A Parallel Genetic Algorithm For Neural Network Tuning - EXAMPLE VERSION - NOT FOR PUBLIC RELEASE

Ben Harsha[1], Nathan Chadderdon[2], Steven Bogaerts[1]

[1]Department of Computer Science, DePauw University
Greencastle, IN 46135, U.S.A.
[2]Department of Computer Science, Knox College
Galesburg, IL 61401, U.S.A.
benjaminharsha_2015@depauw.edu
nchadder@knox.edu
stevenbogaerts@depauw.edu

***Abstract*—One challenge in using artificial neural networks is how to determine appropriate parameters for network structure and learning. Often parameters such as learning rate or number of hidden units are set arbitrarily or with a general "intuition" as to what would be most effective. The goal of this project is to use a genetic algorithm to tune a population of neural networks to determine the best structure and parameters. This paper considers a genetic algorithm to tune the number of hidden units, learning rate, momentum, and number of examples viewed per weight update. Experiments and results are discussed for two domains with distinct properties, demonstrating the importance of careful tuning of network parameters and structure for best performance.**

Keywords: Neural network, genetic algorithm, parallelism

I. Introduction

Things this paper didn't include in its official version:

- Itemized lists
- Enumerated lists

Things I wanted to show you how to do here:

1) Itemized lists
2) Enumerated lists

In the creation of an artificial neural network, low average error on an unseen test set is typically the principle goal. To reach this goal, a network is often trained for many epochs or until a certain average error is achieved on a training set. One risk of such approaches is that they may lead to over-training the network on the training set, hurting its generalizability to the test set. This paper describes a network design technique to reduce average error on a test set by determining ideal structure and parameters. As such, acceptable performance may more likely be reached with reduced need for extra training, and thus reduced risk of over-training. This paper describes a parallel genetic algorithm to tune a population of neural networks, therefore devising an ideal network for a given domain.

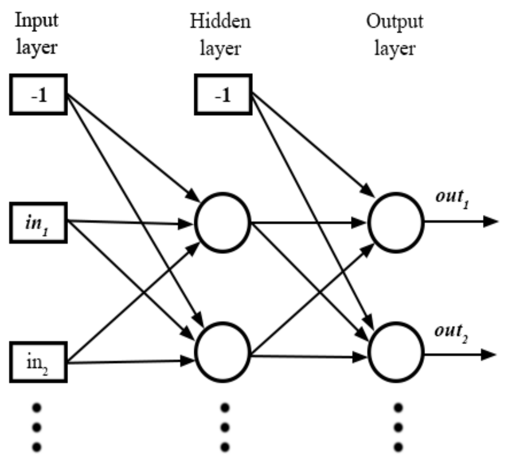

Fig. 1: The structure of a simple neural network

II. Neural Networks

A feed-forward neural network (Russell and Norvig 2009) maps a set of inputs into a set of outputs via multiple levels of weighted sums squashed by an activation function. More specifically, inputs are fed into one or more layers of hidden units, which then feed into the output layer. Figure 1 shows a feed-forward neural network with a single hidden layer. Each unit's output is computed based on its inputs as follows:

$$out_k = S(\sum_{i=1}^{n} W_{ik} in_i)$$

where n is the number of inputs to node k, W_{ik} is the weight from input i to node k, in_i is the input from i, and S is the sigmoid function:

$$S(t) = \frac{1}{1+e^{-t}}$$

A network is considered "trained" when its weights are set such that an appropriate input-output mapping is obtained. For example, a network might take as input the characteristics of a car, and output the value of that car. Back-propagation (Russell and Norvig 2009) is one common training process, in which the network considers a training set of input-output examples repeatedly, modifying the weights slightly in response to each example in order to reduce the error on that example. Thus the training process is a form of gradient descent on the error surface with a dimension for each weight in the network. The hope is that in reducing the error on the representative training set, the network's error will also decrease on unseen examples. A measure of network performance can be obtained by finding the average error of the network on a previously-unseen test set of examples, separate from the training set.

In constructing a neural network, a few parameters must be specified: number of epochs for training, number of hidden units, learning rate, momentum, and batch size. All of these variables together determine how the neural network functions, and in the end will have a significant effect on network performance. The remainder of this section describes the meaning of these parameters.

The number of epochs determines how many times a network will train on a training set. A single epoch involves the network examining every example in the training set and adjusting its weights accordingly. In order to actually learn complex functions, this must occur many times – dozens, hundreds, even thousands, depending on the network and domain.

The number of hidden units determines what kind of functions can be learned by a neural network. Too few means that the target function might not be learnable by the given network. Too many means that the network may take longer to train, may have redundant units that decrease efficiency, or may learn irrelevant details of the inputs that actually harm the generalizability of the network.

The learning rate affects how much the weights are updated during each epoch. If the rate is too high, then the network weights will be adjusted drastically for each example, potentially leading to jumps in the error surface that prevent effective settling at a global minimum. If the rate is too low, then settling may occur quite slowly, requiring more epochs, or even getting stuck in local minima rather than reaching a global minimum.

One can imagine a large boulder rolling among steep hills. It will eventually settle at a minimum, but its momentum will determine how much it will move at any given time. The momentum parameter determines how much the "momentum" of the previous weight change should affect the next. In ideal circumstances, a proper momentum parameter can allow a network to surpass local minima on the way to a global minimum, and can speed up learning in general.

The learning rate and momentum are both used in the weight update rules that follow. For the hidden-output weights:

$$\Delta W_{ji} = \alpha \cdot Err_i \cdot S'(in_i) \cdot x_j + \mu(old\Delta W_{ji})$$

such that

$$in_i = \sum_{j=1}^{n} W_{ji} x_j$$

$$Err_i = f_i(\vec{x}) - S(in_i)$$

For input-hidden weights:

$$\Delta W_{kj} = \alpha \cdot x_k \cdot \Delta_j$$

such that

$$\Delta_j = S'(in_j) \cdot \sum_i W_{ji} \cdot \Delta_i$$

$$\Delta_i = Err_i \cdot S'(in_i)$$

where k is an input unit index, j is a hidden unit index, i is an output unit index, f is the target function (so $f_i(\vec{x})$ is the activation of output unit i in the known example with inputs $\vec{x}$), x_k and x_j refer to the activation of input and hidden units respectively, α is the learning rate, μ is the momentum, W_{xy} is the weight from node x to node y, n is the number of hidden units, $old\Delta W_{ji}$ is the change made to the weight in the previous update, and S' is the derivative of the sigmoid function:

$$S'(t) = S(t) \cdot (1 - S(t))$$

The batch size indicates the number of examples a network examines before updating its edge weights. With batch sizes greater than one, networks will still compute the new edge weights after each example, but will not update them. It instead computes the change between the new edge weight and the old one. It adds these changes together at each example until it has seen the desired amount, when it finally updates the edge weights. Batch training is used in an effort to be quicker in training a function, although it is not always as accurate.

III. Genetic Algorithms

A genetic algorithm (Russell and Norvig 2009) is a machine learning algorithm based on the idea of evolution. A population of "individuals" is rated according to some fitness measure. The most fit individuals are allowed to "reproduce", passing their genes to their offspring through genetic crossover, thus hopefully improving the fitness of the next generation. Random mutations are also introduced to allow for potentially useful diversity. Thus a genetic algorithm, too, can be thought of as a gradient search: a gradient ascent on a fitness surface, with mutation allowing jumps to other areas of the space to avoid local maxima.

There are many ways in which a genetic algorithm can be applied to neural networks. While it is possible to use a genetic algorithm (instead of back-propagation, for example) to train a network, this is not the approach we take here. Rather, we have applied a genetic algorithm to tune the parameters of a population of neural networks, in order to settle on the ideal network characteristics for a given domain. The parameters under consideration include learning rate, momentum, batch size, and the number of hidden units. The genetic algorithm allows the consideration of a large number of networks, in order to identify broad trends in ideal network parameters. It also enables the development of a final "ideal" network for a given domain, with performance improved over more arbitrarily-structured networks.

Of course, this approach is computationally expensive, and so we apply parallelism using OpenMP (OpenMP Architecture Review Board 2011) to improve efficiency. The updating of weights in response to a given example (or batch of examples) is done in parallel, because these updates are independent of each other. In batch learning, multiple examples could also be considered in parallel, with their respective changes accumulated in a shared variable to be applied once at the end of the batch.

While genetic algorithms more commonly store genes as a series of binary values, we chose here to characterize each as a non-negative real number. Thus there are four genes under consideration, corresponding to the network parameters of learning rate, momentum, batch size, and number of hidden units. The genetic algorithm itself also has a number of parameters to determine the operation of the algorithm as applied to a population of neural networks. These parameters are the maximum number of hidden units, maximum learning rate, maximum momentum, maximum batch size, number of individuals, number of generations, removal rate, mutation rate, and testing size. These genetic algorithm parameters are used

All of the maximum values (hidden units, learning rate, momentum, batch size) are used to create the starting population of networks used in the genetic algorithm, with each network parameter for each network set to various random values between 0 and the maximum value for that parameter. This allows the genetic algorithm freedom to create unique combinations of networks while also limiting the values used in these networks to a reasonable range.

The number of individuals specifies the population size: how many networks are created initially, and how many networks are maintained from one generation to the next. The number of generations is used to determine how many times the algorithm removes poorly-performing networks, and allows more fit networks to reproduce. The removal rate is a decimal between zero and one which gives a percentage of individuals to remove each generation, replaced by the offspring of more fit networks. More specifically, the individuals are sorted based on their average error, and the bottom percentage are removed to be replaced by the children of the top performing individuals. Mutations result in a child not inheriting a trait directly from one of its parents, but instead having a random trait between zero and the maximum value for that trait. This maintains the possibility of useful genetic diversity, while crossover allows the genes the most fit to be passed on. Mutation rate determines how often these mutations occur.

The testing size determines how to split up a data set into training and test sets. Training sets are the examples that are given to the network to have it attempt to learn a function. The test set is then used to determine how well the network was able to learn the function. These sets are created by splitting up the initial data set into two parts based on the value given by testing size. It is given by a decimal between zero and one and determines what percentage of the data will be used for testing, with the remaining percentage being used for training.

IV. Experimental Setup

A. *Data Sets*

The data sets used to train the neural networks were acquired from the UCI Machine Learning Repository (Bache and Lichman 2013). Data sets used in experiments include the Car Evaluation Data Set and the

Molecular Biology (Splice-junction Gene Sequences) Data Set. These data sets attempt to classify the value of a car given certain characteristics and the type of gene given a genetic sequence.

B. *Experiments*

Aside from this more general examining of subjective versus objective, experiments were set up to target specific parameters of both the neural networks and the genetic algorithm in order to find which of these parameters affect the outcome in a meaningful way. One of the goals of these experiments was to see how much better neural networks could do with parameters determined by a genetic algorithm. To determine the difference in the answer between the two, initial experiments were run using only a single neural network. The values used for the parameters of the networks in these experiments were chosen based on what is generally considered reasonable when creating a neural network based on research, past experience, and the size of the data set. For the Cars data set, values of either 5 or 20 were used for the number of hidden units in order to get results from a very low amount and a relatively high one; learning rate and momentum were both kept constant at 0.1 and 0.9 respectively; batch size was varied between 1, 10, and 100 in order to evaluate a range of values. For the Splice data set, values of 10 and 100 were used for the number of hidden units; momentum was again kept constant at 0.9; learning rate varied between 0.1 and 0.3; batch size was kept relatively low, with values of 1 and 10.

An issue that was encountered during experiments involved the program reporting incorrect run times. Eventually, this was determined to be caused by the parallelism. Time was being measured using clock time, so when program branched onto different processors the clock time was altered. This resulted in reported times that were generally three or four times the actual run times measured using a stopwatch. This issue was resolved by using the time function instead of the clock function. Using this method, precision was lost since time only reports in seconds, but the reported times were confirmed to be accurate when compared against a stopwatch.

V. Results

A. *Cars Data Set Results*

Figure 2 shows that raising the number of epochs that the networks train on decreases the average error and reduces the variance between average errors calculated by different networks. As shown in the graph above, there is a definite correlation between the number of epochs increasing and the average error decreasing. Along with this, the networks performed more consistently with a higher number of epochs. This is shown by the close proximity of the majority of points at 1000 epochs. It seemed however, that more epochs was not always preferable, as 500 epochs performed better than 1000 on multiple occasions and achieved the lowest average error over all tests.

When examining batch size, it seems that the genetic algorithm prefers a smaller value. Across all experiments run, the average batch size was 16.4 while the average maximum batch size was 72.2. In addition, among 54 experiments run using a maximum batch size of 50, the batch size settled less than or equal to 25 in 40 of them. This shows that a lower batch size resulted in the neural networks performing better overall. Table I summarizes these results.

Similarly, the genetic algorithm tended to settle on

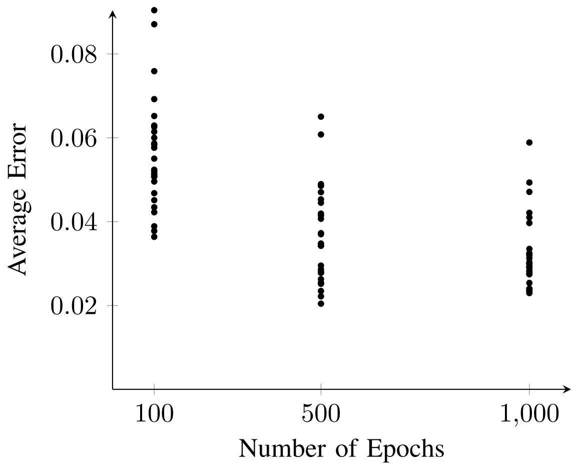

Fig. 2: Cars: Effect of the number of epochs on average error.

TABLE I: Cars: Amount of results settled below 50% of maximum batch size.

Max Batch Size	Total Experiments Run	Below 50% of Max
50	54	40
100	17	17
150	8	7

TABLE II: Amount of results settled below 50% of maximum hidden units, in the Cars domain.

Max Hidden Units	Total Experiments Run	Below 50% of Max
30	46	34
50	8	7
100	17	13
150	8	7

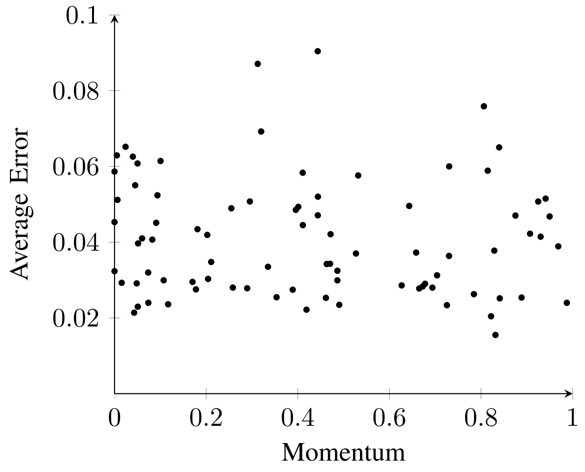

Fig. 3: Cars: Effect of momentum on average error.

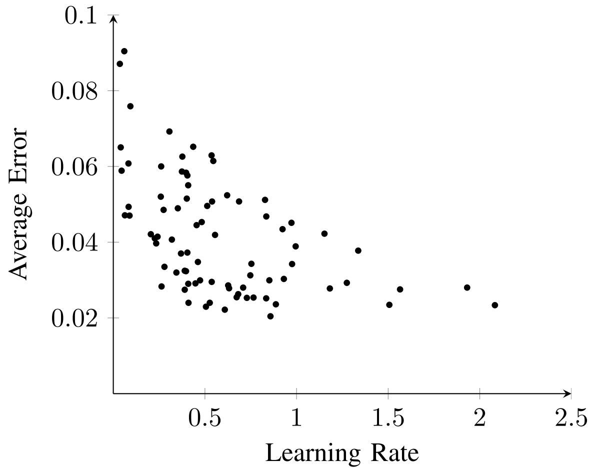

Fig. 4: Cars: Effect of learning rate on average error.

a relatively low number of hidden units. Across all experiments, the average value of the maximum hidden units was 60.4 but the average number of hidden units settled on was only 18.2. This average is only 30% of the average maximum, solidifying that a low number of hidden units was preferable. This clear tendency of settling on low values for number of hidden units is shown by Table II.

The effect of momentum on the neural networks was nearly indiscernible throughout our experiments. As depicted in Figure 3, there was no correlation showing that a high or low momentum affected the average error either positively or negatively. Along with this lack of correlation, there did not seem to be any convergence to a high or low momentum value. Throughout tests, it was seemingly random and did not seem to move in any particular direction. Changing the maximum momentum did nothing to change this trend, as results continued to show seemingly unrelated momentums.

Figure 4 shows the relationship between the learning rate and the average error. A large portion of experiments were run with a maximum learning rate of 1, so there is a higher concentration of points with a learning rate less than 1. A downward trend was identified after running these tests, suggesting that, in the Cars domain, average error decreases as the learning rate increases. In order to determine if the trend continued with higher learning rates, additional experiments were performed using maximum learning rates up to 5. The results of these experiments continued the downward trend showed previously. Generally, it would seem that increasing the learning rate improves the average error.

Further supporting the idea that higher learning rates are preferred are the learning rates settled on based on the maximum learning rates. Experiments run were broken up into groups based on their maximum learning rate, and the average learning rate for each maximum was computed.

TABLE III: Cars: Average learning rate of networks in the final generation, for given maximum possible learning rates.

Maximum Learning Rate	Average Learning Rate
5	1.7267
2	1.4079
1.5	1.2643
1	0.6367
0.75	0.4205
0.5	0.3656
0.3	0.26
0.1	0.0818

For each maximum learning rate Table III except 5, the average learning rate is in the upper 50% of the possible range. This confirms that the genetic algorithm tends to settle on higher learning rates relative to the maximum learning rate, but also shows that there is a threshold in which increasing the learning rate is no longer effective in improving the average error.

Increasing the number of individuals and generations in the genetic algorithm had the effect of making the

TABLE IV: Cars: Average error over all networks with given number of individuals.

Number of Individuals	Average Error
10	0.0421816786
20	0.0347347333
30	0.0283559333

TABLE V: Cars: Average error over all networks with given number of generations.

Number of Generations	Average Error
10	0.0417958082
20	0.0356072
30	0.031888

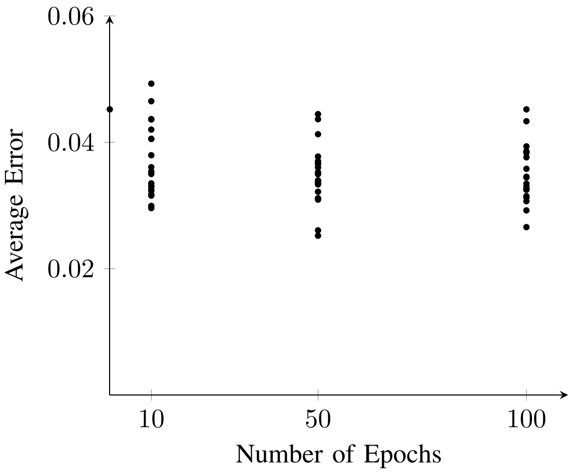

Fig. 5: Splice: Effect of the number of epochs on average error.

networks perform more consistently, but not necessarily better. The majority of tests were run using ten individuals running for ten generations. These amounts were varied in independent experiments in order to determine whether they made any difference on the networks' ability to learn the function. The result of these experiments showed that increasing both of these values had a noticeably positive effect on the networks. While the networks with larger generations and individuals did not get the best error over all tests, they proved to be the most consistent out of all groups tested. Especially those with thirty individuals running ten generations and those with ten individuals running thirty generations were near the top in terms of average error regardless of the number of epochs used to train them.

B. Splice Data Set Results

Similar tests were performed on the Splice data set in order to consider both general trends and trends specific to each data set. As shown in Figure 5, the effect that varying the number of epochs had on the average error was extremely similar to the Cars domain. Average error slightly decreased from 10 epochs to 50 epochs, but the effect was much less apparent from 50 epochs to 100 epochs. Again, there was even some performance lost

TABLE VI: Splice: Amount of results settled below 50% of maximum batch size.

Max Hidden Units	Total Experiments Run	Below 50% of Max
20	29	19
100	23	23

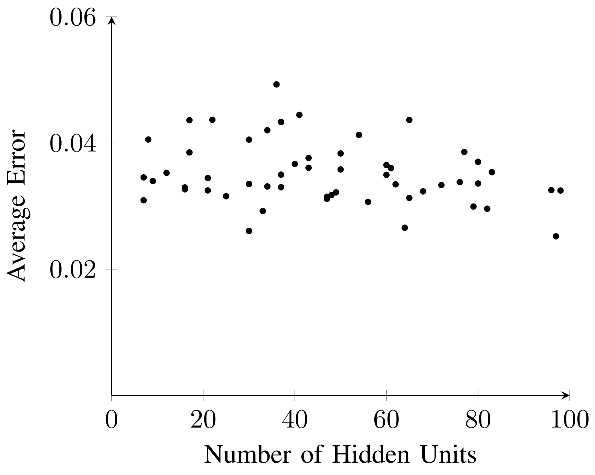

Fig. 6: Splice: Effect of number of hidden units on average error.

when comparing networks with fifty epochs to those with 100.

Also similar to the Cars domain was the genetic algorithm's preference to settle on low batch sizes. Across all experiments run, the average maximum batch size given to the genetic algorithm was 50.1, while the average batch size settled on was just 11.3. Table VI shows this convergence to lower values. Once again, this conveys that the neural networks are able to perform consistently better when given a lower batch size.

Examining the behavior of the number of hidden units showed similarities to those shown by the Cars data set. In general, the algorithm preferred to settle on values that were relatively low given their maximum range. Figure 6 shows the wider range chosen by the ideal networks in the Splice data set.

The effects of momentum were still not discernible from the results gathered using the Splice data set. As shown in Figure 7, momentum's influence was again seemingly random. No trends were found that gave any evidence that momentum affected the average error in either a positive or negative way.

The trend of the algorithm preferring a higher learning rate did not continue over the Splice data set. Even when given freedom to choose higher values, the most effective networks usually used learning rates that were

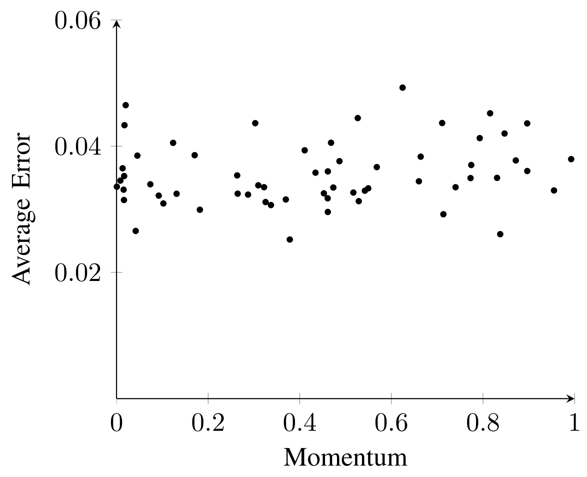

Fig. 7: Splice: Effect of momentum on average error.

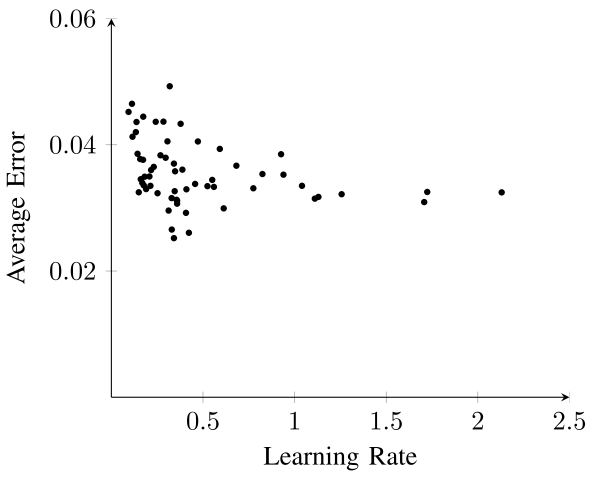

Fig. 8: Splice: Effect of learning rate on average error.

much lower than those used by the Cars data set. Figure 8 shows this convergence toward lower learning rates while also showing that the higher learning rates do not typically out-perform the lower ones.

In Table VII, it is significant that the groups with higher maximum learning rates have such relatively low average learning rates because this shows that even when given the option to have higher learning rates, the genetic algorithm does not choose the higher values. On the other end, however, the groups with lower maximum learning rates do tend to settle on values that are relatively high for the range.

TABLE VII: Splice: Average learning rate of networks in the final generation, for given maximum possible learning rates.

Maximum Learning Rate	Average Learning Rate
5	0.9623
3	0.758
1	0.472
0.5	0.3141
0.3	0.1955

TABLE VIII: Splice: Average error over all networks with given number of individuals.

Number of Individuals	Average Error
10	0.0362218463
20	0.0302012667
30	0.0291261667

TABLE IX: Splice: Average error over all networks with given number of generations.

Number of Generations	Average Error
10	0.0354922167
20	0.0370337333
30	0.0354270333

Increasing the number of individuals and generations had a very similar effect to what was examined with the Cars data set, as seen in Tables VIII and IX. As those two values increased, the networks were able to learn the functions at a more consistent level. Once again, the networks run with higher amounts of generations and individuals did not achieve the best overall average errors, but they consistently performed at a high level.

VI. Analysis

In the results section above, it was determined that a higher number of epochs results in a generally lower average error. From the beginning of the process, this was expected. Results of experiments were able to confirm expectations by showing a clear improvement in average error from 100 to 500 epochs in the Cars data set, and from 10 to 50 in the Splice data set. Neural networks need a large number of epochs to learn functions, and it is intuitive that more epochs would cause the network to learn a function more effectively. What was not expected, however, was that simply adding more epochs does not always improve the performance of the neural networks. In Figure 2, it is apparent that networks with 500 epochs routinely performed just as well or better than networks with 1000 epochs, and a similar trend was shown in Figure 5. It is possible that the higher number of epochs is leading to over-training. This is one drawback of simply training for a certain number of epochs, instead of considering an error-based stopping criterion.

It was also revealed that the genetic algorithm tended to settle on a smaller batch size in both data sets being tested. The purpose of batch training is to make cumulative, less frequent changes to the network, to prevent frequent changes in response to individual examples. On the other hand, making corrections for one example at a time may allow better performance and more refined corrections on the next. Therefore there is a balance to be struck. The genetic algorithm suggests that smaller batch sizes are preferred, at least in the Cars and Splice domains, suggesting that overall each example provides useful information towards the target function and is best applied quickly to the weights.

The results involving hidden units are a bit harder to interpret. In the Cars domain there is a clear negative correlation between the number of hidden nodes and the error, while in the Splice domain the number of hidden units doesn't seem to have a significant impact on the error. There is clearly some difference between domains that makes some more sensitive to the number of hidden units than others, but without further testing it is difficult to say just what that difference might be.

As noted above, momentum did not have a significant effect on the experiments being run, with no correlation between momentum and average error, and no convergence to a specific momentum value across all experiments. The purpose of momentum is to avoid settling on a local minimum. It could be that momentum was not affecting our results because the local minimum was often also a global, or it could be that momentum simply does not have a drastic effect on the datasets being used for these experiments.

It was expected to see that increasing the number of individuals and generations led to more consistently solid performance from the neural networks based on the model presented by the genetic algorithm. Using more individuals gives the algorithm a broader range of data to work with, so it has more options to explore when attempting to evolve the optimal network. It follows then that the resulting networks should be more effective than those coming from a genetic algorithm with fewer individuals. Similarly, using a higher number of generations allows the algorithm more time to evolve and tune the genes. This also results in networks that will have more consistent traits and answers.

When examining the effect of learning rate on the neural networks, it was very surprising that for the Cars domain, a higher learning rate resulted in lower average errors, although there did seem to be a point where a higher learning rate stopped being effective. Only one genetic algorithm finished with a learning rate over 2, even though maximum values of 3 and 5 were used for multiple experiments. Assumptions going into experiments were that a learning rate lower than 0.5 was ideal. As shown in Figure 4, however, this is not always the case. Networks with higher learning rates were able to consistently perform on par or better than networks with learning rates around the previously considered ideal range. This is most likely caused by the relatively small amount of epochs used in the experiments. Since the data sets were so large, it was not feasible to test them for hundreds of thousands of epochs. It was necessary to keep the number of epochs low in order to perform experiments in an efficient manner. Because of these low numbers of epochs, it makes sense that a higher learning rate would benefit the network. It has less time to learn the function, so a higher learning rate speeds up the process by making larger changes each time the network's weights are updated.

VII. Related Work

The parallelization of neural networks has been the focus of much research. For example, (Kim, Asaad, and Linsker 2014) describes a parallel training algorithm similar to the one in this paper, in which the weight updates are computed in parallel by breaking the updates into steps and running them through a pipeline.

Work has also been done in the application of parallelism to genetic algorithms of neural networks. (Gruau 1994) describes a genetic algorithm used to find an optimal neural network to simulate a Turing machine. Parallelism was applied by separating the genetic algorithm population into separate "islands", with the goal of increasing overall genetic diversity. This is in contrast to the work of this paper, in which the members of the next generation are created in parallel, but all individuals were in the same group together.

Another example of a genetic algorithm applied to neural networks can be found in (Soares, Antunes, and Araújo 2012). That work focuses on the creation of neural network ensembles – groups of neural networks that work together to come to a decision. As a part of the ensemble generation, the individual neural networks were optimized using parameters and genes similar to those used in this project. Most similar to the work performed in this paper were the initialization of weights in the network and the use of the number of hidden units as a gene in the genetic algorithm phase.

VIII. Future Work

The work described in this paper leads to many questions for further research. With the limited time available for this research, it was only possible to explore two domains. An expansion of the data sets investigated may uncover more patterns that could guide network design more deeply. It would be very interesting to attempt to develop some guidelines for network design given the domain characteristics; experimentation on many more domains may make that kind of analysis possible. One specific trait that merits further investigation is how much the subjectivity/objectivity of the domain affects the performance of a network. For example, it was noted that in the two domains tested that the Cars domain, which is more subjective, ends up with a higher learning rate than the Splice domain, which is more objective.

More work would also be useful in identifying properties of the domain that affect what parameters like the learning rate should be. The learning rate chosen by the genetic algorithm varied in the two domains, with the Cars domain favoring higher learning rates and the Splice data set favoring more traditional lower learning rates. More work may be able to determine what about the Cars domain makes a higher learning rate favorable, allowing others to know what learning rate will give them the best results without having to simply try them all.

The two domains also differed in their results regarding number of hidden units, while neither seemed influenced by the momentum parameter. We would like to explore possible explanations for this in more detail.

IX. Conclusion

This paper describes results of experiments using a parallel genetic algorithm to tune neural network design. In particular, the algorithm considered the number of hidden units, learning rate, momentum, number of training epochs, and batch size. The domains behaved similarly in some of these parameters, but quite differently in others. Specifically, the Cars domain preferred a higher learning rate and a lower number of hidden units. Both domains preferred a moderate amount of epochs (around 500) and a smaller batch size, and neither seemed particularly influenced by the momentum parameter. Some potential explanations are offered, and these results lead to additional questions to be considered, particularly in applying this work to a greater number of domains.

X. Acknowledgement

This work was supported by NSF grant number CNS-1156893.

References

[1] Bache, K., and Lichman, M. 2013. UCI machine learning repository.

[2] Gruau, F. 1994. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* Ph.D. Dissertation.

[3] Kim, L.-W.; Asaad, S.; and Linsker, R. 2014. A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network. *ACM Trans. Reconfigurable Technol. Syst.* 7(1):5:1–5:23.

[4] OpenMP Architecture Review Board. 2011. OpenMP application program interface version 3.1.

[5] Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd edition.

[6] Soares, S.; Antunes, C.; and Araújo, R. 2012. A genetic algorithm for designing neural network ensembles. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, 681–688. New York, NY, USA: ACM.