AN EVOLUTIONARY METHOD FOR TRAINING AUTOENCODERS FOR DEEP LEARNING NETWORKS

A Thesis

Presented to

The Faculty of the Graduate School

At the University of Missouri

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

SEAN LANDER

Dr. Yi Shang, Advisor

MAY 2014

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

AN EVOLUTIONARY METHOD FOR TRAINING AUTOENCODERS FOR DEEP LEARNING NETWORKS

Presented by Sean Lander

A candidate for the degree of

Master of Science

And hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Yi Shang

Dr. Dong Xu

Dr. Jianlin Cheng

ACKNOWLEDGEMENTS

I would like to first thank my advisor, Dr. Yi Shang, for all of his support, which began before I had even been accepted into this program. His help and guidance over the years has helped me fall in love with the prospects of what academia has to offer and without his help I would not have been able to ever get to where I am now.

I would also like to thank my cohorts and colleagues in Dr. Shang's mobile sensing lab, both those who have left and those who have just begun. Being able to work with such enjoyable people has helped me develop both as a researcher and a developer.

My appreciation and love also goes out to my wife, Becca, who is the entire reason I'm in Columbia, as well as my parents who have always taught me to see things through to the end, bar none.

Finally I would like to thank my committee members Dr. Dong Xu and Dr. Jianlin Cheng for their support on this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Introduced in 2006, Deep Learning has made large strides in both supervised an unsupervised learning. The abilities of Deep Learning have been shown to beat both generic and highly specialized classification and clustering techniques with little change to the underlying concept of a multi-layer perceptron. Though this has caused a resurgence of interest in neural networks, many of the drawbacks and pitfalls of such systems have yet to be addressed after nearly 30 years: speed of training, local minima and manual testing of hyper-parameters.

In this thesis we propose using an evolutionary technique in order to work toward solving these issues and increase the overall quality and abilities of Deep Learning Networks. In the evolution of a population of autoencoders for input reconstruction, we are able to abstract multiple features for each autoencoder in the form of hidden nodes, scoring the autoencoders based on their ability to reconstruct their input, and finally selecting autoencoders for crossover and mutation with hidden nodes as the chromosome. In this way we are able to not only quickly find optimal abstracted feature sets but also optimize the structure of the autoencoder to match the features being selected. This also allows us to experiment with different training methods in respect to data partitioning and selection, reducing overall training time drastically for large and complex datasets. This proposed method allows even large datasets to be trained quickly and efficiently with little manual parameter choice required by the user, leading to faster, more accurate creation of Deep Learning Networks.

1. INTRODUCTION

1.1 Neural Network and Deep Learning Optimization

Artificial Neural Networks (ANNs) have been a mainstay of Artificial Intelligence since the creation of the perceptron in the late 1950s. Since that time, it has seen times of promising development as well as years and decades of being ignored. One of the key reasons that ANNs first lost their appeal was due to the long run times and memory requirements of their training process. Advances in system speed and distributed computing, as well as the discovery of backpropagation [11], made large strides toward removing these barriers and caused a new resurgence in ANN popularity in the 1980s.

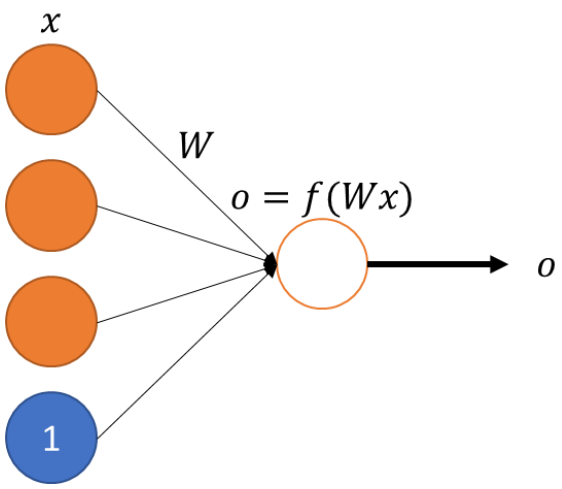

Figure 1. Basic Perceptron

As interest grew, a different issue became apparent: the ANN's inability to find a global optimum. Increases in data size, mixed with the need for multiple runs to find a globally optimal solution, led to attempts to beat locality using other optimization

methods such as the genetic algorithm [4]. Different attempts at using genetic and evolutionary algorithms to optimize ANNs continued over the years, not only trying to achieve a global maximum, but also in order to find optimal hyper-parameters such as network size, the size of hidden layers, activation algorithms, and even different levels of network connectedness [2] [3] [8].

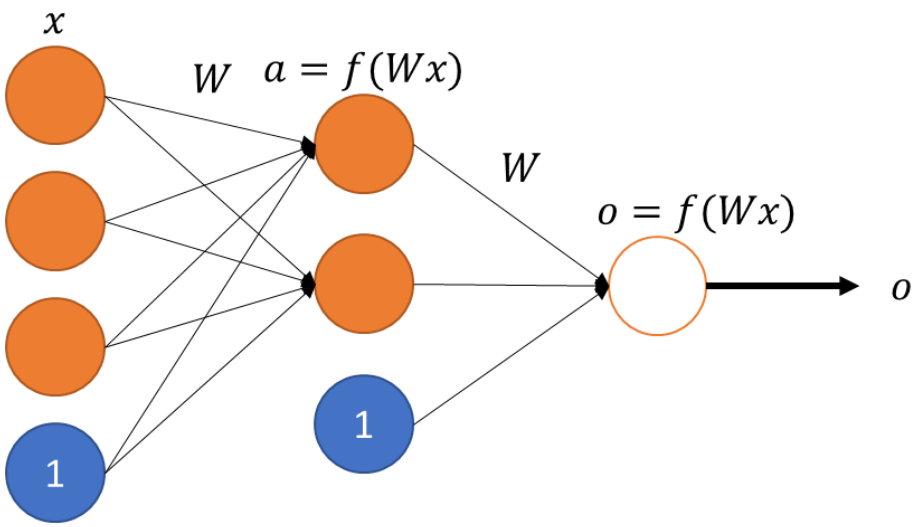

Figure 2. Artificial Neural Network with 1 hidden layer

In 2006, a method was proposed to improve the abilities of backpropagation. This allowed for ANNs with more than two layers, and, more importantly, high accuracy feature abstraction [5]. These Deep Learning/Belief Networks (DLNs) were able to improve both classification and feature creation/abstraction performance over almost all other methods, yet they still suffered from the basic issues surrounding ANNs. Work has since been done to improve ANN training methods, from hyper-parameter selection

guidelines to comparisons on many different optimization methods [6] [7] [9], but these tend to be either problem-specific or highly time-intensive.

1.2 Related/Historical Work

1.2.1 Neural Network Parameter Optimization

While work on optimizing ANNs has since faded due to the popularity of DLNs, they are both, in essence, one and the same. Thus, optimization of DLNs is able to build off of the foundation of ANN research. As far back as 1989, it had been shown that evolutionary approaches to the training of an ANN improved upon traditional methods. Montana and Davis (1989) showed that allowing ANNs to share information, be it weights or nodes, as well as allowing for mutation of the system, improves the overall error when compared with general backpropagation. Though these tests were done before more diverse versions of backpropagation were proposed, such as using conjugated gradient or L-BFGS as opposed to basic gradient descent, it still shows promise in improving the overall quality of an ANN. [4]

Since Montana and Davis's paper others have continued the idea of using genetic and evolutionary techniques to better train and optimize ANNs. In 1997 Yao, et al. (1997) proposed a system, EPNet, for evolving and training ANNs using evolutionary programming. EPNet attempts to better share the behaviors of the parents with the offspring and also includes partial training within generations in order to reduce noise in the children. The research behind EPNet also studied mutation on the node level, preferring node deletion to addition, as well as minimizing the error caused by adding nodes with randomly initialized weights. This was accomplished by sampling for deletion

first, ending mutation should deletion be chosen. Their work also touches on an issue commonly found in evolutionary ANNs, competing conventions, where two ANNs may model the same function by containing the same hidden nodes and weights but with a different ordering of hidden nodes. [2]

Competing conventions is brought up once again by Yang and Kao (2001), who mention that the number of competing conventions grows exponentially with the number of neurons, or hidden nodes. Their method of evolution takes the form of three levels of mutation, each feeding into the next. In this case the encoding of the network differs from other methods in that its chromosomes represent not only the weights but also the different mutation rates to be used, bridging the gap between weight optimization and hyper-parameter optimization. [3]

1.2.2 Deep Learning Parameter Optimization

In 2006 Hinton et al. revived an earlier concept called Deep Networks, in this case stochastic Deep Belief Networks (DBNs), which had been unfeasible to train until this time due to hardware restrictions. One of the major drawbacks of ANNs had been that they were forced to be relatively shallow, at most two hidden layers deep. This is due to backpropagation having diminishing returns the further back it propagated the error. Hinton et al. proposed and introduced autoencoders, single layer ANNs which would reconstruct their input in the output as opposed to classify. Using this method they were able to create a new form of unsupervised learning, allowing ANNs to create high level abstractions of input. This reduced input could be sent to a classifier in a way similar to Principle Component Analysis or even provide pre-trained hidden layers for a DLN.

Backpropagation on this new network, also known as "fine tuning," was much more successful than using an untrained network and allowed for deeper networks than had been previously achieved. [5]

Research then began on analyzing Deep Networks and their layer-wise training approach. Bengio et al. (2007) studied performance of Deep Networks using Hinton's Restricted Boltzmann Machine (DBN) approach. They concluded that improvement was caused by the DLN's power to abstract new features from the data at all levels, something previous ANNs were unable to achieve [9]. Ngiam et al. (2011) continued along this line of study, collecting data on a variety of backpropagation-based learning algorithms including Stochastic Gradient Descent (SGD), Conjugate Gradient (CG), and Limited Memory BFGS (L-BFGS). While CG and L-BFGS are both superior to SGD, it was observed that they scaled poorly with data size as they require batch operations, as opposed to the online ability of SGD. Other methods tested include using MapReduce clusters and GPUs in order to utilize more recent shifts in large scale computing. This inclusive analysis of different techniques and architectures of the DLN training environment resulted in a large push toward distributed GPU-based systems for training of DLNs, as well as a renewed interest in Convolutional Neural Networks. [6] [10]

Though analysis on training methods and hardware environments has been studied, many of the previous drawbacks of ANNs continue to exist in DLNs, specifically the need to manually design and optimize the architecture and hyper-parameters of the ANN/DLN. Bergstra et al.'s (2011) work on hyper-parameter optimization shows that much of the design of an ANN, in this case DLNs, must be done through a mixture of trial-and-error and human optimization. By using greedy search algorithms across an

array of different DLN parameters, everything from the number of layers and the learning rate to whether or not to preprocess the data, Bergstra has shown that an automated approach can be taken to hyper-parameter optimization. Unfortunately, this optimization method suffers from the same issue of local minima as ANNs themselves, leaving room for improvement in the selection and modification of ANNs and DLNs. [7]

2. METHODS

2.1 Neural Networks

2.1.1 Backpropagation using Gradient Descent

Backpropagation is a powerful algorithm with roots in gradient descent, allowing a complex derivative over multiple levels to be run in in $O(N * M)$ time, where N is the size of the input vector and H the size of the hidden layer. An ANN can be described by the variables **W**, the matrices containing the weights between layers, and **b**, and the bias term for each non-terminal layer. Given *m* training samples:

$$(1) \quad \{(x_1, x_M), \dots, (y_1, y_M)\}$$

For each sample (x, y) where $x, y \in [0,1]$ calculate its error:

$$(2) \quad E(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2$$

For all *m* training samples the total error can be calculated as:

$$(3) \quad J(W, b) = \frac{1}{M} \sum_{m=1}^{M} E(W, b; x_m, y_m)$$

This will give an unbiased cost for the current network ***(W,b)***. This equation is prone to over-fitting, however, which can harm the ANN's ability to generalize well. In order to control this, we add a weight decay variable λ to get the final equation:

$$(4) \quad J(W, b) = \left[\frac{1}{M} \sum_{m=1}^{M} E(W, b; x_m, y_m) \right] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2$$

This new end term will compute the summed square of all weights and add a scaled version of that value to the overall cost of the ANN being scored. One key principle of

this decay function is its simple to calculate derivative, a necessity for the backpropagation algorithm.

Now that a cost function has been specified, gradient descent is applied. Once the gradient has been calculated, it can be subtracted from the current weights and biases given some learning rate α:

$$(5) \quad W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}$$

$$(6) \quad b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

This method has a flaw, however, in that it can become stuck such that it will never converge. This is because the algorithm contains no memory, so if the gradient from one iteration to the next negate each other they will continue looping indefinitely. In order to counteract this, a momentum function can be added, otherwise known as the Conjugated Gradient. Using the previous gradient in this way can iteratively narrow down on the minimum in cases where it would otherwise become stuck.

$$(7) \quad \Delta W_{ij}^{(l)t} = \left(\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}\right) + \beta \Delta W_{ij}^{(l)t-1}$$

$$(8) \quad \Delta b_i^{(l)t} = \left(\frac{\partial J(W,b)}{\partial b_i^{(l)}}\right) + \beta \Delta b_i^{(l)t-1}$$

$$(9) \quad W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha(\Delta W_{ij}^{(l)t})$$

$$(10) \quad b_i^{(l)} = b_i^{(l)} - \alpha(\Delta b_i^{(l)t})$$

Using the method above it is possible to train an ANN in either online or batch mode. In online training each gradient is calculated and applied in sequential order such

that no two data points ever train the same ANN. In batch mode every data point in the training set is run through the same ANN and its error calculated. The sum of those errors is then used to update the ANN and the process repeated until convergence. From this point on only batch training will be considered.

For the following equations:

- L is the number of hidden layers
- f is the activation function chosen, in this case sigmoid
- $z^l = W^l . a^{l-1} + b^l$
- z^0 = x
- $a^l = f(z^l)$

$Given\ a\ set\ of\ examples\ \forall i\ \left(x^i, y^i\right), i\ \in m\ and\ ANN(W, b, L, f)$

$Repeat\ until\ convergence$:

1. $For\ each\ sample\ m$:

   $a_{(m)}^{(1)}, \dots, a_{(m)}^{(L)}, z_{(m)}^{(1)}, \dots, z_{(m)}^{(L)} = Feedforward\left(W, b; x_{(m)}\right)$

2. $For\ each\ unit\ i\ in\ layer\ L$:

   $$\delta_i^{(L)} = \frac{\partial}{\partial z_i^{(L)}} \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 = -\left(y_i - a_i^{(L)}\right) f'\left(z_i^{(L)}\right)$$

3. $For\ l = L - 1, \dots, 1\ and\ each\ unit\ i\ in\ layer\ l$:

   $$\delta_i^{(l)} = \sum_{j=1}^{S_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} f'(z_i^{(l)})$$

4. $Compute\ the\ partial\ derivitives$:

   $$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

5. *Compute the overall cost*:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[\frac{1}{m}\sum_{m=1}^{M} \frac{\partial}{\partial W_{ij}^{(l)}} J\left(W, b; x_{(m)}, y_{(m)}\right)\right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{m}\sum_{m=1}^{M} \frac{\partial}{\partial b_i^{(l)}} J\left(W, b; x_{(m)}, y_{(m)}\right)$$

6. *Then update the weights*:

$$\Delta W_{ij}^{(l)t} = \left(\left[\frac{1}{m}\sum_{m=1}^{M} \frac{\partial}{\partial W_{ij}^{(l)}} J\left(W, b; x_{(m)}, y_{(m)}\right)\right] + \lambda W_{ij}^{(l)}\right) + \beta \Delta W_{ij}^{(l)t-1}$$

$$\Delta b_i^{(l)t} = \left(\frac{1}{m}\sum_{m=1}^{M} \frac{\partial}{\partial b_i^{(l)}} J\left(W, b; x_{(m)}, y_{(m)}\right)\right) + \beta \Delta b_i^{(l)t-1}$$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\left(\Delta W_{ij}^{(l)t}\right)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\left(\Delta b_i^{(l)t}\right)$$

2.1.2 Architecture Selection

One of those most difficult parts of creating a neural network is choosing the correct parameters for the problem at hand. Due to the inexhaustibly large number of combinations of parameters, mixed with the time required to train a single large ANN on a large dataset, it is infeasible to sample all options and make a data-driven decision for every problem. Some of the parameters most important to the training are: learning rate, momentum, number of layers, and layer size. A dynamic learning rate can be used to alleviate the issue of tuning and retesting, and the learning rate and momentum together

work to balance out the overall process when used correctly. The actual structure of the ANN is more difficult, however, and is addressed further on in this paper.

2.2. Deep Learning Networks

2.2.1 Probabilistic Deep Belief Networks – Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) were the basis of Hinton's original Stacked Autoencoder system, otherwise known as a Deep Belief Network. RBMs are trained in a stochastic way, using Monte Carlo sampling when moving from one layer to another. The basic process is to use the weights between layers, as well as an activation function, to probabilistically "turn on" nodes, giving them an activation value of either 0 or 1. Sample data is read in and run against the current weights, activating the hidden layer. In a Markov Chain-like process, those hidden layers are then fed back through the weights toward the input layer. This process is repeated for two to three cycles and the error finally calculated between the initial input and the reconstruction. This error is used to update the weights, similar to the method for backpropagation, and the run restarted. This is a simple and powerful algorithm in the toolbox of training Deep Belief Networks.

2.2.2 Deterministic Deep Learning Networks – ANN based Autoencoders

Though powerful, RBMs are just one of the ways to create DLNs. ANN-based autoencoders can be trained using the backpropagation technique above by replacing *y*, the target label, with *x* in order to rebuild the original input.

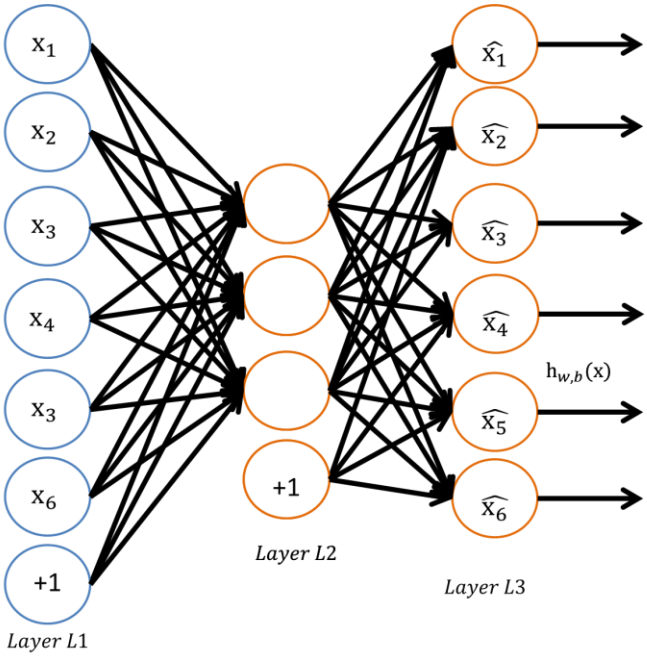

Figure 3. Basic structure of a deterministic autoencoder

In this way, it is possible to build an abstraction of the input data. Once the error rate has been reduced, meaning the reconstruction (output) has a minimal difference from the input, the activation values of the hidden layer will have become an abstract representation of the input data, proven by its ability to rebuild the input. This new representation may be used in the same way as other feature-reduction techniques such as Principal Component Analysis. Hinton et al. (2006) has shown that autoencoder-based feature-reduction can do a much better job than PCA for both clustering and input reconstruction [5].

Autoencoders offer benefits besides the improved reconstruction performance. Autoencoders, unlike some other feature-reduction techniques, provide range-bound output, making them an ideal step in data pre-processing. PCA, by comparison, has no

restrictions during its data projection, meaning output may be out of the accepted input range required by your classifier. While apparently minimal, this assurance of data integrity can be important in large-scale systems and applications.

The time required for training remains a major drawback of this ANN-based approach. Using Autoencoders will effectively increase the training time near linearly with respect to the number of layers being created. Autoencoders also continue to suffer from the issues surrounding ANNs as a whole, mainly the inability to deal with local minima and the difficulty in choosing the correct parameters/architecture of the system.

2.3. Genetic and Evolutionary Algorithms and EvoAE

In both ANNs and autoencoders, avoiding local minima and architecture choice can both be dealt with using evolutionary approaches. Unlike past ANNs, however, we are not looking to maximize classification accuracy but instead are focused specifically on reducing error as much as possible. The purpose has technically changed, but backpropagation works on error, not purpose, so it can be used without alteration in the new environment without issue.

Purpose comes into play in the construction of the chromosomes to be used in the genetic process. Whereas previously researchers have used everything from individual weights to learning rates, autoencoders have a very specific use: feature abstraction. Each hidden node in an autoencoder is an abstraction of the input data and represents a higher level feature found in the data as a whole. By using this knowledge it seems logical to use Montana and Davis' method of nodes-as-genes when creating the chromosomes for each autoencoder [4].

With the chromosomal representation finished a crossover method must be chosen. Crossover can be done many different ways, but in this case consists of a simple bit-map function over the smallest parent, with the remaining features from the larger parent being passed on to the first child.

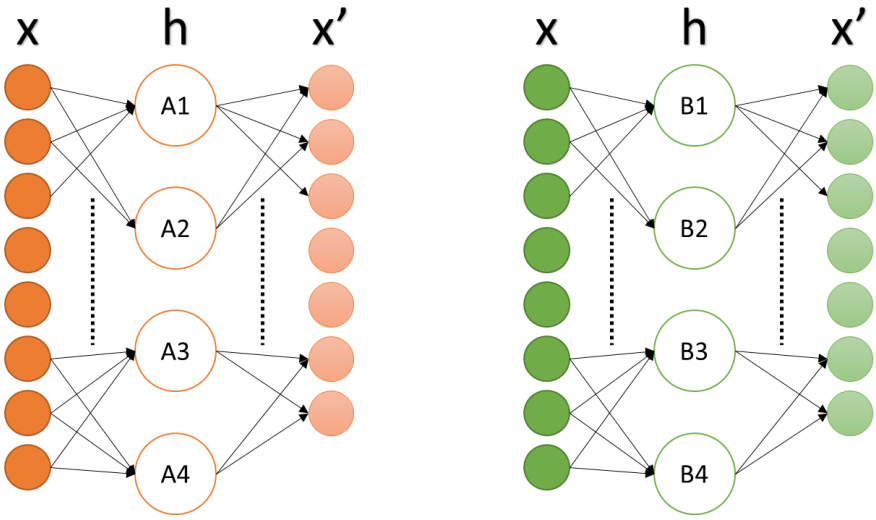

Figure 4. Population consisting of two evolutionary autoencoders A and B

After crossover is complete, the Evolutionary Autoencoder (EvoAE) has a chance for mutation. Though Yao and Liu's (1997) proposal to prefer deletion over addition is sound when attempting to minimize a full ANN, this method gives a 50/50 chance of mutation occurring. This allows for a more fully-explored feature space. While speed is important in the training of autoencoders, the ability to rebuild input is more so.

In order to remove the issue caused by adding a new, untrained node to the autoencoder a third parent is randomly selected to donate their information to the offspring. This will remove the extra noise generate by adding a newly initialized feature as well as expand the chance of shared information outside of just the two parents.

Encoding the hidden nodes in this way allows for the exploration not only of a larger variety of features, but also selection of different autoencoder structures, as each EvoAE may contain a different number of hidden nodes. This is an important feature since it allows for a dual optimality search, as some EvoAE structure/weight initialization combinations will perform better than others. Because we are focusing primarily on optimal reconstruction there is no good argument for fixed-structure autoencoders other than speed of training.

Crossover and mutation are not enough to find a good minimum, however, and local search must be added as a final step during every generation. In local search, backpropagation is used for a set number of epochs. In this way, each set of features is able to make updates based on its newly-neighbored features to improve the EvoAE's reconstruction ability. After a set number of epochs have passed, the overall error of the population is calculated in order to check for convergence and, if the convergence criteria is not met, continue onto the crossover phase once more.

2.4. Distributed Learning with Mini-batches

Speed of training continues to be a large issue with ANNs, and it becomes an ever larger problem with the amount of training data available. While ANNs normally train on labeled data, DLNs are not limited in such a way, and the amount of unlabeled data available is extremely large by comparison. It is because of this that distributed systems and techniques such as MapReduce have become so prevalent in the field of machine learning. DLNs are able to harness the power of distributed systems at a near-linear level in terms of speed increase [6], which allows for faster training and more random restarts.

Large-scale evolutionary and genetic techniques can have very large populations, however, with each training an individual system, which can cause a substantial drop in training speed unless there is hardware capable of supporting it.

Hinton et al. (2006) were able to bypass the issues surrounding memory and speed limitations by using a mini-batch technique [5]. By reducing the data from an input of 60,000 samples to 600 batches of 100 samples per batch it was possible to avoid memory limitations and the slowdowns that can accompany them. This method can be used to speed up individual autoencoders, but doesn't remove the fact of having to train multiple autoencoders simultaneously (or in serial, as is the case in this paper).

A technique is proposed, then, which can train a large population of autoencoders at the same speed as a single autoencoder, given a large enough dataset. As EvoAEs train at the feature level it may be possible to successfully train a single EvoAE in the population using only a fraction of the dataset. It is the features that matter for EvoAE, as opposed to the label, so data distribution based on label is not as important as the overall cohesion of the data itself. Because of this, larger data can be preferred using this method, as an increase in data size just means an increase in population size.

3. PERFORMANCE TESTS

3.1 Testing Setup and Default Parameters/Measurements

All tests took place on a Lenovo Y500 laptop (Intel i7 3rd gen 2.4 GHz, 12GB Ram). All tests were run using 64bit Python 3.3 and the packages specified on this project's Github. Each EvoAE configuration is tested using 5 independent runs. This created a set of 5 "best" AEs both in terms of reconstruction error and training accuracy via linear SVM, as specified in the Key Performance Indicators (KPI) section below. The data shown for "best" in the result graphs are based only on a single AE from each of the 5 runs.

The performance test parameters are as follows:

Parameter	**Wine**	**Iris**	**Heart Disease**	**MNIST**
Learning Rate	0.1	0.1	0.1	0.1
Momentum	2	2	2	2
Weight Decay	0.003	0.003	0.003	0.003
Population Size	30	30	30	30
Generations	50	50	50	50
Epochs/Gen	20	20	20	20
Hidden Size	**32**	**32**	**12**	**200**
Hidden Std Dev	**NULL**	**NULL**	**NULL**	**80**
Hidden +/-	**16**	**16**	**6**	**NULL**
Mutation Rate	**0.5**	**0.5**	**0.5**	**0.1**
Train/Validate	80/20	80/20	80/20	80/20

Dynamic Learning Rate:

The learning rate used in these samples has a diminishing weight in respect to the number of epochs run using the formula:

$$\alpha' = a * (1 + \exp(-epoch))$$

$$W' = W - \alpha' * \Delta W$$

This causes large weights changes in early iterations, while quickly reducing the learning rate to the user-determined value over time.

Convergence Criteria:

Each generation has its validation's sum-squared-error compared against the current best validation sum-squared-error to prevent overfitting. If three (3) generations fail to make an improvement to the population's overall validation error the most recent best population is used as the final result. This is a shared convergence criteria across all tests.

Key Performance Indicators:

The key performance indicators (KPIs) for these tests takes the following factors into account: training time, reconstruction error, classification accuracy of a linear SVM on the reconstructed data.

3.2 Baseline Performance – Single Autoencoder

The baseline for comparison is a single AE with thirty random initializations (30 random initializations). Each AE runs until the convergence criteria is met using two configurations for learning rate: base learning rate and learning rate/10. The use of two settings is due to generic AEs needing manual tweaking of the learning rate for optimal

training. The AE structure is not randomized, instead using the hidden size specified in the table above.

3.3 EvoAE with Full Data

EvoAE with Full Data will use the EvoAE approach with the parameters listed above for the specific dataset. In this case, Full Data means that each EvoAE will train against the full dataset on each epoch.

3.4 EvoAE with Mini-batches

EvoAE with Mini-batches uses the following formula to determine how to split the batches:

$$Batch\ size = \frac{trainingsize}{populationsize}$$

This allows for even comparison of speed between the mini-batch and Evo-batch tests, as well as reducing the memory footprint of the program.

3.5 EvoAE with Evo-batches

EvoAE with EvoBatches uses the technique described in section 2.4, whereby each member of the population works on its own section of the training data. The data is split the same equation as in section 3.4 above, allowing for near-even distribution of data among all members of the population.

3.6 Post-training configurations

Due to mini-batches and Evo-batches only training on a small fraction of the data for each generation, post-training is also implemented and tested on. Two methods of

post-training are used: full data and batch data. For full data post-training, each AE is trained for one final cycle, for the full number of epochs, with the full data set as input. In batch data post-training each AE is trained against every batch, with a full number of epochs of gradient descent used for each batch. In essence, full data post-training institutes a final generation sans crossover or mutation, while batch data post-training institutes a number of generations equal to the population size sans crossover or mutation. This is done so that each AE is able to train against all sample points regardless of when convergence happens, as an early convergence may lead to some data never being seen by the system. Both post-training configurations are tested along with no post-training.

4. RESULTS

4.1 Small Datasets

4.1.1 UCI Wine

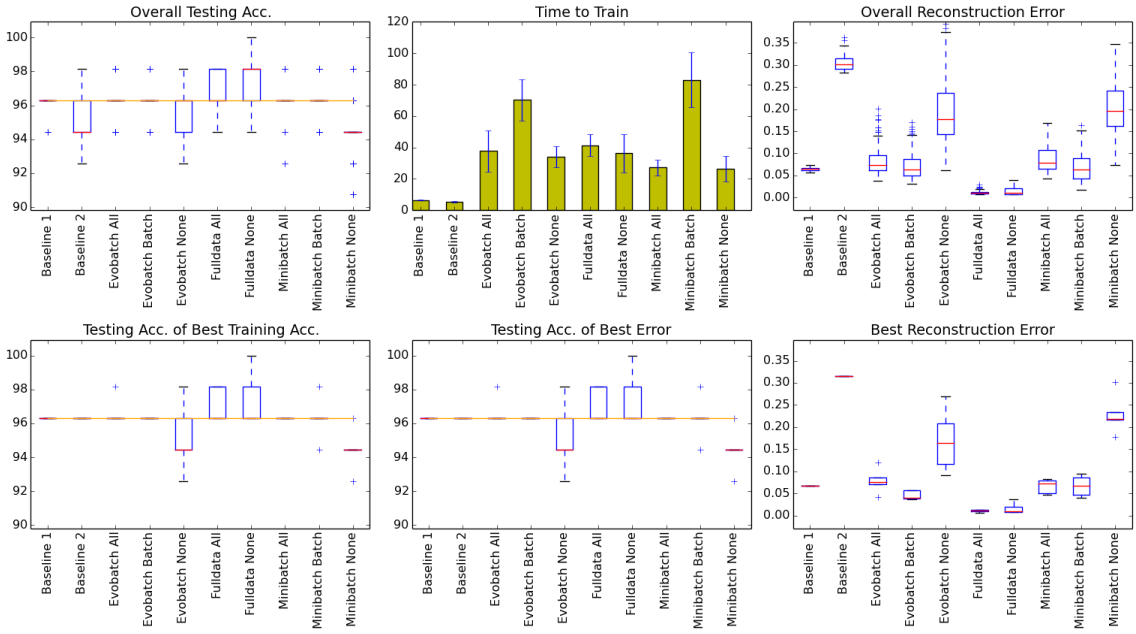

Figure 5. Comparative measure of accuracy, error and speed for UCI Wine dataset

"These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines." [13]

The UCI Wine dataset contains 178 samples with 13 features and 3 classes. Its best training was accomplished using full data with no post-training, though the best error-to-time ratio was achieved using baseline 1.

4.1.2 UCI Iris

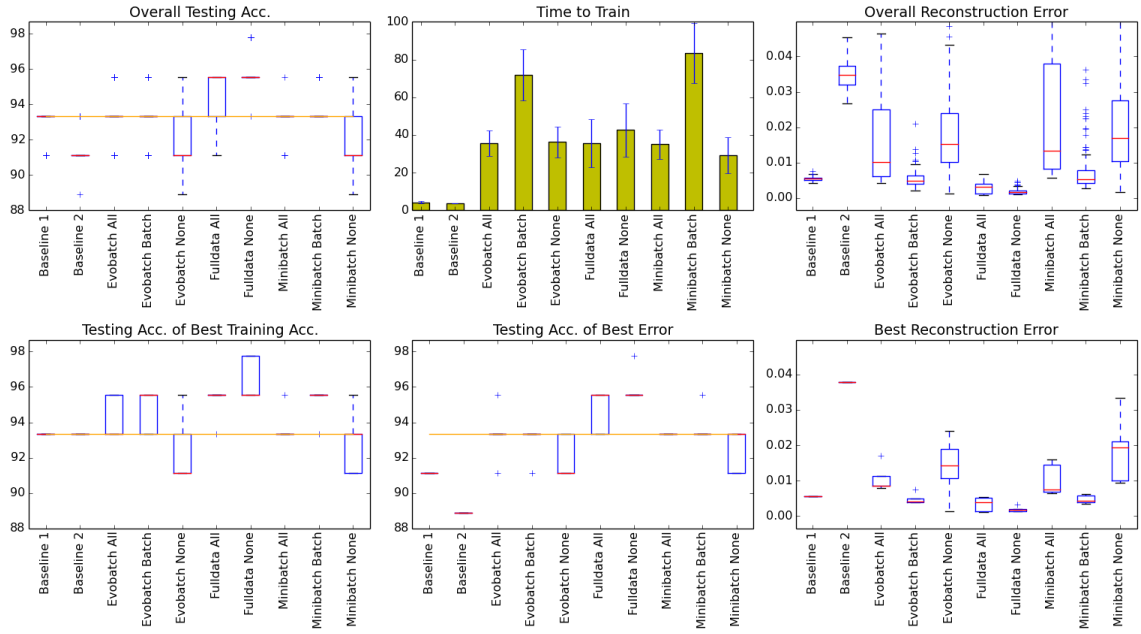

Figure 6. Comparative measure of accuracy, error and speed for UCI Iris dataset

"This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other." [13]

The UCI Iris dataset contains 150 samples with 4 features and 3 classes. As with the UCI Wine dataset the best reconstruction can be found using the full data with no post-training configuration, as well as baseline 1 being the best error-to-time ratio. Unlike UCI Wine, however, EvoAE with full data was able to markedly increase classification after feature abstraction, especially when looking at the AEs with the best training accuracy and reconstruction errors.

4.1.3 UCI Heart Disease

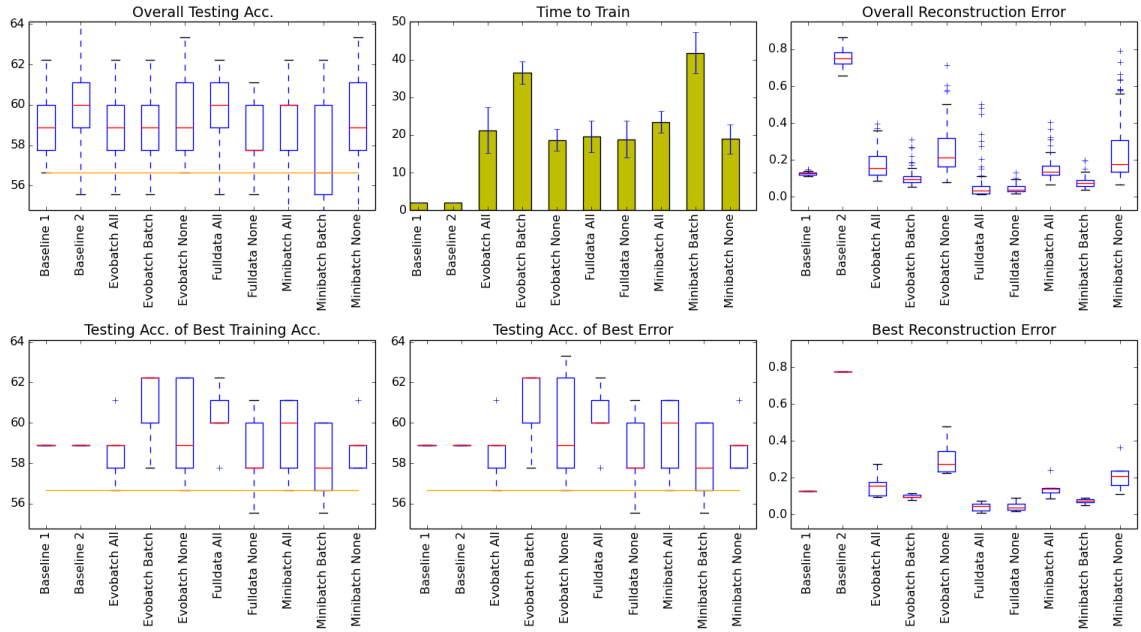

Figure 7. Comparative measure of accuracy, error and speed for UCI Wine dataset

"This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "goal" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4. Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0)." [13]

The UCI Heart Disease dataset contains 297 samples with 13 features and 5 classes, not the 2 classes referenced in the summary. This is the first dataset to diverge from the full data trend above, with both mini-batch and evo-batch with batch post-training configurations delivering close to the same reconstruction errors. However, the time-to-train both of these methods took around double the full data configuration's time-

to-train. This was also an instance where abstracting the data into a higher number of features caused an increase in the ability to correctly classify the data when compared to the baseline.

4.1.4 MNIST 6k-1k Reduced Dataset

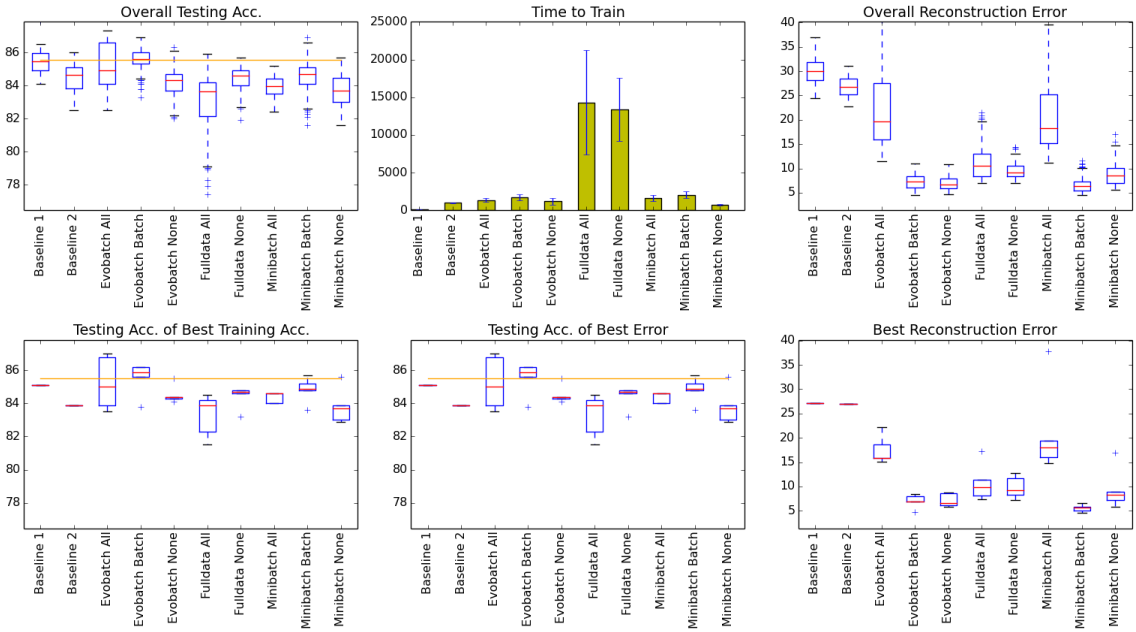

Figure 8. Comparative measure of accuracy, error and speed for reduced MNIST dataset, with 6k training and 1k testing samples

"The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image." [14]

The basic MNIST dataset, as stated above, contains 60k/10k samples for training/testing, respectively, with 784 features (pixels) and 10 classes. In this test 1/10th of the data was used, taking only the first 6000 training samples and 1000 testing

samples. For classification this would cause issues, as the ordering of the samples would cause deviations in data distribution, but as this is specific to reconstruction ability the issue was overlooked in order to decrease testing times.

Of note in this test is both the quality and speed of both mini-batches and evo-batches. Both data size and feature size are much larger than in the previous three datasets, which is apparent when looking at the time-to-train, as a single round of gradient descent is a (6000x784)x(784x~200)x(~200x784)x(784x6000) matrix multiplication on a single 8-core processor.

The reconstruction error for the post-training configurations are interesting as well, with a large increase in reconstruction error after one final round of gradient descents using the full dataset (all errors are the average of the sum squared error in order to normalize). Also interesting is that batch data post-training does not seem to effect the overall reconstruction ability for evo-batches, which is not the case with mini-batches.

4.2 Medium Datasets

4.2.1 MNIST 24k-4k Reduced Dataset

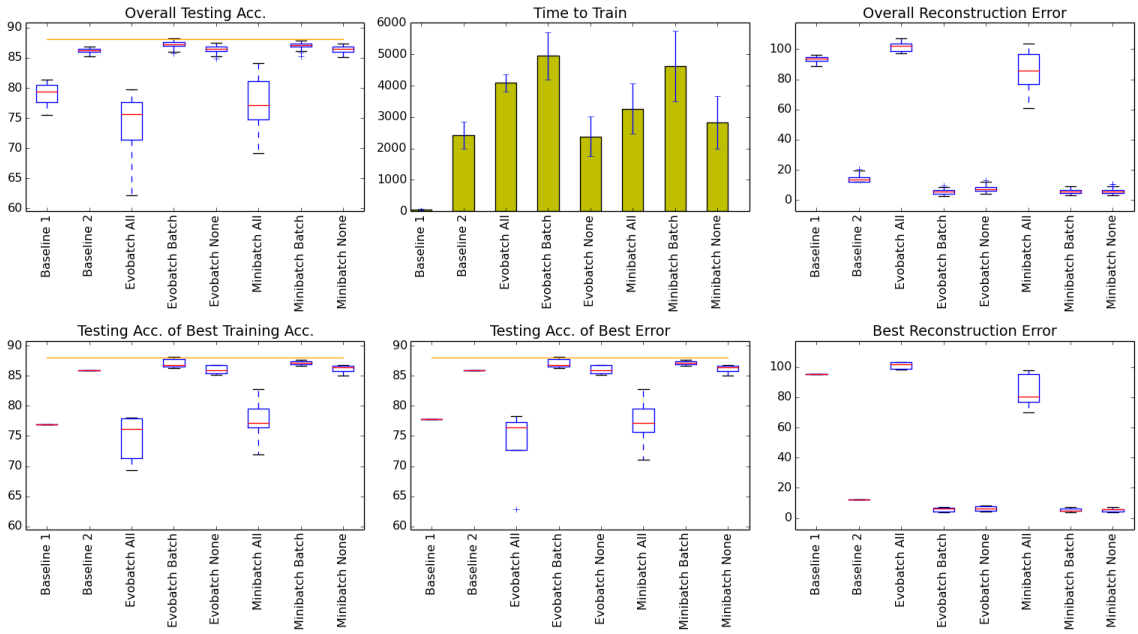

Figure 9. Comparative measure of accuracy, error and speed for reduced MNIST dataset, with 24k training and 4k testing samples

In this test another reduction of the MNIST data was used, but not as drastic as in the above case. It was enough to make training with full data infeasible with the current setup, however, due to that being equivalent 30 AEs being trained 5 times each. As such full data training was removed from this test.

The most noticeable change from previous datasets is that the speed-vs-quality of post-training became much more apparent. Evo-batch and mini-batch with no post-training are almost equivalent to using batch data post-training, while only taking 2/3 the time in the case of mini-batch and less than ½ for evo-batch with near identical reconstruction error both for all cases and best cases. Evo-batch also had a faster training

time than mini-batch, with equivalent reconstruction error and testing accuracies. The ability to parallelize both under different circumstances, both large single machine and in a distributed environment, is a preferable next step.

One important characteristic of this run is when looking at the time-to-train of baseline 2 compared with evo-batch with no post-training. By restricting each AE to train on only a single partition of the data it is possible to obtain quality training across a large number of AEs in the same time as a single AE using traditional methods. Both the average and best reconstruction error of this population are improved over traditional training methods.

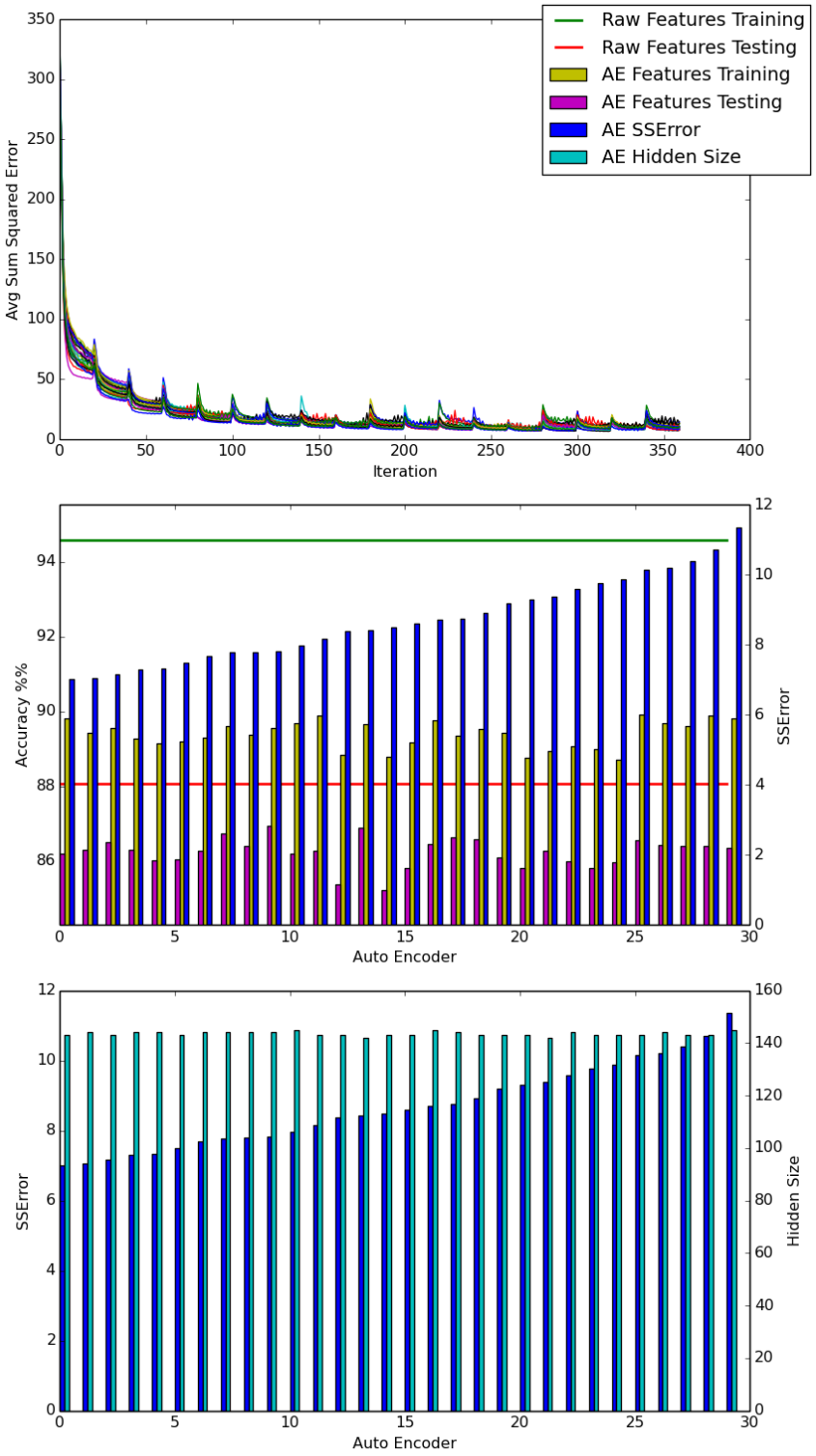

Figure 10. Example training run of EvoAE using evo-batch with no post-training

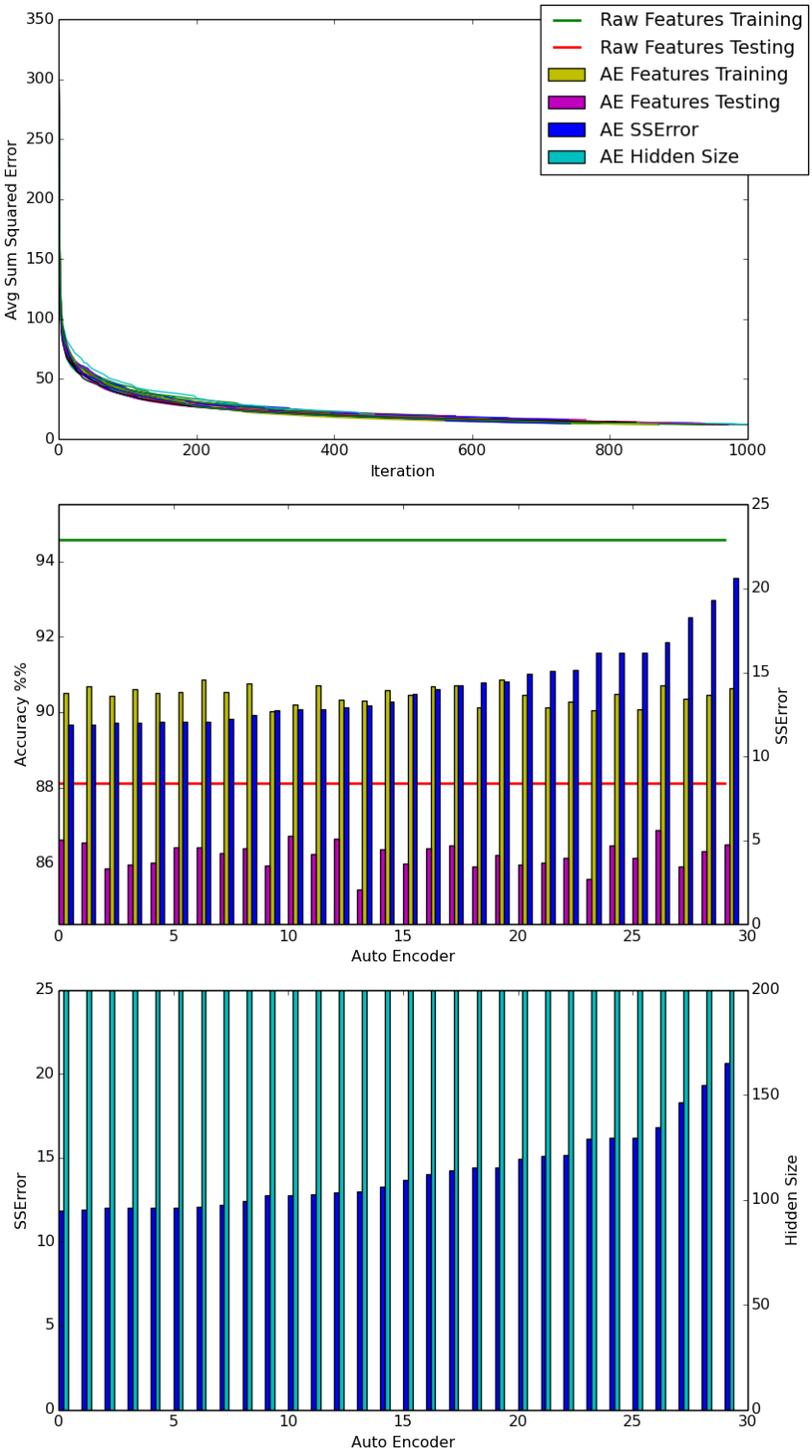

Figure 11. Example training run of 30 AEs using traditional methods

4.3 Large Dataset

4.3.1 MNIST 60k-10k Full Dataset

Due to time constraints the full range of tests was infeasible with the full MNIST data set. Preliminary tests showed that the only viable options were mini-batch and evo-batch in terms of training time. The memory necessary for full data training is above what is available on current mid-range computers (~3.5GB), but both evo-batch and mini-batch can be configured to work on low end systems as well.

4.4 Speed, Error and Accuracy vs Method and Data Size

From the above results it would appear that both mini-batch and evo-batch are non-optimal choices for smaller datasets. However, if time is not an issue, EvoAE using full data training produces the best reconstruction error when compared to traditional trial-and-error methods with random restart.

As data size grows the increases in accuracy of evo-batch and mini-batch become apparent. Hardware limitations are no longer an issue, and the EvoAE approach generally produces higher quality reconstructions than trial-and-error random-restart with lower overall training time.

5. DISCUSSION

From the above results it can be surmised that the proposed method is useful in situations of large amounts of data, yet lacks the ability to effectively replace traditional methods on small scale problems. This is a positive, however, as the exponential growth of data is leading ton continuously more unlabeled data as time goes on. The system lends itself well to the rising cloud-based ecosystem, as many small machines training on a subset of the whole dataset can be shown to train just as quickly, and more effectively, than a single piece of hardware using the testing methods above.

This is most likely due to the system's ability to train an extremely large number of individual new features based on the data regardless of if the current data being trained on is representative of the whole. By allowing each member of the population to extract its own primary features it builds a robust system which is much less likely to over-fit than traditional training methods. As both the baseline and EvoAE systems shared the same convergence criteria, the ability of EvoAE to reach a lower average and overall reconstruction error points to a better abstraction of features than even random restart can afford. Harnessing this power should be able to lead to much more robust and easily tunable deep learning networks in the future.

6. CONCLUSION

In all it has been shown that genetic and evolutionary methods for training neural networks are just as useful now as they were when backpropagation had first come into being. The flaws of traditional neural networks still exist in autoencoders even now that the diminishing returns of backpropagation have been dealt with. By using advances in backpropagation methods, such as conjugate gradient and weight decay, it is possible to effectively train an autoencoder to low levels of reconstruction error. The continued requirement of manual parameter tunings and random restarts still limits both the speed and quality of training.

To deal with this, the addition of evolutionary techniques can be employed to find near-global minima and tune parameters, in this case the structure of the autoencoder. While this removes the issues of manual tuning and random restart, it severely worsens time-to-train overall, especially with large datasets. This can be solved by using the power of the evolutionary algorithm to split training among the population, merging and sharing learned features during each generation in order to optimize an entire population in the time it would normally take to train a single autoencoder. In this way it is possible to create a large population of optimal autoencoders quickly and efficiently.

7. FUTURE WORK

While proof of concept is complete there is much more work which can be done on this topic, both in theory and in practice. Two possibilities quickly come to mind as they have been used effectively on deep learning networks in the past: distributed computing and GPU programming. Speed increases of up to 50% have been seen by switching from CPU to GPU code execution, and a speedup linear to the number of nodes available is expected on a distributed system, both of which would make previously infeasible problems fast and elegant in their execution.

Beyond that, however, is the more pressing matter of standardizing this system for simple integration by researchers, similar to Bayesian Networks and Support Vector Machines are included in many libraries and tools available online. The ability to easily create and use something as robust and powerful as a Deep Learning Network in the same way a tool like WEKA is used would be a boon to fields and industries outside of just Computer Science, and making that vision a reality is now a personal goal.

8. SUMMARY

This thesis explores evolutionary methods in the training of an autoencoder for use in a deep learning network. It explores previous work using evolutionary and genetic methods for training of artificial neural networks and uses similar techniques in the creation of autoencoders for use in input reconstruction.

The proposed method also includes a new form of mini-batch processing of large datasets for training a population of autoencoders in which each autoencoder only trains on a small, unique portion of the data before crossover occurs. The results of these two methods together show to work effectively on large scale data feature abstraction, but is not preferable for small scale datasets.

9. BIBLIOGRAPHY

[1] MNIST handwritten digits image recognition dataset. Available via www. http://yann.lecun.com/exdb/mnist/

[2] Yao, Xin, and Yong Liu. "A new evolutionary system for evolving artificial neural networks." *Neural Networks, IEEE Transactions on* 8, no. 3 (1997): 694-713.

[3] Montana, David J., and Lawrence Davis. "Training Feedforward Neural Networks Using Genetic Algorithms." In *IJCAI*, vol. 89, pp. 762-767. 1989..

[4] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313, no. 5786 (2006): 504-507..

[5] Ngiam, Jiquan, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V. Le, and Andrew Y. Ng. "On optimization methods for deep learning." In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 265-272. 2011.

[6] Bergstra, James, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-Parameter Optimization." In *NIPS*, vol. 24, pp. 2546-2554. 2011.

[7] Ilonen, Jarmo, Joni-Kristian Kamarainen, and Jouni Lampinen. "Differential evolution training algorithm for feed-forward neural networks." *Neural Processing Letters* 17, no. 1 (2003): 93-105.

[8] Bengio, Yoshua, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. "Greedy layer-wise training of deep networks." *Advances in neural information processing systems* 19 (2007): 153.

[9] UFLDL Tutorial. Available via WWW: http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial

[10] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. *Learning representations by back-propagating errors*. MIT Press, Cambridge, MA, USA, 1988.

[11] Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[12] V.A. Medical Center, Long Beach and Cleveland Clinic Foundation:Robert Detrano, M.D., Ph.D. Cleveland Heart Disease dataset found at the UCI Machine Learning Repository.

[13] LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86, no. 11 (1998): 2278-2324.