Colby College
Digital Commons @ Colby

Honors Theses

Student Research

2016

Using Genetic Algorithms to Evolve Artificial Neural Networks

William T. Kearney
Colby College

Follow this and additional works at: http://digitalcommons.colby.edu/honorstheses

Part of the Artificial Intelligence and Robotics Commons, Numerical Analysis and Scientific Computing Commons, and the Theory and Algorithms Commons

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation

Kearney, William T., "Using Genetic Algorithms to Evolve Artificial Neural Networks" (2016). *Honors Theses.* Paper 818.
http://digitalcommons.colby.edu/honorstheses/818

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby. For more information, please contact enrhodes@colby.edu.

Using Genetic Algorithms to Evolve Artificial Neural Networks

Honors Thesis in Computer Science

Colby College

Advisor: Stephanie Taylor

William T. Kearney

May 2016

Contents

Abstract

This paper demonstrates that neuroevolution is an effective method to determine an optimal neural network topology. I provide an overview of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, and describe how unique characteristics of this algorithm solve various problem inherent to neuroevolution (namely the competing conventions problem and the challenges associated with protecting topological innovation). Parallelization is shown to greatly speed up efficiency, further reinforcing neuroevolution as a potential alternative to traditional backpropagation. I also demonstrate that appropriate parameter selection is critical in order to efficiently converge to an optimal topology. Lastly, I produce an example solution to a medical classification machine learning problem that further demonstrates some unique advantages of the NEAT algorithm.

1 Introduction

Recently, ANN's have enjoyed considerable more attention from the research community, particularly in fields such as deep learning (e.g. see [1, 6, 12]) where so-called "deep" ANN's have won numerous contests in pattern recognition, computer vision, natural language processing, and machine learning.

Still, many current machine learning problems can be solved using smaller neural networks. In these cases, NeuroEvolution can prove a useful method to discover an efficient network topology and connection weights. This paper provides an overview of the NeuroEvolution of Augmenting Topologies (NEAT) method [14], an algorithm that allows for the efficient evolution of complex neural network topologies using genetic algorithms. I also discuss the importance and associated challenges of parameter selection, demonstrate how parallelization can be used to greatly improve computation time, and use NEAT to produce an example solution to a medical classification machine learning problem.

1.1 Artificial Neural Networks

Artificial Neural Networks (ANN's) are biologically inspired computational models that seek to mimic the behavioral and adaptive capabilities of central nervous systems. ANN's are used for regression or classification tasks, and are capable of solving highly complex non-linear problems due to their role as universal function approximators. Typically, ANN's are composed of interconnected processing units called nodes or *neurons*. The connection weights between neurons are called *synaptic weights*. Neurons are classified as input, hidden, or output. Usually, there are as many input neurons as there are features in the dataset; likewise, there are as many output neurons as there are classes to predict (one output neuron can be used for regression tasks). The number of hidden nodes needed varies greatly with the task.

The architecture of an ANN is characterized by its *topology*, i.e. the specific structure of nodes and connections that create a certain network. Traditionally, the topology is determined *a priori* by the programmer before training (or a set of possible topologies are evaluated during a validation step). Unfortunately, although two neural networks with different topologies can theoretically represent the same function [2], the most efficient topology might not be readily apparent. Backpropagation (BP)—a portmanteau of "backward propagation of errors"—is used to train the synaptic weights of a

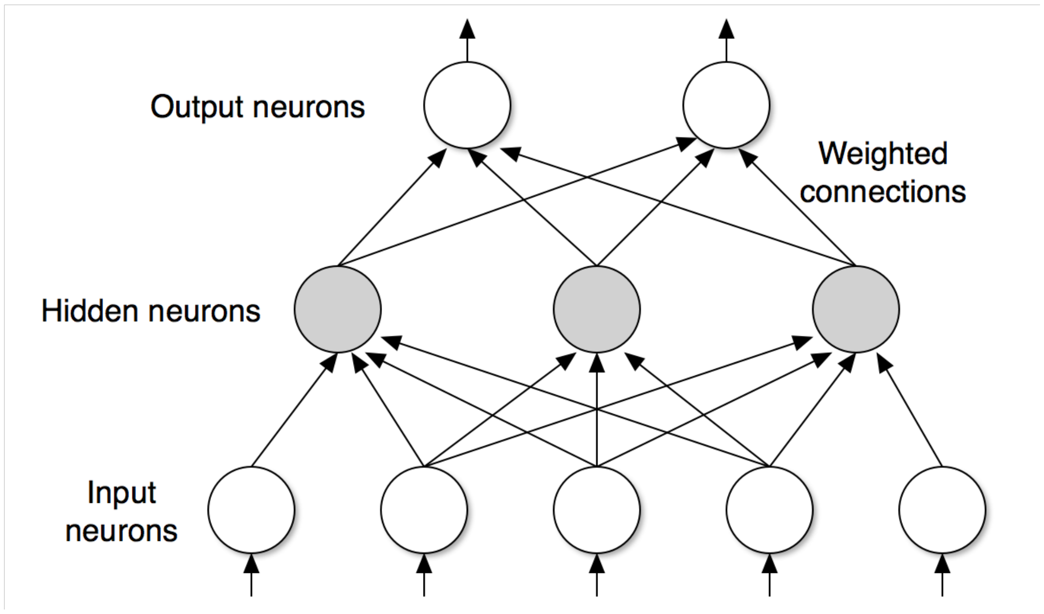

Figure 1: A generic ANN architecture composed of input neurons, hidden neurons, and output neurons connected by synaptic weights.

network, minimizing a cost function by adapting weights. Werbos [15] first described the process of using BP to train neural networks; a few years later, a seminal paper used computational experiments to demonstrate that BP could indeed yield useful internal representations in neural network hidden layers [10]. BP is now the de facto standard algorithm for any supervised learning task.

1.2 Genetic Algorithms

Genetic algorithms are a family of computational models inspired by Darwinian natural selection, and can offer an alternative to backpropagation when finding a good set of weights in a neural network. The original genetic algorithm was introduced and investigated by John Holland [5] and his students (e.g. [3]). A genetic algorithm encodes a potential solution to a problem (the *phenotype*) in a chromosome-like data structure called the *genotype* or *genome*. The canonical genetic algorithm has used binary strings to represent chromosomes [5]. Traditionally, a genetic algorithm creates an initial population of (typically random) genomes, which are materialized as phenotypes

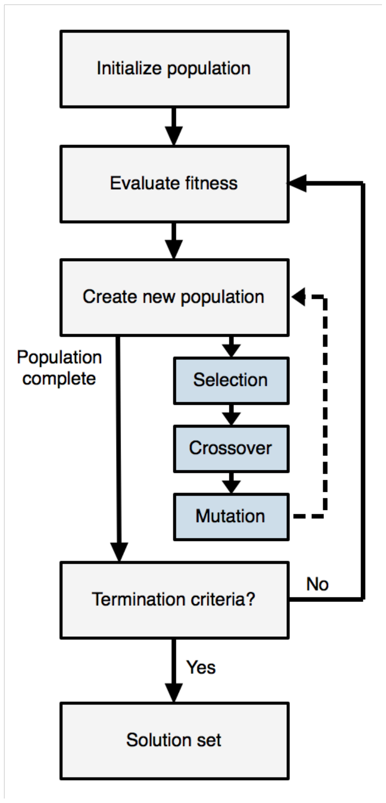

Figure 2: Flowchart of a basic genetic algorithm. *Initialize population*: generate random population of n chromosomes. *Evaluate fitness*: evaluate the fitness $f(x)$ of each chromosome x in the population. *New population*: create a new population by repeating selection, crossover, and mutation until the new population is complete. *Termination criteria*: if end condition satisfied, stop; otherwise, return to evaluate fitness step.

and evaluated on the basis of some fitness function. Those genomes that represent better solutions to the problem at hand are given opportunities to "reproduce", producing genomes for the next generation. Genomes also undergo mutation in order to ensure genetic diversity from one population to the next (analogous to biological mutation) [9].

As a search heuristic, genetic algorithms have some benefits over other forms of search. For example, while gradient descent methods might get trapped in local minima in the error surface, genetic algorithms avoid this pitfall by sampling multiple points on the error surface. Furthermore, genetic algorithms require very little a priori knowledge of the problem domain [13]. See [9] or [16] for more in-depth information on genetic algorithms.

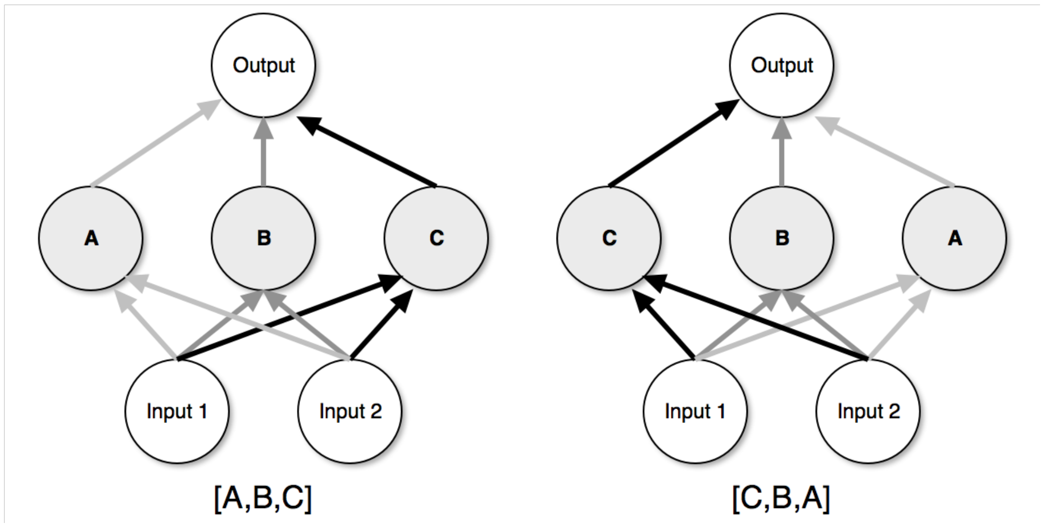

Figure 3: Two neural networks that are functionally equivalent (i.e. compute the same function) but whose hidden nodes appear in different orders, and thus represented by different chromosomes. Using single-point crossover, $[A, B, C] \times [C, B, A]$ yields $[A, B, A]$ and $[C, B, C]$; both offspring are missing one of the 3 main components of the solution (redrawn from [14]).

2 NeuroEvolution

As mentioned previously, the architecture of a neural network and the learning algorithm used to train the model are important decisions when seeking to solve a particular task. NeuroEvolution seeks to solve these problems by using genetic algorithms to evolve the topology of neural networks [4]. However, this solution class comes with its own baggage; namely, the competing conventions problem and the issue of protecting topological innovation. As we'll discuss in a later section, NEAT seeks to resolve these issues primarily through using *historical gene markers*.

2.1 The Competing Conventions Problem

A main concern and well-known problem of NeuroEvolution is the *Competing Conventions Problems* [11]. It is possible that two genomes represent the same solution but with different encodings; in other words, two neural networks that order their hidden nodes differently in their genomes might still be functionally equivalent (see figure 3). This is a problem because differing representations of the same structure are highly likely to produce a damaged offspring during crossover. The issue becomes more serious when we consider the pervasiveness of the problem. Stanley writes, “in general, for n hidden units [neurons], there are $n!$ functionally equivalent solutions” [14].

As an example, consider two functionally equivalent genomes with different structures, represented as $[A, B, C]$ and $[C, B, A]$. A useful (because it is easy to quantify the effects) method of performing crossover during a genetic algorithm is to use single-point crossover, in which a randomly chosen recombination point is used to split each genome into two parts, one of which is swapped with the other parents [16]. Using single-point crossover with our two example genomes yields,

$$\begin{array}{rc} & \text{[A,B,C]} \\ \times & \text{[C,B,A]} \\ \hline & \text{[A,B,A] and [C,B,C]} \end{array}$$

Unfortunately, both $[A, B, A]$ and $[C, B, C]$ are missing a hidden node that was (presumably) important to the solution. Thus, both offspring are likely not good candidate solutions that the algorithm will now needlessly evaluate. Crossing over functionally equivalent genomes needlessly increases computation time by creating damaged children. How can we ensure standard crossover produces viable offspring?

2.2 Topological Innovation

A second major problem exists when we consider the fitness implications from adding new or adjusting existing structure. Take, as an example, the action of adding a new connection in a neural network. Often, such a change will initially decrease fitness before the connection weight is given an opportunity to optimize. If that genome was culled from the population before it was given an opportunity to reproduce in order for its children to develop the new structure further, we might be inadvertently halting a promising structural development. Stanley again turns to biological inspiration for an answer: *speciation* can help protect topological innovation by ensuring genomes only compete within a population niche [14]. Networks with innovative structures are protected by only competing with other networks in their species. Two organisms are said to be in the same species if they represent topologically similar networks. How do we determine if two genomes represent structurally similar networks?

3 The NEAT Algorithm

3.1 Overview

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm was developed by Ken Stanley in 2002 while at the University of Texas at Austin, and is outlined here. The algorithm seeks to resolve some of the shortcomings of previous neuroevolution methods, including evolving neural network topologies along with weights. NEAT proves to be effective due to "(1) employing a principled method of crossover of different topologies, (2) protecting structural innovation using speciation, and (3) incrementally growing from minimal structure" [14].

3.2 Encoding Scheme and Historical Markers

NEAT uses a direct encoding scheme, in which a network architecture is directly encoded into a GA chromosome [9]. Genomes are represented in a list-like data structure that contains *node genes* and *connection genes*. Connection genes contain information about the nodes it connects, the weight of the connection, whether or not the connection is enabled, and a *historical marker* that provides information about the ancestral history of the gene.

Genes that share a historical origin necessarily represent the same structure in a neural network phenotype [14]. Thus, each new gene that is created through structural mutation is assigned a unique *global innovation number*. When individual genes are copied to an offspring genome, they retain their historical marker. This allows genomes to be compared by matching up genes that share a historical marker (and thus represent the same structural component in possibly differing larger network structures). Thus, historical marking seeks to answer the previous questions regarding viable offspring and topological innovation by framing them as topological matching problems.

3.2.1 Crossover

During crossover, genes in each genome are aligned based on their historical marker in a process called *artificial synapsis* [14]. Genes that share an ancestral origin are called *matching genes*; those that don't are called *disjoint genes* or *excess genes* depending on if they exist inside or outside the range of innovation numbers of the other parent, respectively. Thus, although two parents might (and likely do) look different, their historical markers allow us to find structural similarities without any (likely expensive) topological analysis. Matching genes are inherited randomly. Excess genes and disjoint genes are inherited from the more fit parent.

Artificial synapsis helps counteract the negative side effects caused by the competing conventions problem. Compatible organisms mate in a way which allows for the preservation of functional subunits; offspring are likely to be undamaged. Although competing conventions might still exist in the population, NEAT does not spend time evaluating damaged offspring, and thus the main consequence of competing conventions is avoided.

3.2.2 Speciation

Genomes are allowed to compete with populations niches by grouping them into species. Although this appears to be a topological matching problem, historical markers again provide a useful solution. The topological discrepancy between two neural networks can be quantified with the number of excess and disjoint genes between their genomes. Stanley writes, "the more disjoint the genomes, the less evolutionary history they share, and thus the less compatible they are" [14]. The amount of evolutionary history shared between two genomes is measured with a compatibility function:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{1}$$

where δ is the compatibility distance, E is the number of excess genes, D is the number of disjoint genes, $\overline{W}$ is the average weight differences of matching genes, and the coefficients c_1, c_2, and c_3 allow us to adjust the relative importance of the three factors. The compatibility is used as a hard threshold; thus, once an organism crosses it, a new species is created.

3.3 Mutations

There are four main types of mutation operations in NEAT: (1) add a neuron, (2) add a connection, (3) remove a node, (4) remove a connection, and (5) perturb a synaptic weight. The probability of each mutation occurring is controlled by the programmer via a configuration file. When adding a neuron, a random connection is selected and replaced by a new neuron and two new connections. The weights of the two new connections are selected to be effectively the same as the replaced connection in order to avoid overly-drastic movements over the fitness landscape. Adding a new connection works in a similar fashion; two neurons are selected at random, between which a new connection is added. Checks are performed to ensure a connection doesn't already exist between the source and destination neuron. This also highlights two interesting features of NEAT: hidden neurons aren't ordered in a traditional hierarchy of hidden levels and, relatedly, recurrent neural networks are allowed to evolve (and ideally will, if that will solve the problem more efficiently than a feed-forward neural network). For example, when adding a new connection, the source and destination neuron chosen are allowed to be the same neuron.

The removal of neurons and connections is performed similarly. Both connections and neurons can be randomly selected for removal. Hidden neurons are removed if there are no remaining connections interacting with them. When a synaptic weight is randomly chosen to be perturbed, a random number from a zero-centered normal distribution is added to the weight. The standard deviation of the normal distribution is also specified by the programmer. For an analysis of parameter selection, see section 5.

3.4 Complexification

Many genetic algorithms start out with a random collection of initial organisms [5]. This poses a problem when evolving neural networks; it is very possible an organism will be a disconnected graph (i.e. not every node will be connected with the network containing the output). This will take time to remove from the population.

Stanley [14] identifies a more serious problem: "Starting out with random topologies does not lead to finding minimal solutions, since the population starts out with many unnecessary nodes and connections already present." That is to say, randomly generated nodes and connections have not undergone any sort of justification for their existence. It is not known if they contribute to the solution. Conversely, all subsequent structural additions undergo evaluation by the nature of how they change the fitness of the organism. Starting with minimal structure allows the algorithm to search for the solution in a low-dimensional search space, greatly improving performance [14].

4 Parallelization

As Whitley [16] astutely notes, "part of the biological metaphor used to motivate genetic search is that it is inherently parallel. In natural populations, thousands or even millions of individuals exist in parallel." Because genomes are evaluated on their fitness level completely independent of other organisms in the population, genetic algorithms are practically pleading to be implemented in parallel.

4.1 Parallel Python

I used the open source and cross-platform python module Parallel Python (http://www.parallelpython.com), which provides support for the parallel execution of python code on SMP (systems with multiple processors or cores) and clusters (computers connected via a network). Parallel Python has the important capability of overcoming the Global Interpreter Lock that the python interpreter uses for internal bookkeeping with the widely known side effect of allowing only sequential execution of python byte-code instructions, even on SMP computers.

A population of N genomes was divided into N/P sub-populations, where P is the number of available processors. Each processor evaluated each genome in its sub-population (i.e. built a phenotype and propagating the training data through the network) based on a fitness function, reassigning each fitness value and sending the sub-population back to the master processor to perform mutation and crossover.

Genetics algorithms can be computationally expensive, particularly as the population size increases (in other words, "there is no such thing as a free lunch"). Thus, the speed advantage parallelization provides is important. By offering a partial solution to the often-large computation requirements of genetic algorithms, parallel processing returns neuroevolution to the realm of

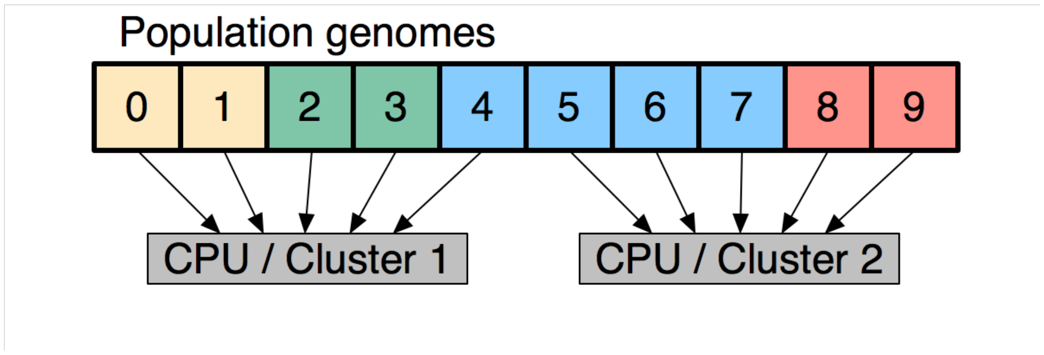

Figure 4: Each sub-population sent to a processor for fitness evaluation. Note how the sub-populations are independent of species (denoted by a different color in the population); the blue species, for example, contains genomes that are being sent to different processors.

efficient optimization techniques—parallelization becomes only increasingly important as data sets continue to grow in size. Even for a relatively simple problem, parallelization reduced the time it took NEAT to find a solution from 25 minutes running sequentially to 8 minutes running in parallel. These types of speedups become more important when solving more complex problems or when working with larger data sets.

4.2 Cluster vs. SMP

When comparing cluster versus SMP performance, it quickly became apparent that keeping all computation on a local machine was more efficient if the fitness evaluation wasn't computationally expensive. This is because the overhead cost of serializing the sub-population, compressing the data, passing the data over the local network to another machine, uncompressing the data, and un-serializing the sub-population was prohibitively expensive when compared to the time spent actually evaluating the sub-population. I suspect that if fitness evaluation became relatively more expensive (i.e. larger, more complex neural networks were needed to solve the problem), then it would begin to make more sense to use a distributed cluster. Because the problem domain I used was fairly simple, all computation was done on a local machine in parallel across 12 cores, preventing the need to send data across a local network.

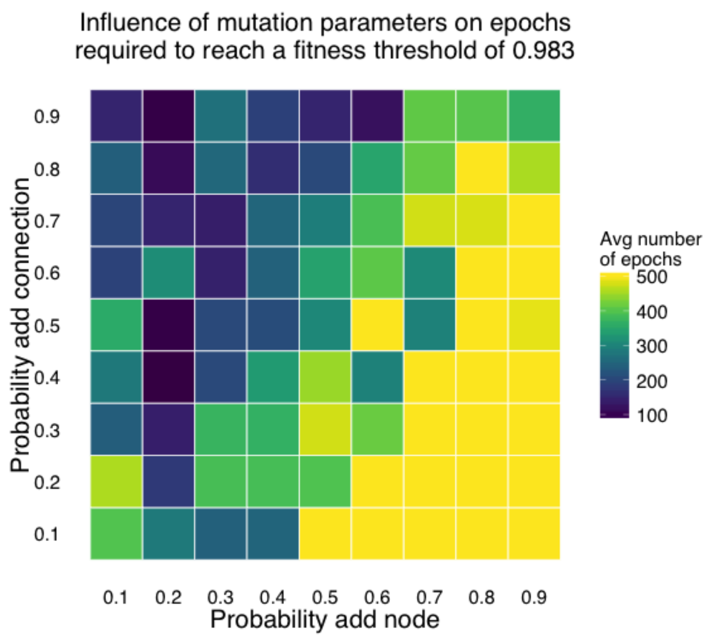

Figure 5: A heat map showing the average number of epochs needed to reach a $1 - MSE$ threshold of 0.983 as a function of the probability of adding a node and the probability of adding a connection during mutation. It suggests that an optimal combination is found when the probability of adding a node is 0.2 and the probability of adding a connection is 0.4 or 0.5.

5 Parameter Selection

It became apparent that choosing appropriate parameters is essential for ensuring the NEAT algorithm makes proper headway towards an effective and efficient neural network topology. My experimentation suggest that the parameters that dictate mutation probability are particularly important because they determine how finely or coarsely the algorithm searches over the search space. Large topological changes are equivalent to large movements over the fitness landscape; if these changes are excessively large, the algorithm will not be able to converge on an optimal or near-optimal solution. Conversely, if mutations don't occur frequently enough, topological diversification doesn't occur as often as desired, and the algorithm can stagnant.

To that end, I performed comparison tests examining the tradeoff between important and related parameters—two of those experiments are discussed here. The NEAT algorithm was run 3 times for each parameter pair in each

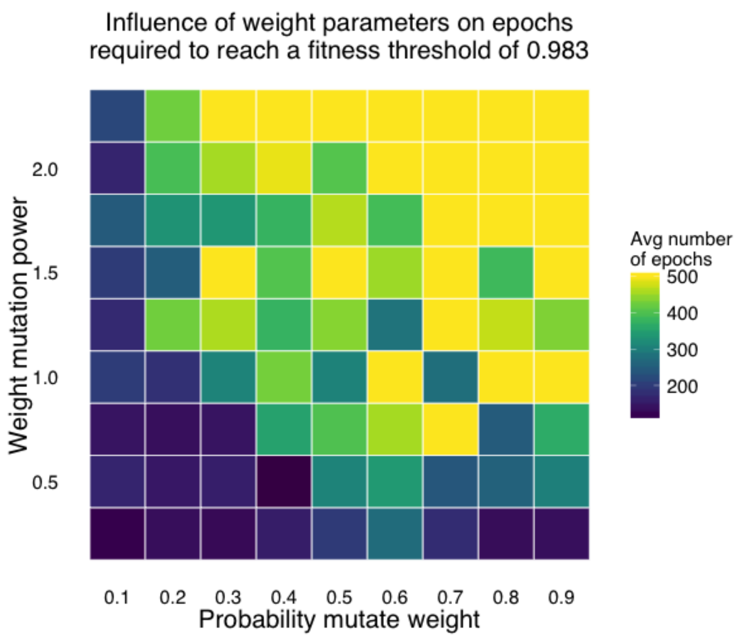

Figure 6: This heat map shows the average number of epochs needed to reach a $1 - MSE$ threshold of 0.983 as a function of the probability of mutating a weight and the weight mutation power (i.e. the standard deviation of the zero-centered normal distribution from which a weight change is drawn) during mutation. When both probabilities are high, the algorithm has difficulty reaching the error threshold.

test. The algorithm reached termination criteria when either $1 - MSE >= 0.983$ or a total of 500 epochs had been evaluated. I produced heats maps that illustrate the average number of epochs (across the three trials) reached for each combination of parameter values. It is important to keep in mind that these parameters are very domain-specific; for a different problem, a different fitness landscape exists, and thus a different set of parameters might be needed to yield optimal results.

The first comparison, illustrated in figure 5, alters the probability of adding a node with the probability of adding a connection. Both values were altered in the range $[0.1, 0.9]$ at intervals of 0.1. This heat map suggests that an optimal combination is found when the probability of adding a node is 0.2 and the probability of adding a connection is 0.4 or 0.5. Furthermore, as the probabilities increase in magnitude, NEAT is unable to make efficient progress towards an optimal network topology. This fits nicely with what intuition might suggest: excessively-large changes in structure mean we are sampling the fitness landscape too coarsely and are thus unable to converge on a optimal topology. The algorithm benefits from a relatively lower probability for adding nodes compared with the probability for adding connections, suggesting that, from a fitness perspective, adding nodes results in a larger topological change than adding connections. This also fits with intuition: adding nodes is mathematically equivalent to introducing non-linear relationships between inputs and outputs. This can possibly manifest as a particuarly large movement across the fitness landscape.

The second comparison, illustrated in figure 6, alters the probability of mutating a connection weight with the weight mutation power. The weight mutation power is the standard deviation of the zero-centered normal distribution from which a weight change is drawn; this weight change is added to the current connection weight. The heat map suggests that an optimal combination of parameters exists when the probability of mutating a weight is 0.4 and the weight mutation power is 0.5. Again, we see that when both probabilities are high the algorithm has difficulty reaching our threshold error within a reasonable number of epochs.

Interestingly, figure 6 suggests that as one of the parameters becomes increasingly small, the other can become increasingly large without adversely affected performance. Again, this is because smaller probability values (and smaller weight perturbations) represent smaller steps across the error surface; when one parameter decreases in magnitude, it allows more leeway in the other parameter to grow in size while still keeping the overall topological change manageable. Sometimes, a relatively small parameter value can compensate for a relatively larger parameter value. This compensation affect is not observed in figure 5 because adding a node represents such a large topological change. When this probability is large, the algorithm will struggle to find an optimal solution regardless of the other parameter value.

6 Data

For this study, I used the Wisconsin Breast Cancer Database (January 8, 1991), obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg [7]. This classic dataset seeks to classify breast cancer masses as benign or malignant. There were a total of 683 instances

(699 total minus 16 due to missing attribute information), of which 65.5% are benign and 34.5% are malignant. 80% of the observations were randomly selected for the training data; the other 20% were used to test results. The following 9 variables were used; each fell within a range $[1, 10]$ but was normalized to fall within $[0, 1]$:

1. Clump Thickness
2. Uniformity of Cell Size
3. Uniformity of Cell Shape
4. Marginal Adhesion
5. Single Epithelial Cell Size
6. Bare Nuclei
7. Bland Chromatin
8. Normal Nucleoli
9. Mitoses

7 Implementation

I used a python implementation of NEAT forked from the project by @MattKallada and further developed by McIntyre (aka CodeReclaimers), who is currently updating it to provide more features and a (hopefully) simpler and better documented API [8]. The python-neat implementation contains a configuration file within which all parameter values must be explicitly enumerated.

7.1 Fitness Function

At the conclusion of each epoch, a measure of the discrepancy between the observed and predicted values of the dependent variable is calculated. Often, this discrepancy is expressed as a mean square error (MSE), which for this study was the error function:

$$E(g) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{2}$$

where $E(g)$ is the mean square error term, N is the number of observation (input) vectors presented to the network, $\hat{y}_i$ is the predicted response, and y_i is the observed response. The N observations constitute the training data set. Fitness is simply $1 - E(g)$; a perfect organism would have a fitness equal to 1. Theoretically, NEAT should find the genome that represents the neural network that minimizes $E(g)$ (i.e. maximizes fitness).

8 Results

Figure 7 visualizes an evolved network with a mean squared error of 0.0248 and a correct classification rate of 97.08% on the test data set. There was no initial structure (i.e. only 9 input nodes and on output node). It took approximately 8 minutes to evolve, running in parallel on 12 cores with a population size of 240 (a similar neural network took 25 minutes to evolve sequentially). After the last generation during training, the genome had a fitness of 0.9868 (or $MSE = 0.0132$) while the population as a whole had an average fitness of 0.9549 (or $MSE = 0.0451$). There were a total of 12 species in the final population. The confusion matrix (see table 1) shows relatively low false positive and false negative rates.

		Actual	
		1	0
Observed	1	34.31%	1.46%
	0	1.46%	62.77%

Table 1: Confusion matrix

This network serves as a useful example for illustrating some interesting features inherent to NEAT. First, we can see that the algorithm employs automatic feature selection. Feature 8 is not connected to the rest of the network, implying that it is not particularly important when predicting output. Though it is possible, of course, that given more time to evolve, the best performing genome would connect feature 8 to the rest of the network. This eliminates the need for the programmer to perform independent feature selection. Secondly, we can see how some features (e.g. 1) are related linearly to

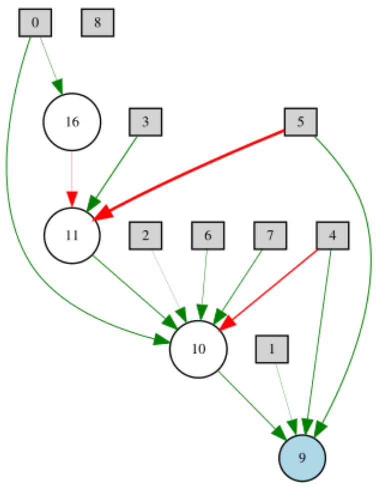

Figure 7: Dark grey boxes represent input nodes; white circles represent hidden nodes; and the blue circle represents the output node. The thickness of the connection corresponds to the magnitude of the weight. Green connections are positive, red connections are negative. Disabled connections are not shown.

the output while others (e.g. 0, 2, 3, 6, etc.) are not. This demonstrates that the algorithm is able to build up complexity over time, even when starting with no initial structure.

The "best genome" initially made very rapid progress (see figure 8), but began to plateau as topological complexity increased. The reason the average population fitness didn't also obviously improve is due to speciation. Although many organisms in the population were not particularly effective when compared to the best genome, they were protected by competing only within their species. This is desirable; it is very possible that sudden jumps in the best genome's fitness were a result of a new genome evolving to become a better predictor than the current best genome (as opposed to a beneficial mutation occurring in the best genome's species, though this is also possible). This kind of topological innovation is only possible through speciation, which can be visualized in figure 9. The initial population (containing 240 organisms) is initially all one species; as generations progress, organisms begin to diverge and, once the compatibility threshold is crossed, a new species is created (for example around epoch 700). After 1000 epochs, there are 12 species present in the population.

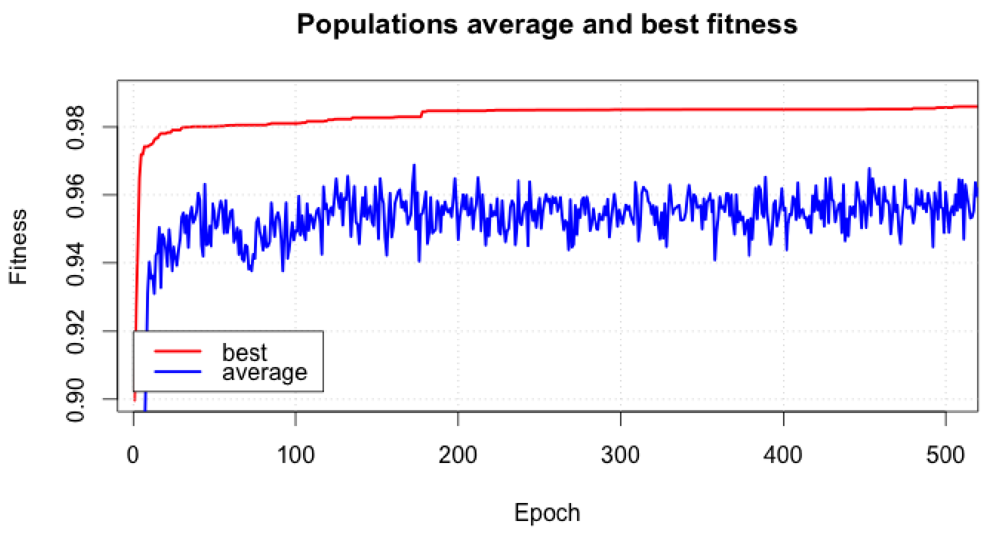

Figure 8: A plot visualizing how fitness improves over generations. The "best genome" initially made very rapid progress, but began to plateau as topological complexity increased.

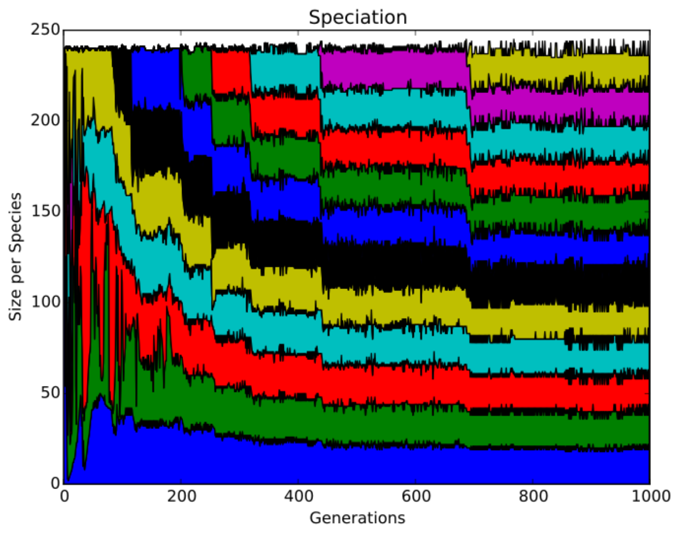

Figure 9: Each color represents a different species in the population. As generations progress, more species are realized as topological diversity occurs.

9 Conclusion

NEAT proves to be an effective method for evolving neural networks that would be unrealistic to create by hand. The algorithm is able to approach comparable classification results in a surprisingly short number of generations. By using historical markers, the competing conventions problems is alleviated and speciation is made computationally efficient. In addition, automatic feature selection is an interesting “bonus” benefit that fits with the NEAT philosophy of preventing a programmer from making potentially sub-optimal decisions. However, it is clear that appropriate parameter selection is critical for evolving networks in an efficient manner. If probabilities of mutation are too large, genomes take excessively large steps over the fitness landscape, and are unable to converge on an optimal or near-optimal solution. This is, of course, a tradeoff: as some parameters decrease in magnitude (materializing as finer search steps), others are allowed to increase in size without sacrificing convergence to an optimal or near-optimal solution.

Like many genetic algorithms, NEAT is particularly well-suited to parallelization. This can greatly improve computation time, as well as allow for larger populations. My results indicate that SMP is preferable to cluster computing as long as the chromosome evaluation remains somewhat trivial; as topological complexity or the amount of data increases, it may prove beneficial to move to a cluster environment.

Acknowledgement

I want to thank my thesis advisor, Stephanie Taylor, for her guidance and for always making herself available to provide thoughtful and useful feedback. Your dedication to your students–and their success—is admirable. I also want to thank Bruce Maxwell for his helpful comments.

References

[1] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989.

[3] Kenneth Alan DeJong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[4] Dario Floreano, Peter Durr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[5] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, May 2015.

[7] O. L. Mangasarian and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News*, 23(5), 1990.

[8] Alan McIntyre. Python implementation of the neat algorithm.

[9] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1998.

[10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[11] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37, 1992.

[12] Jurgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[13] Kenneth O. Stanley. *Efficient Evolution of Neural Networks through Complexification.* PhD thesis, The University of Texas at Austin, August 2004.

[14] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[15] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[16] Darrell Whitley. A genetic algorithm tutorial. Computer Science Department, Colorado State University.