A framework for parallel data mining using neural networks

R. Owen Rogers
rogers@qucis.queensu.ca

November 1997
External Technical Report
ISSN-0836-0227-
97-413

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared December 5, 1997

Abstract

Data mining applications define a class of data analysis problems which require powerful computational tools with reasonable execution times. Parallel neural networks present a logical approach to solving these problems. The two most common data mining tasks, classification and clustering, can be handled respectively by an appropriate selection of supervised and unsupervised neural network techniques. The neural network techniques that will be presented in this report are cross-entropy multi-layer perceptrons and Gaussian mixture model networks. Bulk Synchronous Parallelism (BSP) is chosen as an effective model for parallelization. Several different strategies for parallelizing neural networks are considered using BSP cost analysis. From these analyses, it is shown that concurrently training a

neural network on subsets of a data set, a strategy called exemplar parallelism, yields minimal parallel execution costs. One constraint of these analyses, however, is that the BSP cost model can only demonstrate the optimality of exemplar parallelism on an epoch-by-epoch basis. Thus, batch learning is used to show that exemplar parallelism can attain an optimal convergence speed. A theoretical justification of batch learning is presented from the perspective of deterministic gradient estimation. These theoretical results are verified empirically by examining the parallel execution costs of supervised and unsupervised neural networks on three data mining databases.

Keywords: Data mining, neural networks, parallelism, bulk synchronous parallelism, BSP, cost analysis, supervised learning, unsupervised learning, batch learning, deterministic learning, stochastic learning

1 Introduction

The rapid development of data storage technology in combination with the computerization of business, science and industrial sectors has made it possible and inexpensive for organizations to amass large amounts of information. Researchers have estimated that, with the automation of data collection, the world's data supply is in the process of doubling every 20 months [19]. Given the size and breadth of these data repositories, there is likely to be potentially valuable information buried beneath the raw data. For example, a database of client information may be able to provide insight into future customers; a medical database of previous patients may be able to help diagnose new patients; and a database of market price histories may be used to predict future price levels. With all of this data collated and accessible, there is a growing interest in methods that can be used to 'mine' these databases to extract their profitable information.

Traditional manual statistical techniques and simple database queries are often inadequate for this discovery process. The size of the databases makes exhaustive search prohibitively expensive. In addition, databases are likely to contain corrupt, missing and noisy data causing exact matching to be difficult and unreliable. Thus, new intelligent and automated techniques are required to efficiently, robustly, and thoroughly analyze these databases within a reasonable amount of time. The emerging field of *data mining* is concerned with providing a framework for the process of applying these techniques to large databases [17].

The techniques that are used in the data mining process are generally drawn from diverse areas of research [26][21]. Thus, it is necessary to present a principled and theoretically-sound approach to dealing with data mining applications. In this report, a statistical perspective will be used to characterize the process of data analysis [30]. From this perspective, data mining techniques can be interpreted as models for discovering interesting, informative, and potentially profitable patterns in a database.

A technique that is commonly applied to solving data mining problems is the artificial neural network (ANN). Originally inspired by biological models of mammalian brains, artificial neural networks have emerged as a powerful technique for data analysis [4][22]. Neural networks consist of compositions of simple, nonlinear processing units that are organized in a densely-interconnected graph. A set of parameters, called weights, are assigned to each of the edges of the graph. These parameters are adapted through the local interactions of processing units in the network. By repeatedly adjusting these parameters, the neural network is able to construct a representation of a given data set. This adaptation process is known as training.

A neural network is able to solve highly complex problems due to the nonlinear processing capabilities of its neurons. In addition, the inherent modularity of the neural network structure makes it adaptable to a wide range of applications. In this report, neural networks will be treated as statistical models for data mining. From this perspective, the goal of the neural network is to adjust its parameters to accurately model the distribution of a provided data set.

One of the main limitations of applying neural networks to analyze massive data mining databases is the excessive processing that is required. It is not uncommon for a data mining neural network to take weeks or months to complete its task. This time constraint is infeasible for most real-world applications. However, processing time can be substantially reduced by distributing the load of computation among multiple processors. Thus, parallelism presents a logical approach to managing the computation costs of data mining applications.

This report will consider a particular model for parallel programming known as Bulk Synchronous Parallelism (BSP). BSP is a general-purpose model of parallel computing that allows the development of robust, efficient and portable software. Most importantly, BSP has a cost model that produces accurate estimates of parallel execution times for a wide variety of parallel machines. BSP specifies a structure for developing parallel software which is designed specifi-

cally to handle communication-intensive applications. Given these properties, the BSP model is well-suited to the development of parallel data mining software.

This report considers two approaches for reducing the computation time required to train neural networks on data mining data sets.

- The first approach examines different strategies for parallelizing neural networks. The cost of each of these strategies is compared using the BSP cost model. From this cost analysis, it is asserted that the best parallel speedup can be attained by distributing the data set among the parallel processors and concurrently training a neural network on each subset of the data. This strategy is known as exemplar parallelism.

- The second approach for accelerating neural network training is called batch learning. This technique examines the tradeoff between the frequency and accuracy of neural network weight adaptations and attempts to find the best compromise which minimizes the cost of training.

The organization of this report is as follows. Section 2 presents an overview of the major areas of research which will be dealt with in this document – namely, data mining, artificial neural networks, and general-purpose parallelism. In Section 3, several different methods of partitioning a neural network for parallelization will be analyzed. The BSP cost model will be used to demonstrate the optimality of exemplar parallelism. Section 4 considers batch learning as a method for further accelerating neural network training. The potential speedup of batch learning is demonstrated by conducting several experiments on three data mining databases. Section 5 summarizes the implications of applying batch learning and exemplar parallelism to data mining neural networks.

2 Data Mining, Artificial Neural Networks and Parallelism

2.1 Introduction

The objective of this section is to provide a survey of recent work in the fields of data mining, artificial neural networks, and general-purpose parallelism. The specific techniques dealt with in this paper will be highlighted to show their place within the wider research context.

This section begins with an explication of the field of data mining. The data mining process will be defined from the perspective of statistical parameter estimation. This statistical perspective creates a framework for describing the two most common data mining tasks: classification and clustering. The general similarities and differences between the data mining algorithms used to solve these tasks will be discussed. The material that is discussed in this subsection is a synthesis of the general description of the data mining process presented in [16], and the statistical approach to pattern recognition presented in [30].

Next, artificial neural networks will be described with respect to the data mining tasks of classification and clustering. Two specific neural network algorithms, cross-entropy multi-layer perceptrons and Gaussian mixture models, will be given as effective techniques for solving these data mining problems. Both of these algorithms will be derived and their general properties and capabilities will be summarized. In addition, the distinction between deterministic and stochastic methods will be expressed. The presentation of the cross-entropy multi-layer perceptron represents a synthesis of the work in [22, pp. 138–192], and in [4, pp. 230–240]. The derivations of the Gaussian mixture model follow the material in [42] and [4, pp. 59–73].

The last subsection of this section will assert the need for parallelism in data mining applications. The difficulties of parallel software development will be listed and a set of criteria will be established to facilitate the choice of an appropriate parallel programming model. Bulk Synchronous Parallelism will be introduced as a model which is suitable for designing and implementing parallel neural networks that are applicable to data mining tasks.

2.2 Data Mining

Before the techniques used in data mining can be analyzed in any detail, a formal definition of the data mining process should be stated. The accepted definition of data mining, formulated by Fayyad, Piaetsky-Shapiro and Smyth [16][15], is:

> *Data mining* is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data.

Restated more formally, for a database $\mathcal{D}$, the goal of the data mining process is to identify the patterns $\mathcal{P}$ which satisfy the discovery criteria $\mathcal{I}$.

A discovered pattern is an expression which provides some information about a subset of the database. The information expressed by a discovered

pattern permits a generalized description of the data which is simpler than the data itself. For example, for some supermarket database of customers, a pattern $P_1(D_1)$ may describe the fact that all customers in subset D_1 purchased tofu and lettuce on their last shopping trip. Obviously, this example of a discovered pattern is very simple; however, it still enables one to reason about the subset of customers D_1 by virtue of the fact that they all share the common feature P_1. More complicated patterns may make probabilistic statements about subsets of the database, or may be characterized by arbitrarily complex expressions. Commonly, patterns are expressed in the form of rules, associations, clusters, functions or graphs. Patterns can normally be transformed from one form to another; however, some loss of information may result.

It is not sufficient to simply search a database for any arbitrary pattern; some criteria ($\mathcal{I}$) must be specified to assess the pattern's desirability. The definition of the data mining process provides several conditions to judge if an acceptable pattern has been discovered. These conditions will be defined within the context of data mining:

- *Valid*: the discovered pattern $P(D)$ must be descriptive of least some proportion of the subset D. The level of this validity threshold is contingent on the data mining process; some applications may only accept absolute membership, whereas other applications may be able to handle probabilistic measures. Thus, $V(P, D) \geq v$ for some validity threshold v.

- *Novel*: with many large databases, there is a significant amount of domain knowledge that has already been accumulated; for a discovered pattern to be interesting, it should differ from existing knowledge. Novelty, can thereby be characterized by the metric $N(P, \mathcal{K})$ which compares the new pattern P with the set of existing patterns $\mathcal{K}$.

- *Useful*: to be valuable, the knowledge gained through the discovery of the pattern P should enable the data analyst to perform some set of actions A which were not previously available. Thus, the discovery of the pattern should entail some benefit $G(P, A)$ for the data analyst.

- *Understandable*: because data mining is a human-guided process, it is often important that the data mining operator is capable of understanding the meaning of the discovered pattern. The understandability of the pattern can be measured by the function $U(P)$. The measure is of

course quite subjective and is contingent on the interpretation of the data analyst.

The desirability of the pattern can be characterized by the combination of these conditions $I(P, D, N, V, G, U)$ exceeding some threshold. Generally, only validity and novelty can be measured by the data mining algorithm. The latter two conditions are often subject to evaluation by the data analyst. In addition, it is likely that different data mining processes will weigh the importance of these conditions differently.

2.2.1 Data Mining Models and Statistics

In the previous subsection, data mining was defined as the search for desirable patterns in large databases; however, it still remains to be shown *how* these patterns will be discovered. This subsection will outline a framework for analyzing the way in which data mining algorithms locate patterns.

Before proceeding, it is important to consider the distinction between a pattern and a model. A pattern is an instantiation of a model. Inversely, a model serves as a template for the generation of all possible patterns that can be discovered by a data mining algorithm. Thereby, the structure of the model determines the type and quality of the patterns that can be extracted from a 0database. If one considers the example pattern of tofu and lettuce consumers (P_1) given above, a model $\mathcal{M}$ could, for instance, represent any possible combination of items that could be purchased from the supermarket by all of the customers in D_1. The goal of a data mining algorithm would be to search through the model space $\mathcal{M}$ to find P_1 or any other such pattern which satisfies the search criteria $\mathcal{I}$.

However, it is generally infeasible and unnecessary to search the entire space of possible patterns. Due to the *curse of dimensionality*, the number of instances of a model grows exponentially with the number of items considered [4, pp. 7–8]. For example, assume a supermarket sold i items, and the data mining process was to consider every possible combination of d items that were purchased together on the same shopping receipt. To search this space exhaustively, the model space must have size i^d [4, p. 7]. For a reasonable number of items, searching a model space of this size is clearly prohibitive. However, data mining algorithms incorporate techniques to intelligently restrict the search to a subset of the possible model instances. In addition, domain knowledge is often used to limit the search space to a set of relevant patterns. Thus, the model space $\mathcal{M}$ is generally subset of range of possible

patterns $\mathcal{P}$.

A particular instance of a model can be characterized by a set of parameters $\mathbf{w}$ which, for this example, represent a list of items that may have been purchased by all of the customers in D_1 during their last shopping visit. If one considers the model instance M_1 which describes P_1, M_1 can be represented by the parameters $\mathbf{w}$={tofu, lettuce}. Due to the fact that P_1 holds for all of the shoppers in D_1, the model instance M_1 can be assumed to meet the search criteria $\mathcal{I}$ and provide one of the possible solutions for this data mining application. To summarize, data mining represents the search for the model parameters $\mathbf{w}$ which describe a set of patterns P which satisfy the search criteria $\mathcal{I}$.

It is useful to re-examine this search process from the perspective of statistical pattern recognition and parametric statistics. Each model instance in $\mathcal{M}$ can be represented in terms of the probability that it describes a pattern which satisfies the search criterion. The model search space can be presented as a conditional probability density function $p(\mathbf{w}|\mathcal{D})$ given the database $\mathcal{D}$. This conditional density is referred to as the *posterior distribution*. The posterior distribution measures the quality of a model instance's representation of the data set. M_1, which describes the solution P_1, would have a peak probability in this density function. Thus, the posterior distribution can be used to select a model instance with parameters which accurately identify patterns in the database. The process of selecting the set of model parameters which maximize the posterior distribution is known as *maximum a posteriori (MAP) estimation*.

In order to calculate the posterior probabilities for a particular model instance, the likelihood that the data was generated by the instance must be first determined. For the supermarket example, this would entail examining the items purchased by each customer in D_1 and determining whether they bought the items defined by the model parameters $\mathbf{w}$ (in this case, tofu and lettuce). By calculating the percentage of customers who purchased these items, the likelihood of the model instance can be evaluated. This likelihood measure can be represented by the distribution $p(\mathcal{D}|\mathbf{w})$. The method of evaluating each data example using the model parameters is known as the *likelihood function*. For this example, the likelihood function was a simple comparison of the model parameters with each example in D_1; however, for most data mining algorithms the likelihood function is quite complex. The likelihood function is generally a probabilistic interpretation of the output of a particular data mining algorithm evaluated on the data set. Due to the application of

this statistical framework to data mining algorithms, the outputs of different algorithms can be compared

Bayes' rule can be used to transform the model likelihood into the posterior probabilities. Thus, the MAP model parameters can be selected based on the outputs of the likelihood function applied to the data set.

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})} \tag{1}$$

From this equation, the posterior distribution is expressed as a product of the model likelihood $p(\mathcal{D}|\mathbf{w})$, the prior distribution of the model parameters $p(\mathbf{w})$, and the data distribution $p(\mathcal{D})$. The data distribution is independent of the model parameters and is thus a constant factor in this equation. This distribution serves to normalize the probabilities in the numerator. The prior distribution represents the *a priori* assumptions about the model parameter values in the absence of the data. Because it can be assumed that relatively little is known about the distribution of the data, this density is very broad. Thus, the initial model parameters are drawn from a distribution which will enable the model to be as flexible as possible in response to the data mining application.

In order to determine the model instance with parameters that maximize the posterior distribution, the terms in the numerator of Bayes' rule must be maximized (the denominator can be ignored because it is a constant factor). Equivalently, the negative logarithm of the numerator can be minimized to yield the *cost function* $J(\mathbf{w})$:

$$J(\mathbf{w}) = -\ln p(\mathcal{D}|\mathbf{w}) - \ln p(\mathbf{w}) \tag{2}$$

The first term in this equation is the *log-likelihood*. Most common error functions (such as the sum squared error term, cross-entropy error or Euclidean distance) can be expressed in terms of log-likelihood minimization. Thus, this statistical framework is applicable to a wide range of data mining applications. In the next subsection, the log-likelihood estimate is applied to two of the most common data mining tasks.

The second term in the cost function is the log-prior distribution of the model parameters. This distribution is often used as a measure of model complexity [30]. From the cost function (2), the complexity of the model must be balanced against the likelihood performance improvement gained from the additional complexity. However, it is often assumed that this prior distribution is essentially uniform across the range of possible parameter values. Thus,

the complexity of a model can be ignored, focusing simply on the estimated likelihood of the model. In this context, the goal of the data mining process is to select a model with parameters $\mathbf{w_{ML}}$ which maximize $\ln p(\mathcal{D}|\mathbf{w})$. This process is known as *maximum likelihood estimation*.

Although the log-prior distribution may be removed from the cost function, it is still important to consider the complexity of a model. Intuitively, when choosing a data mining model, it is important that the model is sufficiently powerful to locate patterns which satisfy the search criteria. However, it is also important that the model is not overly complex. Models with many adjustable parameters may be slow to optimize, may overfit the data (leading to poor generalization), and may be more difficult to interpret. This tradeoff between model performance and model complexity is captured by *Occam's razor*; this principle states that the preferred model is the simplest one which is still capable of finding an adequate solution [4, pp. 14–15]. Thus, the data mining process represents not only a search to discover patterns in the database, but also a search to find an appropriate model to represent those patterns. The Bayesian cost function (2) permits both the model complexity and the model parameters to be optimized simultaneously.

The weighting of the tradeoff between performance and complexity is partially contingent on the data mining application. There are two main purposes in data mining: *description* and *prediction*. Descriptive models attempt to produce an understandable representation of the results. Understandability often implies simplicity, and thus, may only be obtained at the expense of the accuracy of the results. Therefore, a simple model with few parameters and good generalization properties may be desirable for a descriptive task. Predictive models, on the other hand, strive to achieve optimal predictive accuracy even though they may be generated by highly complex models. Therefore, when minimizing the cost function (2), descriptive models are likely to penalize complexity more severely than predictive models.

2.2.2 Common Data Mining Tasks

The statistical field of parameter estimation presents a solid framework for characterizing the data mining process. In this section, the log-likelihood cost function (2) will be examined as an approach to solving two of the most common data mining tasks: classification and clustering. Examples of applications for each of these tasks will be given. For a more thorough review of how these tasks relate to specific data mining applications, see [13].

Classification

The goal of classification is to accurately map database examples into one of several predefined classes. For a classification application, a data mining algorithm is provided with a set of database examples $\mathbf{x}$ and corresponding class labels $\mathbf{t}$. The class labels identify each example as belonging to one of C classes $\{\mathcal{C}_1, \ldots, \mathcal{C}_C\}$. By examining the common features of the examples for each class, the data mining algorithm constructs a representation or *class description* for each class. These class descriptions are encoded in the parameters $\mathbf{w}$ of the data mining model.

The class descriptions are formed by presenting the examples and their corresponding class labels (repeatedly) to the model. Each example in the data set is compared with the class descriptions encoded in the model parameters. The example is then mapped to the class by which it is best described. If the assigned classification matches the class label for the example, then the model has been classified correctly. However, if the model has incorrectly labeled the example, then an error results. An error function is used to adjust the model parameters $\mathbf{w}$ so that the model will be better able to correctly classify the input example in the future. The goal of the classification process is, therefore, to optimize the set of model parameters such that the number of examples which are mislabeled by the model is minimized.

This minimization process can be formulated in terms of a likelihood cost function. For a data set of N examples, where $\mathcal{D} = \{\mathbf{x}_n, \mathbf{t}_n\}_{n=1}^{N}$, it can be assumed that the class labels have been assigned to each input example according to some unknown conditional probability density function $p(\mathbf{t}|\mathbf{x})$. This density function represents the probability $P(t_n = k|\mathbf{x}_n)$ that a given input example $\mathbf{x}_n$ belongs to class C_k (where $k = 1 \ldots C$). If the label t_n identifies $\mathbf{x}_n$ as a member of class C_k ($t_n = k$), then the probability P_n will be maximized. Inversely, for the classes C_j which do not match the class label of the input example ($t_n \neq j$), the probability P_n is likely to be very small. Because the conditional data distribution $p(\mathbf{t}|\mathbf{x})$ correctly matches the class labels to each example, the probability P_n will be maximized for every example in the data set.

By modeling the data distribution $p(\mathbf{t}|\mathbf{x})$, a data mining process can maximize the probability of correctly identifying the class labels for each of the input examples. The parameters $\mathbf{w}$ can be used to determine the probability that a model is capable of generating the data distribution, according to the posterior density $p(\mathbf{w}, \mathbf{t}|\mathbf{x})$. By adjusting the model parameters to maximize the posterior distribution, the model can be adapted to fit the data distribution as closely as possible. The fit of a model can be evaluated by determining

the probability of correct classification for each example in the data set. This fit is calculated by the likelihood of the model. If it is assumed that the elements of the data set $\mathcal{D}$ are conditionally independent, then the likelihood can be expressed as the product of the probability estimates for each example:

$$p(\mathcal{D}|\mathbf{w}) = \prod_{n=1}^{N} P(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) \tag{3}$$

By maximizing each probability estimate $P(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w})$, the model will come closer to resembling the data distribution $p(\mathbf{t}|\mathbf{x})$. If the prior distribution $p(\mathbf{w})$ of the model parameters is ignored, the likelihood can be expressed in the form of a cost function $J(\mathbf{w})$.

$$J(\mathbf{w}) = -\ln p(\mathcal{D}|\mathbf{w}) = -\sum_{n=1}^{N} \ln p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) \tag{4}$$

Because the likelihood (3) represents the probability that the model correctly classifies the data set, its negative logarithm can be used to minimize the probability of classification error. Thus, by minimizing the cost function $J(\mathbf{w})$, the model is fitted to the data distribution. The goal of the classification process is therefore to derive a model with an optimal set of parameters $\mathbf{w_{ML}}$ which minimizes the probability of classification error.

At this stage, the method for optimizing the model parameters by using the cost function to fit the conditional data distribution has not been stipulated. Different data mining algorithms will have different approaches to solving this fitting problem. One common approach is to use some form of gradient optimization. Later in this report (Subsections 2.3.2 and 2.3.3), the process of selecting the optimal model parameters will be explored within the context of artificial neural networks. However, regardless of the specific data mining approach, it is important to consider the properties of the distribution of the data $p(\mathbf{t}|\mathbf{x})$. For classification applications, every input example is generally assigned to only one class. Therefore, each class label t_n is a discrete variable. In order to accurately model the conditional data distribution, the model should use a discrete probability density, such as the Bernoulli or multinomial distribution to calculate the likelihood. The consideration of the distribution of data variables is an important step that is often neglected in data mining applications.

Once a classification algorithm has completed the process of optimizing its parameters to fit the distribution of the data, the model should be put

into use for the data mining application. If the purpose of the data mining operation is descriptive, the class descriptions can be extracted from the model parameters $\mathbf{w}_{\mathbf{ML}}$. These class descriptions detail the common properties of the examples belonging to each class. They should be processed to be as succinct and understandable as possible. If the purpose of the data mining operation is predictive, then new, unlabeled data can be applied to the model for classification. If it is assumed that the new data is generated by the same general distribution which created the training data set, then the classification results should be accurate.

Classification is applicable to a wide range of data mining problems. Here are just a few of the potential applications:

insurance fraud detection: the data mining algorithm can construct a representation to distinguish a class consisting of fraudulent claims from a class of legitimate claims.

credit rating prediction: the data mining system could be used to predict the credit rating of a client by assigning each rate to a separate class.

medical diagnosis: the data mining system may be trained to identify the presence (class 1) or absence (class 2) of tumors in X-ray mammogram data.

Clustering

For some data mining applications, predefined classes may not exist and there may be little domain knowledge about the contents and structure of the database. In this case, the function of the data mining algorithm is purely exploratory. Because there is no external guidance to direct the analysis, the data mining algorithm must proceed by examining the structure of the data itself. Structure in the database is distinguished by a consistent set of features that are shared by a group of examples. To identify this structure, the data mining algorithm attempts to group examples by using some similarity or proximity metric. Common metrics include Euclidean distance and nearest-neighbour methods. The set of examples that are grouped together by the algorithm are known as a *cluster*.

From the perspective of statistical parameter estimation, it can be assumed that a data set of N examples, $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^{N}$, has been generated by some unknown, unconditional probability density function $p(\mathbf{x})$. The goal of the data mining process is to estimate this density function as closely as possible – a problem known as *density estimation*. However, it is generally too difficult to

accurately estimate $p(\mathbf{x})$ using a single parametric model. Thus, the problem of estimating $p(\mathbf{x})$ can be simplified by assuming that the data set has been generated by a finite number M of independent distributions or *components*:

$$p(\mathbf{x}) = \sum_{i=1}^{M} \pi_i p(\mathbf{x}|i) \tag{5}$$

where π_i is known as the *mixing proportion*, and $p(\mathbf{x}|i)$ is the density function for each component i. The mixing proportions specify the prior probability $P(i)$ of choosing a component density i to generate a data example.

Because the number of component distributions, M, is finite and is generally considerably smaller than the number of database examples, N, every component density function is likely to represent a number of database examples. An input example $\mathbf{x}_n$ can be evaluated to assess the probability that it was generated by each component density. The component density i with the greatest likelihood can be used to represent the input example. Every example represented by component i is similar in virtue of the fact that it is likely to have been generated by $p(\mathbf{x}|i)$. Thus, each component distribution can be interpreted as describing a cluster in the data set $\mathcal{D}$.

The goal of the clustering process is to construct a model with parameters $\mathbf{w}$ which accurately describe the density function $p(\mathbf{x})$. By decomposing this unconditional distribution into several component densities (5), a set of model parameters $\mathbf{w}_i$ can be used to estimate each component density i. Therefore, the focus of the clustering process is to determine the parameters for each component i that maximize the posterior probability $p(\mathbf{w}_i, i|\mathbf{x})$. The fit of a model to the data distribution can be judged by the calculating the cumulative probability of each component generating each point $\mathbf{x}_n$. By combining each of the component probabilities for every example in the data set, the likelihood of a model $p(\mathbf{x}|\mathbf{w})$ can be determined. If it is assumed that the elements of the data set $\mathcal{D}$ are conditionally independent, then the likelihood of a model can be expressed as:

$$p(\mathcal{D}|\mathbf{w}) = \prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{w}) = \prod_{n=1}^{N} \sum_{i=1}^{M} \pi_i p(\mathbf{x}_n|i, \mathbf{w}_i) \tag{6}$$

Once again, it is straightforward to transform this likelihood equation to the cost function of 2. The maximization of the likelihood equation creates a model which produces the greatest probability of generating each data example. Inversely, this minimizes the probability that an example is poorly fit by

a component distribution. The log-likelihood cost function can be expressed as:

$$J(\mathbf{w}) = -\ln p(\mathcal{D}|\mathbf{w}) = -\sum_{n=1}^{N} \ln \sum_{i=1}^{M} \pi_i p(\mathbf{x}_n|i, \mathbf{w}_i) \tag{7}$$

Clustering problems can be considered as a generalization of classification problems. The complete data set for a classification application (both examples and class labels) can be modeled by the joint probability density function $p(\mathbf{x}, \mathbf{t})$. This joint distribution can be decomposed to yield:

$$p(\mathbf{x}, \mathbf{t}) = p(\mathbf{t}|\mathbf{x})p(\mathbf{x}) \tag{8}$$

The first term of this equation is the conditional data distribution $p(\mathbf{t}|\mathbf{x})$ which is commonly modeled in classification applications. The second term corresponds with the unconditional distribution of the data $p(\mathbf{x})$ which is modeled in clustering applications. Thus, the clustering process is often performed as a precursor to further data mining analysis. Commonly, clustering is used to preprocess the input for a classification application. Because component densities provide an accurate description of a cluster of data examples, they can be used to:

- summarize the data set with the set of component distribution parameters $\mathbf{w}$,
- detect and eliminate noise from the data which occur as outliers to a component distribution, and
- extract the common features of each cluster to provide more meaningful, dimension-reduced data to a classifier.

Alternately, component densities may be used to (automatically) create classes for data sets where no predefined class labels exist [9]. By assuming that each component represents a separate class, the component density function can be used as a class description for each class, and class labels can be created for every data example. The following are two real-world examples of data mining clustering applications:

image compression : a clustering algorithm can be used to compress a database with minimal loss of detail by using cluster prototypes to represent blocks of similar pixels.

signal filtering : a telecommunications database generated by a finite number of noisy components can be filtered using a clustering algorithm to identify the properties of each component and to eliminate signal noise from each example.

2.2.3 Data Mining Algorithms

A wide variety of algorithms exist that have been used to solve data mining applications. Commonly used examples are: decision trees and rules, artificial neural networks, genetic algorithms, association rules, inductive logic programs and mixture modeling [16]. The algorithms for data mining are generally drawn from the fields of machine learning, databases, and statistics [38]. However, the size of the databases and the complexity of the analysis makes data mining distinct from each of these fields. It is important to emphasize that data mining is a process consisting of steps which will often represent a synthesis of techniques from each of these fields.

Despite the differences between data mining algorithms, there are a number of commonalities which unite these algorithms under the data mining framework. The most apparent common feature between data mining algorithms is inductive search – data mining algorithms inherently reason from specific data examples to generalizations about the structure of larger subsets of the database. To form these generalizations, data mining algorithms invariably develop some sort of model. These models, in turn, consist of parameters to be optimized by the inductive search. If conditional independence is assumed among the examples in a database, then this optimization process can be characterized as modeling some conditional or unconditional distribution of the database. Thus, data mining algorithms can generally be interpreted from the perspective of statistical parameter estimation.

In addition to the overarching commonalities, the differences between algorithms can also be characterized in general terms. When a data mining model constructs a generalization about some subset of the database, it, in effect, creates criteria for the membership of that subset. For example, a data example belongs to a class if it is described by the class description, or belongs to a cluster if it is likely to have been generated by the component density. Thus, these membership criteria partition the data set into subsets. Depending on the data mining algorithm, these partitions can either be distinct or overlapping. For example, the k-means algorithm constructs independent tessellations to cluster the data; whereas mixture models generate overlapping distributions. In addition, the boundaries of these partitions may be linear or nonlinear, open or

closed. For example, a decision tree recursively splits a data set into linearly-separable regions; whereas the decision boundaries created by artificial neural networks are nonlinear in nature. These simple characteristics of the partitioning process distinguish seemingly unrelated data mining algorithms. For example, decision trees can be shown to be a variant of hierarchical mixture models [30]. Another important distinction between data mining algorithms is the way that a model represents discovered patterns: genetic algorithms encode results in a genome structure, neural networks represent patterns in their interconnection weights, and decision trees partition the data space using their generated tree structure. Although the results of each of the aforementioned algorithms may appear quite different (even when applied to the same data set), they can generally be transformed into a rule-based format, and by doing so, compared.

2.3 Artificial Neural Networks

The artificial neural network (ANN) is a technique that is commonly applied to solving data mining applications. The aim of this subsection is to provide a general background for neural networks and relate it to the statistical framework for data mining given in the previous subsection. The details of the neural network algorithms that will be used in the report experiments will be given.

2.3.1 What are Artificial Neural Networks?

Although they derive their names from, and are loosely modeled after, mammalian brains, neural networks are, in essence, graph-based parametric models. Artificial neural networks are densely-interconnected graphs of simple processing elements called *neurons*. These graphs can be arranged in an arbitrary structure; however, neural network connections are commonly organized into *layers*, such that each processing element is only connected to neurons in adjacent layers. The neural network model parameters, called *weights*, are represented by numerical values attached to each of the edges in the graph. Due to the graphical structure of the model, information passing through the network must form a path through the weighted interconnections. The result of this graph traversal produces some network output $\mathbf{y}$ which is a function of the input vectors $\mathbf{x}$ and the network weights $\mathbf{w}$:

$$\mathbf{y} = f(\mathbf{x}, \mathbf{w}) \tag{9}$$

Considering the statistical framework presented in the previous subsection, it is important to be able to interpret the network output as a probabilistic likelihood. The neural network algorithms that will be discussed in this subsection are designed to accommodate this perspective.

An artificial neural network attempts to construct an internal representation of its environment through the modification of its weight values. This process of weight adjustment is called *training*. During training, data is repeatedly presented to the network and the network attempts to alter its parameters to model features, patterns and regularities in the data. An iteration through every example $\mathbf{x}_n$ in the data set is known as an *epoch* which is a basic unit of network computation time. The quality of the network representation can be expressed by some cost function $J(\mathbf{w})$ which the network algorithm attempts to minimize. From the perspective of parametric statistics, this search for network parameters to represent patterns in the data can be expressed as modeling the underlying probability distribution of the data[1]. In this context, neural networks can be perceived as algorithms for statistical modeling. Thus, neural networks are applicable to data mining problems using the statistical framework established in the previous subsection.

The two common data mining tasks, classification and clustering, are solved by two different classes of neural network algorithms. For classification applications, the aim of the data mining process is to model the conditional distribution $p(\mathbf{t}|\mathbf{x})$ of a data set consisting of input example vectors $\mathbf{x}$ and corresponding class labels $\mathbf{t}$. Neural networks which are capable of modeling this conditional distribution are called *supervised* networks because they are guided (or supervised) by the class labels in deciding how to classify the inputs. For clustering applications, on the other hand, no predefined classes exist to guide the data analysis process. Thus, clustering algorithms attempt to model the unconditional distribution $p(\mathbf{x})$ to detect underlying structure in the data. The class of neural network algorithms that can be used to model this type of distribution are known as *unsupervised* networks.

The following subsections will present examples of the supervised and unsupervised neural networks that will be used in this report. The network function $f(\mathbf{x}, \mathbf{w})$ and the cost function $J(\mathbf{w})$ will be derived for each neural network algorithm and will be related to the tasks of classification and clustering. The supervised neural network that will be presented is the cross-entropy

[1]Once again, it is important to assume that the data is conditionally independent and time invariant in order to validate the statistical perspective. Neural networks which are applicable to time-series analysis will not be considered.

multi-layer perceptron. The presentation of this algorithm represents a synthesis of the adaptive signal-flow approach to multi-layer perceptrons in [22, pp. 138–192], and the derivations of cross-entropy error in [4, pp. 230–240]. The unsupervised neural network that will be presented is the Gaussian mixture model. The derivations for this network follow the material in [42] and [4, pp. 59–73].

2.3.2 Supervised Neural Networks

Figure 1 depicts the prototypical supervised neural network model: the multi-layer perceptron. The multi-layer perceptron (MLP) neural network commonly consists of two or three fully-interconnected layers of neurons. The input to the network is propagated from the input layer through the weighted interconnections to the output layer. Conversely, error signals are transmitted through the network in a reverse direction. From the figure, it can be observed that there is a layer of neurons which has no direct connection to the input or the output. This layer is known as the *hidden layer* and is responsible for providing the network with its nonlinearity and its ability to construct an internal representation of its input environment.

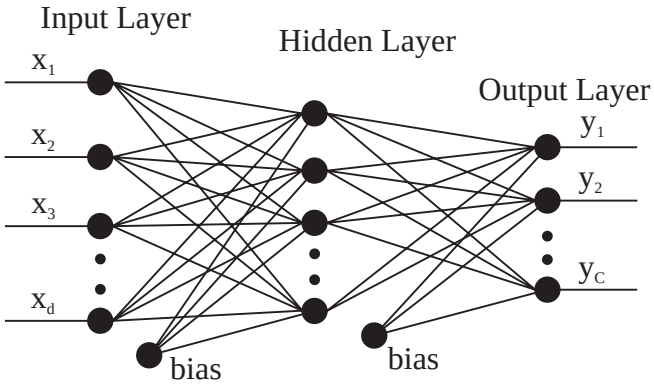

Figure 1: Architecture of a fully connected multi-layer perceptron with a single hidden layer.

The Supervised Network Function

The computations performed by the neurons in a neural network are more or less uniform across the network. Thus, the network function $f(\mathbf{x}, \mathbf{w})$ can be deconstructed in terms of the operations of individual neurons. Each neuron

i in the network generates its output y_i by performing a weighted sum on the input that it receives through its interconnections. This weighted sum is then fed through an *activation* or *basis function* $\varphi_i(v)$ which can be used to transform the range of the output and introduce nonlinearity into the network.

$$y_i(\mathbf{x}) = \varphi_i(\sum_j w_{ij}x_j + w_{i0}), \tag{10}$$

where the variable j is used to indicate the indexes of all neurons connected to y_i. The parameter w_{i0} is called a *bias* which serves to position the basis function in the weight space, and enables the basis function to adapt during training. A basis function can be any piecewise-differentiable function. The most common basis function is the *logistic sigmoid*:

$$\varphi(v) = \frac{1}{1 + e^{-v}} \tag{11}$$

The choice of the logistic sigmoid has important implications for classification

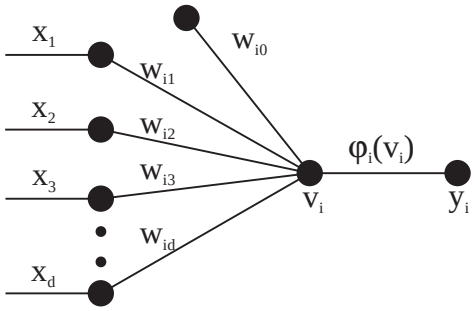

Figure 2: Signal-flow graph for the feed-forward computation performed by neuron i using (10).

applications. By using Bayes' theorem, it can be demonstrated that the logistic sigmoid can be derived from the conditional distribution $p(\mathbf{t}|\mathbf{x})$ for a generic problem with two classes [30]. For classification problems with more than two classes, a generalization of the sigmoid, called the *softmax* activation function can be derived. These derivations yield the important result that the use of logistic sigmoid activation functions allows the output of each neuron to be treated as likelihood probabilities. Thus, the network function $f(\mathbf{x}, \mathbf{w})$ can be used to model the classification data distribution $p(\mathbf{t}|\mathbf{x})$, such that:

$$\mathbf{y} \equiv p(\mathbf{t}|\mathbf{x}, \mathbf{w}) \tag{12}$$

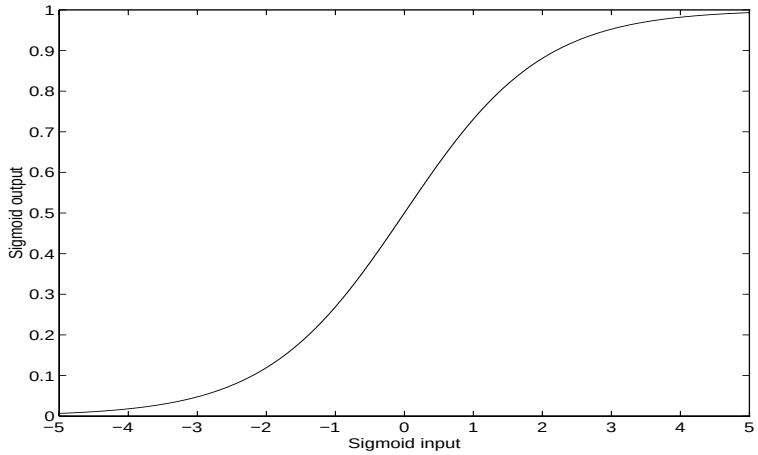

Figure 3: Plot of the logistic sigmoid activation function from (11).

In addition to its importance in statistical modeling, the sigmoidal basis function performs a nonlinear scaling of the output to the range $(0, 1)$. Without this nonlinearity introduced by the basis functions, a multiple layer network could be compressed into the weights of a single layer [22, p. 139]. Another important feature of the sigmoid is its shape. For large and small activations v, the sigmoid asymptotically approaches 0 and 1. At this stage, the sigmoid is *saturated* because changes in the activation value will constitute insignificant changes in the sigmoid output. However, if $|v|$ is small, then the sigmoid is approximately linear and the basis function leaves the activation value v relatively untouched.

The following equation demonstrates the output for a network with a single hidden layer. In this case, the output for all of the neurons j in the hidden layer are used as the input to the neurons in the output layer. Thus, input is generally propagated through the network on a layer-by-layer basis.

$$y_k(\mathbf{x}) = \varphi_k(\sum_i w_{ki}\varphi_i(\sum_j w_{ij}x_j + w_{i0}) + w_{k0}) \tag{13}$$

Classification Cost Functions For Supervised Networks

Now that the network function has been defined for the multi-layer perceptron, the choice of an appropriate error function must be considered. Due to the fact that, for classification applications, the neural network is attempting to model the conditional data distribution $p(\mathbf{t}|\mathbf{x})$, the error function must

reflect the fit of the network to this distribution. As was discussed earlier (in Subsection 2.2.2), the distribution of the class labels $\mathbf{t}$ is discrete because every example is assigned to only one class, and every class is mutually exclusive. For neural networks, class labels are usually presented using a 1-of-C encoding. With this encoding scheme, if the input example $\mathbf{x}_n$ is assigned to class i, then the i^{th} element of $\mathbf{t}_n$ is set to one, and all of the other elements of $\mathbf{t}_n$ are set to zero.

From (12), the network outputs $\mathbf{y}_n$ can be treated as the probabilistic likelihood of the network assigning class label $\mathbf{t}_n$ to $\mathbf{x}_n$. Assuming the existence of multiple classes, the multinomial probability density function presents a natural approach to modeling this distribution. Thus, the likelihood of the model correctly classifying $\mathbf{x}_n$ can be expressed as:

$$p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) = \prod_{k=1}^{C} (y_{k,n})^{t_{k,n}} \tag{14}$$

By taking the negative logarithm of the likelihood estimate for every example in the data set, the cost function $J(\mathbf{w})$ can be constructed:

$$J(\mathbf{w}) = -\sum_{n=1}^{N} \ln p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{C} t_{k,n} \ln y_{k,n} \tag{15}$$

which is known as the *cross-entropy* error function

In order to minimize the cost function, the estimated gradient of the error surface is used to modify the network weights. This gradient is generally expressed as the direction of steepest descent in the error surface. Thus, the gradient can be calculated using the derivative of the cost function with respect to the weights of the network:

$$\nabla J(\mathbf{w}) = \frac{\partial J}{\partial \mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial J_n}{\partial \mathbf{w}} \tag{16}$$

The gradient can be calculated by averaging the derivative to the cost function for each example in the data set. The derivative of the cost function for each example can be computed by using the derivative chain rule to reverse the computation of the network outputs. The partial derivative of the cost function with respect to the weights in the output layer yields the appealing result:

$$\frac{\partial J_n}{\partial w_{jk}} = \frac{\partial J_n}{\partial y_{k,n}} \frac{\partial y_{k,n}}{\partial v_{k,n}} \frac{\partial v_{k,n}}{\partial w_{jk}} \tag{17}$$

$$= -(t_{k,n} - y_{k,n}) \frac{\partial v_{k,n}}{\partial w_{jk}} \tag{18}$$

$$= -y_{j,n}(t_{k,n} - y_{k,n}) \tag{19}$$

Intuitively, the error gradient is equivalent to the difference between the desired response and the network output, multiplied by the input to the weight from the neuron j in the previous layer. For a more thorough derivation of the cross-entropy error gradient, see [29][4, pp. 230–240].

For the neurons in the hidden layers, no explicit output response exists to guide the weight adjustments. Instead, the error gradient is propagated backwards through the network to adjust the network weights in proportion to the influence that they have in determining the network output.

$$\frac{\partial J_n}{\partial w_{ij}} = \frac{\partial J_n}{\partial y_{j,n}} \frac{\partial y_{j,n}}{\partial v_{j,n}} \frac{\partial v_{j,n}}{\partial w_{ij}} \tag{20}$$

$$= -\sum_{k=1}^{C} \{(t_{k,n} - y_{k,n}) w_{jk}\} \frac{\partial y_{j,n}}{\partial v_{j,n}} \frac{\partial v_{j,n}}{\partial w_{ij}} \tag{21}$$

$$= -x_i y_j (1 - y_j) \sum_{k=1}^{C} \{(t_{k,n} - y_{k,n}) w_{jk}\} \tag{22}$$

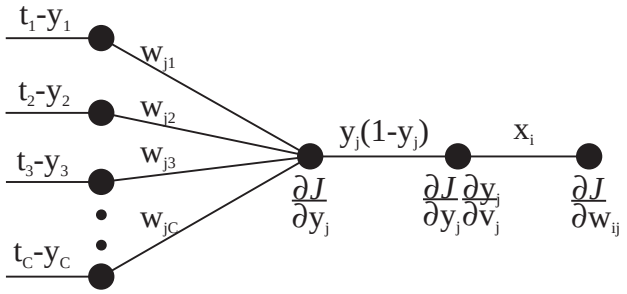

Figure 4: Signal-flow graph for the backward propagation of the gradient estimate to neuron i.

Because the gradient computes the steepest slope of the error surface with respect to each weight, it provides the network with the direction in which each weight should be adjusted to reach a minimum. Thus, the gradient can be used to move the network weights some distance in the direction of the greatest decrease in error. However, the error gradient does not provide a measure of *how far* the network state is from a minimum. If the network is

moved too far in the direction of the gradient, it may overshoot the minimum, potentially leading to an increase in the cost function. Conversely, if the distance that the network is moved is too small, then the network may require many iterations to locate the minimum.

Several algorithms exist to estimate how large a step size is required to reach a minimum. The most common approach is to use a fixed learning rate parameter η that will make the step size proportional to the size of the gradient. The computed adjustment to the network weights is called the delta weight ($\Delta\mathbf{w}$) and is given by:

$$w_{ij} = w_{ij} + \Delta wij = w_{ij} + \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial J_n}{\partial w_{ij}} \tag{23}$$

A variety of alternate approaches exist to determine the size of the delta weight vectors. The simplest variation includes a *momentum* term to accelerate gradient descent when the slope of the gradient is consistent, and to help stabilize the weight adjustments when the slope is oscillating [22, pp. 149–151]. Some approaches use an adaptable learning rate schedule (similar to simulated annealing) to gradually decrease the step size as a minimum is approached [10][11]. Other techniques exploit second-order information about the error surface and use line search to determine an optimal learning rate η [33][14][35]. [47] presents an overview and empirical comparison of most of these techniques.

Through the repeated process of exposing the network to the data set and adjusting the network weights, the neural network will eventually converge to a minimum in the cost function $J(\mathbf{w})$. If the network has converged successfully, its weights $\mathbf{w_{conv}}$ should enable the neural network to approximate the distribution of the data set $p(\mathbf{t}|\mathbf{x})$. Additional issues relating to the speed of network convergence and the quality of the minimum attained will be discussed later in this report.

2.3.3 Unsupervised Neural Networks

Unsupervised neural networks are generally similar to supervised networks in terms of their structure and properties. Both types of networks tend to use regularly structured graphs organized into layers with parameter values attached to each edge. Also, both types of algorithms are iterative and use gradient descent techniques to minimize an error function. However, unlike supervised networks, unsupervised networks are capable of modeling a data set

without the guidance of class information. Thus, unsupervised networks are generally applicable to a wider range of data sets which may be underanalyzed and may have relatively little domain knowledge available. The implication of the independence of class information is that there is no explicit target value for the output of an unsupervised network function to model. Thus, the network error function must be defined relative to the input data. Statistically, this distinction can be summarized by the difference between modeling the conditional distribution $p(\mathbf{t}|\mathbf{x})$ and the unconditional probability distribution $p(\mathbf{x})$.

The unsupervised neural network technique that will be dealt with in this report is the general mixture model. In addition to being one of the more powerful unsupervised neural network techniques, the general mixture model is well suited to the statistical framework established in this report. The introduction to the data mining task of clustering in Subsection 2.2.2 describes the general mixture model approach. This subsection will introduce the algorithm and will present its position within the area of unsupervised neural networks.

Mixture Model Networks

The goal of a general mixture model is to estimate the underlying data distribution $p(\mathbf{x})$. As stated previously, for most data sets, this problem is too difficult to be solved with a single parametric model. However, any density function can be approximated to an arbitrary degree of accuracy by a set of simpler distributions (as long as the number of distributions is sufficiently large) [4, p. 61]. By choosing a finite number of simple distributions, $p(\mathbf{x}|i)_{i=1}^{M}$, mixture models attempt to optimize the parameters of each distribution to fit the unconditional data distribution $p(\mathbf{x})$ as closely as possible. Because mixture models combine the probability estimates of multiple distributions, they are known as semi-parametric techniques.

In a mixture model network, every neuron i represents a simple parametric distribution known as a *component*. The choice of which type of distribution to use in a mixture model depends on the nature of the application and the structure of the data. For example, if the data set consists of real-valued samples, it is advisable to use a continuous density function, such as a Gaussian or log-Gaussian distribution, to model the data. Alternately, if the data is nominal or count-valued, a multinomial or Poisson distribution may be more applicable. Statistical literature details a variety of different types of density functions to suit different data distributions – most of which are applicable to a general mixture model framework [9]. Frequently (especially for data mining databases), a data set will contain a combination of different data

types. To model this heterogeneous data, the mixture model may have to apply different distributions to different input dimensions. For simplicity, this report will only focus on mixture models employing a single type of component: Gaussian distributions. Networks containing Gaussian components will be examined because they are the most common and best understood type of mixture models. In addition, although they are continuous distributions, Gaussian densities can be used to approximate discrete data. Thus, mixture models with only Gaussian components can adequately model heterogeneous data sets.

In a mixture model network, the weights $\mathbf{w}_i$ encode the parameters for the i^{th} component in the network. For Gaussian components, each set of weights specifies the mean, variance, and mixture proportions for a component distribution: $\mathbf{w}_i = \{\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i, \pi_i\}$. To simplify the derivations of the mixture model, it will be assumed that the covariance matrix of the distribution will be constrained to a scalar, such that $\boldsymbol{\Sigma}_i = \sigma^2\mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Thus, the Gaussian distribution is radially symmetric in the input space. Derivations of a Gaussian mixture model using a full covariance matrix can be referenced in [7]. In addition, the mixing proportions π_i satisfy the following constraints:

$$0 \leq \pi_i \leq 1 \tag{24}$$

$$\sum_{i=1}^{M} \pi_i = 1 \tag{25}$$

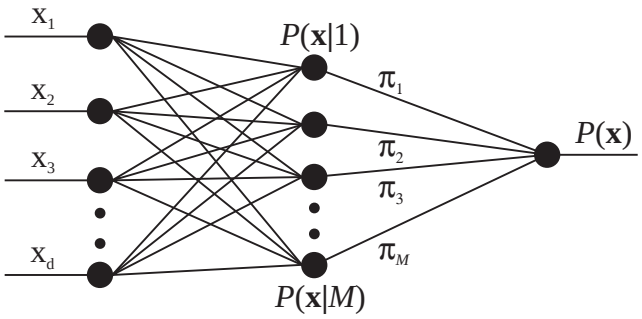

Figure 5: Network representation of a general mixture model.

Figure 5 displays the network representation of a general mixture model. This network consists of two fully interconnected layers of weights. The first layer encodes the parameters for each of the M component distributions. The

output of each neuron in this layer is the probability estimate $P(\mathbf{x}_n|i, \mathbf{w}_i)$ that each component density i generated the input example $\mathbf{x}_n$. The second layer combines the probability estimates for each component with the set of prior probabilities π_i. The resultant output is the cumulative likelihood that input example $\mathbf{x}_n$ was generated by the mixture model. In other words, the network outputs how well the model fits the data distribution $p(\mathbf{x})$. Thus, the network function for a general mixture model can be expressed as:

$$y(\mathbf{x}_n) = p(\mathbf{x}_n|\mathbf{w}) = \sum_{i=1}^{M} \pi_i p(\mathbf{x}_n|i, \mathbf{w}_i) \tag{26}$$

This equation outlines the network function for a general mixture model; however, it does not describe how the component density function $p(\mathbf{x}|i, \mathbf{w})$ is to be evaluated. The evaluation of the component density function is contingent on the type of parametric distribution used by the network. With the assumption that the mixture model network consists of Gaussian components, the component distributions can be specified using the normal density function. By supplying an input vector $\mathbf{x}_n$ to the network, the likelihood function of each component can be evaluated using the component density function and distribution parameters $\mathbf{w}_i$:

$$y_i(\mathbf{x_n}) = P(\mathbf{x}_n|i, \mathbf{w}_i) = \frac{1}{(2\pi\sigma_i^2)^{d/2}} \exp\left\{-\frac{\|\mathbf{x_n} - \mu_i\|^2}{2\sigma_i^2}\right\} \tag{27}$$

The probabilities of each component can be combined with the mixture probabilities using (26) to produce the output of the network function. By combining the network output probabilities for each of the examples in the data set, the likelihood of the model can be estimated (6). Once again, it is straightforward to transform this likelihood estimate into a cost function by taking the negative logarithm:

$$J(\mathbf{w}) = -\sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{w}) = -\sum_{n=1}^{N} \ln\left\{\sum_{i=1}^{M} \pi_i p(\mathbf{x}_n|i, \mathbf{w}_i)\right\} \tag{28}$$

Like supervised neural networks, the error function for mixture models can be minimized by calculating the derivative of the cost function with respect to the network parameters. For mixture models, component distributions often have several different types of parameters which serve different functions in the estimation of the data distribution $p(\mathbf{x})$. The derivative for each type of

parameter must be calculated separately. Below are listed the derivatives for the parameters of a Gaussian mixture model:

$$\frac{\partial J}{\partial \boldsymbol{\mu}_i} = \sum_{n=1}^{N} P(\mathbf{w}_i, i|\mathbf{x}_n) \frac{(\boldsymbol{\mu}_i - \mathbf{x}_n)}{\sigma_i^2} \tag{29}$$

$$\frac{\partial J}{\partial \sigma_i} = \sum_{n=1}^{N} P(\mathbf{w}_i, i|\mathbf{x}_n) \left\{ \frac{d}{\sigma_i} - \frac{\|\mathbf{x}_n - \boldsymbol{\mu}_i\|^2}{\sigma_i^3} \right\} \tag{30}$$

$$\frac{\partial J}{\partial \pi_i} = -\sum_{n=1}^{N} \{P(\mathbf{w}_i, i|\mathbf{x}_n) - \pi_i\} \tag{31}$$

The density $p(\mathbf{w}_i, i|\mathbf{x})$ is the posterior probability distribution of the general mixture model. The posterior probabilities can be computed from the model likelihoods by using Bayes' theorem and (26):

$$p(\mathbf{w}_i, i|\mathbf{x}) = \frac{p(\mathbf{x}|i, \mathbf{w}_i)P(i)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|i, \mathbf{w}_i)P(i)}{\sum_{j=1}^{M} p(\mathbf{x}|j, \mathbf{w}_j)P(i)} \tag{32}$$

where $P(i)$ is the prior probability of component i which is equivalent to π_i. From the equation, it is clear that the posterior probability is simply the normalized version of the likelihood probabilities for each component. Thus, it is ensured that $\sum_{i=1}^{M} \pi_i p(\mathbf{x}|i, \mathbf{w}_i) = 1$.

To minimize the error function with respect to the parameters, the derivatives can be set to zero. Unlike the gradient estimates for supervised networks (19, 22), the derivatives for mixture model networks can be solved directly in terms of the model parameters. Thus, it is possible to determine from the derivatives the parameter values which minimize the cost function $J(\mathbf{w})$. The following set of equations show the estimated optimal values solved for each of the model parameters:

$$\hat{\boldsymbol{\mu}}_i = \frac{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)\mathbf{x}_n}{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)} \tag{33}$$

$$\hat{\sigma}_i^2 = \frac{1}{d} \frac{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)\|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_i\|^2)}{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)} \tag{34}$$

$$\hat{\pi}_i = \frac{1}{N} \sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n) \tag{35}$$

In the first equation, the estimated mean $\hat{\boldsymbol{\mu}}_i$ is equal to the average of the data vectors in $\mathcal{D}$ weighted by the posterior probabilities that each example was

generated by component i. Similarly, the estimated variance $\hat{\sigma}_i^2$ is equivalent to the variance from the component mean to every data vector weighted by the posterior probabilities. Finally, the estimated prior probability $\hat{\pi}_i$ can be determined by averaging the posterior probabilities for every data vector in the data set. Clearly, the derived estimates for the parameters of a Gaussian mixture model present an intuitively satisfying result.

Unlike the gradient calculations for supervised learning, the cost function derivatives for Gaussian mixture models can be solved exactly for each of the network parameters. The implications of this solution is that the direction *and* the distance from the current parameter value to the value which minimizes the derivative $\frac{\partial J}{\partial \mathbf{w}}$ can be determined directly. Thus, there is no need to estimate the distance to the minimum using a learning rate parameter η. However, it is often still useful to transform these equations into the format of the delta weight update equation for supervised networks from (23).

$$\boldsymbol{\mu}_i = \boldsymbol{\mu}_i + \eta(\hat{\boldsymbol{\mu}}_i - \boldsymbol{\mu}_i) \tag{36}$$

$$\sigma_i^2 = \sigma_i^2 + \eta(\hat{\sigma}_i^2 - \sigma_i^2) \tag{37}$$

$$\pi_i = \pi_i + \eta(\hat{\pi}_i - \pi_i) \tag{38}$$

In these equations, the delta weight value ($\Delta \mathbf{w}_i$) is equivalent to the distance from the old weight value to the new estimated value multiplied by the learning rate η. It can be easily observed that if $\eta = 1$, these weight update equations will reduce to (33). The importance of these weight update equations will become apparent later in this report when batch learning is used to calculate new parameter values based on subsets or batches of the data set.

A Brief History of Unsupervised Networks

The general mixture model represents the culmination in the evolution of a variety of unsupervised neural networks. Unsupervised neural networks trace their roots to the humble *competitive layer network*. The deterministic version of this network is equivalent to the well-known *k-means* algorithm [39][12][2][54]. In a competitive layer network, each neuron competes to represent an input example based on some proximity metric. Using a winner-take-all approach, only the weights of the closest neuron are updated to move that neuron proportionally nearer to the input example. At the completion of network training, the weights of each neuron represent the mean of the data examples that are closest to it. The main two difficulties with this network are:

1. the winner-take-all approach permits only a single neuron to respond to each input. This form of update is inefficient and it creates the possibility

that some network neurons may never be updated because they never win the competition (a phenomenon called *neuron starvation*). Thus, it is difficult to choose a good distribution for the initial weight values to ensure that each neuron will be capable of representing an adequate proportion of the input data (this is further complicated by the curse of dimensionality).

2. the network weights provide relatively little information about the examples that belong to each cluster and the boundaries between clusters are not well defined.

Kohonen's *self-organizing feature map (SOFM)* presents several improvements to the simple competitive network that attempt to solve these difficulties [22, pp. 397–434][45][31][37]. In a self-organizing feature map, some form of a topological ordering is imposed upon the neurons in the network. This ordering is generally in the form of a two-dimensional mesh or a line. The implications of this ordering is that the neurons topologically adjacent to the winner will be updated in response to each input. Topological adjacency is determined by a neighbourhood function which initially specifies a very large neighbourhood of neurons to be updated, and which gradually shrinks during network training. This improvement of the SOFM overcomes the first difficulty of competitive layer networks; it is likely that every neuron in the network will be updated in response to some proportion of the inputs. In addition, due to the ordering imposed by the topological structure, adjacent clusters in the network correspond to adjacent locations in the input data. Thus, the examples described by neighbouring clusters are likely to have similar properties. This topological adjacency provides more information about the data than simply the cluster location.

However, as the dimensionality of the data increases, the notion of adjacency among neurons becomes less clear. Adjacency is imposed by some topological structure which, in general, is unlikely to fit the data. Often, the topological structure will become considerably warped as the network is trained to fit the data. For example, if the neurons are organized in a two-dimensional mesh and the data set consists of points uniformly distributed in three dimensions, the mesh will become quite distorted as the network stretches to fill the data space. This distortion effect becomes considerably worse in higher dimensional data sets. Thus, adjacency in high dimensions may no longer correspond to the closest Euclidean distance between neurons. Another effect of this warping is that neurons will often be left in empty space

straddling two separate clusters that are being represented by their topological neighbours. For example, consider a data set with two distinct clusters which is being modeled by a three neuron linear map. It is clear that the topological structure of the map will cause the middle neuron to be mapped to the empty space between the two distributions. This middle neuron is effectively inactive. For high dimensional data sets which are more sparsely distributed, the number of wasted neurons increases considerably. In addition, the self-organizing feature map provides relatively little information regarding the boundaries between clusters or about the generation of the examples in each cluster. Thus, the self-organizing feature map, when applied to high dimensional data, generally provides insufficient information to overcome the second difficulty of competitive learning.

Recently, neural network researchers have begun recognizing the value of the general mixture model approach and its applicability to unsupervised neural network learning [4, pp. 59–73] [42]. General mixture models present a theoretically-sound solution to the two problems of competitive learning mentioned above. As opposed to the winner-take-all approach of competitive networks, mixture models apply what is known as *soft competitive learning* [42][41][40]. With this approach, every neuron in the network is updated in proportion to the probability of having generated the input example. Thus, neurons in the neural network can be interpreted as probability density functions that are compared in terms of their likelihood of having generated the input data. The advantage of the proportional updating of every weight in the network due to soft competition is that the problem of neuron starvation is thwarted. Every neuron in the network is likely to eventually describe some proportion of the data set. In addition, the problem of stranded neurons has been avoided because there is no *a priori* topological structure imposed on the networks.

The problem of the understandability of cluster representations is also neatly solved by mixture models. Because each neuron outputs a likelihood probability estimate, clusters of data examples can be described by a probability density function. Due to the fact that probability distributions are well supported in statistical literature, the results of mixture model training are easily understandable. The parameters of a probability distribution provide considerably more information about the data than the cluster mean and the adjacency information determined by the previous two techniques. Importantly, the type of distribution can be chosen to suit the underlying probability distribution of the data; for example, discrete distributions can be used to fit

nominal data, and continuous distributions can be used to fit real–valued data. Characterizing a data set in terms of a set of probability distributions is also easily comparable with the output of other statistical methods – unlike the results of other unsupervised networks. In addition, the probabilistic likelihood estimate of each neuron is much easier to interpret than distance metrics employed by competitive networks. Lastly, by simplifying a Gaussian mixture model so that every neuron has equal prior probabilities and infinite variances, the mixture model would be theoretically equivalent to a competitive layer network or a self-organizing feature map where the neighbourhood includes just the winning neuron. Thus, there is a natural progression in the evolution of unsupervised neural network techniques to the general mixture model.

2.3.4 Deterministic, Batch and Stochastic Learning

All of the neural network algorithms presented thus far in this report are *deterministic*. During training, they compute each weight update ($\Delta\mathbf{w}$) by using all of the available information in the data set in order to calculate the gradient of the error surface ($\frac{\partial J}{\partial \mathbf{w}}$).

$$\nabla J^{\text{det}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial J_n}{\partial \mathbf{w}} \tag{39}$$

By using the entire data set, the network attempts to come as close as possible to the true gradient to the probability distribution underlying the data. This true gradient is estimated by averaging the instantaneous gradient calculated for each example. Thus, the estimate calculated by the network algorithm will be a noisy but unbiased approximation of the true gradient. Because averaging is a commutative operation, the order that the data examples are presented to the network is irrelevant. Therefore, neural network algorithms which use the entire data set to estimate the gradient for each weight update are deterministic.

The main drawback of deterministic algorithms is that they are inefficient for large databases containing redundancies. Generally, a good estimate of the gradient can be calculated using some subset of the entire data set. Thus, deterministic networks may perform many unnecessary calculations before the network weights will be updated. By relaxing the conditions of deterministic learning, the network algorithm can be made more efficient by allowing more weight updates to occur during an iteration through the data set. Thus, the data set can be partitioned into several subsets or *batches*, and the network

weights can be updated after calculating the gradient for each batch. The gradient for batch b can be calculated from the average of the instantaneous gradients for each of the examples in the batch:

$$\nabla J_b^{\text{bat}}(\mathbf{w}) = \frac{1}{B} \sum_{n=1}^{B} \frac{\partial J_n}{\partial \mathbf{w}}, \tag{40}$$

where B is the batch size and $B \leq N$. For a single epoch, the batch gradient would be evaluated sequentially for each batch b. It is important to note, however, that the training algorithm is no longer deterministic because the order of the presentation of each batch will affect the state of the network.

If the number of batches is equal to the number of examples in the data base, then the network learning algorithm is considered to be *stochastic*. For stochastic learning, the gradient of the error surface is estimated from a single example. Thus, the stochastic gradient for each example n can be expressed as:

$$\nabla J_n^{\text{sto}}(\mathbf{w}) = \frac{\partial J_n}{\partial \mathbf{w}} \tag{41}$$

The stochastic gradient is calculated to update the network weights after the presentation of each example n in the data set. Thus, a stochastic learning network will update its weights N times during a single epoch. Because the gradient, in this approach, is calculated from a single example, it is a very noisy approximation to the deterministic gradient. However, as long as the step size η is chosen appropriately, the noise term of the stochastic gradient is bounded, and the network will be adjusted, on average, in the direction of the deterministic gradient. Unless the learning rate is very small or is decreased using a dynamic learning rate schedule, the noise term in the gradient estimates will prevent the network from settling into a minimum[2]. However, for most applications, the network need not converge to a minimum to provide a good solution.

Despite the difficulty in reaching minimum, there are many advantages to stochastic learning over deterministic learning. First and foremost, stochastic algorithms are likely to be orders of magnitude faster at attaining good results for large, redundant data sets [22, p. 162]. Stochastic algorithms also tend to be better at evading local minima and locating the global minimum in the search space. In addition, stochastic algorithms do not require the storage of the gradient estimates for the entire data set. For applications where data is

[2]The Robbins-Munro formula demonstrates the convergence properties of a stochastic algorithm if $\lim_{n\to\infty} \eta_n = 0$ [4, pp. 46–47].

continuously arriving and requiring processing, stochastic neural networks can handle this data in real-time.

It is important to note that, because the network weights are updated after the presentation of each example, the solution attained by the network is partially contingent on the order that examples are presented to the network. Stochastic learning network tend to be biased in their representation towards the initial examples that are presented to the network. To limit the effects of presentation order, it is often assumed that data examples are selected randomly from the data set or the data set is permuted after every iteration.

This report examines the continuum between the extremes of deterministic and stochastic learning. As will be shown in the next subsection, deterministic learning networks are more amenable to parallel implementation than stochastic networks. A batch learning compromise between these two extremes can harness the parallelization advantages of deterministic learning and the improved convergence speed of stochastic learning.

2.4 Parallelism

Parallelism presents an effective solution for dealing with the computational overhead incurred by data mining applications. However, the development of parallel software is not as simple as executing a sequential algorithm on a multi-processor system. The scarcity of available, widely-used parallel software attests to the fact that the process of parallelization is substantially more complex. Migrating any application to a parallel system requires the fulfillment of several design criteria. In order to successfully meet these guidelines, an appropriate parallel model and parallel platform must be selected. Correctly choosing these will help evade the pitfalls that have caused much parallel software to fall into obscurity.

This subsection will outline some of the properties and challenges of parallel hardware and software. Several design criteria to assist the selection of an appropriate parallel model will be posited. Bulk Synchronous Parallelism (BSP) will be presented as a parallel model for software development which meets these criteria.

2.4.1 Parallel Software Models

The initial step in parallelization entails selecting a parallel model for software development. Skillicorn and Talia outline six characteristics of an ideal model [50]:

- it should be easy to program;
- it should have a well-defined software methodology;
- it should be architecture-independent;
- it should be easy to understand;
- it should be efficiently implementable; and
- it should provide accurate cost analysis.

The selection of a parallel model is assisted by determining how successfully a specific model meets these criteria.

Existing parallel models can be positioned on a range from abstract to low-level. Abstract models conceal the details of parallelization from the developer. Thus, parallel code is easy to develop, analyze and debug (appearing much like a sequential program). In addition, it is also easy to migrate between parallel platforms because the compiler is responsible for handling all of architecture-specific details. However, as a result, it is hard to implement efficient parallel code because the methods of specifying parallelism are quite general. Abstract models are often implemented in high-level languages which have been customized for parallel development and are not widely supported for development in industry.

Low-level models, on the other hand, enable efficient implementation because the developer can customize the code to meet the characteristics of a specific parallel machine. However, the tradeoff of gaining this efficiency is that the developer must specify all of the details of parallelism. Thus, the parallel software becomes difficult to understand, tedious to debug and highly machine-specific. The resulting code is incapable of attaining widespread use because it is too difficult to migrate the software between different parallel machines. Most early parallel software attempts fall into this category.

Clearly, the models at the extremes of this spectrum fail to meet all of the characteristics of an acceptable environment for parallel software development. A mid-range compromise which minimizes the tradeoff between portability and efficiency is desirable. Most contemporary parallel models operate at this medium level of abstraction and differ mainly in terms of their language of implementation and what they choose to make explicit versus what they choose to conceal. For example, a parallel model, such as BSP [24][23][49], requires the software developer to specify which parts of the program should be implemented in parallel; however, it hides the details of how this parallelism

is actually implemented. A parallel model, such as NESL [5][6] which occurs at a higher level of abstraction conceals even the parallel specification of the software from the developer. Linda [8] and Mentat [20], on the other hand, exist at a lower level of abstraction that requires the software developer to explicitly stipulate almost all of the details of parallel implementation. Despite the different levels of abstraction, these models manage to satisfy most of the established criteria to some degree – they are abstract enough to be understandable, portable, cost predictable and easy to program, and yet sufficiently low-level to enable efficient implementation. In addition, most contemporary parallel models are developed as extensions to or libraries for sequential languages commonly used in software development. In fact, some parallel models offer extensions to several standard programming languages. Often these sequential languages will already have a software methodology to complement parallel software development.

2.4.2 Parallel Hardware

Parallel software development also entails the selection of an appropriate parallel architecture on which to execute and evaluate parallel code. However, unlike sequential machines which exhibit more or less homogeneous properties across a variety of applications, different parallel machines may have markedly different performance results on different applications. The performance of a parallel machine on a particular application is a product of the interrelation of its components. By examining the different components and different types of parallel machines, the task of choosing the right parallel architecture can be simplified.

Parallel computers consist of three main components: processors, memory modules and an interconnection network [50]. The choice of each of these components and the way that they are combined determines the type of the parallel machine. There are two major distinctions to be made based on these combinations [18]. The first distinction is contingent on the placement of memory modules within the machine. If each processor has its own local memory, the parallel computer is a distributed-memory system. If the memory modules are detached from an individual processor and are globally accessible by all processors, the machine is a shared-memory system. The second distinction deals with the way in which the processors operate. If each processor executes independently operating on potentially different programs, the parallel machine is multiple instruction, multiple data (MIMD). Alternately, if a single processor executes a program thread and distributes operations to the other

processors in the system (such as in a master-slave paradigm), the machine is considered to be single instruction, multiple data (SIMD).

For a particular algorithm, there may be an optimal number of processors that will ensure that each subtask of a program will be executed concurrently. However, it is expensive and inefficient to choose a parallel architecture that optimally handles the processing for a specific task, and, for most applications, the desired amount of parallelism will far exceed a feasible number of processors. Also, most parallel systems will be used for multiple applications for which the optimal number of processors may not be known *a priori*. To overcome this restriction, contemporary parallel models support *virtual parallelism*. This technique decomposes an application into independent *processes* which can be appropriately distributed among the number of actual *processors* in the system. Thus, an optimal parallel system can be simulated, though without the full benefit of parallel speedup. It is important to distinguish processes from processors: processes are logically independent tasks that can be executed concurrently on a parallel machine, and processors are the physical processing units in a parallel machine.

Communication is at the core of any parallel system. Because inter-processor communication is extraneous to the work done solving a particular problem, it is essential to minimize the cost of communication in relation to the cost of computation. Parallel systems that spend more time in communication than they do in computation largely defeat the purpose of parallelization. The amount of communication to computation required for a particular application will reflect the type of parallel architecture chosen. Applications which are communication-intensive require selecting a system which has low communication costs. Applications which require infrequent communications are less restrictive. Alternately, the parallel encoding of a particular application can be performed so as to meet the communication capabilities of a given parallel machine. For parallel machines with slow communication times, an algorithm can be decomposed so as to maximize the amount of computation performed prior to communication. Inversely, algorithms can use frequent communications to ensure full processor utilization for architectures with low communication costs. Clearly, a parallel model with accurate cost analysis would assist with this process.

The cost of communication for a particular parallel architecture is largely contingent on the type of interconnection network that it employs. An ideal parallel machine would utilize a fully-connected network to establish unique links between every processor. However, in reality, this type of network is in-

feasible because the number of interconnections scales quadratically with the number of processors. Even for a system with a modest number of processors, the required number of interconnections is huge. In addition, there is also the consideration that a processor must be capable of handling messages from all processors simultaneously which is impossible for a large number of processors. Because fully-connected networks are unattainable, a given communication may have to be routed through multiple processors in order to reach its destination. Thus, communication time is largely contingent on the length of time required for a particular message to traverse the interconnection network. Common types of interconnection networks are meshes, hypercubes and arrays. Each of these interconnection networks have different properties. For example, a mesh interconnect requires $O(\sqrt{p})$ to traverse a network of p processors; however, because each processor requires a fixed number of connections, p can grow arbitrarily large. A hypercube interconnect, on the other hand, only requires $O(logp)$ steps to traverse the network; however, it is less scalable because the number of connections increases with $\log p$. Thus, the choice of the interconnection network will affect the properties of the parallel machine.

The interconnection networks specified above are not the only types of communication topologies that can be used in a parallel system; in fact, any communications network can provide general-purpose parallelism. Currently, anything from a Token-ring or Ethernet network of workstations to the Internet can be used to perform parallel computation. For these networks, communication is considerably slower than within a parallel machine, due to the distance between connected processors and the communications protocol. However, the processing power is certainly present. These networks can generally only be employed for applications that require minimal communication. These communication networks, however, have the great advantage of being affordable and widely available. Most companies already have an established communications network, and thus would not have to buy any specific parallel hardware in order to gain parallel speedup. Because network computers are widely used, they are inexpensive to purchase and can be used for any personal computing task. Thus, these communication networks present a compromise between parallel speedup and cost.

2.4.3 Bulk Synchronous Parallelism

Bulk Synchronous Parallelism (BSP) is a relatively new parallel computation model which has been designed at a medium level of abstraction that

greatly facilitates the development and analysis of general-purpose parallel software [49][24][23]. Using BSP, a software developer is only responsible for explicitly specifying what parts of a program should be implemented in parallel. At run-time, the parallel model will implicitly distribute the program's processes among the processors in the parallel machine and will handle the implementation details of communication and synchronization. Thus, the BSP model enables the software developer to write efficient parallel code without getting bogged down in low-level implementation.

The BSP abstract machine assumes that the parallel system contains a series processor-memory pairs linked by some interconnection network. This specification is sufficiently general so that a BSP program can be implemented on any MIMD machine [50]. The generality of the BSP abstract machine enables BSP software to be architecture-independent. Parallelism is specified in a BSP program through a set of generic function calls to the BSP library. At compile time, the compiler will insert the appropriate parallelization code for the target parallel architecture into the developed software. The use of generic functions to specify parallelism enables BSP programs to be portable between different parallel architectures without having to alter their code. The BSP libraries support several of the major software development languages, such as C, C++ and Fortran. Thus, BSP can exploit the software development methodologies of these languages when composing parallel code. Because the specific details of parallel implementation are handled by the BSP library, BSP software also tends to be easy to develop and easy to understand. Other than the references to functions in the BSP libraries, BSP development is quite similar to writing sequential code.

BSP programs are further simplified by enforcing the division of a parallel program into a sequential composition of *supersteps*. A superstep is a parallel construct in which each processor executes the same code concurrently [49]. Every superstep consists of three distinct phases:

1. local computation,
2. global communication,
3. barrier synchronization.

During the local computation phase, all computational operations in the superstep are performed. Computations are specified as being local because each process can only access and operate on the variables that are stored in its memory space at the beginning of the superstep. Any communication

operation which occurs during this phase of the superstep is buffered until the communication phase. The cost of the local computation phase is determined by the computational overhead of the largest process. Thus, it makes sense to balance the amount of computation among each processor in the parallel machine.

All communication between processors is performed *en masse* during the communication phase. BSP treats communication as global, instead of as a series of point-to-point communications. As a result, message routing can be optimized, congestion effects can be minimized and communication performance can be accurately predicted. The global treatment of communication makes BSP well suited to communication-intensive applications. As with the computation phase, the cost of communications is bounded by the process which transmits or receives the largest amount of data; thus, the communication requirements should also be balanced.

The barrier of synchronization phase separates communication from computation. Only after synchronization is complete do the recently communicated values become accessible in the local memories of each process. By ensuring that all communication operations are complete before computation proceeds, there is no concerns of blocking processes, and deadlock is completely avoided. The use of barrier synchronization substantially reduces the complexity of parallel software, making it easier to develop and debug. However, on contemporary parallel architectures, barrier synchronizations are still quite expensive and should be used as sparingly as possible. The cost of the synchronization step can be reduced by ensuring the two previous phases are properly balanced. Regardless, a maxim of BSP software design is to attempt to minimize the number of supersteps in a parallel program.

As a result of the structure and properties of the BSP superstep, the execution performance of a BSP program is highly predictable. Because a BSP program is a sequential composition of supersteps, the cost of the program is simply the sum of the cost of each superstep. The cost of a BSP superstep can be straightforwardly computed from the program text and two architecture-specific parameters.

The first of these parameters, g, measures the permeability of the parallel machine to uniformly-random traffic [49]. In other words, g is a measure of the average communication cost for a fixed length message (one 32-bit word) to traverse the interconnection network under continuous network traffic. This parameter is an attempt to accurately estimate the average cost of communication for a specific parallel machine under normal operating conditions. g is

Computer	Processors	Exec Rate (Mflops)	g (flops/word)	l (flops)
SGI PowerChallenge	4	74	0.5	1902
Cray T3D	4	12	0.8	168
	16	12	1.0	181
	64	12	1.7	148
	256	12	2.4	387
Cray T3E	4	46.7	1.8	357
	16	46.7	1.7	751
IBM SP2	4	26	8.0	3583
	8	26	11.4	5412
Sun	4	10.1	4.1	118

Table 1: BSP parameters for typical parallel computers.

expressed in terms of the number of floating point operations (known as *flops*) that could be executed in the time that it takes for a communicated message to traverse the interconnection network. The second parameter, l, measures the average time required for the parallel machine to complete a barrier synchronization. l is also expressed in terms of floating point operations. Table 1 displays values for g and l on several typical parallel computers [49].

Because each of the three phases of a superstep are independent, the cost of a BSP superstep is simply the sum of cost of each phase. The BSP cost calculations can be expressed formally as:

$$\underset{\text{processes}}{\mathsf{MAX}} \; c_i \; + \; \underset{\text{processes}}{\mathsf{MAX}} \; h_i \; g \; + \; l \tag{42}$$

where, for processor i, c_i is the number of floating point instructions performed during the local computation phase, and h_i is the number of words sent or received by the processor. The cost of a parallel program computed using the BSP cost equation will yield a result in floating point operations; this result can be easily translated into a measure of time by using the parallel computer's execution rate (generally computed in megaflops per second). To simplify BSP cost analysis, it will be assumed that every basic mathematical operation, such as addition, subtraction, multiplication and division, can be computed in a single flop.

The BSP cost equation has been shown to be highly accurate for estimating the execution times of real-world parallel applications – typically within a few percent of the actual value [23]. Thus, the BSP cost model can be used to analyze the performance of a parallel program comparing different parallelization strategies on different parallel machines without laborious implementation and testing.

In summary, bulk synchronous parallelism satisfies the criteria for a general-purpose parallel computation model. It is sufficiently abstract so as to require minimal parallel specification, yet low-level enough to enable efficient implementation. Its superstep structure allows the development of software that is easy to understand and straightforward to implement and debug. BSP is incorporated into conventional programming languages, and thus, allows the adoption of standard programming methodologies with a minimal learning curve. The BSP abstract machine ensures that BSP software is independent of any specific parallel machine, and is highly portable among different platforms. Lastly, the BSP cost model is highly accurate across a diverse range of parallel architectures and applications.

3 BSP Cost Analysis of Artificial Neural Networks

The training of artificial neural networks is computationally-intensive and time-consuming – especially when neural networks are applied to data mining problems where databases can contain megabytes to terabytes of information. Parallelism is a sensible way to reduce the cost of network training. Two main approaches of neural network parallelization have been attempted.

The first approach entails the construction of special-purpose VLSI hardware (so-called *neurocomputers*) to implement a parallel neural network [22][25]. These neurocomputers may be programmable co-processors for accelerating standard neural operations, or they may implement a particular network model directly in silicon.

The second approach to neural parallelism is to develop neural network software for parallel computers. There have been a variety of neural network algorithms designed to operate on specific parallel machines [51][44]; however, there have been very few attempts to develop parallel neural network software using a general-purpose parallel programming model [3]. Therefore, the existing implementations tend to be architecture-specific, proprietarily-structured,

and unwieldy for comparison. Clearly, parallel neural network software could be improved if it was designed using an appropriate parallel programming model.

The BSP model is well-suited to general-purpose neural processing for several reasons. The first reason is that BSP is an excellent model for parallel software development that enables the creation of code which is easy to understand, efficient and architecture-independent. The second reason is that BSP is designed specifically to handle communication-intensive applications due to the global treatment of all communication actions. Considering the densely-interconnected structure of a neural network, any applicable parallel model must be capable of effectively handling broadcast messaging. The third and most important reason for the selection of BSP as parallel computation model for implementing neural networks is the accuracy of the BSP cost model. Neural networks are inherently modular; thus, there are a multitude of ways that a neural network can be divided into concurrently-executable pieces for parallel execution. As yet, there has been little consensus within the neural network community as to the best method of parallelization. Instead, researchers have generally chosen a parallelization strategy which best exploits the capabilities of their favorite (or available) parallel computer [48]. With the BSP cost model, however, different neural network parallelization strategies can be considered across a range of possible parallel computers without requiring tedious implementation and testing. Thus, an optimal method of parallelization can be chosen to suit a particular parallel computer or application. In addition to the benefits of the BSP cost model, the superstep structure of BSP programs effectively limits the range of parallelization possibilities to a manageable number.

It is the focus of this section to analyze several strategies for supervised and unsupervised neural network parallelization using the BSP cost model. The cost of these strategies will be examined theoretically and with application to specific parallel computers. The cost-minimal strategy will be advocated as the best way to implement parallel neural networks.

3.1 Neural Network Parallelization Strategies

There are a variety of different parallelization strategies which have been considered for artificial neural networks [53][46]. These strategies can be considered in terms of their granularity of network decomposition. Due to the modularity of the neural network structure, there are several levels at which neural

network processing can be divided into concurrently-executable components. This report will consider three main levels of neural network parallelism:

1. *Exemplar parallelism (EP)*: this approach uses the existence of a large number of data examples as the source of parallelism; it does not attempt to exploit any of the parallelism present in the neural network itself. For exemplar parallelism, the work of neural network training is reduced by distributing an equal-size partition of the data set to each processor. Every processor trains an identical network on its local set of data examples.

2. *Block parallelism (BP)*: this approach partitions the network into blocks of adjacent neurons that are distributed among the processors.

3. *Neuron parallelism (NP)*: for this approach, each individual neuron is treated as a concurrent process and is randomly distributed among the processors in a parallel machine.

There are two other levels of neural parallelism that are worth mentioning. The first of these is training-session parallelism. This approach entails the simultaneous training of independent neural networks on different processors. For example, every processor in a parallel machine could be used to train a different type of network on a different data set. It is unclear what communication, if any, occurs between the processors in this approach. Thus, training-session parallelism represents a trivial level of parallelism that could be just as easily executed on several different sequential machines. The other approach worth mentioning is known as weight parallelism. Weight parallelism is a simple expansion of neuron parallelism where the weights connected to every neuron in the network are distributed among several processors; in essence, this approach parallelizes the weighted sum computation for each neuron. In contrast to training-session parallelism, the granularity of weight parallelism tends to be too fine for general-purpose parallelism. With this approach, it is unlikely that the computation performed by each processor will outweigh the communication required. Thus, this level of parallelism is rarely efficient for general-purpose parallelism.

3.1.1 Exemplar Parallelism

Figure 6 illustrates the steps of training a parallel neural network using exemplar parallelism. Initially, the data set is distributed among each of the

processors in the parallel computer. In addition, each processor receives an identical copy of the initial neural network. Training begins when each processor iterates through every example in its local data set calculating the error gradients (in the standard sequential fashion). Once a processor has completed its computation, it broadcasts its gradient estimates to every other processor. Thus, there is a total exchange of the complete gradient information from each processor's data subset. After each processor receives the gradient information from every other processor, it updates the weights of its neural network. Thus, once an iteration (or *epoch*) is complete, every processor will have an identical copy of the network with weights that have been trained on entire data set. In effect, each network will be in the same state that it would have been in if it trained on the entire data set. At this point, each processor can begin another iteration through its local data set and training can continue.

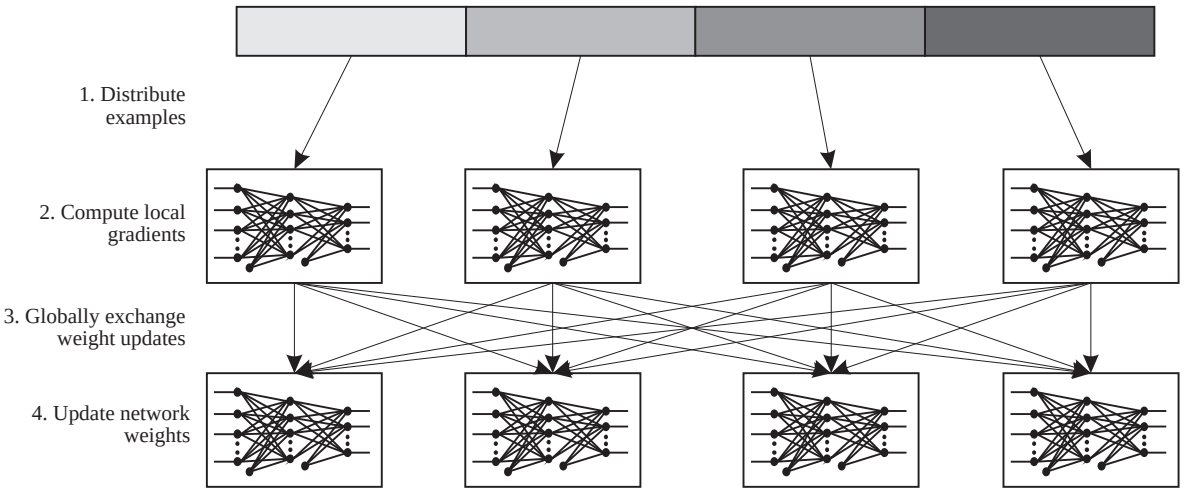

Figure 6: Diagram of the steps in exemplar parallelism. Each processor contains an identical copy of the neural network.

Due to the fact that network weights are only updated after every example has been processed, the neural networks used for exemplar parallelism are necessarily deterministic. For deterministic neural networks, weight adjustments are computed from the average gradient of every example in the data set (see Subsection 2.3.4). Due to the commutativity of the gradient averaging process, the order in which examples are processed is irrelevant. Thus, the gradient for each example can be computed independently and then combined to form

the network weight adjustment. The independence of the computation for each example, means that the processing of each example can be performed in parallel.

For the vast majority of real-world neural network applications, the number of training examples will exceed the number of available processors; thus, exemplar parallelism will be able to fully exploit the capabilities of a parallel machine. An additional advantage of exemplar parallelism is that, because each processor performs the same tasks, it is easy to balance the computation and communication load for each processor. Thus, neural networks implementing exemplar parallelism tend to be quite efficient. A potential limitation of exemplar parallelism is that it requires that the entire set of network weights must be stored in the local memory of each processor. For certain applications which require very large networks, a finer granularity of parallelism may be necessary. However, this possibility is quite unlikely for most parallel machines and neural network applications.

3.1.2 Block Parallelism

An alternate approach to parallelizing artificial neural networks is to exploit the parallelism inherent in the neural network itself. By distributing the task of neural computation among multiple processors, network training can be accelerated. It is the goal of an efficient parallelization strategy to partition the neural network topology in such a way so that every processor is fully utilized and the amount of communication between processors is minimized.

The block parallelism approach divides the neural network into several blocks of adjacent neurons. For simplicity, it will be assumed that these blocks are non-overlapping, and approximately rectangular with depth x and width y. Each block of neurons is distributed among the processors in the parallel computer. The processors containing neurons from the input layer of the network will begin by evaluating the input data. Their resultant output will be propagated to processors containing neurons in subsequent layers. Once the output layer is reached, the flow of computation will be reversed, propagating the error signals back through processors containing neurons in previous network layers.

This approach to parallelization attempts to take advantage of the limited locality that exists between adjacent neurons. This locality can be maximized by storing all of the neurons from a particular layer in a single processor; this approach to block parallelism, known as *layer parallelism*, is displayed in Figure 7. With this approach, each block of neurons only receives M values from

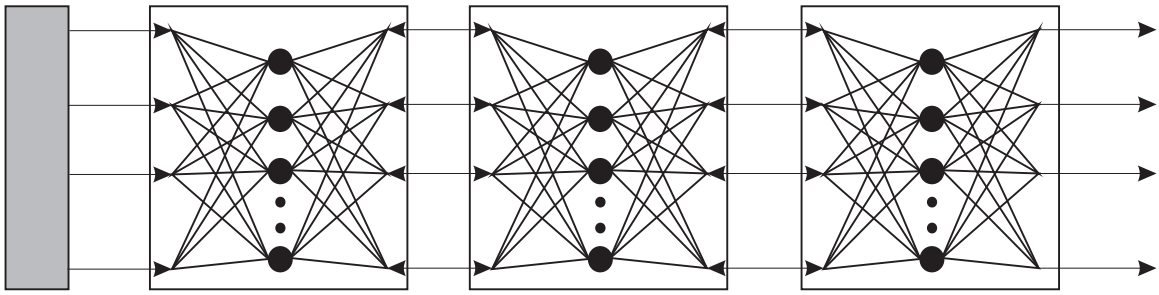

Figure 7: Partitioning of a neural network using layer parallelism ($x = 1$ and $y = M$). Every block corresponds to a processor containing a single network layer.

each adjacent layer (where M is the number of neurons in a network layer). Instead, if each layer is partitioned into several blocks distributed among different processors, then each processor has to receive the same M values. Thus, the number of redundant communications increases proportionally to the number of processors handling each layer. The drawback of layer parallelism is that there are rarely many layers in a neural network. Theoretically, a supervised neural network with a single hidden layer is capable of approximating any continuous multivariate function to any degree of accuracy as long as a sufficient number of hidden neurons are used (a property known as *universal approximation*) [22, p. 182][27]. Therefore, it is theoretically unnecessary to have many hidden layers of neurons. However, there are certain applications, such as pattern recognition [34] and modular neural networks [28], where more network layers are commonly employed. Nonetheless, it is likely that the number of processors in the parallel machine will exceed the number of layers in the neural network; thus, it will often be necessary to partition layers into blocks in order to fully exploit the parallelism of the system.

Due to the flow of computation in a neural network, processing is performed on a layer-by-layer basis. Thus, only processors containing neurons from the active layer will be occupied. By increasing the depth x of each block, so that each processor contains at least one neuron from each layer of the network, every processor will be active during each computation step. However, this approach generally entails multiple partitions for each layer, leading to the redundant inter-processor communications mentioned above. Thus, increasing processor utilization, increases the amount of communications.

There is an alternate approach to block parallelism which avoids this trade-

off between processor utilization and extra communication. The previous approach assumes that only a single example is processed by the network at any time. However, by continuously pipelining data through the network, full processor utilization can be ensured without having to increase the number of partitions for each layer. With pipelining, there is an initial cost associated with feeding an input through the network. Inversely, at the completion of an epoch, there is also the cost of emptying the network of the error gradients of the last example. Thus, pipelining incurs the additional cost of filling and emptying the pipeline; this cost is equivalent to propagating a single example forward through the network and propagating its error gradient backwards again.

One constraint of pipelining is that, as with exemplar parallelism, deterministic learning must be used. If stochastic learning is employed, then by the time an error signal is propagated back to a particular processor, its weights may have already changed. Thus, to ensure that the network is updated consistently, it is essential that the weights should not change between the computation of the error gradient and the updating of the weights in that layer. An additional constraint of pipelining is that every neuron output must be retained in processor memory until the calculated error gradient is received to perform the weight update. If the pipeline is quite long, a potentially large number of waiting neuron outputs must be stored.

3.1.3 Neuron Parallelism

The basic unit of modularity in a neural network is the neuron. Every neuron operates independently, processing the input that it receives, adjusting its weights, and propagating its computed output. Thus, the neuron is a natural level of parallelization for neural networks. For neuron parallelism, every neuron is treated as a parallel process. Each process is distributed among the processors of the parallel computer. Unlike block parallelism, neuron parallelism make no attempt to exploit the locality between neurons in the network. It can be assumed that neurons are randomly allocated to each processor.

Figure 8 shows a simple three-layer network of three neurons per layer, where each neuron is treated as a separate parallel process. Each of these processes may be stored in a single processor; however, for general-purpose parallel computers, it is likely that the number of neurons will exceed the number of processors, and thus, each processor will be assigned many neurons. The flow of computation is similar to block parallelism: processors containing neurons from the first network layer will begin by processing the input,

and then will propagate their output to processors maintaining neurons from subsequent layers. However, unlike block parallelism, there is no geometric relationship between the neurons in each processor. Instead, processors operate on a data-driven basis, such that they are active when they receive information for processing.

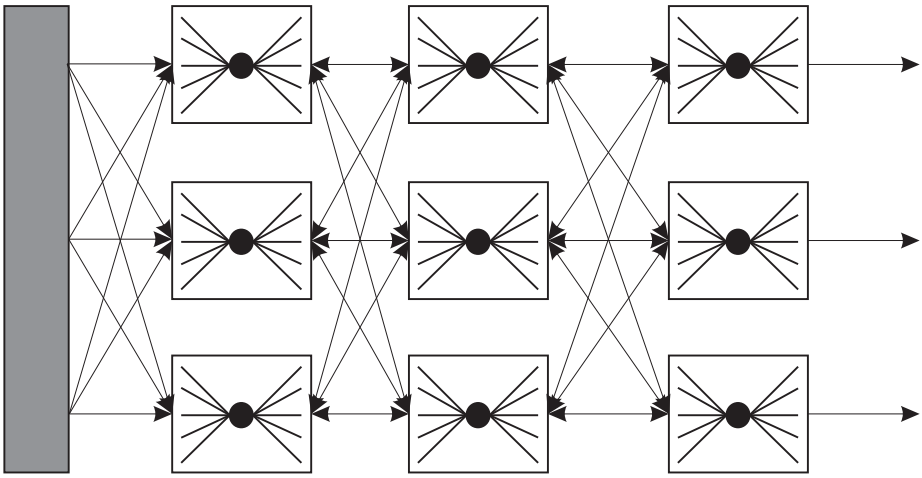

Figure 8: Partitioning of a neural network using neuron parallelism.

Like block parallelism, pipelining is used to ensure full processor utilization. The costs and constraints of pipelining are the same as for block parallelism.

3.2 Neural Network Cost Analysis using BSP

The three parallelization strategies described in the previous subsection specify different ways of distributing the task of neural computation among the processors of a parallel computer. However, none of these strategies alter the actual computations involved. Therefore, the sequential cost of executing the neural network algorithm can be used as a basis for assessing the parallel cost. From the equation of the BSP cost model (42), it is clear that cost of the computation, communication and synchronization phase can each be estimated independently. Therefore, the sequential computation costs of the neural network can be estimated, and then applied to the BSP cost of each of the three network parallelization strategies.

When determining the theoretical cost of an algorithm, the accuracy of the cost estimate is clearly important. However, this estimate should not be

determined at the expense of the clarity and simplicity of the solution. Thus, it is often important to suppress tedious detail and express the algorithm in terms of its most significant components. The terms that are ignored from the cost estimate can generally be modeled by a constant which can then be calculated empirically for a given implementation.

There are two main factors which determine the computational cost for a neural network: the number of examples N in the data set $\mathcal{D}$, and the number of adjustable weights in the network W. The importance of these two factors is clearly expressed by their presence in the network function $f(\mathbf{x}, \mathbf{w})$. Because the cost of neural network computation grows as a factor of the size of the data and the number of weights in the network, it is common to express the estimated cost as a function of N and W[3].

In the following subsections, the theoretical computational costs of supervised and unsupervised neural networks will be calculated. These estimated costs will then be applied to the BSP cost model in order to compare the efficiency of exemplar, block and neuron parallelism.

3.2.1 BSP Cost Analysis of Supervised Neural Networks

The supervised neural network that will be considered in these cost analyses is the cross-entropy multi-layer perceptron. The equations used for implementing this type of neural network are given in Subsection 2.3.2. From these equations, it is possible to obtain the theoretical cost of this supervised neural network. It is important to note, however, that the equations for the cross-entropy multi-layer perceptron are simply used to guide the process of cost analysis; these analyses will be quite general and are applicable to a wide range of supervised neural network algorithms.

For simplicity and generality, the supervised network considered in these analyses will be assumed to consist of L layers with M neurons per layer, where each layer is fully-connected to the neurons in the preceding and succeeding layers. The total number of neurons V is, therefore, LM, and the total number of weights is $W = LM^2$. Although it is unlikely that an actual neural network

[3]As an aside, the number of neurons V in a network also makes a considerable contribution to the computational cost of the network. However, because neurons in a neural network are necessarily interconnected, the number of weights in the network W will increase directly in relation to the number of neurons. On average, the rate of growth of the number of weights is on the order of the square of the number of neurons: $W = \Theta(V^2)$. Therefore, the number of operations for each neuron can be ignored as a lower order term of the number of weights.

will have the same number of neurons in each layer, this assumption will greatly simplify the cost analysis of neural networks using block parallelism. In addition, the rectangular structure of this hypothetical network is a superset of all other network topologies; thus, this structure generalizes any specific network connectivity. It will also be assumed that the data set $\mathcal{D}$ consists of N examples.

To begin, it is necessary to decompose the cost of training a supervised neural network into a series of analyzable steps. On the most abstract level, training consists of iterating a neural network through a data set for a number of epochs. As long as the size of the data set and the structure of the network remain fixed during the course of training, the cost of training for each epoch is constant. Thus, the cost of training can be specified by the product of the number of epochs and the cost of network computation for each epoch. The next section will examine the number of epochs required for network training; this section will strictly focus on the cost of network computation for each epoch.

The computations performed by the neural network during each epoch can be decomposed into three distinct phases:

1. the evaluation of the network function for each input example
2. the calculation of the error gradient with respect to each weight in the network
3. the update of every weight in the network

Each of these phases are executed sequentially; so the cost of training the network for each epoch is the sum of each of these phases. The first two phases are executed for every example in the data set. The third phase will be executed only once for each epoch because all three parallelization strategies are deterministic methods.

The cost of the network function can be determined by considering the processing performed by each neuron. From (10), the output of every neuron is determined by the weighted sum of its inputs followed by the application of a nonlinear activation function (such as the sigmoid function (2.3.2)).

$$y_i(\mathbf{x}_n) = \varphi_i(\sum_j w_{ij}x_j + w_{i0}) \tag{43}$$

$$\text{cost}(\varphi_i(\sum_j w_{ij}x_j + w_{i0})) = a_1 + 2M \tag{44}$$

By decomposing the weighted sum into a multiplication and an addition operation, 2 flops are required for each weight. Every neuron in the network has M interconnection weights to the neuron in the previous layer. Therefore, the total cost of the weighted sum is $2M$. For simplicity, it will be assumed that the cost of computing the activation function can be expressed by the constant a_1 for each neuron. By generalizing the computations of each neuron to the entire network, the cost of evaluating the network function can be specified as:

$$\begin{aligned} \mathbf{y}(\mathbf{x}_n) &= f(\mathbf{x}_n, \mathbf{w}) && (45) \\ \text{cost}(f(\mathbf{x}_n, \mathbf{w})) &= (a_1 + 2M)ML && (46) \\ &= 2W + a_1 ML && (47) \\ &= a_F W, && (48) \end{aligned}$$

using the equality $W = LM^2$. The term $2a_1ML$ can be dropped from the cost equation because it is a lower-order term of the number of weights W. The order of growth of this equation ($\Theta(W)$) is an asymptotically tight bound because the product of the number of weights and some constant a_F can be used to delimit the range of possible costs.

The process of calculating the error gradient is slightly different for neurons in the hidden layer than for neurons in the output layer. Thus, the cost for neurons in each layer should be examined individually. Using (19), the cost for output layer neurons can be determined:

$$\begin{aligned} \frac{\partial J_n}{\partial w_{jk}} &= -y_{j,n}(t_{k,n} - y_{k,n}) && (49) \\ \text{cost}(-y_{j,n}(t_{k,n} - y_{k,n})) &= M + a_3 && (50) \end{aligned}$$

The constant a_3 is used to specify the cost of evaluating the error function for each output. For the cross-entropy error function, this computation involves a single subtraction. Other error functions may be more complex. Next, this computed error is multiplied by the outputs from each of the M neurons in the previous layer $y_{j,n}$ to get the gradient with respect to each weight.

The computation of the hidden layer gradients is only slightly more complicated and increases the cost by a constant factor (22). Because the hidden layer cannot calculate its error directly from the target class labels, it is necessary for the output layer errors to be propagated backwards to the hidden layer. Thus, this step involves the weighted sum of the M errors from the output layer. The constant term a_4 represents the cost of evaluating the inverse of the hidden layer activation function (in this case it is a sigmoid). The

computed error for each hidden layer neuron is then multiplied by each of its M inputs to calculate the error with respect to each weight.

$$\frac{\partial J_n}{\partial w_{ij}} = -x_i y_j (1 - y_j) \sum_{k=1}^{C} \{(t_{k,n} - y_{k,n}) w_{jk}\} \quad (51)$$

$$\text{cost}(-x_i y_j (1 - y_j) \sum_{k=1}^{C} \{(t_{k,n} - y_{k,n}) w_{jk}\}) = M + a_4 + a_5 M \quad (52)$$

By collating the cost of these two equations for every neuron in the hidden and output layers, the total cost of calculating the error gradient can be expressed:

$$\text{cost}(\frac{\partial J_n}{\partial w_{ij}}) = M(L-1)(M + a_4 + a_5 M) + M(M + a_3) \quad (53)$$

$$= W(1 + a_5) + M(a_4 L - a_4 - a_5 M + a_3) \quad (54)$$

$$= a_B W \quad (55)$$

As with the cost of network function, the order of growth of this equation is $\Theta(W)$. a_B is some constant that can be used to asymptotically bound the actual execution cost of this operation.

The final cost to be computed is the cost of updating each network weight. From (23), new values for each weight can be computed from a single addition:

$$w_{ij} = w_{ij} + \Delta wij \quad (56)$$

$$\text{cost}(w_{ij} + \Delta wij) = W \quad (57)$$

Thus, this cost is M for each neuron, and W for the entire network.

Now that the the cost estimates of these three phases have been calculated, they can be combined to determine the total cost of network computation for each epoch. Therefore, by calculating (48) and (55) for each example in the data set, and then updating network weights (57), the cost of computation for each epoch is:

$$C_E = (a_F W + a_B W)N + W = ANW \quad (58)$$

The constant A represents the average cost required to perform all of the computations necessary to update a single weight in the network for each example. Because each weight is only updated once for each epoch, the constant A can be expressed, instead, as the cost of evaluating the network function and computing the gradient for each weight in the network for each example.

It is straightforward to determine A empirically from the execution time of a particular neural network implementation. As a lower bound, $A \geq 5$ can be estimated from the cost equations for the network function (48) and the gradient calculation (55). It can be observed that the only parameter expressing the network topology in the cost equation (58) is the number of weights W in the network. Thus, this cost equation is independent of the rectangular network assumption posited above.

Exemplar Parallelism Cost Analysis

For exemplar parallelism, the task of network training is reduced by distributing the data set $\mathcal{D}$ among several processors so that each processor is only responsible for performing computations on its local subset of the data. Assuming that every processor has equal capabilities, each processor should receive $\frac{N}{p}$ examples to compute, where p is the total number of processors in the parallel system.

From the perspective of BSP, neural network training using exemplar parallelism can be divided into two supersteps. The first superstep computes the weight adjustments for each processor and distributes them globally. The second superstep combines the adjustments received from each processor and updates the network weights[4]. The complete cost of BSP training can be computed by multiplying the sum of both supersteps by the number of network training epochs. It can be assumed that the startup costs of distributing the examples to each processor is negligible in comparison with the cost of network training.

The computational cost of the first superstep is equivalent to the cost of evaluating the network function and computing the error gradient for each example in the local data set. The cost of these two operations is expressed in (48) and (55). By combining these costs computed for each example in the local data set, the computation cost of the first superstep is $\frac{N}{p}(a_F W + a_B W)$.

Once each processor has derived the error gradients using its local data set, these results are communicated globally. Due to the fact that every weight in the network is to be updated by the gradient calculations, the size of the message to be communicated is W. Every processor must broadcast its message to each of the $(p-1)$ other processors in the parallel computer. Thus, the total cost of the communication phase of this superstep is $(p-1)Wg$ (tak-

[4]Technically, this process can be reduced to a single superstep because there is no need for every processor to synchronize after the weight updates. However, for clarity, two separate supersteps will be stipulated; the additional BSP cost incurred by this approach is minimal

ing the BSP communication parameter g into account for a particular parallel machine).

The second superstep involves combining the gradient estimates received from each processor to update the network weights. Each processor receives W gradient values from each processor; thus, the computational cost of combining these estimates is simply $(p-1)W$. No communications need to be performed during this superstep. It is important to note that at the end of this superstep, the neural network stored in each processor will be equivalent.

The total cost of exemplar parallelism for each epoch can be computed using the BSP cost function (42):

$$C_{EP} = \left\lceil \frac{ANW}{p} + (p-1)W \right\rceil + [(p-1)W]\, g + 2l \tag{59}$$

The cost of the first superstep is simplified by using (58).

Block Parallelism Cost Analysis

For block parallelism, the neural network is partitioned into non-overlapping rectangular blocks of neurons which are distributed among the processors of the parallel computer. The initial assumption that the neural network has a rectangular structure will facilitate dividing the network into equally-sized blocks of neurons. It will be assumed that each block has depth x and width y. The depth of the block x corresponds to the number of layers which are spanned by the block. The width of the block y corresponds to the number of neurons in each layer which are part of the same block. Because every block is non-overlapping, equally-sized, and partitions the entire network, it can be assumed that $xyp = LM$ (where LM is the total number of neurons in the network).

The division of a block parallelism neural network into supersteps is somewhat complicated. The propagation of information through a neural network proceeds on a layer-by-layer basis. Thus, a neuron cannot complete its processing until it has received the output from each of the neurons in the connected layer. Consider a block partitioning of a neural network where each processor is assigned only neurons from the same layer ($x = 1$). These processors can be arranged in columns, such that each column contains neurons from the same layer. These processor columns pass information horizontally to processors in adjacent columns. If pipelining is not used, only a single column of processors will be active at any given time. Thus, it would require $\frac{L}{x}$ steps to propagate an input through the network, and the same number of steps to propagate the error gradient back again. With pipelining, information can be continuously

streamed through the network so that each processor is always utilized. Thus, a data set of N examples can be evaluated in $(N-1)$ steps plus the time required to fill and empty the pipeline. The total number of steps required to process a data set of N exemplars is $(N-1)+2\frac{L}{x}$. Each of these steps will be called a *big superstep*.

A big superstep consists of the computation and communication performed by each processor in a column. The cost of the computation phase is simply the forward and backward propagation costs for each of the xy neurons in block. Thus, the cost of computation is $NAxy$, which can be rearranged using the equality $xyp = W$ to equal $\frac{WA}{p}$. The cost of communication is equivalent to sending the outputs of the first and last layer in the block to the processors containing adjacent layers. Thus, $2y$ values are sent to the $\frac{M}{y}$ processors in the two adjacent columns. The cost of the communication phase for each big superstep is therefore $2y\frac{M}{y}$.

However, if each block spans multiple network layers ($x > 1$), then the communication cost of each big superstep becomes more complicated. In this case, additional communication is needed between the processors in the same column containing neurons in each of the x layers. The reason for the additional communication is that each of the x layers in a block requires the complete set of inputs from an adjacent layer in order to compute its output. Because each processor only contains y neurons from each layer, the other $M-y$ values are required to complete the computation for each neuron. This additional set of communications between processors in the same column will be called a *small superstep*. Because there are x layers in each block, $(x-1)$ small supersteps will be required for each big superstep. The communication cost of each small superstep is therefore $(M-y)(x-1)$. It is important to recognize that every process must be synchronized after each small superstep. The number of small supersteps does not, however, change the amount of computation performed by the neural network.

By combining the number of big and small supersteps with the costs of the computation, communication and synchronization phases, the complete BSP cost for block parallelism can be determined:

$$C_{BP} = (N-1) + 2(\frac{L}{x} - 1)\left[\frac{AW}{p} + (2M + (M-y)(x-1))g + xl\right] \qquad (60)$$

Clearly, block parallelism could be made more efficient by reducing the number of small supersteps. If the width of each block y is set to M and the depth x is set to 1 so that each processor is assigned an entire layer, the

number of small supersteps can be eliminated. This version of block parallelism is known as layer parallelism (mentioned previously in Subsection 3.1.2). By making the assumption about the dimension of each block that ($x = 1$ and $y = M$), the BSP block parallelism cost equation can be reduced to:

$$C_{BP} = (N-1) + 2(L-1)\left[\frac{AW}{p} + 2Mg + l\right] \tag{61}$$

Neuron Parallelism Cost Analysis

For neuron parallelism, there is no attempt to exploit locality in the network topology. The work of neural network training is reduced by randomly distributing the neurons of the network among the processors in a parallel computer. Due to the modularity of the network structure, the cost of the computations and communications performed by each neuron are equivalent. Assuming that each processor in the neural network is equally powerful, an equal number of neurons ($\frac{ML}{p}$) can be assigned to each processor.

Because there is no attempt to exploit locality, neuron parallelism represents an extreme form of block parallelism where each block contains a single neuron ($x = 1$ and $y = 1$). Without locality, there are no big supersteps to be performed. Thus, each superstep is equivalent to one small superstep. Each neuron must send and receive activation from every neuron in the adjacent layers. In order to ensure full processor utilization, pipelining is used. The cost of pipelining from block parallelism can be used by setting $x = 1$. Therefore, neuron parallelism performs $(N-1) + 2(L-1)$ supersteps per epoch.

Because each neuron operates independently and performs the same computational operations, the computation cost of neuron parallelism can be derived from (58) for each example. This cost is distributed among each of the processors in the parallel machine so that each processor performs $\frac{ML}{p}^{th}$ of the total computations. Therefore, the total cost of the local computation phase for each superstep is $\frac{AW}{p}$. It can be observed that this cost is identical to the cost of the computation phase for block parallelism because both strategies assume that each processor receives an equal proportion of the number of neurons in the network.

The cost of the communication phase is more expensive for neuron parallelism than for block parallelism due to the absence of locality. Because of the random allocation of neurons to processors, the parallel implementation can make no assumptions about which neurons reside on which processors. In the worst case, each layer of neurons may be partitioned among every processor in the parallel machine. Thus, for a neuron to transmit its output to an ad-

jacent layer may require communicating with all p processors. Assuming that each neuron must propagate its output and computed error gradients to M neurons in the preceding and succeeding layers, the cost of communication for each neuron is $2M$. Because this cost is shared by all of the $\frac{ML}{p}$ neurons in each processor, the total communication cost for each superstep is $\frac{2M^2L}{p}$.

The costs of the computation and communication phases can be combined with the number of supersteps to yield the BSP cost for neuron parallelism:

$$C_{NP} = (N-1) + 2(L-1)\left[\frac{AW}{p} + \frac{2WL}{p}g + l\right] \tag{62}$$

The cost of the communication is simplified by using $W = LM^2$.

Comparison of Network Parallelization Strategies

Now that the BSP cost equations have been determined for each of the three network parallelization strategies, the cost of their parallel implementations can be compared. The actual execution times can be predicted using the BSP parameters g and l for a variety of parallel computers (such as those given in Table 1).

To facilitate the comparison of these methods of parallelization, it is helpful to consider certain practical guidelines and limitations. For example, because processors are a finite and expensive resource, the number of processors is likely to be small relative to the size of the network. As a limit, the largest number of processors that can be considered using these parallelization strategies is the number of neurons in the network ($p \leq ML$). In this case, block and neuron parallelism are equivalent because every processor is assigned a single neuron. An additional constraint is the size of the network in relation to the number of data examples. As a rule of thumb, the size of the data set should be proportionally greater than the number of weights by some constant factor ϵ [22, p. 179].

$$N \gg \frac{W}{\epsilon} \tag{63}$$

The factor ϵ is inversely proportional to desired accuracy of the network. Therefore, the number of examples N is likely to be considerably larger than the number of weights for network training applications. This is especially true for data mining applications.

Using these practical constraints, there are several ways that the BSP cost equations for each method of parallelization can be simplified. Due to the fact that the size of the data set is likely to be quite large, the cost of filling and empty the pipeline for block and neuron parallelism is generally insignificant.

	EP	LP	NP
total computation	$\frac{N}{p}AW$	$N\frac{AW}{p}$	$N\frac{AW}{p}$
total communication	$W(p-1)g$	$N2Mg$	$N\frac{2W}{p}g$
total synchronization	$2l$	Nl	Nl

Table 2: Parallel implementation costs for supervised neural networks.

Therefore, the number of supersteps for these two methods is roughly N. By making this simplification, it can be observed that the cost of computation for block and neuron parallelism are equivalent. Similarly, by ignoring the computation term for the second superstep in the exemplar parallelism cost equation, the cost of computation becomes equal for each parallelization strategy. The results of these simplifications are intuitively appealing because the process of parallelization does not change the actual neural network algorithm, it simply distributes the work of computation among several processors. For these cost comparisons, layer parallelism will be considered instead of general block parallelism because layer parallelism is the most efficient method of partitioning a neural network into blocks (as long as the number of processors is less than the number of layers). The results of these simplifications are summarized in Table 2.

It is clear that the main distinctions between the three parallelization strategies is the amount of communication required and the number of barrier synchronizations performed. From the simplified equations in the table, the superiority of layer parallelism over neuron parallelism becomes apparent. Both techniques require an equal number of barrier synchronizations. However, neuron parallelism requires ML more communications per epoch. This reduction in communication overhead is created by the locality exploited by layer parallelism. Comparing layer and exemplar parallelism, it is not so ap-

parent as to which technique is superior. Layer parallelism definitely requires more barrier synchronizations. From the BSP cost equations it can be observed that exemplar parallelism performs less communication than layer parallelism if $N > \frac{1}{2}ML(p-1)$. By applying the rule of thumb 63 that the number of examples generally exceeds the number of weights in a neural network by a constant factor, it can be inferred that the cost of communication is more expensive for layer parallelism. This is especially true for data mining applications where N is huge. Therefore, from this reductionist analysis, exemplar parallelism is the best technique for parallelizing supervised neural networks.

It is helpful to generate some predictions of training costs for real parallel computers in order to demonstrate the superiority of exemplar parallelism. Figure 9 shows the predicted parallel implementation costs of a supervised neural network using the three methods of parallelization. These predictions are generated for data sets with a varying number of examples. It is assumed that the neural network has 16 layers of 32 neurons and is implemented on a 16 processor Cray T3E parallel computer with $g = 1.7$ and $l = 751$. The neural computation constant A is assumed to be 6. For the purpose of comparison, an unrealistically large number of layers is used in this example. Even in this case, exemplar parallelism is better than layer parallelism for modest numbers of examples. Figure 10 compares the communication costs of the three parallelization strategies. From this figure it is clear that even for a very small number of examples (much smaller than the required number of examples specified by the rule of thumb) and a very large network, the cost of communication for layer parallelism quickly outgrows the communication required by exemplar parallelism.

3.2.2 BSP Cost Analysis of Unsupervised Neural Networks

The BSP cost analysis of unsupervised neural networks will proceed in a similar fashion to the previous subsection: the sequential computational cost of the unsupervised network will be calculated, and then this cost will be applied to the strategies for network parallelization. The BSP cost of each strategy will be compared to show which method is best suited to parallelizing unsupervised neural networks.

The unsupervised neural network that will be considered in this cost analysis is the Gaussian mixture model. The equations for this type of unsupervised neural network are given in Subsection 2.3.3 and will be used to guide the cost analysis. Although a specific type of unsupervised neural network will be analyzed, the results of this cost analysis are sufficiently general that they can

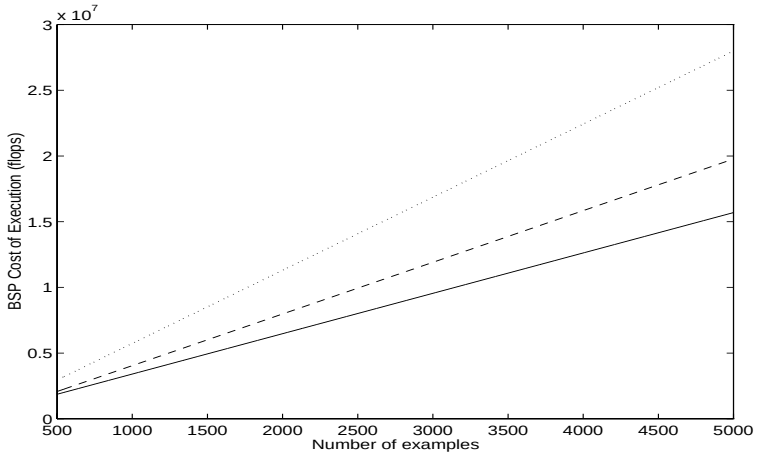

Figure 9: Comparison of the predicted parallel execution costs of exemplar, layer and neuron parallelism using a supervised neural network. The solid line represents the cost of exemplar parallelism, the dashed line represents the cost of layer parallelism, and the dotted line represents the cost of neuron parallelism.

be applied to a wide range of unsupervised neural network algorithms.

It is important to note that the mixture model that has been presented in this report consists of a single layer of components. Because there is no locality that can be exploited in a single-layer network, block parallelism will not be considered as an option for parallelizing unsupervised neural networks.

Once again, for simplicity and generality, it will be assumed the network has a rectangular topology. The mixture model will consist of M components, each connected to M inputs. There are three types of weights in a mixture model corresponding to the mean, variance, and mixture proportions of each component distribution i: $\mathbf{w}_i = \{\boldsymbol{\mu}_i, \sigma_i^2, \pi_i\}$. Every component mean $\boldsymbol{\mu}_i$ has dimension M because it positions the mixture distribution in relation to the M-dimensional input vectors. In continuity with the description of the Gaussian mixture model presented in Section 2, it will be assumed that the variance for each component is specified by a single parameter rather than by the full covariance matrix. The mixture proportion is also specified by a single parameter value for each component. Thus, each component has $M+2$ weights. Considering all of the parameters for each component, the total number of weights in the network is $W = M^2 + 2M$.

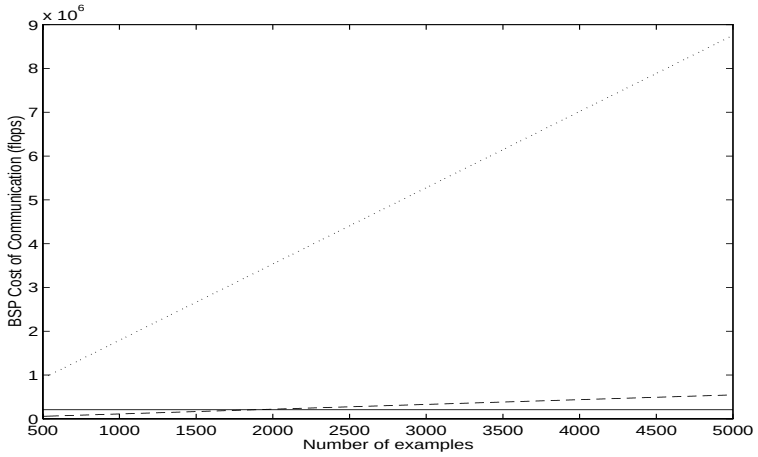

Figure 10: Comparison of the predicted communication costs of exemplar, layer and neuron parallelism using a supervised neural network. The solid line represents the cost of exemplar parallelism, the dashed line represents the cost of layer parallelism, and the dotted line represents the cost of neuron parallelism.

The presence of different types of weights in the network will complicate the cost analysis somewhat. For supervised networks, the cost of network operations could be expressed as a product of the number of network weights because every weight operation could be treated equivalently. For the cost analysis of unsupervised networks, if an operation incurs a cost that is bounded by $\Theta(M^2)$, then it will be assumed the cost can be expressed as AW, where A is some constant factor[5].

To proceed with the cost analysis of the Gaussian mixture model, it is necessary to divide the process of training into a series of analyzable steps. As stated in the previous subsection, on the most abstract level, training consists of a number of epochs. Because the operations performed by the neural network remain unchanged for each epoch, it can be assumed that the cost of each epoch is constant. By further decomposing the training process, each epoch consists of iterating the neural network through every example in a data set. Again, because the neural network performs the same operations on

[5]It is important to note that this constant A will be different for an unsupervised neural network than for a supervised neural network. The same variable is used to show similarity between the parallel implementations for both types of networks.

each example, the cost of computation for each example is fixed. These cost analyses will trace the computations for a single example $\mathbf{x}_n$ as it is processed by the neural network.

For a Gaussian mixture model, there are four main stages to processing each input example:

1. calculating the likelihood for each example by evaluating the network function,
2. determining the posterior probabilities for each example by normalizing the likelihoods,
3. deriving the gradient of the posterior probability for each parameter, and
4. updating each parameter to maximize the *a posteriori* probability of the network.

The cost of each of these stages is uniform for each mixture component in the model. To simplify this analysis and to facilitate the parallel cost analysis, each component will be examined individually. It is straightforward to compute the cost for the entire network by summing the cost of each of these stages and multiplying by the number of components M in the network.

A mixture model begins processing an example by evaluating the network function. Considering that the components of the model are Gaussian distributions, the normal probability density function is used to compute each likelihood. From (27), the cost of determining the likelihood for each component can be analyzed:

$$\begin{aligned} y_i(\mathbf{x}_n) &= P(\mathbf{x}_n|i,\mathbf{w}_i) && (64)\\ &= \frac{1}{(2\pi\sigma_i^2)^{d/2}}\exp\left\{-\frac{\|\mathbf{x}_n-\boldsymbol{\mu}_i\|^2}{2\sigma_i^2}\right\} && (65)\\ \text{cost}(y_i(\mathbf{x}_n)) &= a_1 + a_2 + \{a_3 M + a_4\} && (66)\\ &= a_F M && (67) \end{aligned}$$

where a_i is some constant cost incurred by performing operation i. For this analysis, it will be assumed that the cost of an exponentiation function is constant if its arguments are constant. Because the variance of the component distribution is a single parameter, the normalizing constant (the first term in the equation) has constant cost a_1. Similarly, the denominator of the exponential ($2\sigma_i^2$) has constant cost a_4. The cost of performing the Euclidean distance

calculation ($\|\mathbf{x_n} - \boldsymbol{\mu}_i\|^2$) has cost $a_3 M$ because the component mean and the input example are both M-dimensional vectors (where a_3 is the cost of each distance calculation). Because the output of the Euclidean distance calculation is a single value, the cost of the exponential is constant. By bounding the growth of (67), the cost of calculating the likelihood for each component can be expressed as the product of M and a constant a_F.

Once the network function has been evaluated for an input example, the posterior probability can be calculated by normalizing the output probabilities of each component. The computation of the component-conditional probabilities is given in (32). Using this equation, its cost can be determined:

$$p(\mathbf{w}_i, i|\mathbf{x}_n) = \frac{p(\mathbf{x}_n|i, \mathbf{w}_i)\pi_i}{\sum_{j=1}^{M} p(\mathbf{x}_n|j, \mathbf{w}_j)\pi_i} \tag{68}$$

$$\text{cost}(p(\mathbf{w}_i, i|\mathbf{x}_n)) = a_5 + a_5 M \tag{69}$$

$$= a_N M \tag{70}$$

The posterior probability $p(\mathbf{x}_n|i, \mathbf{w}_i)$ for component i is normalized by dividing its value by the sum of the posterior probabilities for every component in the network. The numerator performs a single multiply, and thus has a constant cost a_5. The sum on the denominator combines the posterior probabilities for each of the M components. Therefore, this sum incurs cost $a_5 M$.

Having determined the posterior probabilities, the gradient calculation for each component parameter can be computed. The calculation of the gradient is different for the mean, variance and prior probability of the distribution. Thus, the cost of each of these calculations will be examined individually. The equations that will be used for these gradient calculations are expressed in (33), (34) and (35). It is important to note that these equations calculate the gradient by performing a weighted sum of the posterior probabilities for every input example in the data set. It is straightforward to re-express these equations by using a moving average to combine each posterior probability individually [40]. This moving average equation will not be given here; however, the important consideration is that the gradient can be evaluated for each input example in turn. Therefore, the cost of this calculation can be expressed for each input example.

The cost of the gradient calculation for $\boldsymbol{\mu}_i$ is:

$$\hat{\boldsymbol{\mu}}_i = \frac{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)\mathbf{x}_n}{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)} \tag{71}$$

$$\text{cost}(\hat{\boldsymbol{\mu}}_i) = N(a_6 M) + NM \tag{72}$$

$$= \quad N(a_6 M + M) \tag{73}$$

Because this cost is equal for each input example, the cost of evaluating the gradient for a single example is $(a_6 + 1)M$. Intuitively, the cost of calculating the gradient requires a constant number of operations for each of the M weights of the parameter $\boldsymbol{\mu}_i$.

The cost of the gradient calculation for σ_i^2 is:

$$\hat{\sigma}_i^2 = \frac{1}{d} \frac{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n) \|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_i\|^2)}{\sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n)} \tag{74}$$

$$\text{cost}(\hat{\sigma}_i^2) = N(a_7) + N \tag{75}$$

$$= N(1 + a_7) \tag{76}$$

For this computation to have constant cost for each input example, it is assumed that the Euclidean distance calculation ($\|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_i\|^2$) has been retained from the computation of the component probabilities (65).

The cost of the gradient calculation for the mixture prior probabilities is:

$$\hat{\pi}_i = \frac{1}{N} \sum_{n=1}^{N} p(\mathbf{w}_i, i|\mathbf{x}_n) \tag{77}$$

$$\text{cost}(\hat{\pi}_i) = N(a_8) \tag{78}$$

It is clear from this equation that the cost for each input example is constant.

The cost of the gradient calculation for each type of parameter can be combined to express the total cost of this stage:

$$a_6 M + M + a_7 + a_8 = a_B M, \tag{79}$$

for some constant cost a_B.

The final stage to be analyzed is the update calculation for each weight in the network. The equations from (36), (37), and (38) detail this updating procedure. The update of each parameter may be performed immediately (as is the case for stochastic learning) or may be deferred for several examples (as in deterministic or batch learning). As would be expected, the cost of updating each parameter is simply some constant times the number of weights to be adjusted. Thus, the cost for the mean of the component distribution is $\Theta(M)$, and the cost for the other two parameters is $\Theta(1)$.

$$\boldsymbol{\mu}_i = \boldsymbol{\mu}_i + \eta(\hat{\boldsymbol{\mu}}_i - \boldsymbol{\mu}_i) \tag{80}$$

$$\begin{aligned}
\text{cost}(\boldsymbol{\mu}_i + \eta(\hat{\boldsymbol{\mu}}_i - \boldsymbol{\mu}_i)) &= a_9 M & (81)\\
\sigma_i^2 &= \sigma_i^2 + \eta(\hat{\sigma}_i^2 - \sigma_i^2) & (82)\\
\text{cost}(\sigma_i^2 + \eta(\hat{\sigma}_i^2 - \sigma_i^2)) &= a_{10} & (83)\\
\pi_i &= \pi_i + \eta(\hat{\pi}_i - \pi_i) & (84)\\
\text{cost}(\pi_i + \eta(\hat{\pi}_i - \pi_i)) &= a_{11} & (85)
\end{aligned}$$

The total cost of the weight updating procedure can be determined by combining the individual costs for updating each type of parameter.

$$a_9 M + a_{10} + a11 = a_U M, \tag{86}$$

for some constant a_U.

The cost of each of the stages that have been analyzed are given in terms of a single component. Because the operations performed by each component is identical, this cost can be generalized to the cost of the entire network by multiplying by the number of components M. The cost of each of these stages can be collated by totaling the computation cost for an entire epoch. It will be assumed that the parallel neural networks to be analyzed have been implemented using deterministic learning. Therefore, they only perform a single weight update per epoch. This is why the cost of weight updating is not multiplied by the number of examples in the data set for this cost equation. The total cost of executing a Gaussian mixture model for each epoch is:

$$N(a_F M^2 + a_N M^2 + a_B M^2) + a_U M^2 = N\Theta(M^2) \equiv NAW \tag{87}$$

Considering that the total cost of each phase of computation is determined by the product of M^2 and some constant, its order of growth is bound by $\Theta(M^2)$. In order to express the total cost of computation in terms of the number of weights W, it has been assumed that $\Theta(M^2)$ can be expressed as AW, for some constant A. A represents the cost of the computations performed by each weight in the mixture model network.

Exemplar Parallelism Cost Analysis

Exemplar parallelism exploits the presence of a large number of data examples in the data set as the source of parallelism. Thus, the cost of training a neural network is reduced by decomposing the data set into several subsets, and distributing these subsets among the processors in a parallel computer. Exemplar parallelism does not attempt to partition the network, itself, among the processors – each processor trains a complete network. Therefore, the process of developing a parallel implementation using exemplar parallelism is

independent of the specific neural network algorithm being used. The only constraint is that the neural network algorithm must be capable of deterministic or batch learning.

In virtue of the algorithm independence of exemplar parallelism, the analysis of a parallel unsupervised neural network using this method will be the same as for a supervised network. Thus, the superstep structure, and the communication and synchronization requirements will be equivalent for a parallel unsupervised network implementation. By using this information and simply substituting the computation costs for the unsupervised neural network into the BSP cost equation, the cost of a parallel unsupervised network can be computed.

The results from the supervised exemplar parallelism cost analysis will be summarized briefly. For exemplar parallelism, only two supersteps are needed: the first evaluates the network function and computes the gradient for each example in the processor's local subset, and the second updates the weights of the network using the calculated gradient information.

From (87), the total cost required to update every network weight for each example is AW. Each processor operates on its local subset of $\frac{N}{p}$ examples. Therefore the cost of the computation for the first superstep is $\frac{N}{p}AW$. It can be noted, that this cost is identical to the computation cost of a supervised network during the first superstep (only with different values of W and A).

At the end of the first superstep, each processor broadcasts its calculated weight adjustment to every other processor. Thus, the communication volume handled by each processor is $(p-1)W$. Upon receiving this weight adjustment information, each processor must merge these updates. This merging operation requires constant cost for each weight adjustment, and thus, incurs the cost $(p-1)W$. Once these values have been merged, the weights of the neural network local to each processor can be updated.

Totaling each of these costs using the BSP cost equation yields:

$$C_{EP} = \left[\frac{ANW}{p} + (p-1)W\right] + [(p-1)W]\,g + 2l \tag{88}$$

which is equivalent to the cost of exemplar parallelism for supervised neural networks (59).

Neuron Parallelism Cost Analysis

For neuron parallelism, the neurons (components) of the unsupervised neural network are randomly distributed among the processors of a parallel computer. Assuming that the neurons are distributed equally, each processor is

assigned $\frac{M}{p}$ neurons.

Each neuron operates independently, processing the information that it receives through its interconnections. Because it is assumed that the mixture model consists of a single layer of components, there is no concern of propagating information through the network. Each neuron operates directly on the input vectors. Because each processor receives an equal number of neurons, the cost of computation can be shared among each of the processors. Thus, the cost of computation is $\frac{AW}{p}$.

The need for inter-neuron communication is created by the calculation of the posterior probabilities. This probability is determined by normalizing the component probabilities for each example (32). To perform this normalization operation, the output probability from every component is needed. Because different components reside on different processors, the component probabilities must be exchanged between each processor. Thus, each processor must broadcast $\frac{M}{p}$ probabilities to the other $(p-1)$ processors. The total volume of communication is $\frac{M}{p}(p-1)$. Once component probabilities have been exchanged and the posterior probabilities have been calculated, the gradient and weight adjustments can be evaluated for each component.

Thus, the task of training can be divided into two supersteps for each example: the first is to calculate and exchange the posterior probabilities, and the second is to update the parameter weights accordingly. By combining the calculated computation and communication costs for each example, the BSP cost equation can be expressed.

$$N\frac{AW}{p} + N\frac{M}{p}(p-1)g + 2Nl \tag{89}$$

Comparison of Network Parallelization Strategies

The cost equations for exemplar and neuron parallelism applied to unsupervised neural networks are illustrated in Table 3. These cost equations can be used to predict the actual execution times of parallel unsupervised neural networks for a variety of parallel computers by using the BSP cost model. Thus, the cost of parallel implementations of exemplar and neuron parallelism can be compared both theoretically and empirically using the BSP cost model.

The comparison of the cost of these two implementation strategies is facilitated by simplifying their cost equations. By eliminating the post-processing computation term in the exemplar parallelism cost equation, the computational cost of the two parallelization strategies are made equivalent. It is acceptable to ignore this term because it is insignificant in comparison with the cost of evaluating the network function.

	EP	NP
total computation	$\frac{N}{p}AW + (p-1)W$	$N\frac{AW}{p}$
total communication	$(p-1)Wg$	$N\frac{M}{p}(p-1)g$
total synchronization	$2l$	$2Nl$

Table 3: Parallel implementation costs for unsupervised neural networks.

By making the computation terms equivalent for the two parallelization methods, only their communication and synchronization costs need to be compared. The synchronization cost of neuron parallelism is clearly larger than for exemplar parallelism. The communication costs of the two strategies can be more simply compared by factoring out the terms g, $p-1$, and M (using the equality $W \approx M^2$). The remaining factors are M for exemplar parallelism and $\frac{N}{p}$ for neuron parallelism. For a clustering application, the number of data examples is necessarily much larger than the number of mixture components. Therefore, the communication cost of neuron parallelism exceeds the cost for exemplar parallelism. In summary, exemplar parallelism is a less expensive parallelization strategy than neuron parallelism because it requires less communication and synchronization.

The superiority of exemplar parallelism is illustrated using the predicted execution cost of an unsupervised neural network on an actual parallel computer. Figure 11 displays a comparison of the running times for a 32 component Gaussian mixture model network on a 16 processor Cray T3E parallel computer ($g = 1.7, l = 751$). In Figure 12, only the communication costs of these methods are compared. From these figures, the superiority of exemplar parallelism is readily apparent.

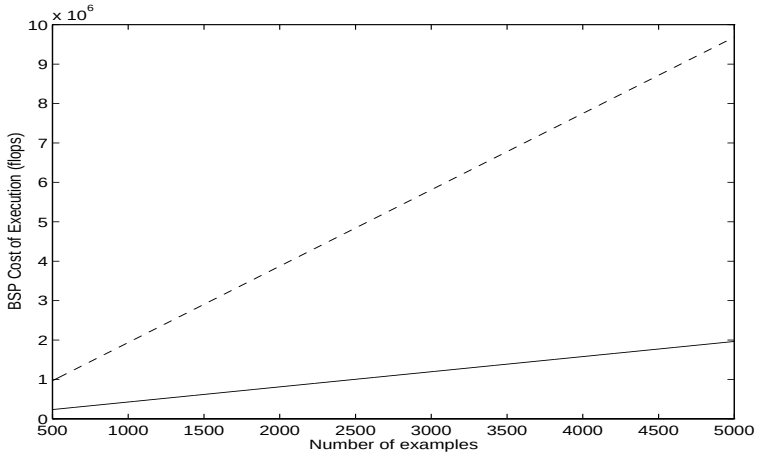

Figure 11: Comparison of the predicted parallel execution costs of exemplar and neuron parallelism using an unsupervised neural network. The solid line represents the cost of exemplar parallelism, and the dashed line represents the cost of neuron parallelism.

4 Batch Learning: In theory and practice

In the previous section, exemplar parallelism was established as the cost-minimal strategy for supervised and unsupervised neural network parallelization. The superiority of exemplar parallelism stems from the fact that it minimizes the BSP cost, and hence, the execution time of a parallel neural network for every epoch. In other words, exemplar parallelism is faster on an epoch-by-epoch basis than any of the other parallelization techniques considered.

However, this epoch-by-epoch cost analysis does not consider the speed of network convergence. If there exists an alternate parallelization strategy which is more expensive for each epoch, but has a faster speed of convergence, it may exhibit a lower overall cost than exemplar parallelism. Thus, the speed of convergence of neural networks must be examined.

One of the constraints of the exemplar parallelism strategy is that it requires deterministic updating of neural network weights. It has been noted previously in Subsection 2.3.4 that deterministic learning is generally much slower than stochastic learning schemes which perform multiple weight updates for each iteration through the data set.

Batch learning represents a compromise between the accelerated speed of

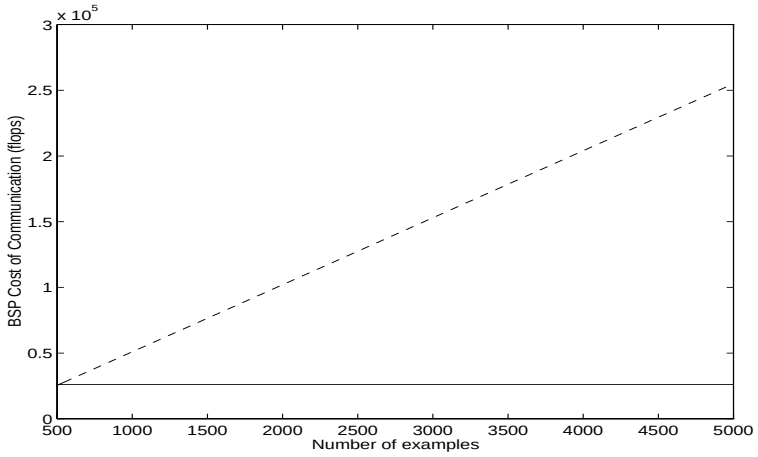

Figure 12: Comparison of the predicted communication costs of exemplar and neuron parallelism using an unsupervised neural network. The solid line represents the cost of exemplar parallelism, and the dashed line represents the cost of neuron parallelism.

convergence of stochastic learning and the postponed updating of deterministic learning. By combining exemplar parallelism with batch learning, there will be an increase in the per-epoch cost of EP; however, the increased speed of convergence will lower the overall BSP cost.

The goal of this subsection is to demonstrate the optimality of exemplar parallelism using batch learning, both in theory and in practice. This subsection begins by examining the reasons for the slow convergence speed of deterministic learning schemes. Batch learning is then established as a method which overcomes the convergence limitations of deterministic learning, and yet is still amenable to exemplar parallelism. The results of these theoretical analyses will then be verified empirically using three data mining databases.

4.1 Batch Learning in Theory

To recapitulate, the goal of an artificial neural network is to model some unknown, underlying probability distribution of the data – $p(\mathbf{t}|\mathbf{x})$ for supervised networks, and $p(\mathbf{x})$ for unsupervised networks. At every iteration, the neural network evaluates its cost function $J(\mathbf{w})$ to determine how close it is to simulating the data distribution. The network then attempts to improve its representation of the distribution by estimating the gradient $\nabla J(\mathbf{w})$ of the

cost function to update its parameters. By repeatedly re-evaluating the error gradient and updating the network weights, the network will eventually come to represent the data distribution to some designated degree of accuracy.

There are two obstacles which affect the quality of the network's ability to represent the underlying distribution. The first obstacle, which has been mentioned earlier, is the capabilities of the neural network model. If the network is insufficiently powerful (too few hidden units or mixture components) or if it is poorly suited to the application (using a Gaussian density to model a discrete distribution), then the network is unlikely to be capable of accurately modeling the data distribution. The other obstacle to training accuracy is the size of the data set. It is important to recognize that a data set is only a sample of the possible observations from an underlying data distribution. For most distributions, it is impossible to enumerate every possible value in the population; thus, a fixed-size sample must be generated. The representative quality of the sample (assuming an unbiased sampling process) is generally contingent on its size. If this sample is too small, it may be impossible for the network to gain a good representation of the data distribution and the network may be prone to overfit the sample. In general, the larger the sample, the more likely the neural network will be able to accurately model the data distribution.

Assume that it is possible to attain some estimate, $\nabla G(\mathbf{w})$, which measures the true gradient of the difference between the underlying data distribution and the network's approximation. $\nabla G(\mathbf{w})$ can be determined by evaluating the gradients for the entire population of observations that can be drawn from the data distribution. This estimate represents the best attainable weight adjustment to the network using the gradient information. When training a neural network, clearly it is desirable to have the computed error gradient $\nabla J(\mathbf{w})$ approximate the true gradient $\nabla G(\mathbf{w})$ as closely as possible. Although the true gradient cannot be calculated directly, it can generally be estimated by allowing the network to train on the largest available data set:

$$\nabla G(\mathbf{w}) = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1} \nabla J_n(\mathbf{w}) \tag{90}$$

Due to the fact that the data set is generally a fixed size, a neural network algorithm which uses every example to compute the gradient attains the best available approximation of the true error gradient. The class of neural network algorithms which meet this requirement are called deterministic networks (deterministic networks have already been described briefly in Subsection 2.3.4).

For deterministic neural networks, weight adjustments are computed from the gradient of every example in the data set.

$$\nabla G(\mathbf{w}) \approx \nabla J^{\text{det}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1} \nabla J_n(\mathbf{w}) \tag{91}$$

Therefore, deterministic neural networks are capable of making the most accurate weight adjustments at each iteration to model the given data distribution.

Batch learning and stochastic learning are two alternate approaches which compromise the accuracy of their gradient estimate in favour of greater speed of convergence. The batch learning approach partitions the data set into several batches on which the network trains sequentially. Thus, the network updates its weights after evaluating the gradient $\nabla J_b^{\text{bat}}(\mathbf{w})$ for each batch b (40). The goal of batch learning is to sample these batches so that the computed gradient for each batch still approximates the deterministic gradient.

$$\nabla J^{\text{det}}(\mathbf{w}) \approx \nabla J_b^{\text{bat}}(\mathbf{w}) = \nabla J_b^{\text{bat}}(\mathbf{w}) + \varepsilon \tag{92}$$

$$\mathbf{w}^{\text{det}} = \mathbf{w} + \Delta \mathbf{w}^{\text{det}} \tag{93}$$

$$\approx \mathbf{w} + \Delta \mathbf{w}^{\text{bat}} \tag{94}$$

The gradient of each batch can be interpreted as a noisy estimate of the deterministic gradient, with the magnitude of the noise indicated by some noise term ε. If the magnitude of ε is sufficiently small, then the batch delta weights $\Delta \mathbf{w}^{\text{bat}}$ will be approximately equal to the deterministic delta weights $\Delta \mathbf{w}^{\text{det}}$. Therefore, the network will be in the same state after iterating through a batch as when iterating through the entire data set. For example, if a data set was divided into two equal-sized batches, and $\varepsilon \approx 0$ for each batch gradient, then a batch network after one epoch would have the same weights as a deterministic network after two epochs. Thus, for this example, the batch network would converge twice as fast as the deterministic one. As long as the batch gradient continues to approximate the deterministic gradient, the data set can be partitioned into smaller and smaller batches further speeding the convergence of the batch network.

Stochastic learning represents the limit of batch size reduction such that each batch contains a single example. A stochastic learning network updates its weights after estimating the error gradient for each example (41). Thus, a stochastic learning network performs N weight updates per epoch to the single weight update of the deterministic network. If the stochastic gradient estimate accurately approximates the gradient across the entire data set, then a

stochastic network would be N times as fast as the deterministic one. However, for any non-trivial data set, it is unlikely that the estimate of the gradient from a single example is going to be a very accurate approximation of the deterministic gradient. Thus, the error term ε for stochastic learning is likely to be very large. The presence of this error term causes the network update to be misdirected from the true gradient update. Figure 13 illustrates this divergence between deterministic and stochastic gradient calculations. The effect of the error term is that the stochastic learning network requires several steps in order to reach a comparable state to that of the updated deterministic learning network. Due to the influence of ε, the stochastic learning network from the figure converges to the error minimum B in $\frac{N}{8}^{th}$ the speed of the deterministic network.

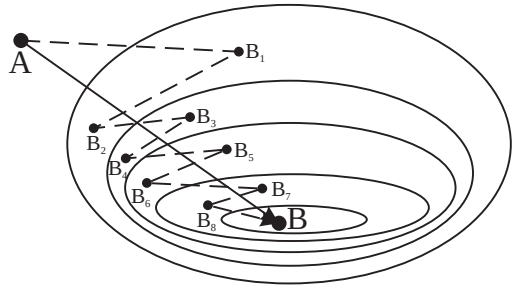

Figure 13: Comparison of deterministic and stochastic gradient descent in a topological error surface. Point A represents the initial network state. Point B represents the minimum in the weight space. The straight line shows the network update performed by the deterministic learning network. Due to the fact that the deterministic gradient approximates the true gradient of the error space, the network can proceed directly to the minimum in a single step. The dotted, jagged line indicates the path through the error space taken by a stochastic learning network. The influence of ε prevents the network from correctly estimating the gradient. The stochastic gradient tends to oscillate around the deterministic gradient because every new gradient estimate has to compensate for the error of the previous iteration. This network required 8 steps to reach the minimum B.

From the example of stochastic learning, it is clear that the speed of convergence will not necessarily increase as smaller and smaller batches are used. At some point, the estimated gradient ceases to be a *sufficiently accurate* approximation of the true gradient $\nabla G(\mathbf{w})$. Unless the database is completely

redundant, there is simply insufficient information in each example to accurately estimate the true gradient. Due to the size of the noise term in the stochastic gradient calculation, a relatively small learning rate step size is required to prevent the network weights from overshooting the minimum or being updated too far in an inaccurate direction. Thus, the learning rate must decrease as the presence of the noise term increases. The decrease in learning rate step size means that the network will require more updates to reach a minimum. Therefore a tradeoff exists between the accuracy of the gradient estimate, and the number of weight updates per epoch. Somewhere between the extremes of deterministic and stochastic learning exists an optimal batch size which will maximize the convergence speed of the network.

4.2 Batch Learning in Practice

From the previous subsection, several theories were posited about the behaviour of deterministic, batch and stochastic approaches to neural network learning. This subsection will attempt to support these hypotheses by creating a set of experiments that will train supervised and unsupervised neural networks using a range of different batch sizes. The supervised neural network that will be used in these experiments is the cross-entropy multi-layer perceptron. The unsupervised neural network that will be used is the Gaussian mixture model. These neural networks will be applied to classification and clustering tasks for three data mining data sets.

This subsection will begin by describing the properties of each of the data mining databases. Next, the experimental framework will be presented for each type of neural network. The results and observations for each experiment will be presented. Finally, the full cost of BSP implementation using exemplar parallelism will be determined for each experiment.

4.2.1 Data Mining Data Set Descriptions

There are three databases which will be considered in these experiments: the Wisconsin breast cancer database, the Daimler-Benz thyroid database, and the 1994 US Census Bureau database. Each of these databases was gathered from the UCI (University of California at Irvine) data repository[6] This data repository contains many databases that have been collected from real-world

[6]These databases can be downloaded from the UCI web site at *ftp://ftp.ics.uci.edu/pub/machine-learning-databases/README*.

sources and have been analyzed using data mining techniques. The reason for selecting these data sets is to show that results of these experiments are practical and are applicable to standard data mining applications. In addition, the databases that will be explored in these experiments are not proprietary and are part of the public domain. Thus, the analyses presented in this report can be compared with existing publications and can be used as a reference for future explorations mining these databases.

The Wisconsin Breast Cancer database consists of a collection of mammogram data from 699 patients with discovered breast lumps. The results of the mammogram for each patient are described by 9 continuous-valued attributes. These attributes numerically describe properties of each breast lump, such as its size, shape, and consistency, from X-ray data. The purpose of this database is to classify each breast lump as being malignant or benign based on the attribute information. The collation and initial analysis of this database are demonstrated in [52][36]. These initial experiments were able to attain 93.7% classification accuracy when using 1-nearest neighbour clustering. This database is well-suited to the preliminary testing of a data mining algorithm because it is a reasonable size, it is easy to attain good classification accuracy and it is easy to analyze for clustering applications (the database can be described quite accurately using two Gaussian distributions).

- Database size: 699 examples
- Classes:
  - Class 1 = Benign (458 examples (65.5%))
  - Class 2 = Malignant (241 examples (34.5%))
- Attributes:
  - 9 continuous-valued attributes with domain (1, 10)
  - missing values were replaced with the mean value
  - scaled to range (0, 1)

Table 4: Description of the Wisconsin breast cancer database.

The Daimler-Benz thyroid database is larger and more complex than the previous database. The database contains information about 3772 patients who were examined for thyroid conditions. The patient information consists

of a combination of 21 binary and continuous attributes. There are three possible diagnoses for each patient: normal, hyper-, and hypothyroid. This classification is complicated by the fact that the vast majority of cases exhibit a normal thyroid. Therefore, there are relatively few examples of the hyper- and hypothyroid groups to construct a class description from. In addition, the classifier must attain a very high classification accuracy ($> 92.47\%$) in order to be recognized as successfully classifying any of the non-normal cases. This database was used as a benchmark to compare the performance of a variety of neural network algorithms in [47].

- Database size: 3772 examples
- Classes:
  - Class 1 = Hyperthyroid (93 examples (2.47%))
  - Class 2 = Hypothyroid (191 examples (5.06%))
  - Class 3 = Normal thyroid (3488 examples (92.47%))
- Attributes:
  - 21 attributes: 15 binary and 6 continuous
  - no missing values

Table 5: Description of the Daimler-Benz thyroid database.

The third database that will be examined in these experiments is a collection of census data polled from over 30000 US citizens during 1994. This database contains a variety of demographic information, from education to race, describing each of the polled individuals. This information is represented by a mixture of nominal and continuous variables. In order to identify the discrete values of the nominal variables, a 1-of-n encoding scheme is used to transform the attribute[7]. The resultant data set contains 47 input attributes.

The classification task for this database is to predict the income level of each individual citizen based on their census information (probably for taxation purposes). There are two possible levels of income that are considered:

[7]By performing a 1-of-n encoding, a nominal attribute with C possible values could be represented by a C bit binary string of zeros with a single 1 at the position corresponding to the attribute value.

individuals with a high income earning over $50,000 per year, and individuals who earn less than that amount. The classification performance of several data mining algorithms applied to this database are summarized in [32]. This database is very noisy and it is quite difficult to attain a high degree of classification accuracy. Also, due to its size, it requires long training times.

- Database size: 30162 examples
- Classes:
  - Class 1 = High income ($> \$50{,}000$) (7474 examples (24.78%))
  - Class 2 = Low income ($\leq \$50{,}000$) (22688 examples (75.22%))
- Attributes:
  - 12 attributes: 6 binary and 6 continuous
  - binary attributes encoded using 1-of-n encoding
  - total of 47 inputs attributes

Table 6: Description of the 1994 US Census database.

4.2.2 Experimental Framework

To recapitulate, the theoretical analysis of batch learning from Subsection 4.1 posited three main hypotheses regarding the rate of convergence of neural networks:

1. batch learning will be faster than deterministic learning.
2. the number of training epochs decreases linearly with the number of batches as long as the batch gradient resembles the deterministic gradient.
3. there is a lower limit to the number of training epochs required to train a network using batch learning

In order to create empirical support for these claims, a set of experiments needs to be designed. The goal of these experiments is to examine the implications for batch learning on the convergence speed of neural network training.

The simplest way to examine the effects of batch learning is to train several neural networks, each with a different batch size, and examine how many epochs they each require to meet some performance criterion. Using this approach, a set of specifications can be drafted to ensure that each hypothesis will be address by the experiment.

According to the first hypothesis, neural networks that use batch learning will require fewer epochs to train than networks trained using deterministic learning. This hypothesis is simple to examine by training several networks using deterministic learning, and several neural networks that partition the data set into a number of batches. By comparing the average number of training epochs that are required by each type of network, the validity of the hypothesis can be examined.

In order to examine how the number of required training epochs changes with the number of batches, several neural networks can be trained with different batch sizes. If the average number of epochs that they require increases linearly with the number of batches, then this hypothesis can be supported.

The third hypothesis can be explored in a similar fashion to the previous two hypotheses. By training several neural networks using a large number of batches, a point is sought beyond which the number of training epochs required remains constant or increases.

Given the fact that each of these experiment specification have similar requirements, a single experiment can be designed to examine all of the hypotheses. In general, this experiment will train several networks with various numbers of batches and examine the number of epochs that are required to meet some performance criterion. This experiment can be conducted for both cross-entropy multi-layer perceptrons and Gaussian mixture models. Both types of neural networks will be trained on each of the three data mining databases specified in the previous subsection.

However, before these experiments can be conducted, several design decisions must be addressed. These are:

- choosing a performance criterion to halt network training,
- determining the learning rate,
- initializing the network, and
- selecting an appropriate number of hidden neurons or components.

The choice of the relevant parameters to meet these design decisions are listed for each experiment in Tables 7, and 8.

- Cross-entropy multi-layer perceptrons:
  - Breast cancer database:
    - Number of training trials for each batch size: 50
    - Number of hidden neurons: 6
    - Classification accuracy threshold: 98%
  - Thyroid database:
    - Number of training trials for each batch size: 50
    - Number of hidden neurons: 10
    - Classification accuracy threshold: 99%
  - US census database:
    - Number of training trials for each batch size: 10
    - Number of hidden neurons: 20
    - Classification accuracy threshold: 85%

Table 7: Batch learning experiment parameters for cross-entropy multi-layer perceptron networks.

Choosing a Performance Criterion

Before beginning neural network training, it is important to establish a criterion for deciding when to stop training. In the ideal, neural network training would be halted once the network has converged to a minimum in the error space. However, it is both exceedingly time-consuming and unnecessary for a neural network to settle into a minimum. Generally, comparable performance can be gained by halting training before a minimum has been reached. In addition, for batch learning networks, convergence to a minimum may not be possible due to the noise term in the gradient estimate.

Thus, it is generally preferable to establish a criteria for halting neural network training based on the cost function $J(\mathbf{w})$ or some other measure of network performance. Considering that both cross-entropy multi-layer perceptrons and Gaussian mixture models are concerned with modeling probability distributions, it is logical to choose a convergence criterion that is based on some probability measure.

For cross-entropy multi-layer perceptrons, the most common measure of network training performance is classification accuracy. This measure calcu-

- Gaussian mixture models:
  - Breast cancer database:
    - Number of training trials for each batch size: 50
    - Number of components: 4
    - Log-likelihood threshold: 3550
  - Thyroid database:
    - Number of training trials for each batch size: 50
    - Number of components: 32
    - Log-likelihood threshold: 150000

Table 8: Batch learning experiment parameters for Gaussian mixture model networks.

lates the percentage of examples which have been classified correctly by the network. This measure provides a consistently-ranged $(0-1)$ and intuitively-understandable representation of the network's performance. Classification accuracy is also suitable because it is commonly used by many different data mining techniques so that the results of the neural network are easily compared with other methods.

It is more difficult to define an appropriate convergence criterion for the Gaussian mixture model. Because there is no predefined target value to which the network output can be compared, the performance of the network is based on its ability to represent the input. Thus, the measure that will be used is the log-likelihood. The log-likelihood estimate is calculated by taking the sum of the logarithms of the output of the network function (28) for each input example. Thus, the larger the log-likelihood value, the better the performance of the network.

There are two main problems with using log-likelihood as a convergence criterion. The first is that it is not particularly intuitive to interpret because it is not defined on a fixed range. The second is that mixture models which maximize likelihood are susceptible to singularities [43]. If a component is assigned to a single data example, it will attempt to represent that point exactly by letting its variance go to zero. In effect, this will drive the output of the component to infinity (27). This will cause the log-likelihood value to be arbitrarily large even though the network may be a poor representation of the

data. The effect of singularities can be minimized by preventing the variance of each component from dropping below a certain value. Despite these shortcomings, log-likelihood provides a good statistical measure of the convergence and is the measure that will be used for the mixture model experiments.

For each experiment, an appropriate value must be chosen as a threshold for the performance measure. Each threshold was selected by training several networks on each data set for a large number of epochs. Based on the results of these training trials, a convergence threshold was chosen which was attainable, but difficult for the networks to reach. For supervised neural networks, the convergence threshold was set at one or two percent below the best-attained classification accuracy. For unsupervised neural networks, the convergence threshold was set at ninety percent of the best log-likelihood. Because mixture models can solve the gradient directly for each parameter (33–35), deterministic mixture model are capable of converging to a minimum within relatively few epochs. Due to the influence of the noise term ε in the batch gradient estimates, it is quite difficult for batch learning mixture models to approach this minimum. Therefore, the performance threshold for mixture model networks is generally quite far (10%) from the largest log-likelihood.

Because these experiments are strictly concerned with how quickly a neural network trains, it is not essential for the networks to be trained to reach the best possible minimum. By allowing networks to train to different performance levels, comparing the convergence speed becomes more difficult. Thus, the use of a fixed convergence criteria to halt network training is very important.

Determining the Learning Rate

In order to update the neural network weights in the direction of the gradient, a parameter η, called the learning rate, is used to determine the magnitude of this update. It is important to accurately determine this parameter because a value that is too small will lead to slow convergence, and a value that is too large may prevent the network weights from reaching the performance threshold. It is assumed that the learning rate parameter remains constant during the course of each neural network training session.

Technically, for a deterministic mixture model, it is not necessary to determine a learning rate value because the network is capable of directly solving the gradient with respect to each weight in the network (33–35). Thus, the updated weight value for the next iteration can be determined directly. However, for a mixture model that uses batch learning, this update is not necessarily accurate because it is calculated using only a subset of the data. Therefore, it necessary to use the learning rate parameter to provide an incremental tran-

sition between the calculated update and the current network state. From the mixture model weight update equations (36), it can be assumed that deterministic mixture models have a learning rate of 1 because they base their weight update entirely on the newly-estimated weight value. For batch learning mixture models, a learning rate parameter value in the range $(0, 1)$ is used to provide an incremental transition from the current network state to the new one.

For cross-entropy multi-layer perceptrons, the gradient only provides information about the direction of the update for each weight. Therefore, it is necessary to use the learning rate to specify the size of the update in the direction of the gradient. The learning rate parameter may be set to any postive real value.

In these batch learning experiments, it is desirable to determine the fewest possible training epochs required for convergence for each different batch size. This way, the training performance of each neural network can be compared in an unbiased fashion. In order to satisfy this constraint, it is necessary to optimize the learning rate for each neural network trial. The best learning rate is chosen by training several neural networks with a range of different learning rate parameters. The learning rate that yields the fewest average number of training epochs for convergence is chosen.

A momentum term is commonly used when training supervised stochastic learning networks. However, no momentum was used in these experiments for two reasons. The first reason is that the addition of a momentum constant increases the number of parameters that must be optimized for each batch size. Because it requires training many networks to determine the optimal learning rate for each batch size, the process of determining an optimal parameter is quite time consuming. With the addition of a momentum term, both parameters would have to be optimized simultaneously. This greatly increases the size of the search space. Thus, co-optimizing a momentum constant is very time consuming. The second reason for not including a momentum term is that from my experiments, I found that a network with well-chosen learning rate converged just as quickly as a network which used momentum. Thus, the addition of a momentum term is unlikely to have a significant effect on the speed of network training

Initializing the Neural Network

The weights of a neural network are generally randomly initialized at the start of each training trail. For supervised neural networks, weights are generally set to normally-distributed values around zero with a small variance. For

unsupervised neural networks, the means $\boldsymbol{\mu}_i$ of each distribution are set to values that are normally-distributed around the center of the input distribution. The variances and mixing proportions for mixture models are generally set to the same value for each component.

One of the consequences of random initialization is that it is common for neural networks to converge to different solutions on different trials. Therefore, it is necessary to train several neural networks in order to accurately assess their training performance. For these experiments, multiple neural networks were trained for each different batch size, and their average number of training epochs were calculated.

Selecting the Number of Hidden Neurons or Components

The number of hidden neurons determine the processing capability of the cross-entropy multi-layer perceptron. Networks with more hidden neurons are able to model more complex problems. Similarly, for mixture models, the number of components determines the network's ability to represent the input data. The more component distributions that are used in a mixture model, the more clusters can be identified by network.

The appropriate number of hidden neurons and components in a network is the smallest number that enables a neural network to reach the performance criterion. This number of neurons or components limits the risk of over-fitting the data while still providing adequate performance. For each database, the appropriate number of hidden neurons or components was determined experimentally.

4.2.3 Batch Learning Experiment Results

In this subsection, the results of the batch learning experiments described above will be presented for each database. Each experiment will be analyzed to examine whether the experiment hypotheses are supported. The results for the supervised and unsupervised neural networks will be discussed separately. The observations for each experiment will be used to explore the validity of the experiment hypotheses.

Batch Learning Experiment Results for Supervised Neural Networks

Figures 14, 15, and 16 illustrate the number of training epochs required by the cross-entropy multi-layer perceptron applied to the breast cancer, thyroid and US census databases. In each of these figures, the average number of training epochs is plotted against the batch size of the network. An asterisk on the graph indicates the result of training a batch learning network for a

particular number of batches. These figures emphasize the convergence speed for networks with large batch sizes. Because their batch size is sufficiently large, these neural networks should be capable of accurately approximating the deterministic gradient from the data in each batch. Therefore, in theory, they should converge to a solution b times more quickly than a deterministic neural network, where b is the number of batches used by the batch learning neural network.

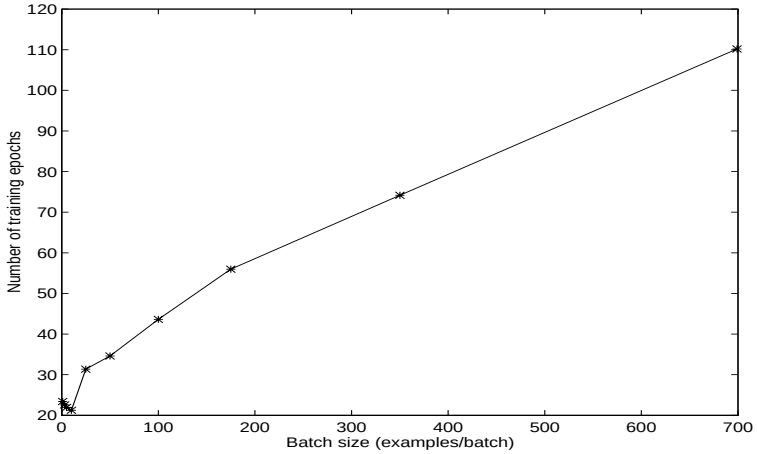

Figure 14: The number of training epochs required by cross-entropy multilayer perceptrons applied to the breast cancer database plotted against batch size.

For the training results on the thyroid and US census databases, it is clear that the number of epochs required to train batch learning networks with sufficiently large batches does, indeed, decrease with $\frac{E_D}{b}$. E_D is the average number of epochs needed to train the deterministic neural network on these two databases. This observation can be easily verified using the numerical results for these experiments which are presented in Appendix A. For the breast cancer database, the reduction of the number of training epochs due to batch learning is not as large. However, it can be noted from the figure that the number of training epoch still decreases approximately linearly with the number of batches. These observations are sufficient to support the first two hypotheses of the experiment (faster convergence using batches and a near-linear relation between batch size and training epochs).

In order to examine the validity of the third hypothesis, it is helpful to invert the x-axis of the figures so that the number of training epochs is plotted

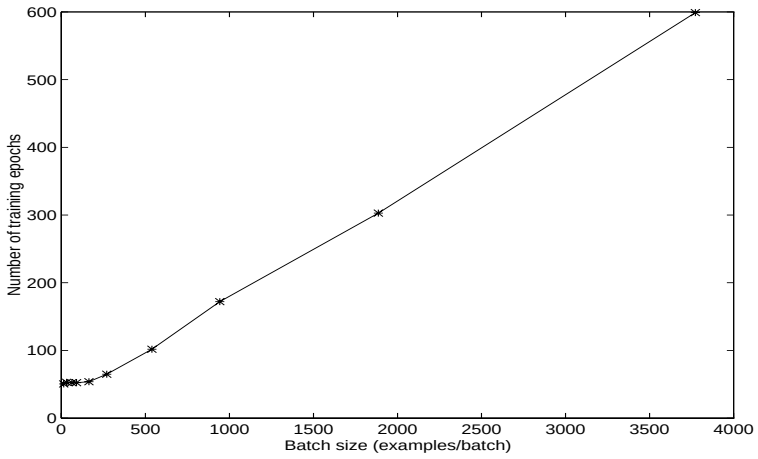

Figure 15: The number of training epochs required by cross-entropy multi-layer perceptrons applied to the thyroid database plotted against batch size.

against the number of batches. The batch size B is inversely proportional to the number of batches b ($B = \frac{N}{b}$), so these figures simply invert the x-axis so that the training results of networks with a large number of batches can be examined more closely. Figures 17, 18, and 19 illustrate these plots for each of the three data mining databases. These figures emphasize the results of cross-entropy multi-layer perceptrons that are trained using a large number of batches. In theory, there should be a lower limit to the number of epochs that are required. Further, increasing the number of batches once this limit has been reached will not lead to any further improvement in convergence speed. The number of training epochs that are required will remain constant or may even increase.

All three of these figures clearly indicate a lower limit to the number of epochs that are required to successfully train a neural network on each database. In each of these experiments, this limit is reached by a batch learning neural network with an intermediate number of batches. From the figures it can be observed that by increasing the number of batches beyond this point yields no further decrease in the number of training epochs. Therefore, the third hypothesis is supported by these experiments.

Because the lower limit to the convergence speed is reached by a neural network trained with an intermediate number of batches, these experiments are amenable to exemplar parallelism. The constraint imposed by exemplar

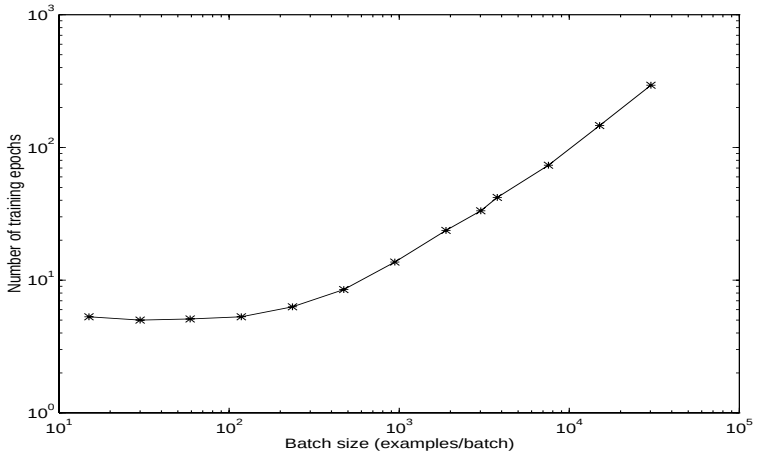

Figure 16: The number of training epochs required by cross-entropy multilayer perceptrons applied to the US census database plotted against batch size. Note that the both x- and the y-axis have logarithmic scale.

parallelism is that the batch size must be larger than the number of processors. For these experiments, the batch size of the network requiring the fewest training epochs is generally larger than the number of processors for most general-purpose parallel machines. In the next subsection, the optimal number of batches will be determined more accurately using the BSP cost model.

It is also important to consider how the optimal learning rate varies with changes in the batch size. Figures 20, 21, and 22 plot the optimized learning rate against the batch size. Each of these figures have approximately the same shape (the US census database figure is distorted somewhat due to the logarithmic scaling of the axes). The learning rate is constant for networks trained with large batch sizes. Once a certain batch size is reached, the optimized learning rate drops quite rapidly. By comparing these graphs with the Figures 14, 15, and 16, it can be observed that the learning rate begins to drop off as the increase in the convergence speed begins to falter.

Considering the theoretical analyses of batch learning, it has been posited that the batch learning gradient attempts to approximate the deterministic gradient. So far, this theory has been stated without proof. However, this theory can be supported by examining these plots of the optimized learning rate. Recall that the calculated gradient for supervised learning provides the

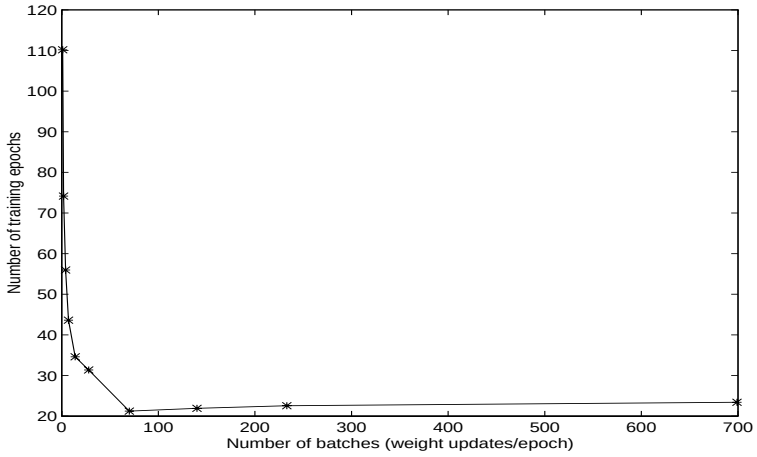

Figure 17: The number of training epochs required by cross-entropy multilayer perceptrons applied to the breast cancer database plotted against the number of batches.

direction in which weight should be adjusted so as to minimize the error; however, it does not dictate the magnitude of this update. The size of the update is specified by the learning rate parameter. Now, if the batch gradient accurately estimated the deterministic gradient, both gradients would have the same direction. Assuming that a learning rate parameter was chosen to optimally update the weights using the deterministic gradient, this same learning rate parameter would be optimal for the batch gradient as well. Therefore, considering that the learning rate remains constant for batch sizes which have a linear decrease in the number of training epochs required, it can be inferred the batch gradient is an accurate approximation of the deterministic gradient.

Batch Learning Experiment Results for Unsupervised Neural Networks

Because the results of these batch learning experiments for training Gaussian mixture models are practically identical to the results given above, only the figures from training on the first two databases are provided. The numerical results of these experiments are listed in Appendix B.

The validity of the first two hypotheses can be examined by viewing the plots of the number of training epochs against the batch size. These plots are displayed in Figures 23, and 24. The figure containing the training results from the breast cancer database illustrates an approximately linear speedup in

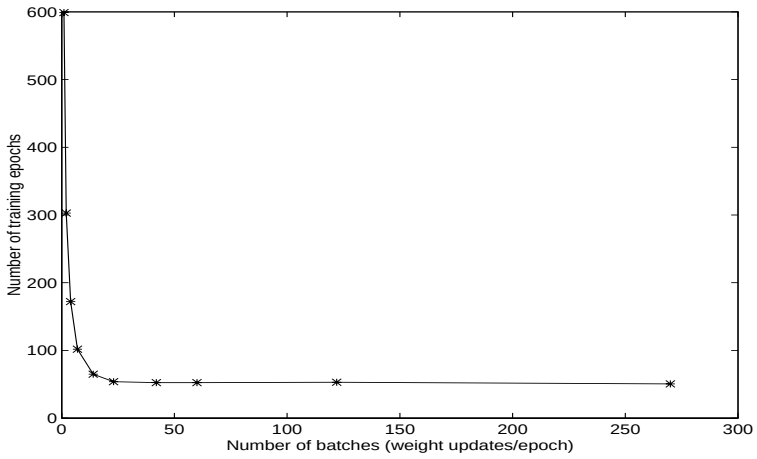

Figure 18: The number of training epochs required by cross-entropy multilayer perceptrons applied to the thyroid database plotted against the number of batches.

the convergence speed for mixture models trained with large batch sizes. For the thyroid database, although the decrease in the number of training epochs is not linear, the speed of convergence definitely increases with the number of batches. The magnitude of this speedup can be examined by using the numerical results from Appendix B. These two figures demonstrate that the use of batch learning with Gaussian mixture models can lead to an increase in the convergence speed and a decrease in the number of training epochs that are required for training.

The lower limit to the number of required training epochs rate can be examined by plotting the number of epochs against the number of batches. These plots are displayed in Figures 25 and 26. In these figures, it is clear that the number of training epochs approaches a limit as the number of batches increases. In both plots, the number of epochs reaches a maximum once a critical number of batches are used. For both of these experiments, it was impossible to train networks to meet the convergence criterion that partitioned the data set into more than 64 batches; the noise term in the gradient estimate for these networks was too large to enable them to reach the convergence criterion. The presence of this limitation to the decrease in the number of training epochs supports the third hypothesis.

Like the experiments for supervised neural networks, the limit to the number of training epochs is reached by neural networks trained using an interme-

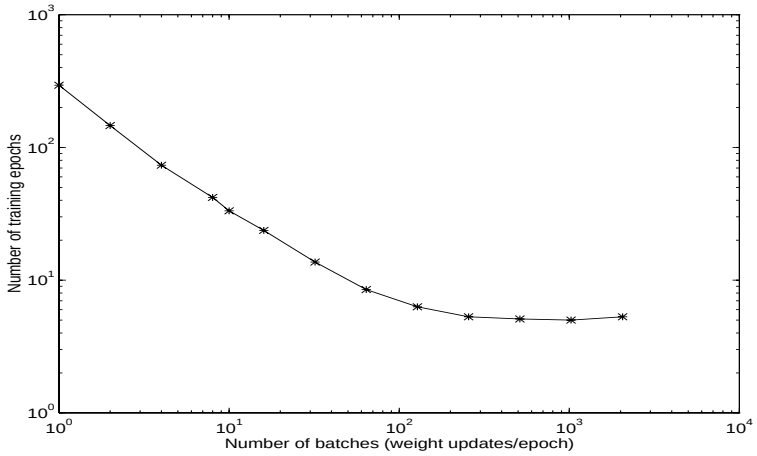

Figure 19: The number of training epochs required by cross-entropy multi-layer perceptrons applied to the US census database plotted against the number of batches. Note that the both x- and the y-axis have logarithmic scale.

diate batch size. This observation means that batch learning Gaussian mixture models are suitable for parallel implementation using exemplar parallelism.

The last set of figures to be examined plot the learning rate against the batch size. Figures 27 and 28 display these results. These figures are very similar to the plots of the optimized learning rate for supervised networks. While the number of training epochs decreases approximately linearly in relation to the number of batches, the optimized learning rate retains a fixed value. Once the number of epochs ceases to decrease linearly with the number of batches, the learning rate begins to decrease. The same analysis of the implications of the variation in the optimized learning rate parameter that were presented for supervised neural networks can be applied to unsupervised neural networks.

4.2.4 BSP Cost Analysis and Batch Learning

In the previous subsection, several neural network training experiments were conducted examining the effect of batch size on the number of training epochs. The results of training neural networks with different batch sizes were compared based on the number of epochs that were required to reach a performance threshold. On a sequential computer, these results are sufficient to indicate that network training time can be reduced by using batch learning. However, this is not as clear for a parallel neural network implementation because the

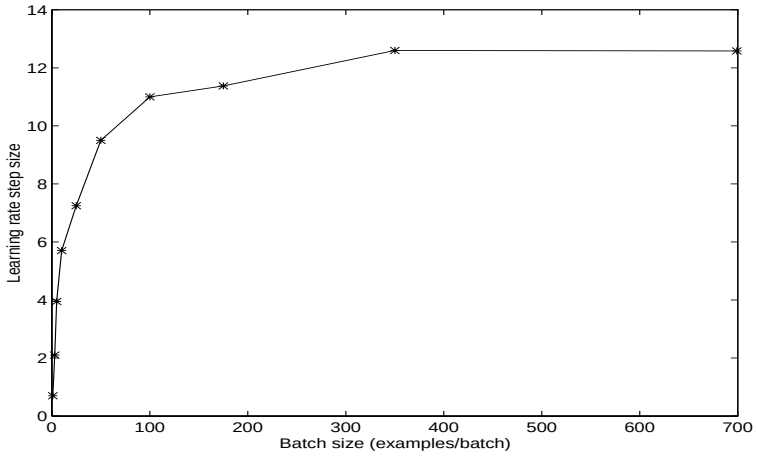

Figure 20: The optimized learning rate of cross-entropy multi-layer perceptrons applied to the breast cancer database plotted against batch size.

use of multiple weight updates per epoch incurs the cost of additional communications and barrier synchronizations for each epoch. By applying the batch learning network training results to the BSP cost model, the magnitude of the cost of these additional operations can be determined. In addition, the execution time of these batch learning networks can be accurately predicted for a range of parallel computers.

Before the parallel execution times can be computed, the BSP cost equations for supervised and unsupervised neural networks need to be tailored to work with batch learning. Considering the theoretical and empirical results from Section 3, exemplar parallelism was deemed the superior strategy for parallelizing neural networks. Thus, this subsection will only consider the application of exemplar parallelism to batch learning neural networks. However, the general conclusions should be applicable to other neural network parallelization strategies. The BSP cost equations for exemplar parallelism are given in Subsections 3.2.1 and 3.2.2. It can be observed that the BSP cost equations are equivalent for supervised and unsupervised neural networks; thus, the alterations to these cost equations required by batch learning are applicable to both types of neural networks.

In order to minimize the number of supersteps, the most cost-efficient implementation of exemplar parallelism performs a single weight update for each epoch. Thus, the BSP cost equations for exemplar parallelism assume that

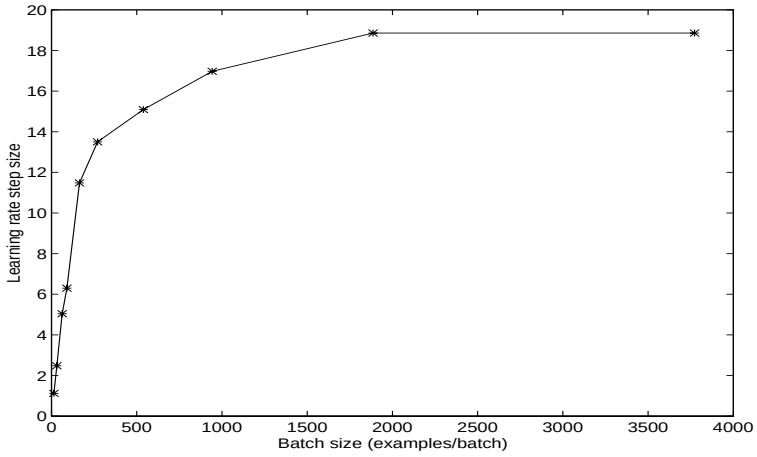

Figure 21: The optimized learning rate of cross-entropy multi-layer perceptrons applied to the thyroid database plotted against batch size.

deterministic learning is being used. For batch learning, however, b weight updates are performed for each epoch, where b is the number of batches. Therefore, the number of supersteps performed by a parallel neural network implemented with batch learning is increased by a factor of b for each epoch. Thus, a batch learning implementation of exemplar parallelism is will have a higher BSP cost because it will require b times more communications and synchronizations for each epoch. However, if the decreased number of training epochs of batch learning outweigh these additional costs, then a parallel batch learning network may indeed have a faster execution time.

Because batch learning divides the data set into b batches, the parallel processors operate on a single batch for each superstep. Thus, the cost of computation for each superstep is simply the cost of processing $\frac{N}{b}$ examples. The costs of communication and synchronization are independent of the number of examples being processed, therefore their costs remain unchanged from the original BSP equation. Making these minor changes, the BSP cost of exemplar parallelism for batch learing neural networks can be stated:

$$b\left[\frac{N}{b}\frac{AW}{p} + (p-1)Wg + 2l\right] \tag{95}$$

It can be observed from this equation that, by multiplying through by the number of supersteps b for each epoch, the computation cost becomes equivalent to

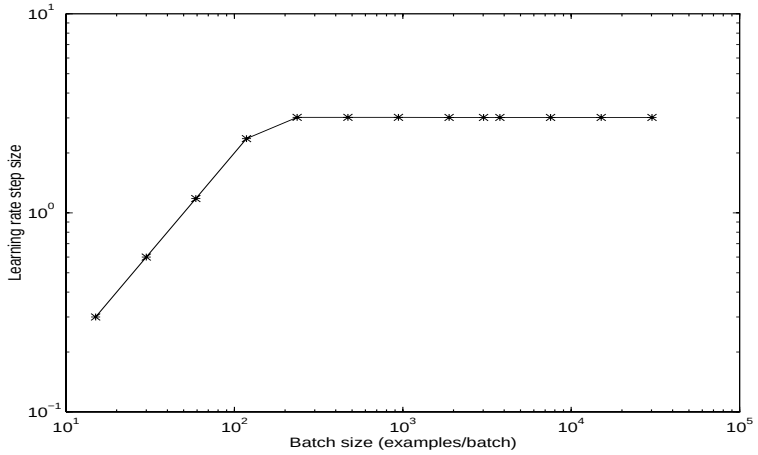

Figure 22: The optimized learning rate of cross-entropy multi-layer perceptrons applied to the US census database plotted against batch size. Note that the both x- and the y-axis have logarithmic scale

the computation cost in the original equation, and the costs of communication and synchronization are increased by a factor of the number of batches b.

The total cost for neural network training is the number of epochs E multiplied by the BSP cost for each epoch. Thus, the total BSP cost of training a neural network using exemplar parallelism and batch learning is:

$$E\left[\frac{NAW}{p} + b(p-1)Wg + 2bl\right] \tag{96}$$

The theoretical and empirical analyses of batch learning given above hold important implications for the interpretation of this cost equation. It has been demonstrated that the number of training epochs decreases linearly with the number of batches, as long as the batch gradient is an accurate estimate of the deterministic gradient. Thus, the number of epochs E required for training is inversely proportional to b. Due this proportionality, it can be assumed that:

$$E \propto \frac{1}{b} \equiv E = \frac{c}{b}, \tag{97}$$

for some constant c. By applying this result to (96), this cost equation can be reduced to:

$$E\frac{NAW}{p} + c(p-1)Wg + 2lc \tag{98}$$

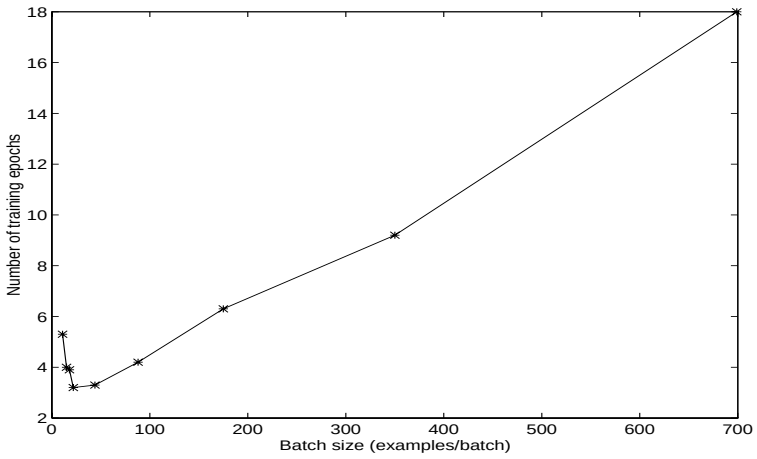

Figure 23: The number of training epochs required by Gaussian mixture model networks applied to the breast cancer database plotted against batch size.

Thus, it can be observed that the cost of communication and synchronization remains constant in relation to the number of batches that are used. However, the computation term is still a product of the number of epochs that are required for training the network. Because batch learning networks require fewer epochs to train, the BSP cost of batch learning will be less than the cost of deterministic learning as long as the decrease in the number of training epochs for the batch learning network outweighs the constant cost of communication and synchronization. Therefore, it can be stated that the predicted execution time for parallel neural networks trained using batch learning is less than for comparable networks trained with deterministic learning methods. Despite the additional communication and synchronization costs, the use of batch learning still leads to a reduction in parallel execution time.

The above result only holds as long as the number of training epochs decreases linearly with the number of batches. As has been demonstrated previously, this decrease in the number of epochs only holds up to a certain point. Beyond this point, the error term in the batch gradient calculation ε is too large to permit accurate estimation of the deterministic gradient. Therefore, no further decrease in the number of training epochs can be attained. For supervised neural networks, this error term has relatively little influence and the number of epochs tends to remain constant as the number of batches is increased further. For unsupervised neural networks, the presence of the error

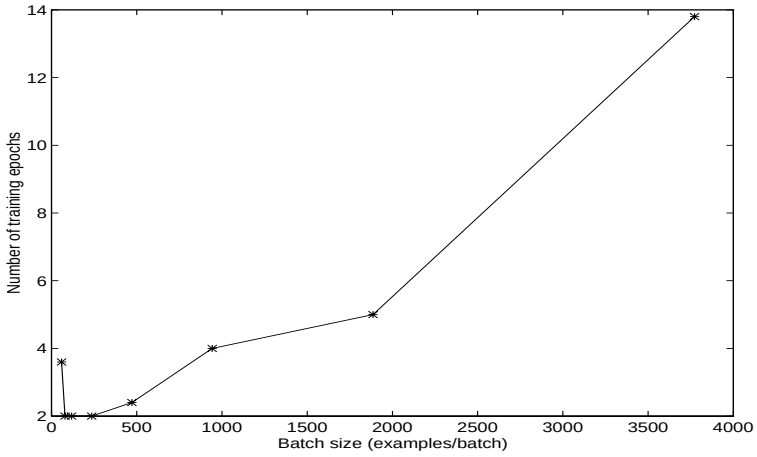

Figure 24: The number of training epochs required by Gaussian mixture model networks applied to the thyroid database plotted against batch size.

term may lead to an increase in the number of training epochs or may even prevent the network from converging to the specified performance threshold.

As a lower bound, the implications of a constant number of training epochs can be examined using (96). Because the number of training epochs is fixed as the number of batches increases, the cost of computation remains constant, and the cost of communication and synchronization increase linearly with the number of batches. Therefore, the total BSP cost of network training also increases linearly with the number of batches. For the case of unsupervised neural networks where the number of training epochs increases, it is clear that an even larger increase in the total BSP cost would occur.

Therefore, there is a well-defined minimum in the BSP cost equation for batch learning that occurs once the number of training epochs stops decreasing. Clearly, it is desirable to determine the number of batches b that corresponds to this minimum. This number of batches yields an implementation with minimal execution time on both sequential and parallel computers.

Figures 29 and 30 show a breakdown of the computation, communication and synchronization costs of training several supervised batch learning neural networks on the thyroid and US census databases. These figures represent the training results from the previous subsection applied to the BSP cost equation for batch learning. Both of these figures display the costs for parallel implementation on an IBM SP2 with 4 processors. From these figures, the

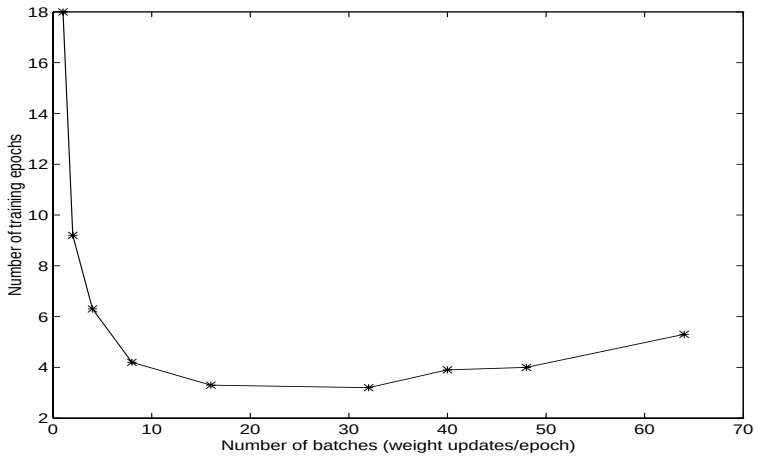

Figure 25: The number of training epochs required by Gaussian mixture model networks applied to the breast cancer database plotted against the number of batches.

minimum in the cost function is clearly apparent[8]. The figures show that where the number of epochs is decreasing, the cost of computation decreases linearly and the cost of communication and synchronization remain constant. It is also clear that in the regions where the number of epochs is constant, the cost of computation remains fixed and the cost of communication and synchronization increases linearly. These figures provide an empirical demonstration of the analysis of the BSP cost equation for batch learning presented above.

4.2.5 Batch Learning Parallel Execution Times

In the previous subsection, the BSP cost equation for exemplar parallelism applied to batch learning neural networks was derived. Now, the parallel execution times of the results from Subsection 4.2.3 can be predicted for a range of parallel computers by using the BSP cost model. Parallel execution times will be presented using 4 parallel computers: the CrayT3E, the IBM SP2, the SGI PowerChallenge, and the multiprocessor Sun computer. The BSP parameters for each of these parallel computers are listed in Table 1. In order to compare the execution times of each of these computers, the BSP cost in floating point operations can be divided by the execution rate to yield the

[8]It is useful to compare these figures with the network training results from Figures 18 and 19.

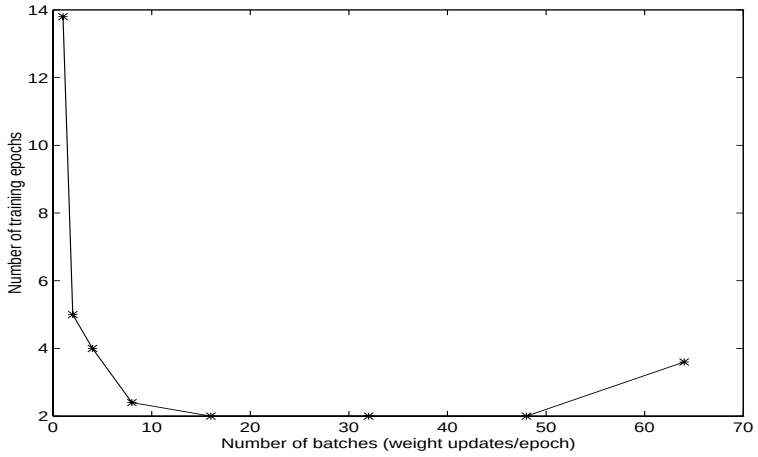

Figure 26: The number of training epochs required by Gaussian mixture model networks applied to the thyroid database plotted against the number of batches.

execution time in seconds. It will be assumed that each of these computers has 4 processors ($p = 4$).

In this subsection, the results of supervised and unsupervised network training will be presented separately. The BSP parallel execution time will be given for the training results of each of the three data mining databases. The optimal number of batches will be discussed for each experiment, and the parallel speedup attainable by using batch learning will be presented.

BSP Execution Times for Supervised Networks

Figures 31, 32, and 33 display the predicted parallel execution times for batch learning neural networks trained on the breast cancer, thyroid and US census databases. For the figures showing results from the thyroid and US census databases, a well-defined minimum in the BSP cost function is apparent. This minimum corresponds to the optimal number of batches for minimizing the parallel execution time. For these two figures, the location of this minimum is relatively consistent for each of the parallel computers considered. There is more variation between the location of the minimum in Figure 31. This variation is largely a product of the small size of the breast cancer database.

In addition, these figures can be used to compare the performance of the different parallel computers. For each experiment, the CrayT3D yielded the best performance. The IBM SP2 had the slowest execution time for each

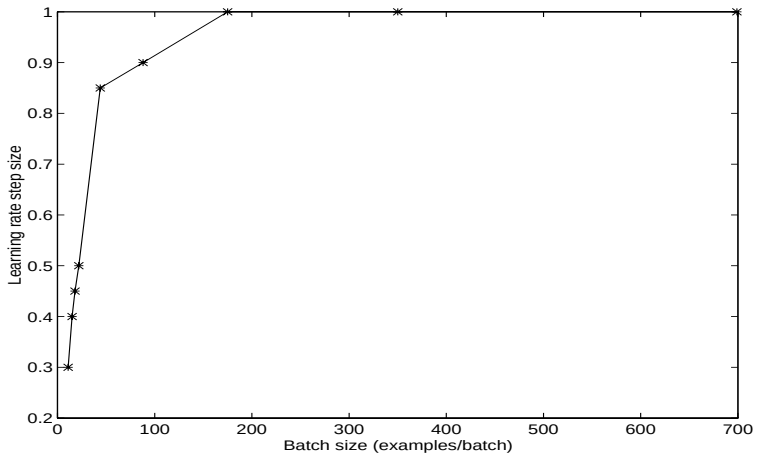

Figure 27: The optimized learning rate of Gaussian mixture model networks applied to the breast cancer database plotted against batch size.

experiment due to its large cost of performing a barrier synchronization.

The results of these figures are summarized in Tables 9, 10, and 11. In each table the optimal number of batches is shown for each parallel computer. It is important to note that there is relatively little variation in the optimal batch size between each of the parallel computers. The tables also show the fastest and the slowest times execution times for each parallel computer. By comparing these two times, the speedup of batch learning over deterministic learning can be calculated. The average speedup for each database is: 3.6 for the breast cancer database, 10.3 for the thyroid database, and 52.4 for the US census database. If these parallel computers can be taken as a small, but representative sample of the performance of average parallel computers, it can be stated that batch learning is potentially 52.4 times faster on average than deterministic learning when applied in parallel to the US census database. It is important to note that the speedup increases as the size of the database increases. If this pattern is indicative of a trend, then batch learning becomes increasingly important for accelerating network training as the size of the database increases.

BSP Execution Times for Unsupervised Networks

Figures 34 and 35 show the BSP execution times for the training results of the Gaussian mixture models applied to the breast cancer database and the thyroid database. Comparing these figures to the same types of figures

Computer	# Batches	Best Cost (Mflops)	Best Cost (seconds)	Worst Cost (seconds)	Speedup
CrayT3D	70	2.2	0.18	0.64	3.5
IBM SP2	7	5.7	0.47	2.36	5.0
SGI PowerChallenge	7	4.2	0.35	1.12	3.2
Sun	14	2.9	0.24	0.65	2.7

Table 9: BSP cost of training cross-entropy multi-layer perceptron networks on the breast cancer database.

Computer	# Batches	Best Cost (Mflops)	Best Cost (seconds)	Worst Cost (seconds)	Speedup
CrayT3D	42	73.0	6.09	67.82	11.1
IBM SP2	23	89.0	7.42	68.42	9.2
SGI PowerChallenge	23	78.2	6.51	67.98	10.4
Sun	23	77.0	6.41	67.93	10.6

Table 10: BSP cost of training cross-entropy multi-layer perceptron networks on the thyroid database.

Computer	# Batches	Best Cost (Mflops)	Best Cost (seconds)	Worst Cost (seconds)	Speedup
CrayT3D	512	233.1	19.43	1087.5	56.0
IBM SP2	256	276.6	23.05	1088.1	47.2
SGI PowerChallenge	512	239.9	19.99	1087.5	54.4
Sun	256	251.7	20.97	1087.7	51.9

Table 11: BSP cost of training cross-entropy multi-layer perceptron networks on the US census database.

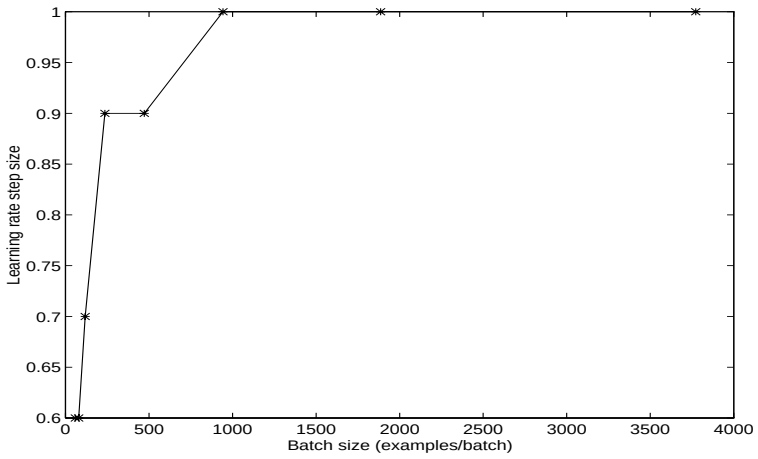

Figure 28: The optimized learning rate of Gaussian mixture model networks applied to the thyroid database plotted against batch size.

for the supervised networks show a similarity in the shape of the plots. The main distinction between the unsupervised and the supervised results is that the number of training epochs increases after a certain point for unsupervised neural networks rather than remaining constant. The similarity between these figures indicates that there is a consistency in the database that is exploited by using batch learning, regardless of the particular neural network technique.

From these figures, the execution time can be compared for the different parallel computers. Once again, the Cray T3D outperforms the other machines and the IBM SP2 is the slowest due to its expensive barrier synchronizations.

Tables 12 and 13 summarize the results displayed in the figures. The optimal number of batches and the best execution time is specified for each parallel computer. It can be observed that the optimal number of batches is relatively consistent among each of the parallel computers for both experiments. The Worst Cost column refers to the execution time for deterministic learning (although deterministic learning does not necessarily have the worst cost in the breast cancer example). By dividing the worst cost by the best cost, the speedup enabled by batch learning can be determined. For the breast cancer database, the average speedup is 3.4. For the thyroid database, it is 6.6.

These speedup results are not as large as the speedup results for supervised results. In addition, the optimal batch size is smaller than for supervised networks. These results are largely a product of the superior convergence

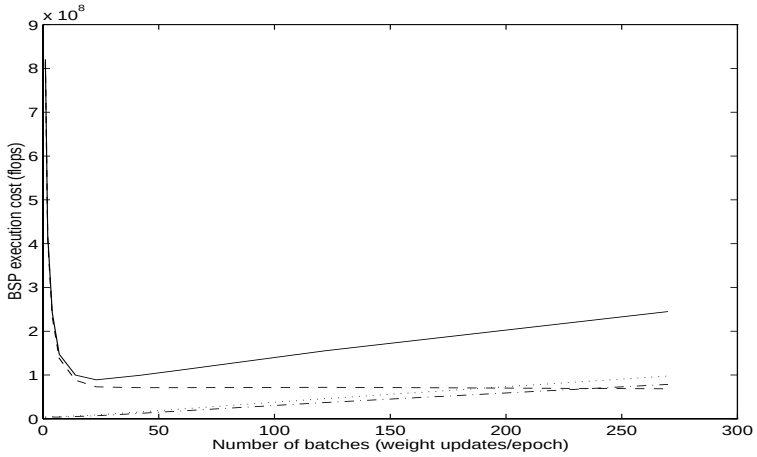

Figure 29: Breakdown of the BSP cost for training supervised batch learning neural networks on the thyroid database using a 4-processor IBM SP2. The solid line represents the total BSP cost of parallel implementation. The dashed line is the cost of computation, the dot-dashed line is the cost of communication, and the dotted line is the cost of synchronization.

properties of a deterministic mixture model. Because a mixture model can use the gradient directly to solve the best new value for each parameter, it can converge quickly and accurately to a minimum. Therefore, the batch gradient estimates have to be very accurate to enable a linear decrease in the number of training epochs. However, clearly there is still some speedup to be gained by using batch learning for mixture model networks.

5 Summary and Conclusions

In this report, I have presented a novel synthesis of several disparate areas of computing that are on the cusp of converging in industry. The fields of data mining, neural networks and parallelism are three areas of rapid development and topical research. The growing abundance of data mining applications has created a need for powerful data analysis tools that can process massive databases in a reasonable amount of time. The hybridization of parallelism and neural networks presents a powerful approach to solving these problems.

However, before parallel neural networks can gain widespread acceptance and application in industry, a framework needs to be established for their

Computer	# Batches	Best Cost (Mflops)	Best Cost (seconds)	Worst Cost (seconds)	Speedup
CrayT3D	16	0.1	0.01	0.05	4.7
IBM SP2	8	0.4	0.04	0.07	1.9
SGI PowerChallenge	8	0.2	0.02	0.06	2.6
Sun	16	0.2	0.01	0.06	4.3

Table 12: BSP cost of training Gaussian mixture model networks on the breast cancer database.

Computer	# Batches	Best Cost (Mflops)	Best Cost (seconds)	Worst Cost (seconds)	Speedup
CrayT3D	16	7.7	0.64	4.37	6.8
IBM SP2	16	8.4	0.70	4.40	6.3
SGI PowerChallenge	16	7.8	0.65	4.38	6.7
Sun	16	7.9	0.66	4.38	6.7

Table 13: BSP cost of training Gaussian mixture model networks on the thyroid database.

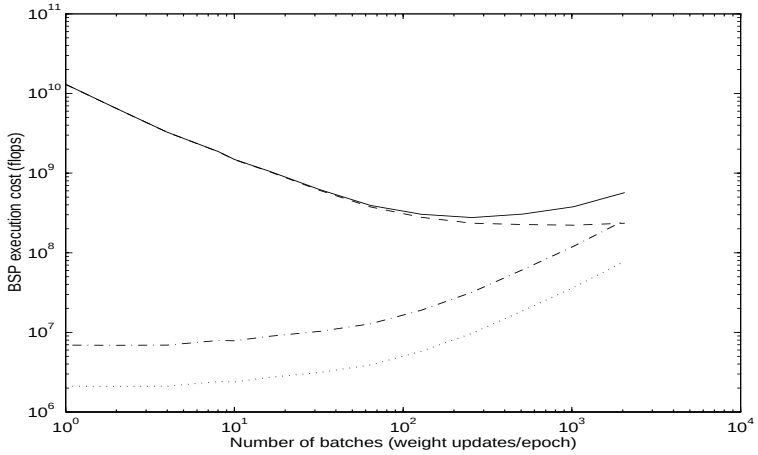

Figure 30: Breakdown of the BSP cost for training supervised batch learning neural networks on the US census database using a 4-processor IBM SP2. The solid line represents the total BSP cost of parallel implementation. The dashed line is the cost of computation, the dot-dashed line is the cost of communication, and the dotted line is the cost of synchronization. Note that the both x- and the y-axis have logarithmic scale.

development and analysis. In this report, I assert that statistical parameter estimation and bulk synchronous parallelism are two suitable foundations for this framework.

Statistical parameter estimation presents a theoretically-established approach to dealing with data analysis tasks. As I have demonstrated in this report, this approach is particularly well-suited to the description of supervised and unsupervised neural networks for clustering and classification applications. In particular, the development of the cross-entropy error term and the progression of unsupervised networks to mixture models are a product of the influence of statistics on the field of neural networks. This statistical perspective provides a much-needed context for the analysis and comparison of neural networks with other data analysis techniques.

The framework for the design and development of parallel neural network software is provided by bulk synchronous parallelism. As I have detailed, the BSP model of parallel computation is well suited to parallelizing neural networks for several reasons: it enables the creation of code which is easy to understand, efficient, and architecture-independent, it is designed specifically

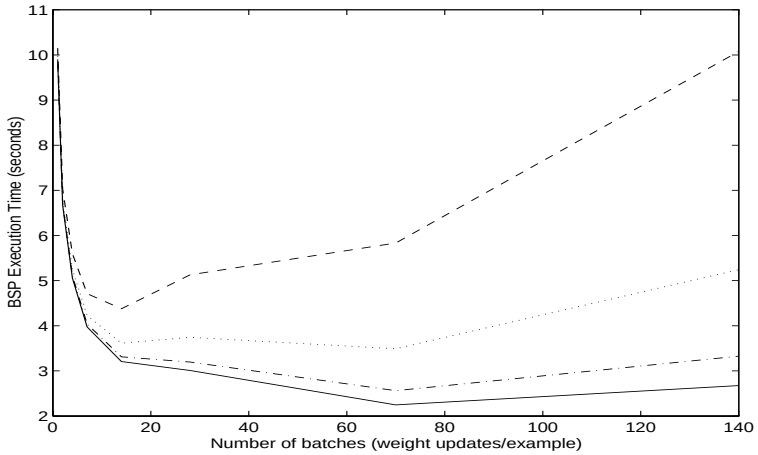

Figure 31: The parallel execution time for training cross-entropy multi-layer perceptrons on the breast cancer database is plotted against the number of batches. Training time is displayed for several different parallel computers. Legend: solid line - CrayT3D, dashed line - IBM SP2, dot-dashed line - SGI PowerChallenge, dotted line - Sun.

to handle communication-intensive applications, and it has a highly accurate cost model. By using BSP to develop the network software, general-purpose parallel neural networks can gain wide application in industry.

Using the framework established by statistical parameter estimation and bulk synchronous processing has enabled me to make several discoveries in the area of parallel neural networks. I will briefly summarize each of these contributions.

I have used the BSP cost model to perform detailed analyses of the cost of parallelizing supervised and unsupervised neural networks. Three granularities of parallelism were examined and compared in this report: exemplar, block and neuron parallelism. I analyzed each of these techniques theoretically and with application to specific parallel computers by using the BSP cost model. The results of these analyses revealed that exemplar parallelism is the superor technique under practical training considerations. The superiority of exemplar parallelism is even more apparent when the size of the data set is large, as it is in data mining applications. Most existing parallel neural network applications arbitrarily choose a method of parallelization to suit their specific parallel computer or their particular application. The theoretical and empirical results

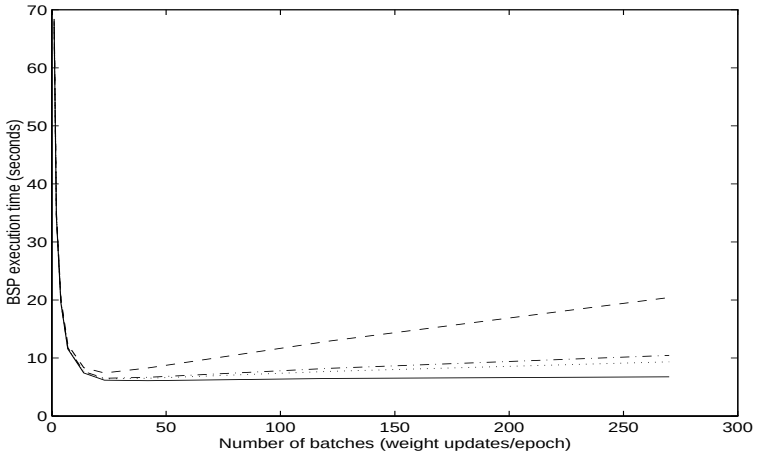

Figure 32: The parallel execution time for training cross-entropy multi-layer perceptrons on the thyroid database is plotted against the number of batches. Training time is displayed for several different parallel computers. Legend: solid line - CrayT3D, dashed line - IBM SP2, dot-dashed line - SGI Power-Challenge, dotted line - Sun

in this report show that exemplar parallelism is an optimal form of parallelism in neural networks.

However, the results of the BSP cost analysis only prove optimality on an epoch-by-epoch basis. Each of the parallelization strategies were optimized to minimize the cost of training for each epoch. This optimization process entailed limiting the number of weight updates for each epoch. Thus, the three parallelization strategies assumed that the neural network implementations used deterministic learning. However, by minimizing the number of weight updates for each epoch, the speed of neural network training is slowed considerably. Thus, a neural network implementation which is optimal on an epoch-by-epoch basis may end up incurring a higher cost than a network which is more expensive for each epoch but requires more epochs to reach the convergence criteria. Thus, it is important to consider the speed of network convergence in order to assert the superiority of exemplar parallelism.

To examine the potential convergence speed for a particular neural network algorithm, I analyzed the implications of the number of weight updates per epoch on the speed of network training. This is the only paper that I am aware of that conducts this type of examination. I created a set of experiments

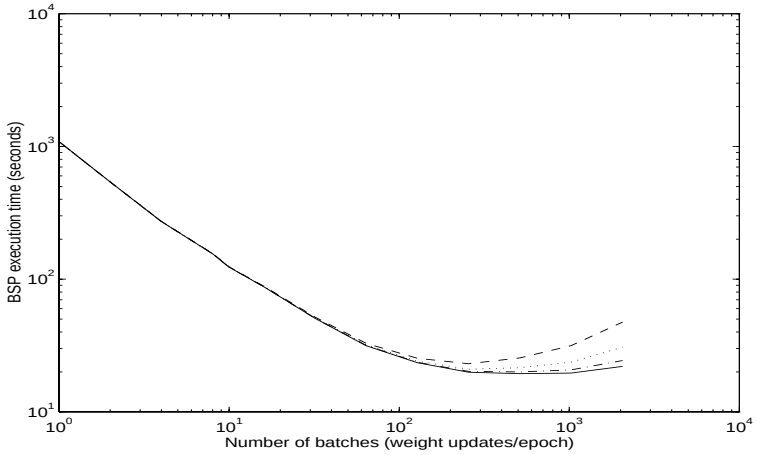

Figure 33: The parallel execution time for cross-entropy multi-layer perceptrons on the US census database is plotted against the number of batches. Training time is displayed for several different parallel computers. Legend: solid line - CrayT3D, dashed line - IBM SP2, dot-dashed line - SGI PowerChallenge, dotted line - Sun.

to examine the extent of the variation in the number of training epochs by dividing the data set into a number of subsets or batches. I trained several neural networks, each with a different batch size, and examined the number of epochs that were required to reach a certain performance threshold. Three reasonably large data mining data sets were used to train the neural networks.

The phenomenon that I observed throughout these experiments was that the required number of training epochs decreases linearly with the number of weight updates up to a point. Beyond this point, the number of training epochs stopped decreasing. This result was consistent for both supervised and unsupervised neural networks trained on each of the three data sets. Invariably, this point described some intermediate batch size between deterministic and stochastic updating.

In order to explain this phenomenon, I conducted an analysis of the method of batch learning. I define batch learning as an attempt to approximate the deterministic gradient using a subset of the data set. As long as the gradient estimate is accurate, then a batch learning network will be in the same state after each weight update as a deterministic learning network at the end of an epoch. Therefore, a deterministic network that requires E epochs to converge

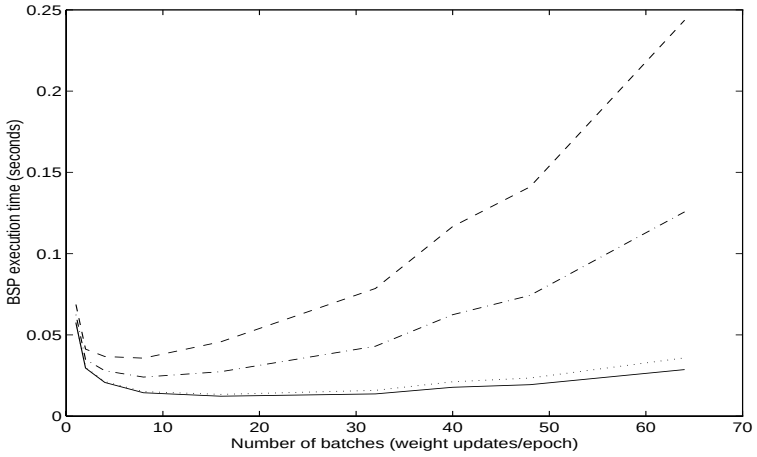

Figure 34: The parallel execution time for training Gaussian mixture model networks on the breast cancer database is plotted against the number of batches. Training time is displayed for several different parallel computers. Legend: solid line - CrayT3D, dashed line - IBM SP2, dot-dashed line - SGI PowerChallenge, dotted line - Sun.

to a solution would require only $\frac{E}{b}$ epochs for convergence if it was implemented using batch learning to divide the data set into b batches (assuming that the error gradient for each batch approximates the deterministic gradient). From this result, it is clear that the number of training epochs of a batch learning network decreases linearly with the number of batches.

However, because the batch estimate of the gradient is noisy and incomplete, there is a minimum number of examples in a batch that are required in order to determine an accurate estimate of the deterministic gradient. Thus, there is a bound to the number of training epochs required that can be surpassed by further dividing the data set into smaller and smaller batches. Since smaller batches imply more weight updates and weight updates are expensive in a parallel environment, it is important to determine the batch size at which the number of training epochs stops decreasing.

By incorporating the results of these experiments into the BSP cost equations for exemplar parallelism, the parallel execution cost of neural network training for each batch size can be determined. In each case, the BSP cost was minimized for a batch size which approximates the optimal number of training epochs. Therefore, because the cost of exemplar parallelism is minimal for

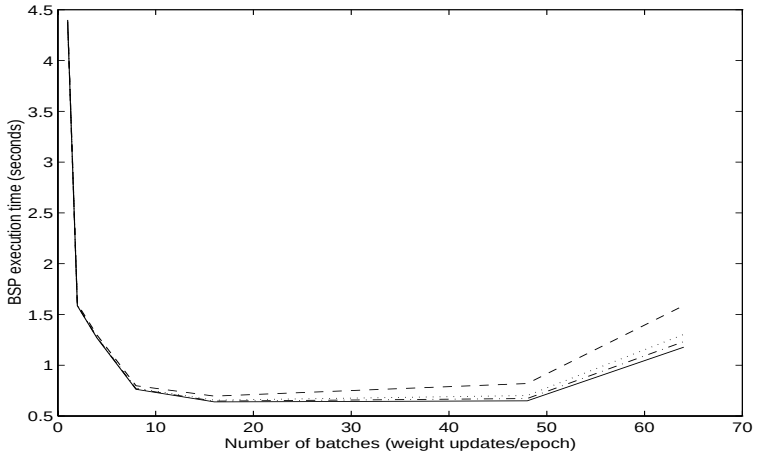

Figure 35: The parallel execution time for training Gaussian mixture model networks on the thyroid database is plotted against the number of batches. Training time is displayed for several different parallel computers. Legend: solid line - CrayT3D, dashed line - IBM SP2, dot-dashed line - SGI Power-Challenge, dotted line - Sun.

each epoch and because exemplar parallelism is nearly optimal for the fewest number of epochs, exemplar parallelism can then be asserted as the optimal strategy for parallelizing neural network training.

Empirically, by using batch learning to train neural networks, I was able to demonstrate a considerable acceleration in the speed of convergence. For the largest data mining training set, I was able to reduce the training execution time by two orders of magnitude as a result of batch learning when training a cross-entropy multi-layer perceptron.

In summary, the results that I have presented in this report demonstrate that exemplar parallelism and batch learning are two theoretically-sound methods for accelerating neural network training. In addition, I assert that both of these methods are particularly well-suited to data mining applications where the training sets are very large and are likely to contain many redundancies. Therefore these two optimization techniques have immense potential for increasing the viability of neural networks for data mining problems.

References

[1] M.A. Arbib, editor. *The Handbook of Brain Theory and Neural Networks.* MIT Press, Cambridge, Massachusetts, 1995.

[2] S. Becker and M. Plumbley. Unsupervised neural network learning procedures for feature extraction and classification. In F. Pineda, editor, *Journal of Applied Intelligence*, volume 6, pages 1–21, Boston, 1996. Kluwer Academic Publishers.

[3] M. Besch and H.W. Pohl. Flexible data parallel training of neural networks using MIMD computers. In *Third Euromicro Workshop on Parallel and Distributed Processing.* San Remo, Italy, 1995.

[4] C.M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[5] G. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie-Mellon University, 1995.

[6] G. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[7] J.M. Buhmann. Data clustering and learning. In Arbib [1], chapter 3, pages 278–282.

[8] N. Carriero and D. Gelernter. Data parallelism and Linda. Technical Report TR-915, Yale University, 1992.

[9] P. Cheeseman and J. Stutz. Bayesian classification (AutoClass): Theory and results. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180. AAAI Press, 1996.

[10] C. Darken, J. Chang, and J. Moody. Learning rate schedules for faster stochastic gradient search. In *Neural Networks for Signal Processing 2 – Proceedings of the 1992 IEEE Workshop*, Piscataway, NJ, 1992. IEEE Press.

[11] C. Darken and J. Moody. Note on learning rate schedules for stochastic optimization. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 832–838, Palo Alto, CA, 1991. Morgan Kaufmann.

[12] J. Dayhoff. *Neural Network Architectures: An Introduction*. Van Nostrand Reinhold, New York, 1990.

[13] K.M. Decker and S. Focardi. Technology overview: A report on data mining. Technical Report TR-95-02, Swiss Scientific Computing Center, 1995.

[14] S.E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie-Mellon University, 1988.

[15] U.M. Fayyad. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11), November 1996.

[16] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In Fayyad et al. [17], pages 1–36.

[17] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[18] M.J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, Massachusetts, 1995.

[19] W. J. Frawley, G. Piatetsky-Shapiro, and C.J. Matheus. Knowledge discovery in databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, chapter 1, pages 1–27. AAAI Press, 1191.

[20] A. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report TR-91-07, University of Virginia, 1991.

[21] J. Han. Data mining techniques. In *ACM-SIGMOD'96 Conference Tutorial*, 1996.

[22] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, New Jersey, 1994.

[23] J.M.D. Hill, P.I. Crumpton, and D.A. Burgess. Theory, practice, and a tool for BSP performance prediction. In *Europar'96*, volume 1124 of *LNCS*, pages 697–705. Springer-Verlag, 1996.

[24] J.M.D. Hill and D.B. Skillicorn. Lessons learned from implementing BSP. In *High-Performance Computing and Networks*, Springer Lecture Notes in Computer Science Vol. 1225, pages 762–771, April 1997. Also appears as Oxford University Computing Laboratory Technical Report TR-96-21.

[25] M.A. Holler. VLSI implementations of learning and memory systems: A review. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 993–1000, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[26] M. Holsheimer and A. Siebes. Data mining: the search for knowledge in databases. Technical Report CS-R9406, CWI, 1994.

[27] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. In *Neural Networks 2*, pages 359–366. 1989.

[28] R.A. Jacobs and M.I. Jordan. A competitive modular connectionist architecture. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 767–773, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[29] M. Joost and W. Schiffmann. Speeding up backpropagation algorithms by using cross-entropy combined with pattern normalization. Technical Report to be submitted, University of Koblenz-Landau, 1997.

[30] M.I. Jordan and C.M. Bishop. Neural networks. In A. Tucker, editor, *CRC Handbook of Computer Science*. CRC Press, Boca Raton, Florida, 1996.

[31] S. Kaski. *Data Exploration Using Self-Organizing Maps*. PhD thesis, Helsinki University of Technology, 1997.

[32] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, page to appear, 1996.

[33] A.H. Kramer and A. Sangiovanni-Vincentelli. Efficient parallel learning algorithms for neural networks. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1, pages 40–48, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[34] Y. le Cun. Generalization and network design strategies. Technical Report CRG-TR-89-4, University of Toronto, 1989.

[35] Y. le Cun, P.Y. Simard, and B. Pearlmutter. Automatic learning rate maximization by on-line estimation of the Hessian's eigenvectors. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 156–163, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[36] O.L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis. In T.F. Coleman and Y. Li, editors, *Large-scale numerical optimization*, pages 22–30. SIAM Publications, 1990.

[37] R.A. Mann. Application of the Kohonen self-organising feature map to radar signal classification. Master's thesis, McMaster University, 1990.

[38] H. Mannila. Data mining: machine learning, statistics, and databases. In *Eighth International Conference on Scientific and Statistical Database Management*, pages 1–8, Stockholm, 1996.

[39] J.L. McClelland and D.E. Rumelhart. *Explorations in Parallel Distributed Processing.* MIT Press, Cambridge, Massachusetts, 1988.

[40] S.J. Nowlan. Max likelihood competition in RBF networks. Technical Report CRG-TR-90-2, University of Toronto, 1990.

[41] S.J. Nowlan. Maximum likelihood competitive learning. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 2, pages 574 – 582, Palo Alto, California, 1990. Morgan Kaufmann Publishers.

[42] S.J. Nowlan. *Soft Competitive Adaptation: Neural Network Learning Algorithms based on Fitting Statistical Mixtures.* PhD thesis, Carnegie-Mellon University, 1991.

[43] D. Ormoneit and V. Tresp. Improved gaussian mixture density estimates using bayesian penalty terms and network averaging. Technical Report FK1-205-95, Technical University of Munich, 1995.

[44] D.A. Pomerleau, G.L. Gusciora, D.L. Touretzky, and H.T. Kung. Neural network simulations at Warp speed: How we got 17 million connections

per second. In *IEEE International Conference on Neural Networks*, pages 143–150, San Diego, 1988.

[45] H. Ritter. Self-organizing feature maps: Kohonen maps. In Arbib [1], chapter 3, pages 846–851.

[46] R.O. Rogers and D.B. Skillicorn. Strategies for parallelizing supervised and unsupervised learning in artificial neural networks using the BSP cost model. Technical Report TR-97-406, Queen's University, 1997.

[47] W. Schiffmann, M. Joost, and R. Werner. Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical Report TR 16/1992, University of Koblenz, 1992.

[48] N. Serbedzija. Simulating artificial neural networks on parallel architectures. In *Computer*, volume 29, pages 53–63, 1996.

[49] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, to appear. Also appears as Oxford University Computing Laboratory, Technical Report TR-15-96, November 1996.

[50] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. In *Computing Surveys*, 1997. to appear.

[51] M. Witbrock and M. Zagha. An implementation of back-propagation learning on GF11, a large SIMD parallel computer. Technical Report CMU-CS-89-208, Carnegie-Mellon University, 1989.

[52] W.H. Wolberg and O.L. Mangasarian. Multisurface methods of pattern separation for medical diagnosis applied to breast cytology. In *Proceedings of the National Academy of Sciences*, volume 87, pages 9193–9196, 1990.

[53] T. S. Wong. Partitioning strategies for parallelizing neural networks. Master's thesis, Queen's University, 1993.

[54] A.L. Yuille and D. Geiger. Winner-take-all mechanisms. In Arbib [1], chapter 3, pages 1056–1060.

NB	BS	LR	EP	STD	Speedup
1	699	0.018	110.18	36.09	0
2	350	0.036	74.16	22.96	1.48
4	175	0.065	55.96	22.06	1.97
7	100	0.11	43.62	11.25	2.53
14	50	0.19	34.6	9.79	3.18
28	25	0.29	31.36	17.56	3.51
70	10	0.57	21.24	5.66	5.19
140	5	0.76	21.92	6.79	5.03
233	3	0.7	22.56	5.33	4.88
699	1	0.7	23.4	6.84	4.71

Table 14: Batch learning training results for the cross-entropy multi-layer perceptron on the breast cancer database.

A Batch Learning Results for Supervised Networks

Legend:

- *NB*: number of batches (number of weight updates for each epoch)
- *BS*: batch size (number of examples in each batch)
- *LR*: the optimal learning rate for each batch size
- *EP*: average number of epochs required for convergence
- *STD*: the standard deviation of the number of training epochs for each batch size
- *Speedup*: the speedup attained over the rate of convergence of the deterministic neural network

B Batch Learning Results for Unsupervised Networks

Legend:

NB	BS	LR	EP	STD	Speedup
1	3772	0.005	598.94	115.39	0
2	1886	0.01	302.88	95.45	1.98
4	943	0.018	172.1	27.63	3.48
7	539	0.028	101.84	14.2	5.88
14	270	0.05	64.92	11.66	9.23
23	164	0.07	53.78	7.574	11.14
42	90	0.07	52.32	6.976	11.44
60	63	0.08	52.45	6.357	11.42
122	31	0.08	52.94	7.702	11.31
270	14	0.08	50.54	6.866	11.85

Table 15: Batch learning training results for the cross-entropy multi-layer perceptron on the thyroid database.

NB	BS	LR	EP	STD	Speedup
1	30162	0.0001	294.3	15.68	0
2	15081	0.0002	146.2	8.49	2.01
4	7541	0.0004	73.5	3.68	4.00
8	3771	0.0008	42	1.49	7.01
10	3017	0010	33.3	2.58	8.84
16	1885	0.0016	23.7	1.41	12.42
32	943	0.0032	13.7	0.67	21.48
64	472	0.0064	8.5	0.85	34.62
128	236	0.0128	6.3	0.67	46.71
256	118	0.02	5.3	0.48	55.53
512	59	0.02	5.1	0.5676	57.71
1024	30	0.02	5	0.667	58.86
2056	15	0.02	5.3	0.483	55.53

Table 16: Batch learning training results for the cross-entropy multi-layer perceptron on the US census database.

NB	BS	LR	EP	STD	Speedup
1	699	1.0	18	7.00	0
2	350	1.0	9.2	2.25	1.96
4	175	1.0	6.3	1.95	2.86
8	88	0.9	4.2	1.03	4.29
16	44	0.85	3.3	0.68	5.45
32	22	0.5	3.2	0.92	5.63
40	18	0.45	3.9	1.33	4.62
48	15	0.4	4.0	1.45	4.50
64	11	0.3	5.3	2.35	3.40

Table 17: Batch learning training results for Gaussian mixture model networks on the breast cancer database.

- *NB*: number of batches (number of weight updates for each epoch)
- *BS*: batch size (number of examples in each batch)
- *LR*: the optimal learning rate for each batch size
- *EP*: average number of epochs required for convergence
- *STD*: the standard deviation of the number of training epochs for each batch size
- *Speedup*: the speedup attained over the rate of convergence of the deterministic neural network

NB	BS	LR	EP	STD	Speedup
1	3772	1.0	13.8	10.22	0
2	1886	1.0	5.0	2.22	2.76
4	943	1.0	4.0	1.25	3.45
8	472	0.9	2.4	0.55	5.75
16	236	0.9	2.1	0.37	6.57
32	118	0.7	2.0	0.13	6.90
48	79	0.7	2.0	0.2	6.90
64	59	0.6	3.6	6.89	3.83

Table 18: Batch learning training results for Gaussian mixture model networks on the thyroid database.