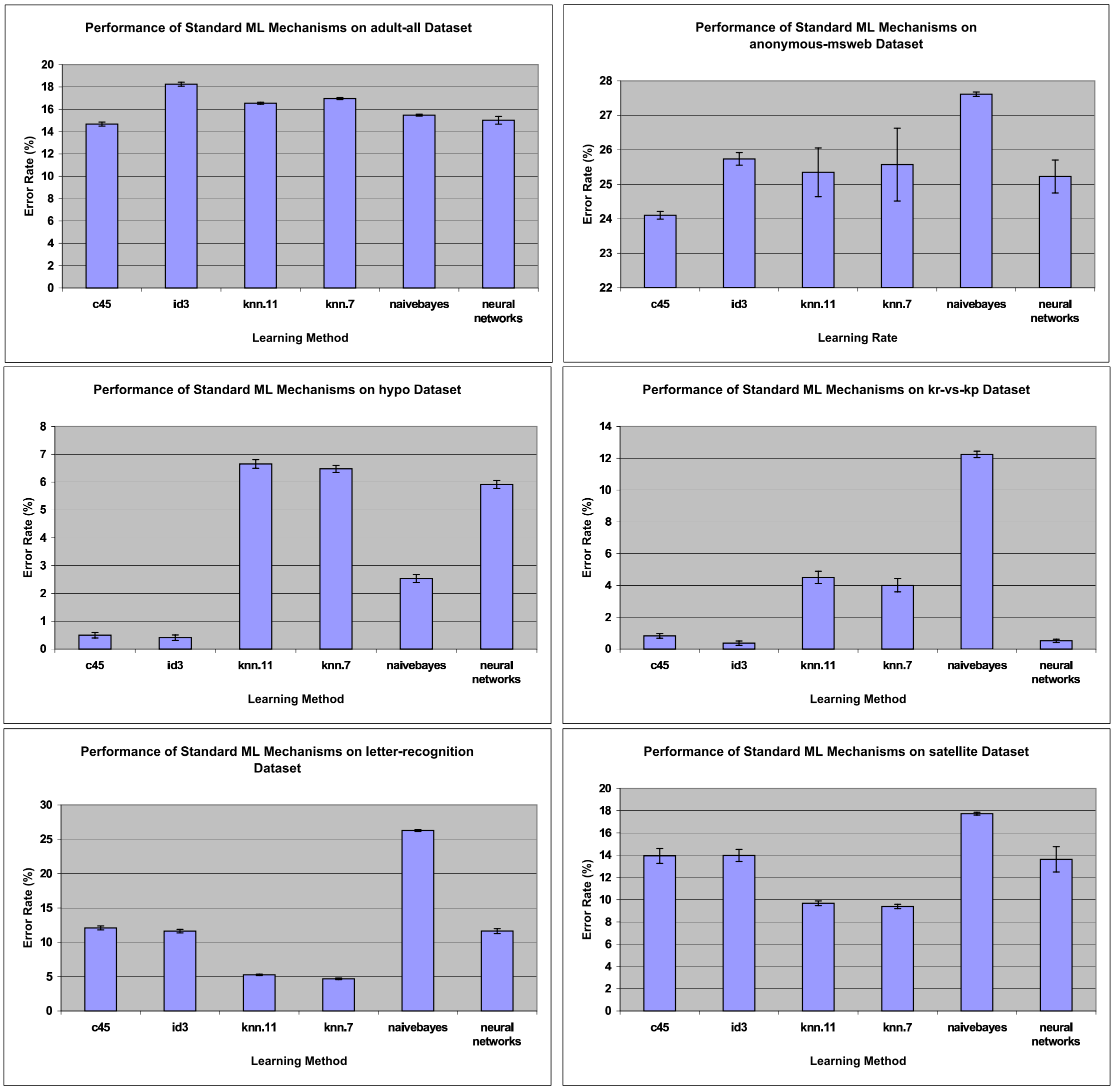
As a baseline we first investigate the performance of several standard ML algorithms. The algorithms we tested are the C4.5 decision tree learning method [Quinlan, 1983], the ID3 decision tree learning method [Quinlan, 1986], the K-Nearest Neighbor learning method [Dasarathy, 1991] with K values of 7 and 11, and the naive Bayes learning method. We compare neural networks learning method with all these approaches.

In Table 3, we present the results with these machine learning approaches. These results show a varied pattern. Different ML mechanisms perform well on different datasets. For example, the datasets hypo and sick are well adapted to decision trees. That is the error rate for these datasets is significantly lower with C4.5 and ID3 algorithms, whereas other ML mechanisms perform well but not as good as these decision tree methods. This may be because these datasets have a skewed distribution of data. The K-Nearest Neighbor learning method seems to perform better than all the other learning methods for letter and satellite datasets. On the other

Table 3: Table showing the error rates of various learning mechanisms. The values shown are averaged over 20 runs. The values in parenthesis indicate the standard deviation. The results with neural networks are obtained by training for 100 epochs and using the maximum number of hidden units that can be used respective datasets.

Dataset	C45	ID3	KNN-11	KNN-7	naive Bayes	NN
adult	**14.68**(±**0.11**)	18.24(±0.10)	16.54(±0.05)	16.94(±0.05)	15.48(±0.05)	15.01(±0.20)
anon-msweb	**24.10**(±**0.06**)	25.74(±0.11)	25.35(±0.41)	25.57(±0.61)	27.61(±0.04)	25.08(±2.87)
hypo	0.50(±0.06)	**0.41**(±**0.06**)	6.65(±0.09)	6.48(±0.07)	2.53(±0.08)	5.92(±0.08)
kr-vs-kp	0.83(±0.08)	**0.37**(±**0.08**)	4.51(±0.22)	4.01(±0.24)	12.24(±0.12)	0.52(±0.06)
letter	12.09(±0.17)	11.63(±0.15)	5.27(±0.06)	**4.68**(±**0.07**)	26.29(±0.08)	11.64(±0.20)
satellite	13.94(±0.39)	13.98(±0.31)	9.68(±0.12)	**9.39**(±**0.11**)	17.73(±0.08)	13.63(±0.67)
segmentation	5.32(±0.31)	5.22(±0.25)	5.43(±0.19)	**5.08**(±**0.16**)	14.96(±0.30)	3.74(±0.22)
shuttle-all	0.04(±0.01)	**0.02**(±**0.00**)	0.19(±0.00)	0.15(±0.01)	0.57(±0.00)	0.70(±0.17)
sick	**1.04**(±**0.13**)	1.07(±0.09)	3.97(±0.09)	3.67(±0.09)	2.87(±0.09)	3.50(±0.24)
splice	5.83(±0.17)	13.04(±0.43)	17.52(±0.40)	19.64(±0.41)	4.44(±0.08)	**4.34**(±**0.14**)

hand, all the learning methods seem to perform better than this learning method for the splice dataset. Figures 4 and 5 show the graphs obtained by plotting the error rate with various ML learning methods for the datasets listed in Table 2. It is evident from the graphs that different ML mechanisms perform well on various problems and in general the decision tree and neural networks algorithms often perform well. The results from these experiments lead us to the following conclusions: (1) the decision tree learning methods are generally effective in many problems, (2) artificial neural network learning methods often perform well and are generally better with larger number of hidden units, and (3) the naive Bayes and K-Nearest Neighbor algorithms perform well on some problems and poorly on some problems.



Performance of Standard ML Mechanisms on adult-all Dataset

Performance of Standard ML Mechanisms on anonymous-msweb Dataset

Performance of Standard ML Mechanisms on hypo Dataset

Performance of Standard ML Mechanisms on kr-vs-kp Dataset

Performance of Standard ML Mechanisms on letter-recognition Dataset

Performance of Standard ML Mechanisms on satellite Dataset

Figure 4: A plot of test set error rate rate for standard ML mechanisms for first set of problems. The standard deviation is represented as the error bars (by calculating 95% critical region) in the graphs.

In the next section we concentrate on neural network learning and present numerical results showing the error rate at various stages of neural learning.

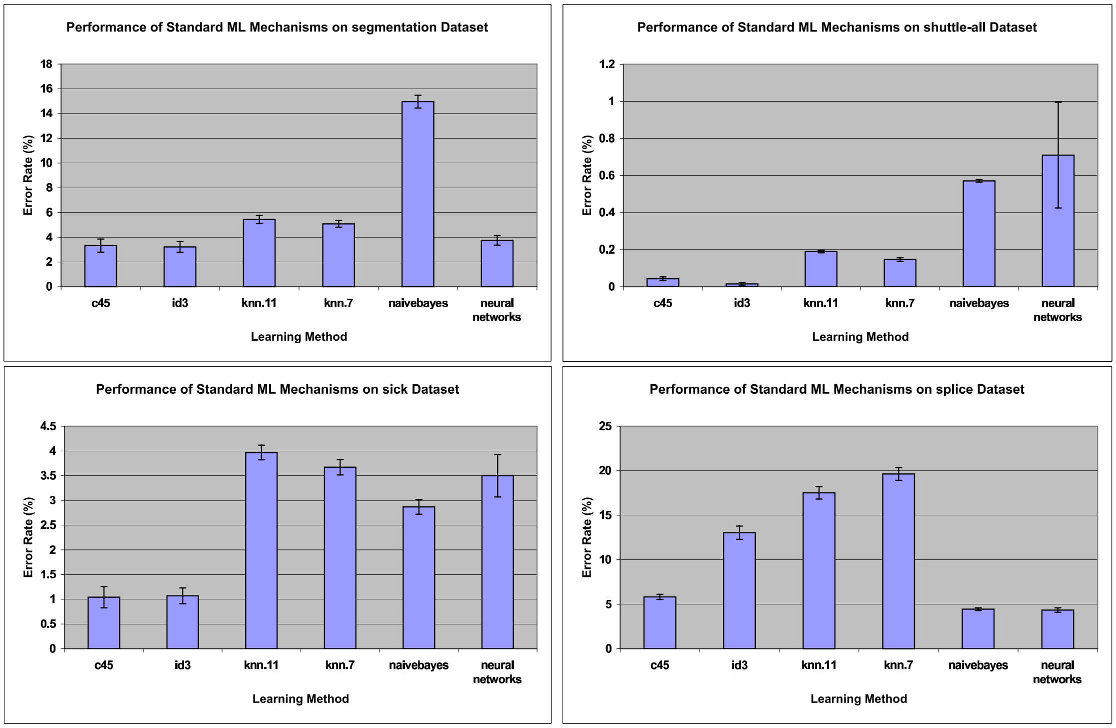


Figure 5: A plot of test set error rate rate for standard ML mechanisms for first set of problems. The standard deviation is represented as the error bars (by calculating 95% critical region) in the graphs.

3.3 Experiments with Convergence Time in Neural Networks

One question we examine with neural networks is convergence rate. Though neural networks require significant time to converge, they often produce a good solution within fewer number of epochs, especially when there is huge amount of data. To demonstrate this hypothesis, we examine the performance of neural networks at various stages of learning. Table 4 shows the results after various stages of neural network learning for various datasets. For each dataset, the value where the error seems to plateau is shown in bold face. Figures 6 and 7 show the plots that indicate the error rate at regular intervals in the neural network learning for the datasets

Table 4: Convergence results for the datasets used in our experiments. We show the test-set error rate after 0.1, 0.5, 1, 2, 3, 5, 10, 15, 30, 50, and 100 training epochs. Note that in many cases, the error after only one epoch is comparable to the error after many training epochs.

	Error Rate after N Epochs										
Dataset	N=0.1	N=0.5	N=1	N=2	N=3	N=5	N=10	N=15	N=30	N=50	N=100
adult	17.68	16.79	16.37	16.10	15.97	15.67	15.12	14.91	14.77	14.93	15.01
anon-msweb	29.55	27.81	27.12	26.04	24.65	24.49	24.39	24.34	24.65	24.87	25.08
hypo	7.72	7.72	7.72	7.70	7.33	6.52	6.29	6.22	6.08	6.02	5.92
kr-vs-kp	19.30	7.03	5.52	3.95	2.94	1.74	0.85	0.66	0.58	0.54	0.52
letter	94.80	77.93	47.24	30.74	26.25	22.24	18.53	16.90	14.52	12.98	11.64
satellite	33.56	20.86	19.64	18.80	18.50	17.72	16.61	15.71	15.29	14.21	13.63
segmentation	72.95	36.77	17.51	10.78	9.50	8.39	7.11	6.58	5.34	4.33	3.74
shuttle-all	9.56	5.64	4.84	3.62	3.13	2.36	1.57	1.34	1.19	1.01	0.70
sick	6.21	6.26	6.20	6.23	6.14	6.14	5.80	5.47	4.71	4.28	3.50
splice	40.87	14.18	9.05	6.20	5.41	4.75	4.44	4.43	4.30	4.37	4.34

described in Table 2. Note that error rate comparable to what can be achieved with a large number of epochs (100) can sometimes be achieved with a much smaller number of epochs. For datasets adult, anonymous-msweb, kr-vs-kp and splice, the convergence is obtained before training the neural network for 100 epochs. For the other datasets, the minimum error is found at 100 epochs but the error after a few epochs is comparable to the error after 100 epochs (i.e., within the 95% confidence region). This suggests that for some datasets it may be possible to adjust learning in a simple manner to achieve good results with a very small number of epochs. In the next section we present experiments to test this hypothesis.

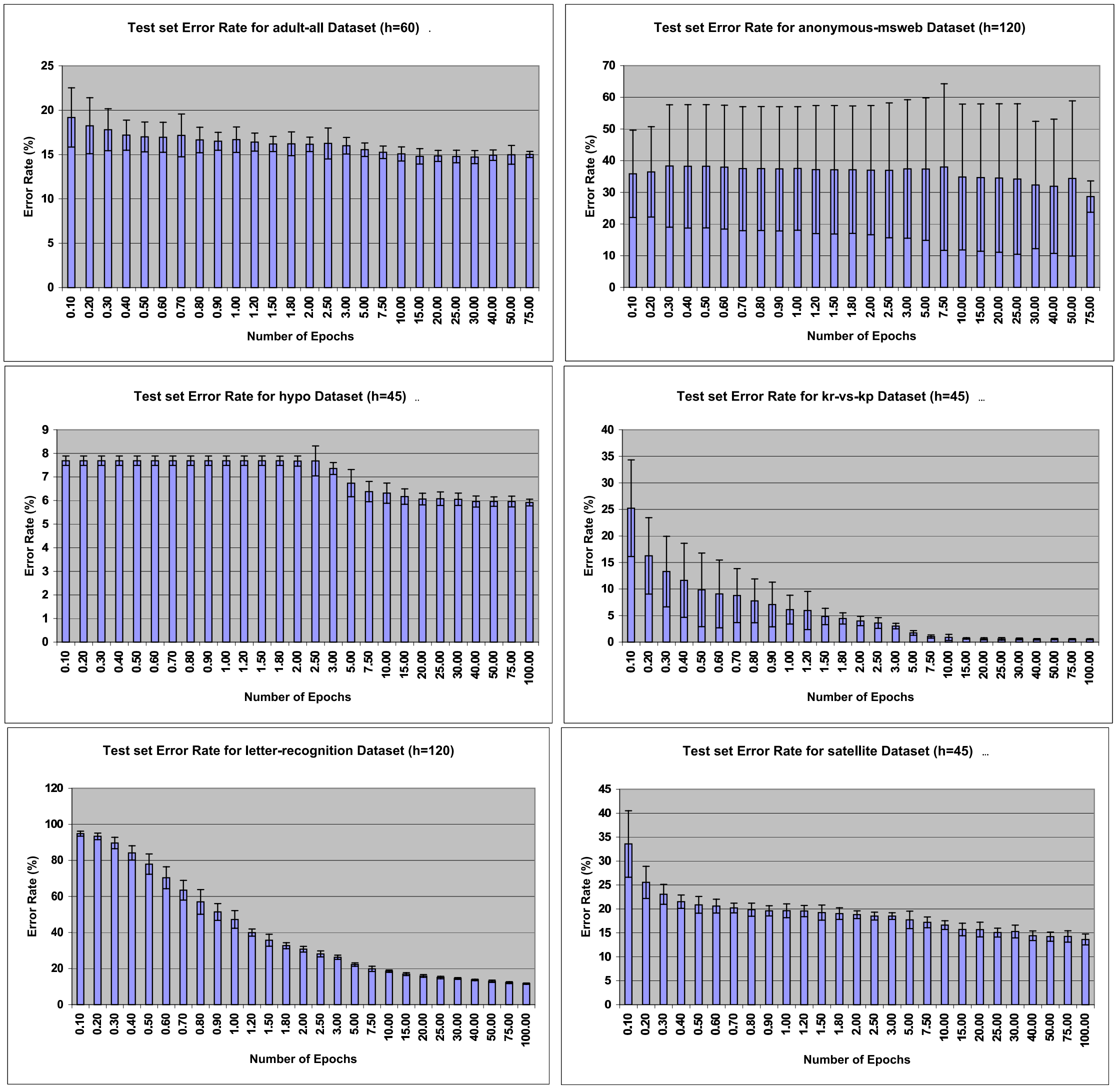


Figure 6: A plot of test-set error rate with respect to the amount of training for first set of datasets. An epoch consists of a presentation of all the data once. So 0.2 of an epoch is the presentation of 20% of the data. Note that the x axis showing the number of epochs is not a linear scale but a sliding scale showing much of the performance during the first epoch of training and then performance at a number of later points in training. The standard deviation is represented as error bars (by calculating 95% critical region) in the graphs.

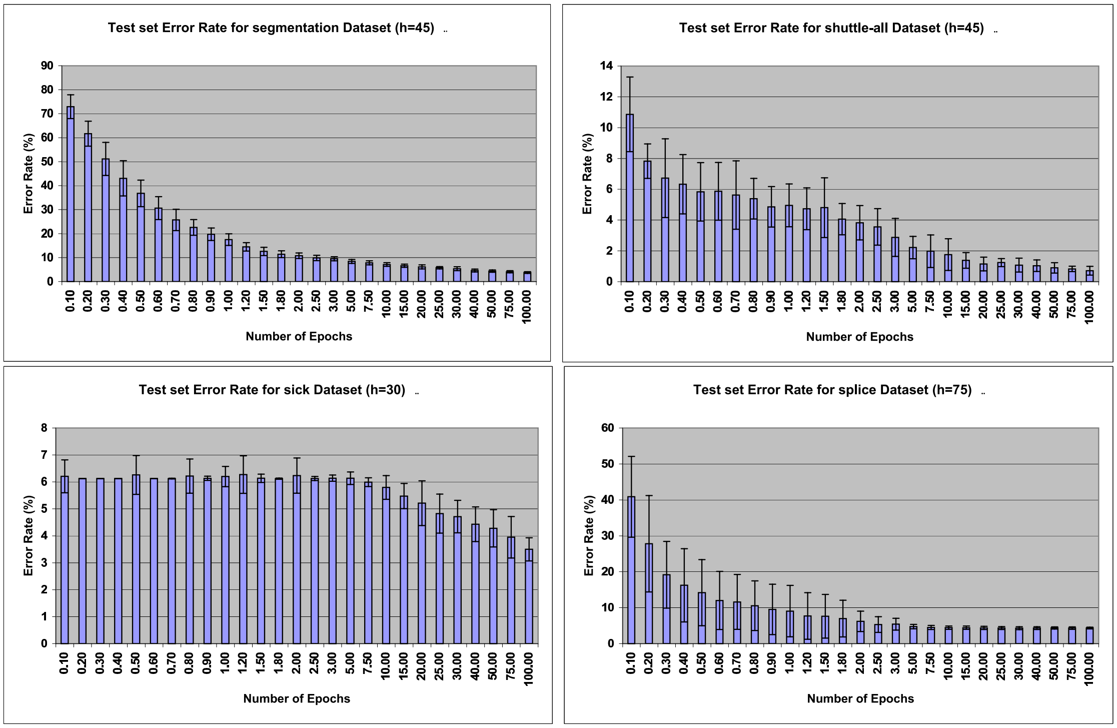


Figure 7: A plot of test set error rate rate with respect to the amount of training for second set of datasets.

3.4 Using Larger Numbers of Hidden Units

One key aspect of neural network learning is the effect of hidden nodes. Determining the neural network architecture for a given problem is a complex task. One common strategy is to use a larger number of hidden nodes than the number of input and output units. We conducted experiments with different neural network architectures for a number of datasets that we selected. Table 5 shows the test set error rates for a number of datasets with different neural network architectures. Figures 8 and 9 show the graphs for the test-set error over the first few training epochs for different values of hidden units. These graphs indicate an interesting behavior that as the number of hidden units are increasing, the convergence towards the minimum value

Table 5: Results showing the test set error rate for the NN(Baseline) algorithm for various numbers of hidden units including a large number of hidden units. All the results except ones with large number of hidden units are recorded after 100 epochs.

	NN(Baseline) with H hidden nodes					
Dataset	# Hiddens	Error	# Hiddens	Error	# Hiddens	Error
adult	20	14.89(±0.11)	40	15.14(±0.12)	60	15.01(±0.20)
msweb	40	25.08(±0.20)	80	25.03(±0.28)	120	28.67(±2.86)
hypo	15	5.92(±0.09)	30	5.89(±0.08)	45	5.92(±0.08)
kr-vs-kp	15	0.58(±0.09)	30	0.54(±0.07)	45	0.52(±0.06)
letter	40	16.18(±0.27)	80	12.98(±0.24)	120	11.64(±0.20)
satellite	15	15.96(±1.21)	30	15.31(±1.33)	45	13.63(±0.66)
segmentation	15	4.16(±0.39)	30	3.78(±0.26)	45	3.74(±0.22)
shuttle-all	15	0.70(±0.12)	30	0.68(±0.08)	45	0.71(±0.17)
sick	10	3.60(±0.44)	20	3.48(±0.34)	30	3.50(±0.25)
splice	25	4.75(±0.25)	50	4.47(±0.22)	75	4.34(±0.14)

gets delayed (i.e., the error rate starts decreasing with an initial delay). But for many of the datasets, the network with more hidden units produces lower error rates than the ones with less hidden units at 100 epochs. A possible reason for this behavior is that the neural network takes a few epochs to adapt the network architecture with the problem at hand and this initial number of epochs often increases as the number of hidden units increases. Our experiments indicated that the larger the number of hidden nodes in the network, the more accurate the network. In the next section we investigate whether it is possible to alter neural network training so that the convergence can be achieved more quickly. We present experiments by varying the learning parameters of the neural network.

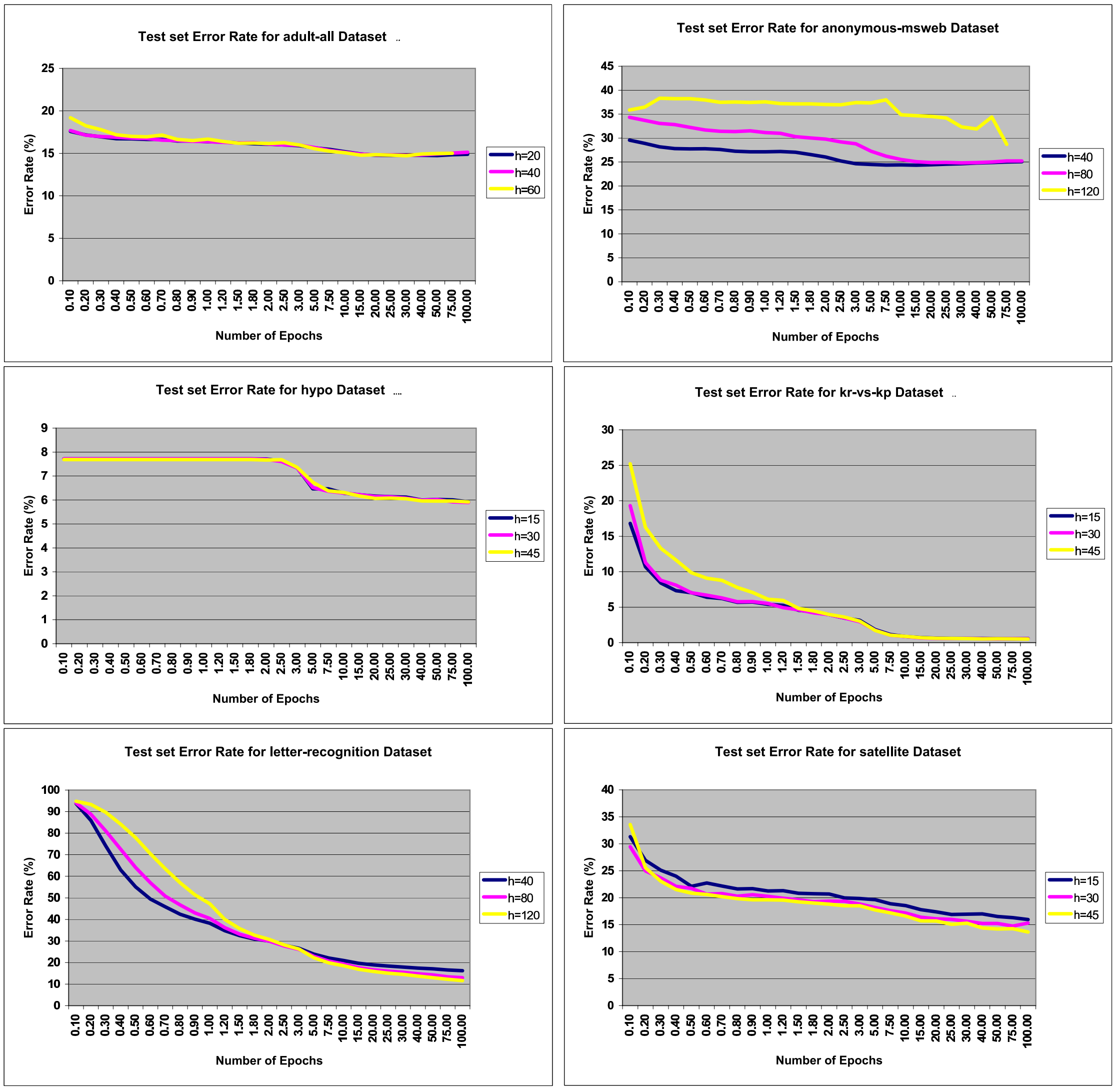


Figure 8: A plot of test set error rate rate with respect to the number of training epochs used for first set of problems for different values of hidden units.

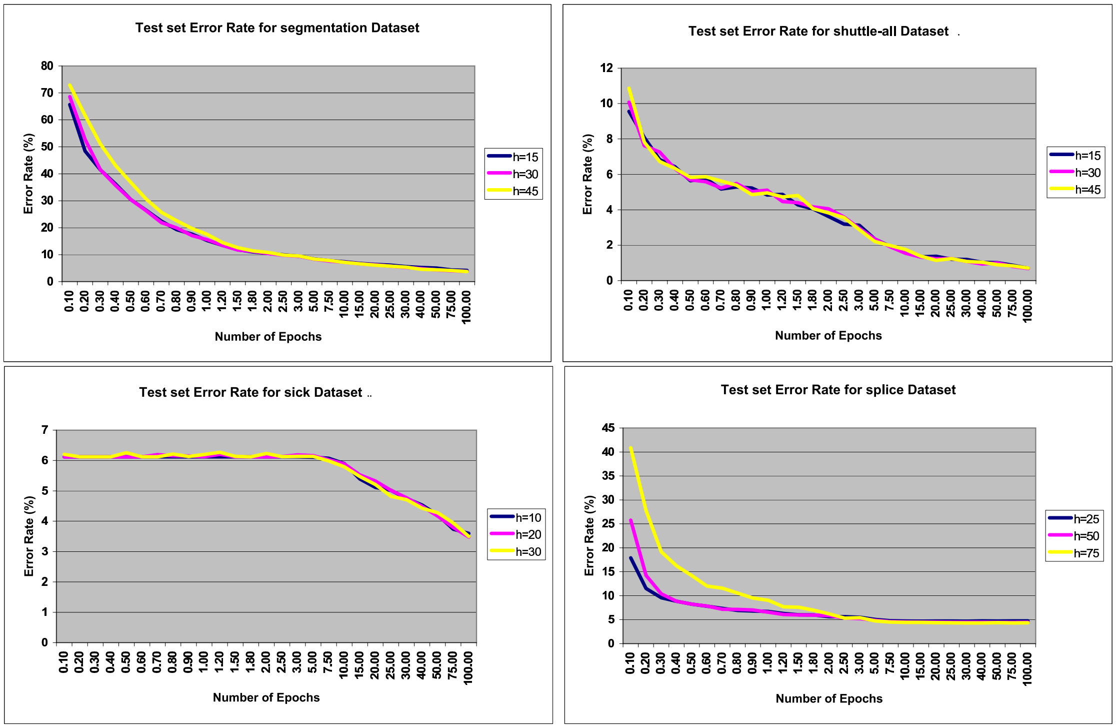


Figure 9: A plot of test set error rate rate for different values of hidden nodes for second set of datasets

3.5 Varying Learning Parameters

One possible approach for handling very large datasets is to use different values of two key parameters that greatly affect convergence time: the learning rate and the momentum. We investigate these parameters to try to discover values that could be used to achieve high accuracy in a very small number of epochs. To test our results we performed experiments on a number of datasets.

3.5.1 Varying the Learning Rate

One simple aspect of neural learning which could lead to faster convergence is to alter the learning rate. The learning rate controls the weight updates at each time step. If the learning rate is large, the error may oscillate wildly because the weight updates are large (possibly overshooting the minimum). If the learning rate is small, the weights move very slowly towards their optimum value due to smaller steps. Estimating the optimum learning rate for a given problem is a complex task. We experimented with various values of learning rate in the range 0.15 to 0.5.

Figures 10 and 11 show the test-set error over a number of training epochs for different values of the learning rate on various datasets. For many of the datasets, the smaller learning rate values resulted in smaller error rates (and in better results) but no single value is effective in all cases. For hypo and sick datasets, all the values of learning rate seem to produce similar error rates. In many cases, the learning rate value 0.15 seems to produce reasonable error rates. From our experiments we concluded that choosing a single learning rate to speed convergence is impossible, and next focused on investigating the momentum.

3.5.2 Varying the Momentum

The momentum adds a fraction of previous weight update value to the current one. Momentum helps to avoid local minima by speeding up the learning when the weights are moving in a single direction continuously by increasing the size of steps. In our experiments we used various values for momentum from 0.8 to 0.99. As with our experiments with changing the learning rate, we found that altering the momentum could significantly affect the convergence time, but that different values worked better for different problems. Figures 12 and 13 show the convergence of error against the training time for various problems. These graphs show that in general larger values of momentum result in lower error rates but again no single value is effective in all cases. Again for hypo and sick datasets, we found that different values of momentum result in almost similar error rates. For the letter

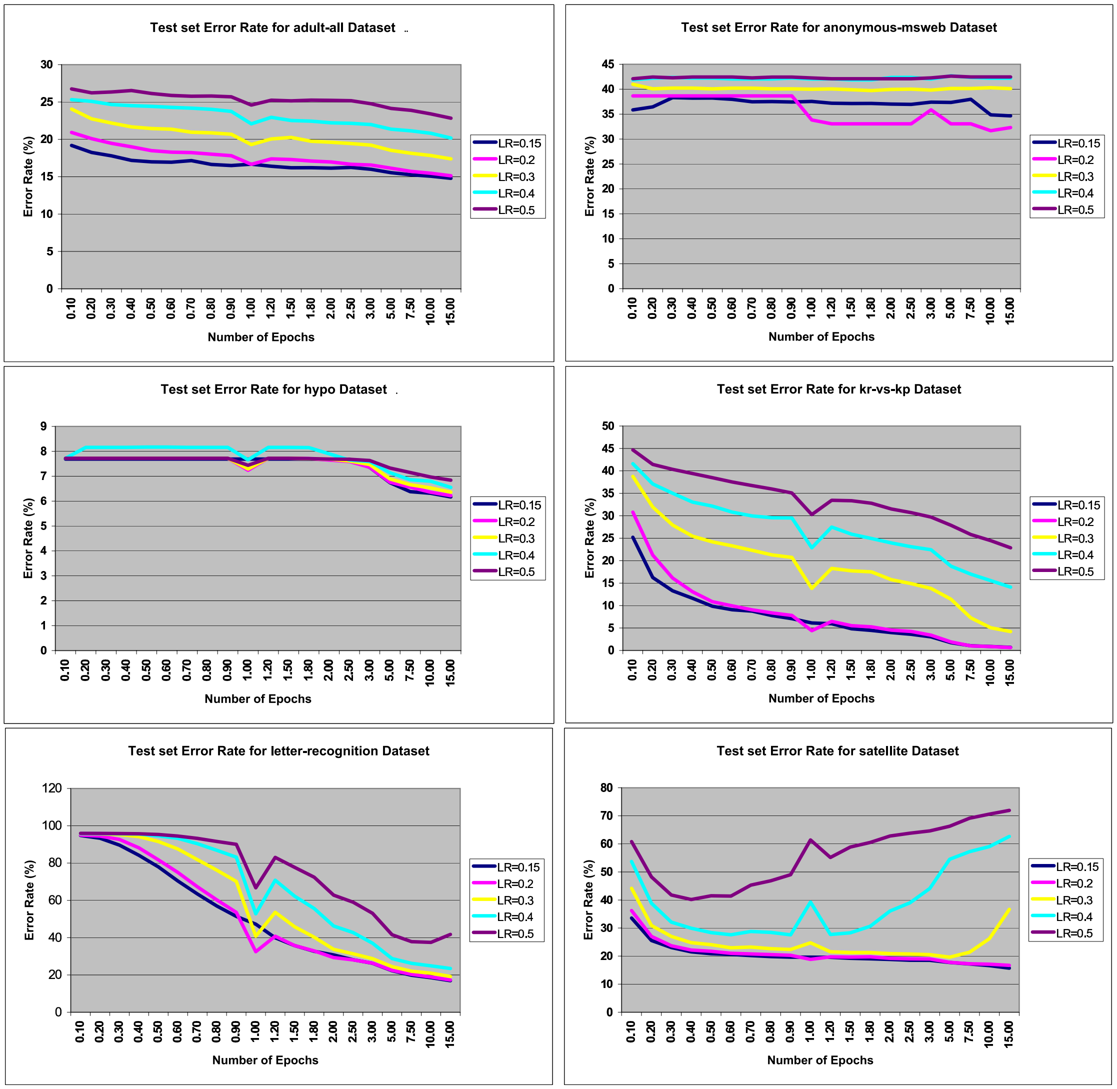


Figure 10: A plot of test set error rate rate with respect to the number of training epochs used for first set of problems for different values of the learning rate.

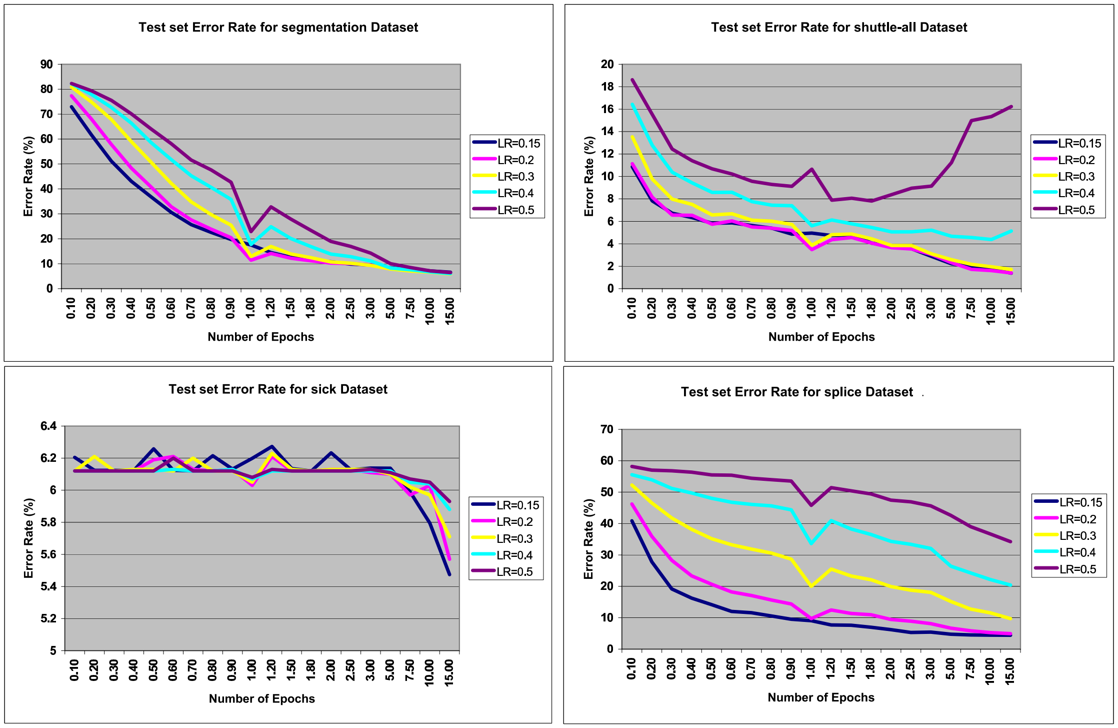
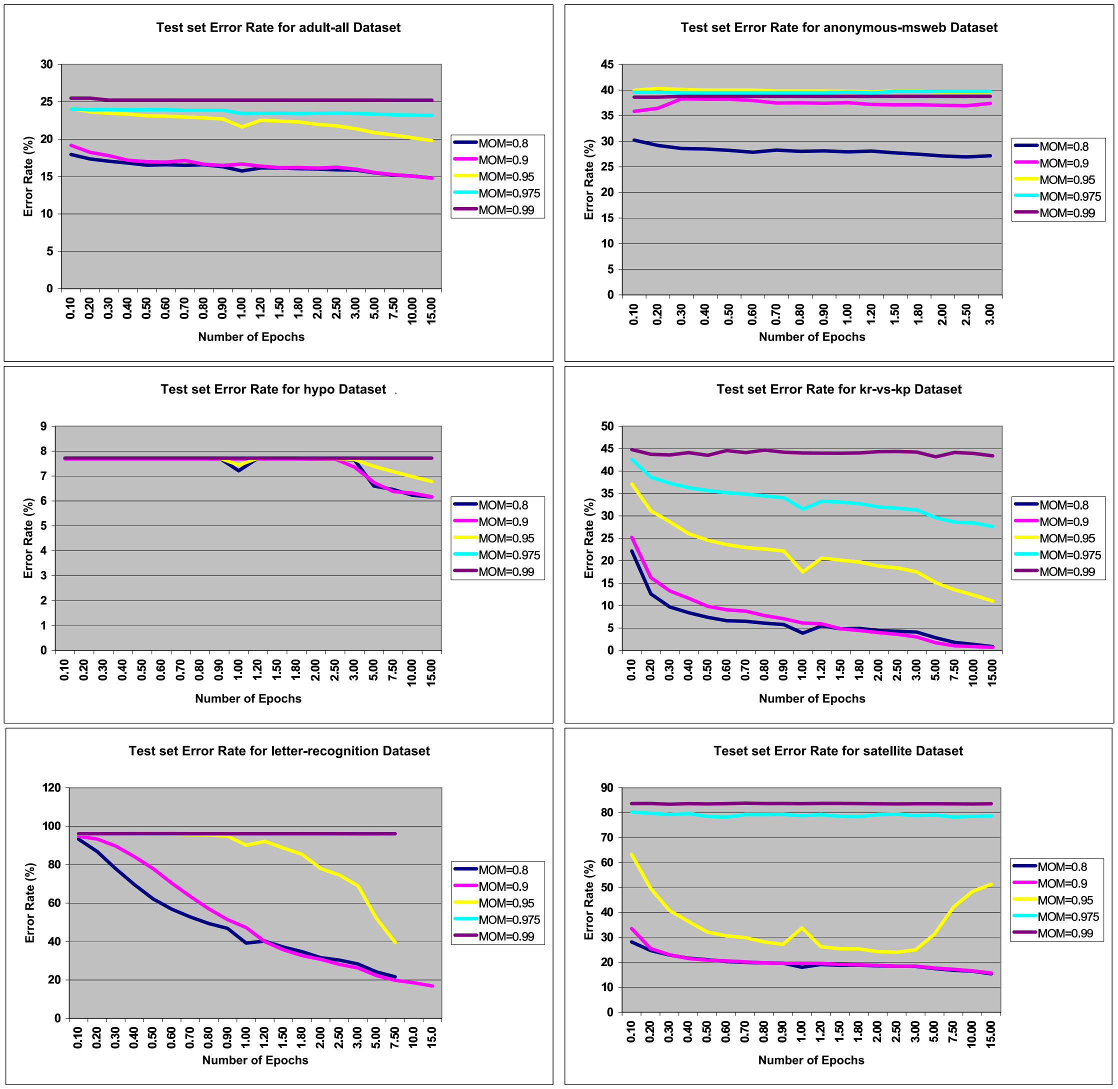


Figure 11: A plot of test set error rate rate for different values of learning rate for second set of problems

dataset, the learner does not learn at all for larger values of momentum, keeping it above 90%. For the adult dataset, the larger values of momentum seem to reduce the error rate, but still the smaller momentum values produce better results. We found that no single value of momentum outperforms our standard momentum 0.9 in terms of convergence time over all of the datasets.

Our empirical results show that varying the learning rate and momentum can produce faster convergence in some cases, but that no single value of either parameter is effective for all problems. In pilot studies using other learning methods such as Quickprop [Fahlman, 1988], we found similar effects. We therefore decided to investigate an approach that could take advantage of the type of problem we are



Test set Error Rate for adult-all Dataset

Test set Error Rate for anonymous-msweb Dataset

Test set Error Rate for hypo Dataset

Test set Error Rate for kr-vs-kp Dataset

Test set Error Rate for letter-recognition Dataset

Teset set Error Rate for satellite Dataset

Figure 12: A plot of test set error rate rate with respect to the number of training epochs used for first set of datasets for different values of the momentum parameter.

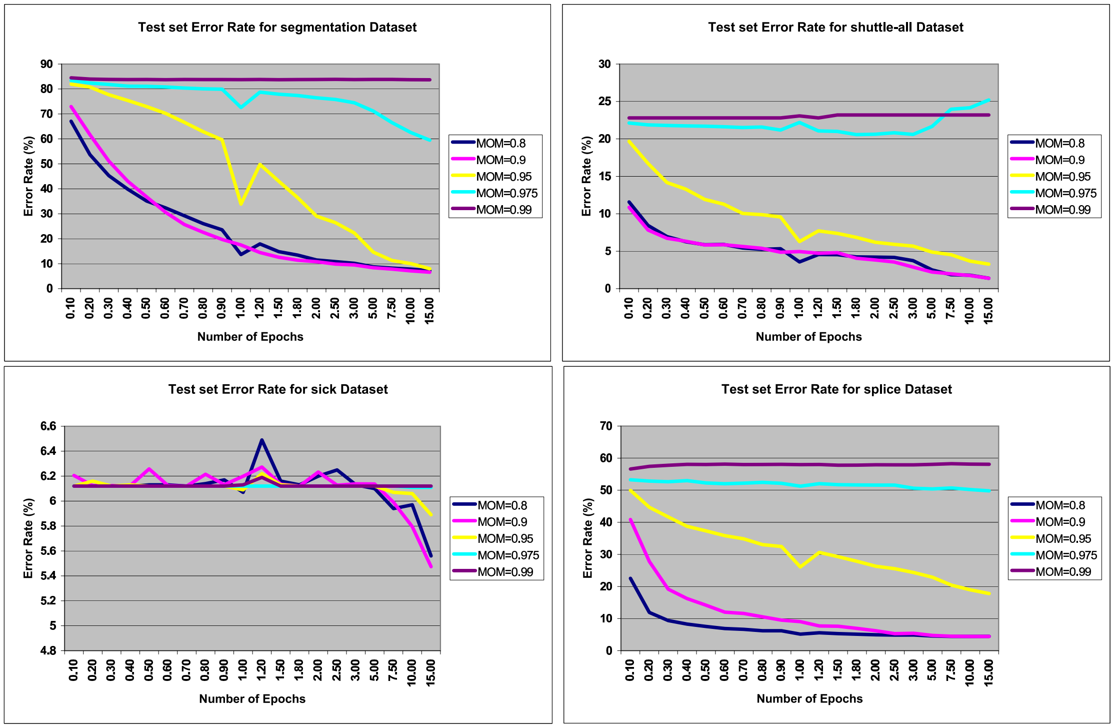


Figure 13: A plot of test set error rate rate for different values of momentum for second set of datasets

interested in – one where there is a large amount of data.

4 Learning from Subsets of the Dataset

In order to be able to learn a network in a single pass through the data we decided to use an approach of reading the data into memory one group at a time and learning on that group. Using this approach we can select the group size (the subset of the data) so that it fits into memory. We can then set the number of groups to the number we would need to read each data point into memory once. Catlett [1991a, 1991b] notes that one of the drawbacks to focusing on subsets of the data is that a model built only using that subset may be inferior to a classifier using all of the data. And Fahlman and Lebiere [1990] in their research on Cascade Correlation noted the *moving target problem* which occurs when the dataset changes characteristics (which may occur for different subsets), causing the network to unlearn its current weights.

In our approach, we use subsets of the data to train the neural network, but we focus on producing a single neural network that is based on all of the data. Thus we hope to overcome the limitation noted by Catlett. To address the problem of having the network unlearn the information it got from training on previous subsets of data, we examine an approach related to the learning algorithm Cascade Correlation [Fahlman and Lebiere, 1990]. In Cascade Correlation, a new hidden unit is introduced once training error appears to plateau. This process continues until error no longer appears to decrease. In our work, we add one or more hidden units after each subset of data. We also set the learning rate on the weights for the earlier hidden units to much lower values, so the next subset of data is unlikely to change the weights of those hidden units (thus preserving the knowledge of the earlier hidden units).

Note that we refer to this as a one pass algorithm because it only requires only one pass over the dataset in terms of reading the data into memory. The algorithm still requires multiple passes over the data for learning while in memory, but each data page gets read only once. Thus, if the main bottleneck is disk usage, this algorithm will fare very well.

4.1 Training the Network One Subset of Data at a Time

Our first approach to learning using subsets of the data is the straightforward approach of simply training the current network on each successive subset of the data, we call it NN(Subset). Table 6 summarizes this algorithm. Although this approach may be very effective, it has the potential that with each subset of data we may unlearn the weights from previous subsets.

4.2 Training Dynamically Growing Network with Subsequent Subsets of Data

To address this potential learning problem we introduce another approach to learning from subsets of the data which we call NNGrow(Subset). Table 7 gives the algorithm for this method. The main difference between this method and the previous algorithm is how the hidden units are set and used. In the NN(Subset) approach the number of hidden units is set initially and the initial network is created with that number of hidden units. The learner is then free to adjust the weights in this network with each new subset of data. In the NNGrow(Subset) approach we begin initially with a perceptron model (no hidden units) which is used to learn the initial subset of data. After each subset (except the last) we then add a set of hidden units to the network. Furthermore, we set the learning rate on the previous weights to be much smaller so that while learning is still possible, the weights will not change as much. The learning rate for weights is updated with the following rule,

$$\eta_{w_{j \to k}} = R \times \eta_{w_{j \to k}}, \forall j \notin NewlyAddedHiddens \tag{11}$$

where R is the reduction factor by which the learning rate is reduced.

Table 6: The NN(Subset) algorithm for training a neural network on subsets of the data in one pass through the dataset.

- Determine the number of pages P of the data that can fit into memory.
- Determine how many groups of pages G will be needed to read all of the data into memory once.
- Initialize the neural network.
- For each of G partitions:
  - Randomly select (without replacement) P pages of the dataset.
  - Train the neural network for N epochs on the current subset of the data.
- Output the resulting neural network.

In this way we hope that any changes made by the learner for each subset will be focused on the new set of hidden units that we introduced just before that subset. Figure 14 shows a sample of how a network would grow during the training for a simple problem. We introduce this method to counteract the potential problem that with each new subset, the learner may unlearn the weights from the previous subset.

One further question we should address is the distribution of the data. For this work we assume that the data is roughly randomly distributed across the pages for our experiments. In separate experiments [Maclin, 2002] using subset methods we found that the results did not statistically differ even when the data was ordered. Partly, the ordering problem is addressed by selecting random pages of data. But in the case where the data is ordered on a feature that is critical to the model, and where the number of pages we are able to hold in memory is small but the number

Table 7: The NNGrow(Subset) algorithm for training a neural network on subsets of the data. The main difference between this algorithm and the NN(Subset) algorithm presented previously is that this algorithm attempts to preserve the knowledge learned in the previous steps by adding hidden units that focus on the current subset of the data. Differences between this algorithm and the NN(Subset) algorithm are shown in *italics*.

- Determine the number of pages P of the data that can fit into memory.
- Determine how many groups of pages G will be needed to read all of the data into memory once.
- Initialize the neural network with some number of starting hidden units.
- For each of G partitions:
  - Randomly select (without replacement) P pages of the dataset.
  - Train the neural network for N epochs on the current subset of the data.
  - *If this is not the last of the G partitions:*
    - *Set the learning rate of the current weights in the network to the standard value (0.15) times a reduction factor (R)*
    - *Add a set of H hidden units to the network that have connections from the input units and previous hidden units and connections to the output units where the learning rate on these connections is set to the standard value (0.15)*
- Output the resulting neural network.

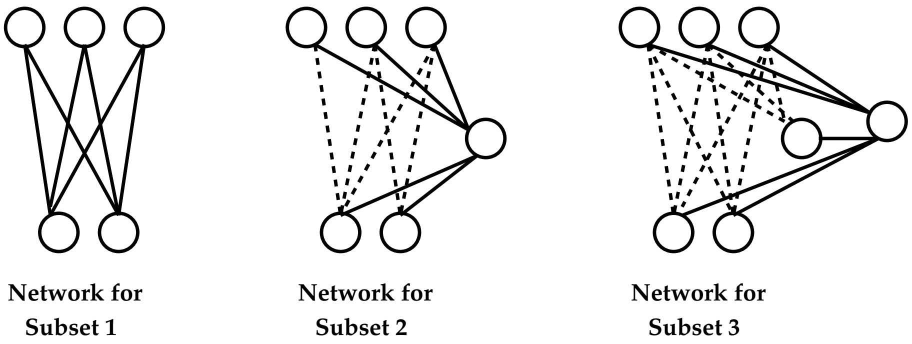


Figure 14: A sample of the NNGrow process for adding hidden units where there are three inputs, two outputs, and one hidden unit is added after each subset. For each subset, the weights shown with solid lines have the normal learning rate and the weights with dashed lines have a significantly reduced learning rate.

of data points on a page is large, we could get subsets of the data that are skewed. For this case we would suggest an alternate method for selecting the subsets of the data that would use overlapping subsets. Initially, we would select a random set of pages to read into memory. Then, we would randomly swap data points in memory so that each memory page was somewhat random. Then we would train the network on this subset. Next, we would select some percentage of the pages in memory (say 20%) and replace these pages with random pages from the remaining pages in the dataset. We would then swap the data points on the pages just added to memory randomly with other data points in memory. Then we would train the network with this subset. This process would continue until all of the data is read. Each succeeding subset would have 20% of its data points from new pages brought into memory and a sampling of the data points seen previously, generally leading to a fairly random distribution.

Using this approach we could also examine an alternate method for selecting which data points to delete. Breiman [1999] looks at a subset-oriented method where data points are selected to be kept in memory when the classifier does not predict them well. For our method, we could follow a similar model and select data points where the neural network has little error. We could accumulate these data points on a group of pages and replace these pages with new pages from the remaining dataset. Thus each subset would concentrate on data points we are not predicting well, which could lead to increases in accuracy similar to those shown for Boosting methods [Freund and Schapire, 1996].

4.3 Experimental Results with Subset Approach

To test our new methods NN(Subset) and NNGrow(Subset), we performed experiments on several large datasets from the UCI ML repository. As noted earlier, we expect that the datasets our algorithm would be applied to would be significantly larger. We chose the datasets shown so that we could compare our results to results for neural networks trained on the entire dataset. For a larger set of data, we expect that we would be able to use much larger subsets than we use in our experiments, which would generally help our networks converge much more quickly (thus perhaps requiring fewer training epochs while in memory).

We chose to set the subset size of the datasets so that each dataset would be divided into 10 subsets. For some of the datasets with smaller numbers of points, this in fact significantly handicaps the learner (since larger subsets of the data should help with generalization), but it was desirable that the algorithm have small groups of data to work with to determine whether it would compare favorably with training on all of the data. For the larger datasets (the last two in the Table 2), we used a paging approach to train them. In general, the regular training mechanisms require that all the data is kept in main memory which can increase the memory requirement of the process significantly thereby slowing down the process. To overcome this

Table 8: Results showing the test-set error rate for the NN(Subset) algorithm for various numbers of training epochs compared to results for a similar neural network trained on all of the data.

Dataset	NN(Subset) and NN(Baseline) with E Epochs					
	E=10		E=20		E=30	
	NN(Subset)	NN(Baseline)	NN(Subset)	NN(Baseline)	NN(Subset)	NN(Baseline)
adult	16.26(±0.13)	15.12(±0.46)	16.66(±0.17)	14.85(±0.36)	16.82(±0.15)	14.77(±0.09)
anon-msweb	33.39(±3.05)	24.39(±0.14)	33.15(±3.12)	24.44(±0.14)	30.06(±1.02)	24.64(±0.21)
hypo	6.37(±0.12)	6.29(±0.25)	6.32(±0.23)	6.06(±0.14)	6.32(±0.18)	6.08(±0.15)
kr-vs-kp	1.56(±0.22)	0.85(±0.33)	1.42(±0.14)	0.61(±0.12)	1.40(±0.19)	0.58(±0.10)
letter	18.92(±0.34)	18.53(±0.37)	16.42(±0.28)	15.88(±0.45)	15.33(±0.31)	14.52(±0.27)
satellite	16.62(±0.60)	16.61(±0.53)	15.61(±0.66)	15.68(±0.89)	15.35(±0.52)	15.29(±0.78)
segmentation	7.58(±0.29)	7.11(±0.45)	6.57(±0.52)	6.11(±0.48)	6.15(±0.35)	5.34(±0.47)
shuttle-all	1.67(±0.28)	1.57(±0.59)	1.21(±0.20)	1.15(±0.26)	1.01(±0.22)	1.19(±0.26)
sick	5.89(±0.35)	5.80(±0.25)	5.73(±0.50)	5.21(±0.48)	5.27(±0.53)	4.71(±0.35)
splice	5.23(±0.31)	4.44(±0.28)	5.15(±0.27)	4.49(±0.26)	5.21(±0.24)	4.30(±0.23)
forest-cover	33.05(±1.09)	34.73(±0.27)	30.64(±1.28)	28.52(±1.45)	25.38(±1.32)	21.50(±0.25)
synthetic	89.19(±3.44)	85.14(±2.35)	88.88(±0.88)	83.17(±2.65)	82.25(±3.12)	82.07(±9.47)

difficulty, we implemented a simulator that takes into account the main memory size and performs its own paging.

Initially, we compare the simplest subset algorithm we developed, NN(Subset), with training on the entire dataset. The main parameter we have to set in NN(Subset) is how many training epochs each subset of the data gets while in memory. We varied this parameter over the possible values 10, 20, 30, 50, 75, and 100. Tables 8 and 9 show results from these experiments. As can be seen, for

Table 9: Results showing the test-set error rate for the NN(Subset) algorithm for various numbers of training epochs compared to results for a similar neural network trained on all of the data.

Dataset	NN(Subset) and NN(Baseline) with E Epochs					
	E=50		E=75		E=100	
	NN(Subset)	NN(Baseline)	NN(Subset)	NN(Baseline)	NN(Subset)	NN(Baseline)
adult	16.99(±0.20)	14.99(±0.14)	16.98(±0.11)	15.01(±0.20)	17.07(±0.12)	15.01(±0.19)
anon-msweb	31.43(±1.65)	24.87(±0.18)	33.92(±0.00)	25.01(±0.18)	33.92(±0.00)	25.08(±2.87)
hypo	6.42(±0.18)	6.02(±0.12)	6.33(±0.14)	5.96(±013)	6.42(±0.18)	5.92(±0.08)
kr-vs-kp	1.39(±0.16)	0.54(±0.07)	1.41(±0.19)	0.53(±0.06)	1.38(±0.16)	0.52(±0.06)
letter	14.08(±0.23)	12.98(±0.37)	13.57(±0.23)	12.20(±0.29)	13.34(±0.26)	11.64(±0.20)
satellite	15.06(±0.38)	14.21(±0.55)	14.87(±0.51)	14.25(±0.69)	15.49(±1.08)	13.63(±0.67)
segmentation	5.69(±0.38)	4.33(±0.27)	5.58(±0.31)	4.08(±0.26)	5.30(±0.45)	3.74(±0.22)
shuttle-all	0.96(±0.17)	1.01(±0.19)	0.86(±0.15)	0.83(±0.10)	0.74(±0.18)	0.70(±0.17)
sick	5.04(±0.36)	4.71(±0.60)	5.91(±0.41)	3.95(±0.44)	4.62(±0.34)	3.50(±0.24)
splice	5.17(±0.23)	4.30(±0.17)	5.17(±0.35)	4.32(±0.20)	5.19(±0.17)	4.34(±0.14)
forest-cover	24.18(±1.11)	19.34(±0.34)	24.02(±1.40)	18.49(±0.87)	22.32(±2.62)	17.14(±0.26)
synthetic	82.11(±0.52)	81.42(±0.56)	81.24(±1.29)	80.89(±1.12)	80.12(±1.87)	78.34(±1.23)

smaller number of training epochs, the NN(Subset) results are similar to the results obtained for the entire dataset, and only statistically significantly worse for few datasets (adult, kr-vs-kp, and splice). The NN(Subset) method outperforms the NN(Baseline) method for anonymous-msweb dataset for smaller number of epochs. A possible explanation for the poor performance of NN(Subset) algorithm for some datasets would be that the distribution in these datasets may be skewed. But for many cases the results with NN(Subset) method are comparable to NN(Baseline)

Table 10: Results showing the error rate for NNGrow(Subset) approaches along with the baseline results. Shown are different values of N for the subset and baseline approach.

Dataset	NNGrow(Subset) and NN(Baseline) with E Epochs					
	E=10		E=20		E=30	
	NNGrow	NN(Baseline)	NNGrow	NN(Baseline)	NNGrow	NN(Baseline)
	(Subset)		(Subset)		(Subset)	
adult	15.43(±0.10)	15.12(±0.46)	15.58(±0.14)	14.85(±0.36)	15.69(±0.00)	14.77(±0.09)
anon-msweb	24.74(±0.26)	24.39(±0.14)	25.41(±0.19)	24.44(±0.14)	25.67(±0.00)	24.64(±0.21)
hypo	6.39(±0.12)	6.29(±0.25)	6.22(±0.12)	6.06(±0.14)	6.13(±0.05)	6.08(±0.15)
kr-vs-kp	2.68(±0.18)	0.85(±0.33)	2.01(±0.32)	0.61(±0.12)	1.81(±0.12)	0.58(±0.10)
letter	21.30(±0.28)	18.53(±0.37)	17.27(±0.27)	15.88(±0.45)	15.44(±0.18)	14.52(±0.27)
satellite	15.26(±0.32)	16.61(±0.53)	13.34(±0.33)	15.68(±0.89)	12.83(±0.49)	15.29(±0.78)
segmentation	8.34(±0.31)	7.11(±0.45)	7.19(±0.24)	6.11(±0.48)	6.47(±0.47)	5.34(±0.47)
shuttle-all	1.52(±0.24)	1.57(±0.59)	1.13(±0.22)	1.15(±0.26)	0.88(±0.06)	1.19(±0.26)
sick	6.07(±0.07)	5.80(±0.25)	5.59(±0.32)	5.21(±0.48)	5.17(±0.21)	4.71(±0.35)
splice	4.93(±0.27)	4.44(±0.28)	4.83(±0.28)	4.49(±0.26)	4.67(±0.30)	4.30(±0.23)
forest-cover	38.26(±1.98)	34.73(±0.27)	35.76(±2.48)	28.52(±1.45)	28.02(±0.80)	21.50(±0.25)
synthetic	89.22(±0.27)	85.14(±2.35)	85.35(±2.46)	83.17(±2.65)	82.98(±1.42)	82.07(±9.47)

method.

In our second set of experiments, we compare results from our NNGrow(Subset) algorithm to the results from a neural network trained on the entire dataset. We set the number of training epochs to 10, 20, 30, 50, 75, and 100 and the reduction factor R to 0.1.

Tables 10 and 11 shows results from these experiments. From these results, it is evident that the results for NNGrow(Subset) are statistically significantly worse

Table 11: Results showing the error rate for NNGrow(Subset) approach along with the baseline results. Shown are different values of N for the subset and baseline approach.

Dataset	NNGrow(Subset) and NN(Baseline) with E Epochs					
	E=50		E=75		E=100	
	NNGrow	NN(Baseline)	NNGrow	NN(Baseline)	NNGrow	NN(Baseline)
	(Subset)		(Subset)		(Subset)	
adult	16.41(±0.16)	14.99(±0.14)	16.76(±0.13)	15.01(±0.20)	16.94(±0.13)	15.01(±0.19)
anon-msweb	25.86(±0.26)	24.87(±0.18)	25.97(±0.18)	25.01(±0.18)	26.05(±0.24)	25.08(±2.87)
hypo	6.18(±0.15)	6.02(±0.12)	6.23(±0.13)	5.96(±013)	6.24(±0.12)	5.92(±0.08)
kr-vs-kp	1.75(±0.25)	0.54(±0.07)	1.66(±0.18)	0.53(±0.06)	1.66(±0.23)	0.52(±0.06)
letter	14.94(±0.27)	12.98(±0.37)	14.99(±0.27)	12.20(±0.29)	15.07(±0.33)	11.64(±0.20)
satellite	12.28(±0.37)	14.21(±0.55)	12.02(±0.37)	14.25(±0.69)	12.18(±0.50)	13.63(±0.67)
segmentation	5.84(±0.26)	4.33(±0.27)	5.36(±0.42)	4.08(±0.26)	5.10(±0.34)	3.74(±0.22)
shuttle-all	0.54(±0.10)	1.01(±0.19)	0.39(±0.07)	0.83(±0.10)	0.36(±0.03)	0.70(±0.17)
sick	4.63(±0.27)	4.71(±0.60)	4.48(±0.33)	3.95(±0.44)	4.30(±0.36)	3.50(±0.24)
splice	4.85(±0.23)	4.30(±0.17)	4.87(±0.24)	4.32(±0.20)	4.89(±0.34)	4.34(±0.14)
forest-cover	27.45(±0.37)	19.34(±0.34)	25.12(±2.09)	18.49(±0.87)	23.34(±1.76)	17.14(±0.26)
synthetic	82.04(±0.40)	81.42(±0.56)	81.42(±0.83)	80.89(±1.12)	79.24(±1.58)	78.34(±1.23)

than NN(Baseline) results only for the kr-vs-kp dataset and for all the other datasets, both the results are comparable. For anonymous-msweb, satellite, and shuttle-all datasets the NNGrow method produces significantly lower error rates than both the NN(Subset) and NN(Baseline) methods. For the datasets letter, segmentation NN(Subset) method produces lower error rates than NNGrow(Subset) but the margin between between both the results is very small. In general NNGrow(Subset) performs well on most of the datasets and in many cases the accuracy achieved with this method is very close to one obtained with the standard neural networks trained on all of the data.

Overall, our results suggest that a neural network can be built in one pass through a large dataset that is generally comparable to one built in many passes over the same dataset. For these results, we do not report timing data since we chose datasets that can fit into memory (and therefore swapping is not an issue), but both of our methods still perform faster than the neural network method trained on all of the data.

5 Additional Related Work

Our work can be related to work from a number of areas. First we compare our approach against the cascade correlation algorithm proposed by Fahlman and Lebiere [1990]. Then we discuss one of the primary areas of our related research that focuses on training on subsets of the data. Another interesting related research area is work on building classifiers for very large databases. The third and significant research area which is directly related to our work is research on investigating methods for speeding up the learning of a neural network. Researchers have also examined several other areas that relate to our work in various ways. In this section we will discuss all these research areas in detail and compare our approach with that work.

5.1 The Cascade-Correlation Algorithm

First we compare our method against the cascade correlation architecture proposed by Fahlman and Lebiere [1990] . The basics of this algorithm are shown in Figure 15. It begins with a perceptron network and it keeps adding hidden units one by one until the learning is almost complete. For each new hidden unit, the output is correlated to the magnitude of the residual error that is to be eliminated (that is, at each step, the error that has to be eliminated, called residual error, is computed and compared against the output). Once a new hidden unit is added to the network, its input weights do not change and only the output weights are repeatedly trained using the Quickprop algorithm [Fahlman, 1988]. Thus each hidden unit may capture a certain feature in the output layer.

The performance of this algorithm is tested with benchmark problems such as two-spirals and N-input parity and cascade correlation outperforms backpropagation in terms of the convergence rate. The main advantage of this approach is that

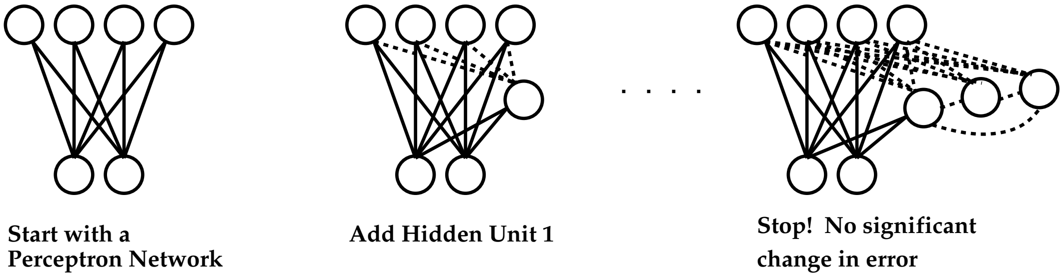

Start with a Perceptron Network
Add Hidden Unit 1
Stop! No significant change in error

Figure 15: Steps of the Cascade-Correlation algorithm. The network begins as a perceptron (no hidden units). After a certain number of epochs (controlled by the user), a hidden unit is added and the error is measured. This process repeats until the error is very small. Whenever a new hidden unit is added, its input weights are frozen (represented by dashed lines in the figure). Only the perceptron weights, weights from the hiddens to the output units and other weights are trained. Note that each new hidden unit is connected to all of the existing hidden units.

the network architecture for a given arbitrary classification problem need not be predetermined and the algorithm determines the size and depth of the network.

Our NNGrow (Subset) algorithm resembles this approach in how it adds hidden units, but it differs significantly in that our algorithm uses lower learning rate for older weights in the network, instead of completely freezing those weights. Also the process of adding the hidden units stops when we complete training all the data, rather than when the error reaches the optimum value. In NNGrow, the network architecture is predetermined (based on the number of subsets and the number of hidden units to add each time).

5.2 Training subsets of data

Catlett [1991a, 1991b] explored the concept of using subsets of a dataset for training, especially when there is a large amount of data. He found that classifiers trained on subsets were generally inferior to classifiers trained on all of the data, despite investigating a number of subset methods. But he did not focus on methods for making use of all the data as a series of subsets. Our work differs in that we use all of the data for training, but we keep only a subset of data in memory and train it.

More recently, Breiman [1999] has introduced a method for creating an ensemble classifier in a small number of passes through the data. His method also works by selecting subsets of the data and then training a separate classifier on each subset. In Breiman's approach, a new classifier is grown on each sample of data and aggregated with the previous classifiers. He suggested two methods for pasting votes: *pasting Rvotes* and *pasting Ivotes*. Pasting Rvotes or random votes is simple in the way that each example has equal probability of getting selected, but is less effective in terms of accuracy. Pasting Ivotes or important votes works by selecting the examples based on whether the out-of-bag classifier classifies them correctly. An out-of-bag classifier is constructed by first forming bootstrap training sets then and collecting the bagging predictor votes from the examples omitted by the bootstrap sample. This method of pasting Ivotes proves to be a more accurate method than pasting Rvotes. The error estimate after adding each classifier is compared against the previous error and the process continues as long as each classifier seems to improve the overall accuracy. Breiman's method includes a test that allows the learner to decide when to terminate learning (possibly before even one pass of the data has been completed). One of the key idea in this algorithm is that increased weight is applied to those examples that are more likely to be misclassified. The time and memory estimates for large datasets with this algorithm show that it can scale up to terabyte databases.

This algorithm is similar to our approach in a way that the entire data base is never in the main memory at any time. But our approach significantly differs in that we are able to use all of the data in building our single classifier, possibly resulting in a more accurate single classifier and a faster classification method. Also we need only sequential access to the data on disk.

Street and Kim [2001] examined an ensemble approach where the data is selected similarly to our approach. Their work builds a classifier for each subset and then decides whether to add that classifier to the ensemble. In this approach a fixed size ensemble is maintained, each of the component classifiers are combined into an ensemble and once the ensemble is full, each subsequent classifier is added only if it satisfies a particular criterion and in that case one of the existing component classifier is removed. A key aspect with this algorithm is to decide which existing classifier to remove when a new classifier is added. The classifiers that do well on those points the ensemble misclassifies are favored in this process. That is, if both the ensemble and the new classifier agree on an example and they classify it correctly, then the quality measure for that classifier is increased and the quality measure is decreased otherwise. One of the drawbacks of this method is that this algorithm is not very good at handling concept drift, a phenomenon that indicates the change in the concept domain when the knowledge is out of date. The similarity of this approach with our algorithm is that this method builds the final ensemble in just a single pass through all the data blocks, but in a less obvious way. This approach resembles the Breiman's approach in a way that only part of data is in main memory at any point of time. But again both these algorithms do not make use of all the available data for training which might sometimes result in a less accurate classifier. Our algorithms significantly differ from these two approaches in that we focus on producing a single classifier incorporating all of the data, possibly resulting in a more accurate one.

5.3 Learning on Very Large Databases

Several learning algorithms have been suggested which work efficiently on very large databases. Most of these algorithms are based on decision trees and association rules. Neural networks have received less attention in this area compared to other machine learning techniques. Our work differs mainly in that we focus on neural network learning, a learning method that has proven to be very accurate for many datasets.

Toivonen [1996] proposed a method for discovering association rules, which are one of the most popular KDD techniques, for large databases. Association rules are if-then else rules that have probabilistic information attached to them. For example "beer + chips ⇒ diapers (80%)" is an association rule which implies that 80 percent of customers who buy beer and potato chips also buy diapers. The essence of this algorithm is that it finds the association rules in just one pass over the database. The idea is to pick a random sample of data, develop the association rules and then verify those rules with the rest of the database. But this algorithm suffers from the problem that the classifier built with these association rules is inferior to the one built using the whole database. For example, an association rule that is very frequent in the whole database might not be frequent in the random sample selected which results in omission of that association rule. That is, the accuracy is sacrificed against the efficiency. Our algorithm helps avoid this trade off by training on the whole database, but one sample at a time.

Gehrke et al., [1999] looked at a method that constructs an optimal decision tree for a large dataset in only two scans through the data. Many of the current decision tree construction algorithms work by making one pass through the data to develop one level of the tree. To address this issue, they introduced a new algorithm called BOAT (**B**ootstrapped **O**ptimistic **A**lgorithm for **T**ree Construction). In this approach, an initial decision tree is constructed from a random sample of the large dataset that can fit into the memory. This tree is refined with the other data in dataset by making corrections each time the tree fails to predict correctly. Using bootstrapping, this initial tree is modified to obtain a new tree which is close to the one built using all of the data. The possible advantages with this approach are that: (1) this is an incremental method which enables modifications to the tree upon the addition of new examples of training data rather than reconstructing the tree from scratch and (2) it develops the tree in fewer passes through the data making it efficient to work with large datasets. The similarity of this approach with our algorithm is that only a small subset of data that can fit in memory is trained at a time. But our algorithm is able to use all of the training data for training instead of training for a sample of data and improving the model with the other data. In this way we possibly have a more accurate classifier as we do not leave out any data from training.

Recently Fung and Mangasarian [2000] proposed an efficient method to learn from a large dataset using support vector machines. Support vector machines [Vapnik, 1995] are based on the Structural Risk Minimization (SRM) principle that helps to minimize the upper bound on the expected risk, where as the Empirical Risk Minimization (ERM) principle which is generally employed by neural networks helps to minimize the error on training data. Support vector machines have proved to be a successful classification scheme especially in pattern recognition and regression problems. The learning in support vector machines works by determining the support vectors which help to draw the decision boundary. These support vectors are constructed by solving a set of equations that optimize the expected risk on the test set (probability of test set error). The complexity of the algorithm depends on the number of support vectors. Fung and Mangasarian proposed an approach called minimal support vector machine that selects a small subset of support vectors that classify the whole data. They attempt to construct a decision boundary that is at equal distance between the edges of the two surfaces (that determine the two classes of the problem) thus having a maximum margin between them. This method ensures the creation of a small vital subset of data points from the original large dataset that can be used for incremental methods to merge with future data. Our approach differs in the way that we focus on neural network learning for classification. Also we attempt to train on all of the data (a subset at a time) and produce a classifier, whereas this minimal support vector machine approach attempts to find a selection of data that represents the features of the whole dataset.

5.4 Methods for speeding up the learning in neural networks

A third area of related research includes other methods for speeding up the learning of neural networks. Researchers have proposed methods for more quickly updating the weights in the network.

For example, Riedmiller and Braun [1993] suggest an approach the adapts the weight update based on examining the derivative of the error with respect to the weight. This method, known as RProp (**R**esilient **Prop**agation), is a well known but simple variant of backpropagation. This method is based on the change of sign of the partial derivative of the error function with respect to each weight, $\frac{\partial E}{\partial w_i}$ in the previous step to the partial derivative in the current step. First the weight update amount is computed for each weight. If the sign of the derivative changes, then that means the local minima got skipped and the weight update change is retained to its original value (Figure 16). If the sign did not change, then the update is optimal and the weight update value is increased slightly in order to achieve quicker convergence.

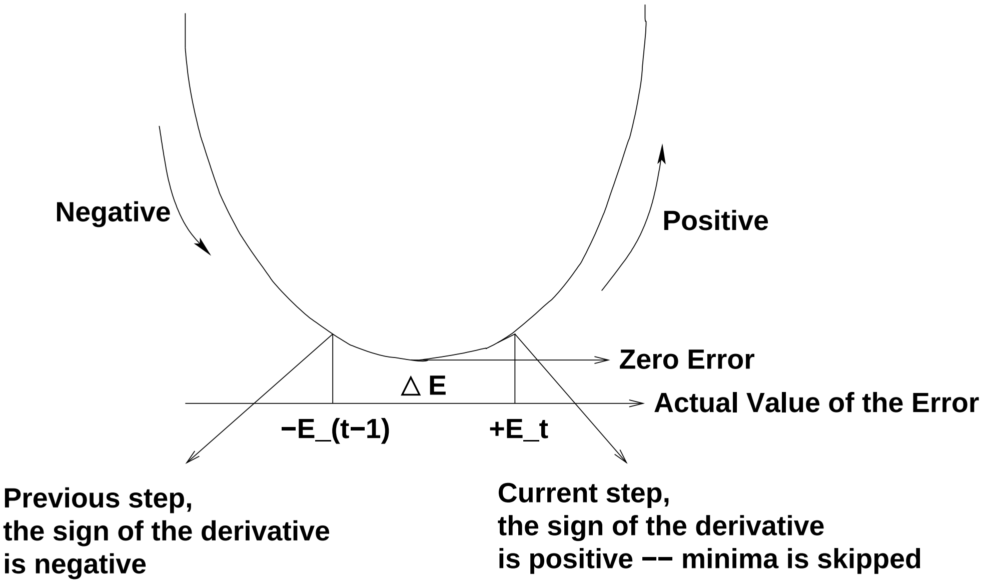

Negative
Positive
Zero Error
△ E
Actual Value of the Error
−E_(t−1)
+E_t
Previous step,
the sign of the derivative
is negative
Current step,
the sign of the derivative
is positive −− minima is skipped

Figure 16: The effect of change of sign of the derivative of the error in RProp. Before applying the weight update, the sign of the derivative is negative. After the weight update amount (ΔE) is added, then the the sign of the derivative is now positive which is an indication that the minima is skipped. The figure demonstrates the transition from negative error to the positive error without reaching the zero point.

Once the weight update amount is adapted, then it is determined whether to add this amount or to subtract this amount. This is done by observing the sign of $\frac{\partial E}{\partial w_i}$. If this sign is positive, it means that the error has increased and this weight update is subtracted from the weight. If the sign is negative, the update is added to the weight. RProp converges in 19 epochs on average 10 - 5 - 10 encoder problem, where as backpropagation requires 121 epochs. Here the 10 - 5 - 10 encoder refers to a neural network with three layers of units, two layers of modifiable weights (also called hidden layers), 10 input units, 5 hidden units and 10 output units. For the 12

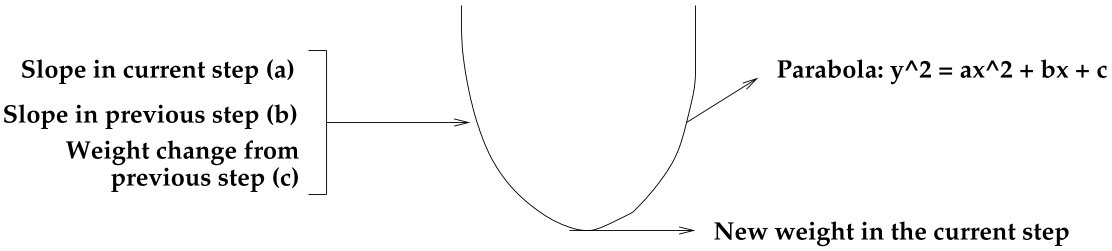

Slope in current step (a)
Slope in previous step (b)
Weight change from previous step (c)
Parabola: y^2 = ax^2 + bx + c
New weight in the current step

Figure 17: Weight update rule in QuickProp. The weight update amount is computed using the parabola constructed using the three parameters.

- 2 - 12 problem RProp takes 322 epochs and backpropagation over 15,000 epochs. One of the similarities of this approach with our algorithm is that both the methods work to speed up the error convergence, but our approach is significantly different from this method as it does not manipulate the basic workings of backpropagation.

Fahlman [1988] looked at a method, that he calls **Quick Prop**agation, that attempts to model the error surface as a parabola, and uses this surface to try to jump to the appropriate weight value in one step. QuickProp works by trying to adapt each weight in the neural network to its optimum value. For each weight, a copy of the partial derivative of the error in the current step, the gradient in the previous step, and the weight change from the previous step are maintained. Using these three parameters (a, b, c), he constructs a parabola as $y^2 = ax^2 + bx + c$ (see Figure 5.4). Then the new weight in the current step is taken as the minimum point on the parabola. With this algorithm, the new update value is less than the previous value if it is positive and it is greater than the previous value if it is negative (i.e., the weights always move towards their optimum value). Quickprop often works significantly faster than the backpropagation. But the approach is not guaranteed to converge. Moreover, he assumes a quadratic function (parabola) with

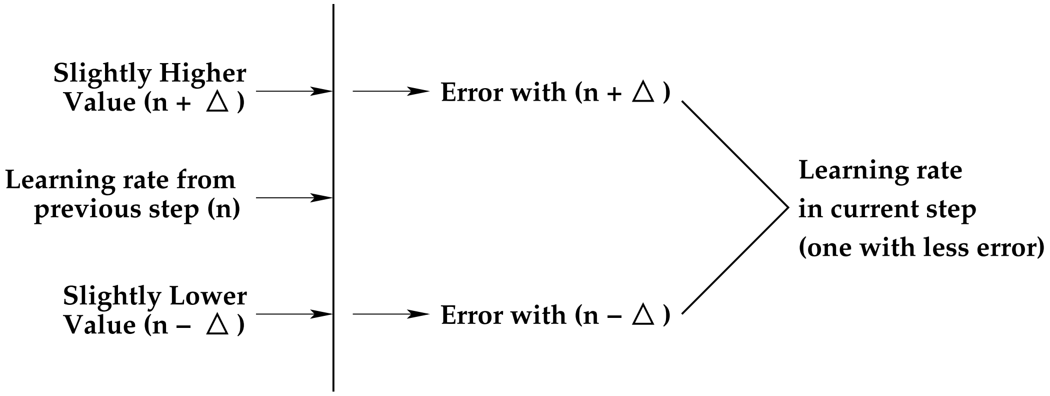

Slightly Higher Value (n + △)
Learning rate from previous step (n)
Slightly Lower Value (n – △)
Error with (n + △)
Error with (n – △)
Learning rate in current step (one with less error)

Figure 18: Dynamic adaptation of learning rate in Salomon and van Hemmen's method. The error is computed for slightly higher and lower values of learning rate in previous step and the one with less error is set for the learning rate in the current step.

three parameters (the two slopes and the weight update), but we can guarantee one only when a local minimum is nearby. The main difference between this approach and our algorithm is that our algorithm does not change the way the weight update amount is computed.

Researchers have also looked at methods for adapting parameters such as the learning rate and the momentum.

Salomon and van Hemmen [1996] looked at a method for adapting the learning rate (η) dynamically. First, a copy of the learning rate from the previous step (η_{t-1}) is maintained and the error is computed for a value slightly higher than η_{t-1} and for a value slightly less then η_{t-1}. Then the learning rate in the current step (η_t) is set with the one that gives the lowest error. This algorithm is demonstrated in Figure 5.4. A convergence proof is given for this algorithm. This method can also be applied to the momentum (α) term as well as the learning rate.

Salomon and van Hemmen [1996] have proposed another method to adapt both the learning rate and the momentum. This method is similar to the earlier approach except that it adapts both the parameters alternatively. These two methods, called η self-adaptation and η and α self-adaptation perform well in terms of the number of epochs taken to converge on classical encoder problem and other standard datasets. The drawback with this approach is that it does not solve the problem of getting stuck in a local minima. In one way, our NNGrow(Subset) algorithm is similar to this method as it uses lower learning rates to the older hidden units and high learning rates to the hidden units that are being added. But the key aspect in our algorithm is to capture the features in the subset of data in the weights of newly added hidden units and we do not focus on adapting the learning rate to a global value that works best for the whole dataset.

Osowski et. al., [1993] looked at a method that uses optimization techniques at each step to try to adapt the learning rate. The implementation of this algorithm involves two steps. First the direction of search (i.e., whether to increase or decrease the learning rate) is determined and then the directional minimization is applied, (i.e., the amount that will be added or subtracted to the learning rate is determined). This algorithm makes use of the variable metrics method [Davidon, 1959] and the conjugate gradient direction of search with a suitable line search. This algorithm works relatively fast with large datasets and large networks. It often converges in fewer epochs as the optimal learning rate is reached very early. One of the disadvantages of this approach is that it involves too many complex mathematical operations and uses sophisticated optimization techniques which can take significant processor time.

Schmidhuber [1989] proposed to search for solutions in the error surface instead of searching for the minimum value of the gradient of the error function. The interesting aspect of this approach is that the learning rate is computed dynamically after seeing every training example instead of modifying it after one entire epoch. The computation is based upon the approximation of the error function as a tangential hyper plane. Effectively the weights are also changed after each pattern presentation. If there is a zero-point in the error function, then that will be the global minima. The search for zero points never ends up in a local minima and this is the strategy used here.

If the error function does not have zero-points, then it is redefined so as to have zero-points by adding a very small negative constant to the error. The number of epochs have been significantly reduced with this approach, but the existence of zeros for error surface even with the redefinition of the error function is not guaranteed.

Our approach differs from these methods mainly in that we are employing a simpler learning method, which we will believe will be more robust over a wider range of problems. Furthermore, these techniques are in some sense complimentary to our approach, since we could employ them to possibly reduce the amount of training required while in memory.

5.5 Other Methods for Speeding up Neural Learning

Researchers suggest that one of the reasons for the slowness of backpropagation is due to its cost function, the Sum of Squared Error (SSE). Some authors have tried to redefine this SSE function in order to overcome the problems with it.

Some of the problems with SSE function are caused by a phenomenon called flat-spots [Fahlman, 1988]. Flat-spots are the regions where the derivative the sigmoid function approaches zero. In these regions the weight updates become negligible, in spite of the presence of error. In backpropagation, the weight update is computed as the derivative of the sigmoid function multiplied by the error seen by that unit. And the derivative is given by $o_i(1 - o_i)$, where o_i is the output of the unit j. This derivative (called as sigmoid prime function) approaches zero as the output of the unit approaches 0 or 1 and therefore the weight update becomes negligible.

Flat-spots can occur even when the error is maximum, making the weight changes negligible. This can slow down learning, as the units take long time to get out of the flat-spot. To handle these flat-spots, Fahlman [1988] proposed to add an offset of 0.1 to the derivative of the sigmoid function. This modification eliminates flat-spots because the the sigmoid prime function never goes to zero. Balakrishnan and Hanovar [1992] proposed another method to eliminate flat-spots by redefining the error function as the mean sum squared error over the inputs of the output units instead of the standard error function which takes the mean sum of the squared error over the outputs of output units. This modification can speed up backpropagation considerably. Our work can be differentiated from these approaches in that our algorithm does not increase the convergence speed by changing the SSE function, but instead achieves better classification by using the available memory efficiently.

Research has also examined the issue of building a neural network in parallel [Zhang et al., 1989]. This approach is very effective if a large parallel machine is available that can fit all of the data, but it is difficult to adapt if the data does not fit (and of course, requires that the user have access to a parallel machine).

6 Future Work

A number of areas for possible future work are suggested by our results. We plan to apply our methods to larger datasets, both to further test our system and to determine whether we could change other parameters (for example, on much larger datasets we might be able to greatly reduce the number of times data is presented while in memory). We also plan to investigate some of the other training methods mentioned in Section 5 that can speed learning time to see if the combination of our approach and these alternate methods can further reduce training time.

Another issue we intend to consider is how to determine when to stop learning. It may be the case that an acceptable model can be obtained after only considering a fraction of the data. One approach we could pursue would be to use the next subset of the data as a *validation set* for the network. If the error estimate we get by looking at the current subset does not seem to be changing much, we might conclude that we can stop training now. For best results, we would likely want to consider the trajectory of the error over several subsets to determine if the error indeed appears to have reached a minimum, but in any case, this approach might make it possible to terminate learning without having to read the entire dataset.

We also intend to explore an approach similar to that of Breiman [1999] as mentioned earlier. In Breiman's approach, the subset of the data is maintained and new data points are added to the subset. To make room for the new data points, other points are thrown out based on whether the classifier already does well on that point (in which case further training is not needed). This leads to a subset approach that can work like Boosting [Freund and Schapire, 1996]. We intend to look at a similar mechanism for our training method, keeping overlapping subsets of the data, where the previous points with high error are kept in our subset.

We also intend to look at using the large amount of data to select appropriate values for other learning parameters. For example, we may use subsets of the data as if they were validation sets to try various numbers of hidden units in order to find an appropriate network topology. Similarly we may apply this approach to set other parameters (learning rate, momentum, regularization parameters, etc.). This would reduce the amount of human intervention needed to learn an effective network.

7 Conclusions

Neural networks are a very effective method for classification, but are not often used in KDD applications because they often require many presentations of the dataset before converging on a model. In this thesis, we examined two novel methods for constructing a neural network that allows the neural network to be built in one pass through the data (i.e., where each data point is read into memory only once). Our algorithms read in a subset of the data at a time and then train on that subset of the data for a number of epochs. This process repeats until all of the data has been read in once.

We look at two variations of our basic method. In the simplest variation, NN(Subset), we simply initialize our neural network with some appropriate number of hidden units and then train the network for each succeeding subset of the data. One potential drawback of this approach is that the information learned for previous subsets may be lost when training on succeeding subsets of the data. We address this problem in our second approach, NNGrow(Subset), which attempts to have each subset become associated with a set of hidden units in the network. This is done by introducing a new set of hidden units for each subset of the data and decreasing the learning rate for the previous weights in the network, thus causing the learner to focus on using the new hidden units for the new subset of the data.

Experiments with our methods show that NN(Subset) produces results that are comparable to training on the entire dataset in only one pass through the data. For some of the problems, this method does perform as well as neural networks learning method trained on all of the data. This may be because the the distribution of the data is skewed. But in many cases, the accuracy achieved with NN(Subset) method is comparable to the one obtained with NN(Baseline) method. The NNGrow(Subset) method performs well in many problems. In many problems the results obtained with NNGrow(Subset) algorithm are comparable with the results obtained with NN(Baseline) algorithm. In general NNGrow(Subset) method performs better than NN(Subset) method and produces results that are very close to the NN(Baseline) algorithm. Overall, our results suggest that a neural network can be built for a very large dataset in a single pass through the data, making neural networks a feasible alternative for classification in data mining tasks.

References

[Balakrishnan and Honavar, 1992] Balakrishnan, K. and Honavar, V. (1992). Faster learning in multi-layer networks by handling output layer flat-spots. In *In Proceedings of the International Joint Conference on Neural Networks (IJCNN'92)*, Beijing, China.

[Blake and Merz, 1998] Blake, C. L. and Merz, C. J. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[Bommaganti, 2001] Bommaganti, H. (2001). Feature boosting: A novel feature subset selection method. Master's thesis, University of Minnesota Duluth, Duluth, MN.

[Breiman, 1999] Breiman, L. (1999). Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36:85–103.

[Catlett, 1991a] Catlett, J. (1991a). Megainduction: A test flight. In *Proceedings of the Eighth International Machine Learning Workshop*, pages 596–599, Evanston, IL.

[Catlett, 1991b] Catlett, J. (1991b). *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, Department of Computer Science, University of Sydney, Australia.

[Craven, 1996] Craven, M. (1996). *Extracting Comprehensible Models from Trained Neural Networks.* PhD thesis, University of Wisconsin, Madison, WI.

[Dasarathy, 1991] Dasarathy, B. V. (1991). Nearest Neighbor (NN). In Dasarathy, B. V., editor, *Norms: NN Pattern Classification Techniques*, Los Alamitos, CA. IEEE Society Press.

[Davidon, 1959] Davidon, W. (1959). Variable metric methods for minimization. Technical Report ANL-5990, Argonne National Laboratory, Argonne, Illinois.

[Fahlman, 1988] Fahlman, S. (1988). An Empirical Study of Learning Speed in Back-propagation Networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie-Mellon University.

[Fahlman and Lebiere, 1990] Fahlman, S. and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D., editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Los Altos, CA. Morgan Kaufmann.

[Freund and Schapire, 1996] Freund, Y. and Schapire, R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156, Bari, Italy.

[Kohavi and John, 1997] Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324.

[Maclin, 2002] Maclin, R. (2002). One-pass learning for large datasets using ensembles trained on subsets of the data. Technical Report 02-01, University of Minnesota Duluth.

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, Boston, MA.

[Opitz and Maclin, 1999] Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198.

[Osowski et al., 1996] Osowski, S., Bojarczak, P., and Stodolski, M. (1996). Fast second order learning algorithm for feedforward multilayer neural networks and its applications. *Neural Networks*, 9:1583–1596.

[Quinlan, 1983] Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, San Mateo, CA. Morgan Kaufmann.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

[Riedmiller and Braun, 1993] Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

[Rumelhart et al., 1986] Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. and McClelland, J., editors, *Parallel Distributed Processing (Vol. 1)*, pages 318–363, Cambridge, MA. MIT Press.

[Salomon and van Hemmen, 1996] Salomon, R. and van Hemmen, J. (1996). Accelerating backpropagation through dynamic self-adaptation. *Neural Networks*, 9:589–601.

[Schmidhuber, 1989] Schmidhuber, J. (1989). Accelerated learning in backpropagation nets. In *Connectionism in Perspective*, pages 439–445, Amsterdam. Elsevier.

[Shavlik et al., 1991] Shavlik, J., Mooney, R., and Towell, G. (1991). Symbolic and neural net learning algorithms: An empirical comparison. *Machine Learning*, 6:111–143.

[Stager and Agarwal, 1997] Stager, F. and Agarwal, M. (1997). Three methods to speed up the training of feedforward and feedback perceptrons. *Neural Networks*, 10:1435–1443.

[Thrun, 1995] Thrun, S. (1995). Extracting rules from artificial neural networks with distributed representations. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 505–512. MIT Press, Cambridge, MA.

[Vapnik, 1995] Vapnik, V. (1995). *The Nature of Statistical Learning Theory.* Springer-Verlag, New York.

[Zhang et al., 1989] Zhang, X., Mckenna, M., Mesirov, J., and Waltz, D. (1989). An efficient implementation of the back-propagation algorithm on the connection machine CM-2. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (NIPS-89)*, pages 801–809, San Mateo, CA. Morgan Kaufmann.