Combining Artificial Neural Networks with Genetic Algorithms in Order to Recognize Handwritten Digits from the MNIST Database

Curtis Andrew Schaff

Abstract

Computer vision is an important part of the field of computer science. Historically, image recognition has posed a major challenge to those working in this field. This challenge arises from the difference between the way the human brain processes images and the way a computer processes images. Of the many approaches taken to solve this problem, artificial neural networks show the most promise. However, it can be difficult to determine the best structure of an artificial neural network to achieve this goal. The purpose of this paper is to investigate the following question using genetic algorithms as an optimization strategy: **How can computer program be designed to discover the optimum neural network structure in order to recognize handwritten digits?** This essay assumes a basic knowledge of the operation of neural networks. For information on neural network construction and training, see Appendix A.

In order to investigate this research question, a series of programs were created to implement a basic genetic algorithm designed to evolve a population of neural networks with the goal of recognizing handwritten digits provided by the Mixed National Institute of Standards and Technology (MNIST) dataset. After sufficient testing, the test program was started, running through 25 iterations of evolution before terminating.

This investigation concludes that it is possible to create a program designed to optimize the structure for a neural network with the goal of recognizing handwritten digits. However, after some initial improvement, the genetic algorithm began to evolve populations with decreasing accuracies. This fall in performance was likely due to a small error in implementation, rather than a fundamental issue in the methodology of this investigation.

(270 Words)

Table of Contents

Table of Figures

Introduction

Artificial Intelligence (AI) is an important field, benefiting many and often working behind the scenes in our normal lives, from the navigation software in a GPS to the facial recognition software on websites such as Facebook. AI research has also contributed much to our lives, from better speech recognition engines in our phones to the development of self-driving cars.

A significant amount of AI research is dedicated to creating intelligent programs (agents) that can improve on existing programs. Doing so involves applying techniques such as machine learning, natural language processing, and machine vision, among others. Artificial Neural Networks make up one particular approach to the development of AI.

Artificial Neural Networks (ANNs) are not a particularly new concept in the field of AI, but development has been hampered in the past by a lack of efficient methods by which to train the networks. Fortunately, the recent breakthrough of backpropagation has allowed research to move forward. ANNs are mathematical models that can be 'trained' to recognize patterns, imitate functions, and perform analysis. Much like a brain, a neural network consists of neurons, whose connections can be manipulated to store and process information. In the case of an ANN, each neuron represents a mathematical formula which is applied to a signal before it is passed on to the next neuron or neurons in the network.

When designing a network, selecting the correct structure is crucial to arriving at good training performance and activation. However, determining the correct structure for an ANN can be difficult, as there are no real formulas or precise techniques to assist in this process (Nielsen, 2016). At the moment, there are a few heuristics that can help approach the correct structure, but further tweaking is always necessary to fully optimize performance.

Although there may not be an analytical solution to the problem of neural network design, there is a computational solution. Genetic Algorithms (GAs) can be used to optimize ANNs through a process of simulated evolution. A series of networks are created, tested, and sorted. The best performing networks are 'bred' and move on to the next iteration of evolution, leaving the worst networks behind (Larson, 2009).

The focus of this essay is to investigate the following question through the use of genetic algorithms: **How can a computer program be designed to discover the optimum neural network structure in order to recognize handwritten digits?** To address this, I have created a program in Python that works to evolve a feedforward network that can recognize handwritten digits from the MNIST dataset.

The CAPTCHA system takes advantage of the difficulty computers have in image recognition. By displaying warped text that only humans should be able to read, automatic systems are prevented from creating unwanted accounts to many online services.

The efficacy of the CAPTCHA system is due to the fundamental difference between the way humans see images and the way a computer 'sees' an image. When a computer loads an image into memory, the information it has is restricted to individual pixels, their intensities in various channels, and their positions relative to other pixels. In this way, the computer must work from the ground up, using solely pixel relationships, to pick out patterns and identify features. However, when a human sees an image one of the first processes to occur in the visual cortex is edge detection, an important first step in feature recognition (BrainHQ, 2015). In this case, simple features (i.e. lines and shapes) are detected by small sets of neurons, with more complex features (i.e. faces and objects) detected by larger networks of neurons working together. This difference in processing technique from pixel-based to feature-based poses one of the greatest challenges in the field of computer vision. ANNs are able to overcome this difficulty by processing images more like the human brain, by looking at patterns, rather than just pixels.

Artificial Neural Networks are mathematical models with the power to change a number of elements in the field of computer science, including computer vision. Their structure is simultaneously the key to their power and the reason they are difficult to use. This is especially true with image recognition: a neural network is very effective in pattern recognition, but the correct structure must be obtained in order to take advantage of this power. This essay explores the use of genetic algorithms to determine the optimum structure for a neural network which has a similar goal to pattern recognition, that of recognizing handwritten digits.

MNIST Dataset

The MNIST dataset is made up of approximately 70,000 images of handwritten digits. The National Institute of Standards and Technology (NIST) compiled this database with the intention of using it in machine learning applications. Each image is a 28 × 28 pixel gray scale image, as shown in Figure 1.

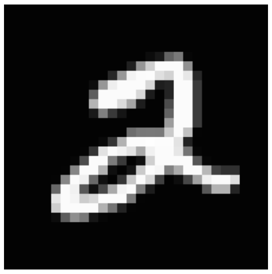


Figure 1 - A sample from the MNIST training set

The datasets used by the test program are stored in `mnist.pkl`, a file provided by Michael Nielsen for use in his online textbook *Neural Networks and Deep Learning*. Contained within this file are three different datasets: a training set, a validation set, and a test set. The training set is intended for use only when training neural networks. Similarly the validation and test sets are intended solely for evaluating neural networks and performing final tests, respectively. Each one of these datasets is made up of a list of 2-element tuples, each containing image data in a 28 × 28 matrix and an integer representing the number written in the image.

Implementation

The majority of the programming behind the algorithms described in this paper was done in the programming language Python 2.7 with some external libraries based on programs written in C, a comparatively faster programming language. Overall, the research shown here was performed with around 2000 lines of Python code spread over eight files, excluding external libraries. The main programs and their tasks are detailed in Table 1. Each of the programs in bold is available online on GitHub via the following link: https://github.com/CurtisAndrewSchaff/Extended-Essay.

Filename	Description
`__init__.py`	A starter program. Received population details as an input and initiated the genetic algorithm
`GA.py`	Handled all evolution and evaluation tasks.
`networkLib.py`	Provided a simple system to keep track of network sizes and training performances, as well as simplifying the process of network breeding.
`Graphics.py`	A library to handle the main progress window. This provided a visual representation of training and evolution status.
`mnist.py`	Created all datasets using a downloaded copy of the MNIST dataset.
`Custom_Evolution.py`	Allowed for the creation, training, and evaluation of custom networks.
`Grapher.py`	Created performance graphs after the test program had terminated.

Table 1 – Main Python Programs. All programs listed with bold filenames formed of the main program loop, all other files provided assistance or post-processing capabilities.

Dataset Management

The datasets used to train and evaluate each neural network were processed by the program `mnist.py`. This program handled the setup of the three MNIST datasets that were included in `mnist.pkl`. Each element of each dataset was stored as a member of the `mnist.number` class. This class was created to facilitate the processing and display of image data, as well as the setup of the lists needed to train and evaluate networks. The image data was stored in a list containing 784 elements, with each element representing the intensity of a gray scale pixel in the image. The network structures in use dictated this storage format, as none of the networks could accept inputs as a 28 × 28 pixel 2D array.

As well as the image data, the `mnist.number` class stored the desired output of the network as supplied by the MNIST dataset. However, in order to evaluate the network, the integer provided must be converted into a form that matches that of the network's output. In order to do this, the `mnist.number` class creates a list of numbers, setting all elements to zero with the exception of the element that pertains to the desired number, which is set to 10. This value is then stored with the image data for use in network evaluation.

Neural Networks

All of the programming relevant to the creation, storage, training, and activation of networks was accomplished using the `Pybrain` library. `Pybrain` is a freely available artificial intelligence-oriented library of which neural networks make up only a part. The file `networkLib.py` provided a simplified interface to the `Pybrain` library, easing the process of network training, evaluation, and storage.

However, `Pybrain` did not provide all of the functions required by this paper's program. In order for the genetic algorithm to operate, each network needed an evaluation algorithm and the ability to store its performance. Similarly, the genetic algorithm required a simple way to breed networks, preferably with simple lists to keep track of network structures, as opposed to the system provided by `Pybrain`, which was deemed unsuitable for this application. These are some of the reasons behind the development of `networkLib.py`. This program was created to permit the manipulation of networks in a more object-oriented fashion, as well providing a simplified breeding and evaluation process.

The `networkLib.netObj` class provides a simpler way to activate, train, evaluate, and store networks. This class greatly simplifies the process of creating and testing neural networks, which is essential for efficient operation of the genetic algorithm.

Network training was provided through the `networkLib.netObj.train` method. This method trains the network using `Pybrain`'s `BackpropTrainer` for 10 training epochs or until the network's Mean Squared Error (MSE) drops below 1, the target MSE for this networks discussed in this paper. After each epoch, the network's MSE is stored, enabling another program to return and analyze the network's performance after the genetic algorithm has completed. Following training, the network is evaluated to determine its success relative to other members of the population.

Easy network evaluation is essential when gauging the efficacy of the genetic algorithm. The `networkLib.netObj` class makes evaluation very simple through its `evaluate` method. This method provides two algorithms for network evaluation. The first algorithm, selected by setting the argument `getSimilarityIndex` to `True`, judges the network based on how similar its normalized output is to the normalized form of the correct output, as provided by the `mnist.number` object. This metric was used by the genetic algorithm, but is relatively meaningless elsewhere. The second algorithm, selected by setting the argument `getSimilarityIndex` to `False`, returns a value that can be multiplied by 100 to determine the network's percentage accuracy. Regardless of the metric selected, the underlying evaluation process is the same: the network is tested against every image in the MNIST validation set and its success recorded. In the case of the first metric, a lower number is more desirable, as this represents a higher similarity between the normalized output of the network and the desired output from the given input. Conversely, a higher value is preferable for the second metric, as that would indicate a higher success rate in determining the handwritten numbers in its input images.

`networkLib.py` also provides a simple breeding method, which relies on the parent networks' stored structures to create a new child network with a hybrid structure. It is necessary to store the parents' structures independently from `Pybrain`'s network object, as doing so allows for a much simpler breeding process. The details of this algorithm are described later on in this paper.

Graphics

The graphical elements of this program were created using two libraries: `Pygame` and `matplotlib`. Both of these libraries are completely open-source and freely available online. `Pygame` was used to create a graphical progress indicator, which became indispensable during long training sessions. `matplotlib` was used to produce the training progress graphs after all networks had been trained and evaluated. These graphs display the Mean Squared Error (MSE) of each network after each training epoch.

The file `Graphics.py` provided a simpler interface to the `Pygame` library. This script organized the process of drawing and updating the values on the graphical progress window.

Genetic Algorithms: An Introduction

A Genetic Algorithm is an algorithm that mimics the process of biological evolution to optimize a population of objects. In this case, the genetic algorithm was created to optimize the performance of neural networks by breeding the networks that performed better to create a series of well-adapted network. Almost all of the networks discussed in this paper were generated by a genetic algorithm.

Genetic algorithms are made up of three main elements: a population generator, a breeding algorithm, and an evolution algorithm. When run in a loop, these elements make up a powerful optimization system.

Population Generator

The population generator is the first part of a genetic algorithm. This function generates a predetermined number of objects to create the initial population. In some cases, such as those in which the initial population is already supplied, this element may be unnecessary.

In the test program, this function generates a neural network with a random set of hidden layers, each with a random size. However, in order to keep network size (and, as a result, training time) low, both of these parameters were restricted to a range of values. At the start, the maximum number of neurons in a layer was set at 25 and the maximum number of hidden layers was set to one.

Breeding Algorithm

The breeding algorithm is the heart of a GA. This algorithm facilitates the creation of a new object from two parent objects. As a result, the design of a breeding algorithm has an immense impact on the success of the GA. If the breeding algorithm fails to produce a well-designed child for two similarly well-designed parents, the entire GA will fall apart, running in an endless loop and failing to improve on its initial population.

As well as being responsible for the creation of new objects, a breeding algorithm must also ensure genetic diversity through the occasional random mutation of the parent objects' genetic information. These mutations are usually performed by adding more data, subtracting data, or changing existing data. The probability of a mutation occurring should be set relatively low, so as to create genetically diverse objects at times, but to produce well-functioning children the majority of the time. Performing mutations is a crucial part of a genetic algorithm, as it can prevent the GA from becoming stuck at local maxima (as shown in Figure 2), populations where each member is well-adapted to its environment, but not optimally adapted. Without mutations, the GA can never exit a local maximum, an act that, though undesirable in the short run, could potentially lead to the discovery of a higher global maximum.

The GA utilized for this paper uses the breeding algorithm described in Figure 3. As shown, the algorithm (known as `networkLib.breedNetworks`) begins by building up a child network from the parent networks' genetic information (the size of each of the parent networks' hidden layers). This particular stage utilizes a bias that determines which parent gives more genetic information. This can be useful if a preferential selection is desired, but should be kept at 0.5 when not needed.

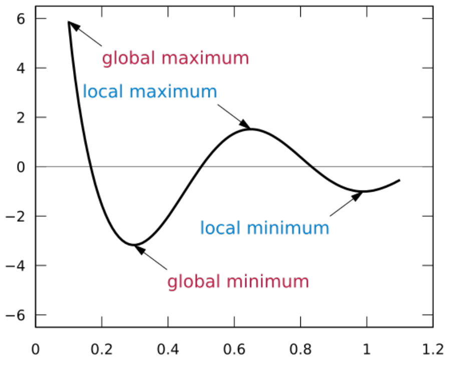
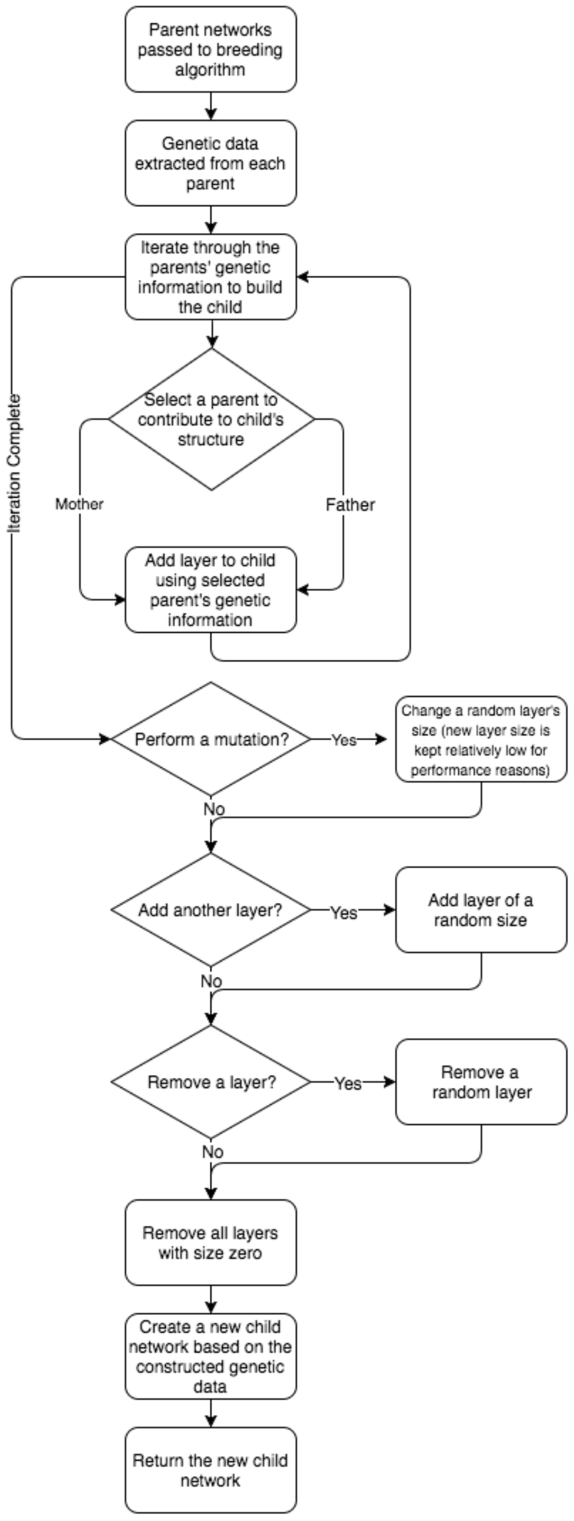


Figure 2 - Local and Global Maxima (Image Credit: Wikipedia, 2007)



Parent networks passed to breeding algorithm
Genetic data extracted from each parent
Iterate through the parents' genetic information to build the child
Select a parent to contribute to child's structure
Mother
Father
Add layer to child using selected parent's genetic information
Iteration Complete
Perform a mutation?
Yes
Change a random layer's size (new layer size is kept relatively low for performance reasons)
No
Add another layer?
Yes
Add layer of a random size
No
Remove a layer?
Yes
Remove a random layer
No
Remove all layers with size zero
Create a new child network based on the constructed genetic data
Return the new child network

Figure 3 - The Breeding Algorithm

After the initial child is constructed, operations may be performed on its structure. These operations include the mutation of an existing element, the addition of a new layer, or the subtraction of an existing layer. As discussed above, these operations should occur infrequently, but help to ensure genetic diversity.

Finally, any empty layers are removed from the child, as such layers will cause errors when the child becomes a fully operating neural network.

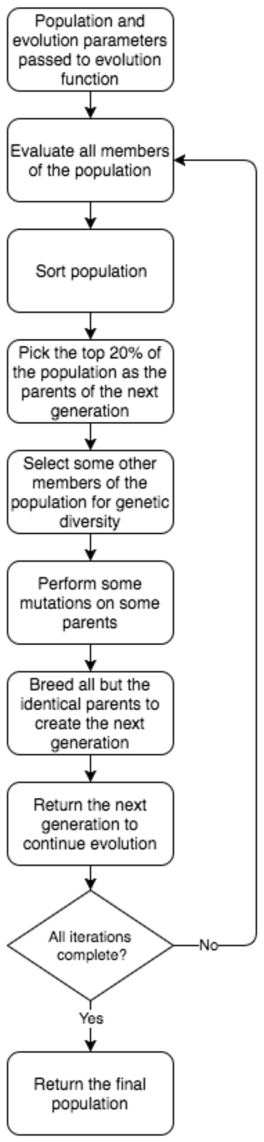
Evolution Algorithm

The evolution algorithm is what makes a GA different from other optimization algorithms. This algorithm facilitates the evaluation of a population, and the construction of a new population based on the strongest members of the previous population.

The evolution algorithm employed in this paper works as shown in Figure 4. This algorithm (which makes up the function `GA.evolve)` starts by evaluating the population, sorting the population based on each member's performance, and selecting the best 20% of the population as the parents of the next population. After this, the algorithm may select a few random lower-performing members of the population to encourage genetic diversity. Finally, the algorithm generates the next generation of networks by breeding the selected parents using the aforementioned algorithm.

After the new generation of networks has been generated, the cycle repeats. The new generation trained and then fed back into the evolution algorithm, where the networks are once again evaluated, sorted, and bred. Finally, the GA completes after a set number of iterations with the final population taken to be well optimized.

In order to determine how effective the population is, we must determine the grade of the population. After every member has been created, each is trained and tested with its error or score recorded. Following this, these values are averaged across the entire population, producing what is known as the population's grade. In the case of my algorithm, the values recorded from each network consisted of the network's MSE after its final epoch of training. As such, the target of my genetic algorithm was to minimize the population's grade.



Population and evolution parameters passed to evolution function
Evaluate all members of the population
Sort population
Pick the top 20% of the population as the parents of the next generation
Select some other members of the population for genetic diversity
Perform some mutations on some parents
Breed all but the identical parents to create the next generation
Return the next generation to continue evolution
All iterations complete?
No
Yes
Return the final population

Figure 4 - The Evolution Algorithm

Program Output

The test program terminates after it completes the requested number of iterations of evolution. All of the networks involved in this process can then be found in a nearby directory, where each network has been classified by the iteration it belonged to and the order in which it was evaluated. These networks have been saved after training and evaluation, allowing one to reconstruct the network in another program, view training performance or test the network with specific inputs. This allows another program (`Grapher.py`) to generate individual performance graphs for each network, providing a visual representation of the efficacy of its structure.

Testing

To ensure the functionality of the programming behind this paper, I tested it incrementally, gradually increasing population size and number of iterations until I was confident that the test program was operational. The rationale behind starting with small populations and numbers of iterations relates to the speed of the test program. Training a neural network normally takes upwards of 15 seconds per epoch, with larger networks taking far longer to train than smaller networks. In order to ensure that the test program ran as well as possible, it was tested with small populations of small networks, which allowed me to discover bugs that could arise during the transition between generations without having to wait for large networks to be trained and evaluated.

Testing in this manner proved to be beneficial, as it revealed a simple, but significant flaw in my programming. The issue related to the way that the datasets used to test and train networks were stored. Originally, these datasets were stored with the networks, with one copy per network. However, this quickly exceeded Python's memory allocation when using large populations, and caused the test program to crash. This was rectified by removing a single keyword, ensuring that all of the networks accessed a single shared dataset, drastically reducing the memory required to run the test program, and significantly increasing its speed.

Results

After a sufficient amount of testing was conducted, I started collecting data. The genetic algorithm generated a population of 25 networks, each starting with at most 1 hidden layer with as many as 25 neurons. The test program ran for 25 iterations before terminating, producing the data shown in Figure 5 in the process.

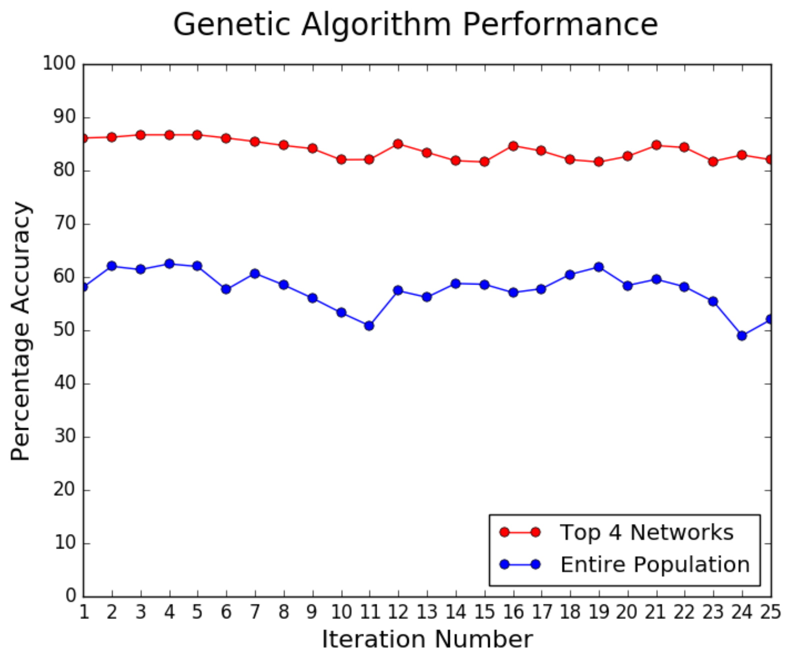


Figure 5 – Percentage accuracy of best four networks compared to that of the entire population

Figure 5 shows the percentage accuracy of the top four networks compared with that of the overall population. In this case, the percentage accuracy is based on the metric described earlier – the percentage of correct determinations from the MNIST validation set. As can be seen, the best four networks from each iteration averaged around 85% to 87% success at identifying handwritten digits. The overall population was slightly more volatile, oscillating between 49% and 62%.

However, after the first five iterations, performance started to drop, which could be due to one of several reasons. A likely conclusion is that, as the networks became more complicated, the ten training epochs allocated were insufficient for the network to perform adequately, suggesting that the network could have performed better with a longer training period. However, this fall in performance could also indicate more significant issues in the genetic algorithm. In this case, the most likely scenarios are failures in the breeding algorithm or the parent selection process of the GA.

If this error stemmed from the parent selection process, it is relatively easy to fix. All that is required is to ensure the algorithm is stricter when selecting parents (i.e. only selecting parents with errors above a predetermined threshold or by simply retaining fewer parents across iterations).

Alternatively, if the issue is a result of the breeding algorithm, the algorithm may have to be modified slightly or completely rewritten, depending on the location of the error. If the decrease shown in Figure 5 is a result of networks slowly growing and losing performance in training, the breeding algorithm would only need to be slightly modified to preserve the number of hidden layers. Alternatively, this error may be indicative of a much more fundamental issue, such as a poor recombination of genetic information, requiring significant changes. Such a fix would likely require further research into approaches such as Neuroevolution through Augmenting Topologies or NEAT (O. Stanley & Miikkulainen, 2002).

However, a steady decrease in performance does not necessarily indicate a failure. During its run, the genetic algorithm retained every single network from every iteration involved in the data collection process. This means that the networks produced by the final iteration do not represent the full output of the genetic algorithm. Rather, another program can search through the data, looking for the network with the best performance out of any other network from any iteration. In doing so, the network shown in Figure 6 was found. After training, this network had an accuracy of around 87.5%. As a comparison, some experimentation with custom neural network structures yielded the structure shown in Figure 7, which had an accuracy of around 72.8%. This network represents the highest accuracy I could achieve by designing the network structure myself. I used the program `Custom_Evolution.py`, which allows for the creation and testing of neural networks with custom structures to conduct this experimentation. The fact that the genetic algorithm managed to beat my best network structure by 14.7 percentage points proves the utility of the combination of genetic algorithms and neural networks.

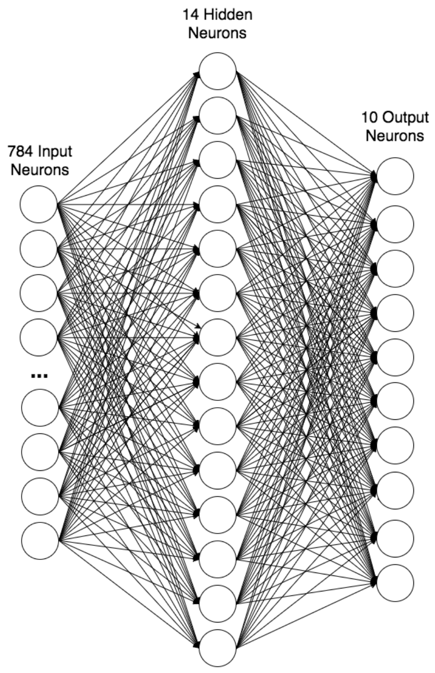


Figure 6 - Best neural network found by GA

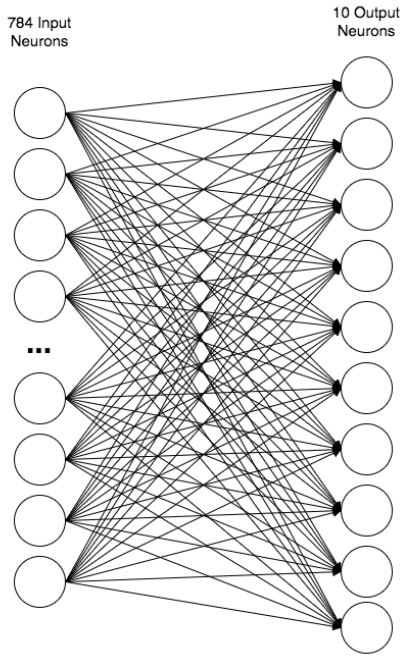


Figure 7 - Best neural network found through experimentation

Conclusion

These results are proof of the efficacy of the combination of genetic algorithms and artificial neural networks. After the first five iterations the population's performance begins a downward trend, with periods of improvement followed by further decreases in performance. This, however, does not indicate a failure in the approach. As the genetic algorithm used in this program was set up to save each population after training and evaluation, another program can search through the accumulated data to find the best network out of every iteration, as described earlier. This allows the algorithm to arrive at a near-optimal network structure without having to have it in the final population. The fact that the genetic algorithm can outperform a human operator in some cases demonstrates the validity of using genetic algorithms to determine the optimum structure of neural networks designed to recognize handwritten digits. These results prove the utility of this approach, even if the final population isn't the highest performing, the best structure found can still be obtained by looking through the data generated from previous iterations.

This shows that it is possible to design a computer program that uses genetic algorithms to discover the optimum neural network structure in order to recognize handwritten digits. This topic shows potential for future investigations. For example, can this technique be used to optimize neural network structures for other applications? Could genetic algorithms be combined with reinforcement learners to create an agent with a brain that optimizes its own structure? Individually, genetic algorithms and neural networks are effective and adaptable solutions to many problems in the field of artificial intelligence. However, when combined, they can be used to create powerful systems capable of advanced feats, such as the recognition of handwritten digits. Artificial Intelligence is a promising field and the combination of neural networks and genetic algorithms is a powerful approach to many of the dilemmas it poses.

Bibliography

BrainHQ. (2015, July 28). *How Vision Works*. Retrieved April 1, 2016, from BrainHQ: http://www.brainhq.com/brain-resources/brain-facts-myths/how-vision-works

DeepLearning.TV. (2015, December 15). *Restricted Boltzmann Machines - Ep. 6 (Deep Learning SIMPLIFIED)*. Retrieved July 16, 2016, from YouTube.com: https://www.youtube.com/watch?v=puux7KZQfsE

KSmrq. (2007, June 21). *Maxima and Minima*. Retrieved June 17, 2016, from Wikipedia: https://commons.wikimedia.org/wiki/File:Extrema_example_original.svg

Larson, W. (2009, January 2). *Genetic Algorithms: Cool Name & Damn Simple*. Retrieved July 2015, from Irrational Exuberance: http://lethain.com/genetic-algorithms-cool-name-damn-simple/

National Institute of Standards and Technology. (n.d.). Mixed National Institute of Standards and Technology Database.

Nielsen, M. (2016, January). *Using Neural Networks to Recognize Handwritten Digits*. Retrieved May 12, 2016, from Neural Networks and Deep Learning: http://neuralnetworksanddeeplearning.com/chap1.html

O. Stanley, K., & Miikkulainen, R. (2002). *Evolving Neural Networks through Augmenting Topologies.* The University of Texas at Austin, Department of Computer Sciences. Austin: The MIT Press Journals .

Appendix A – Neurons, Neural Networks, and Neural Network Training

Neurons

A neural network is made up of many of individual neurons where each neuron is responsible for transforming its input data and passing it on to every neuron in the next layer. These transformations are performed by a single equation, with different equations for different types of neurons. These equations produce the neuron's output based on three variables: the input vector (x), the weight vector (w), and the neuron's bias (b).

The two variables that relate to the connections between networks are the input and weight vectors, where the weight vector represents the weight multipliers applied to each input to the neuron. Both the input and weight vectors are constructed such that each input and each weight corresponds to one component of the vector. By taking the scalar product of both vectors, one can obtain a value for the total signal the neuron receives from the layer before it.

Finally, the neuron's bias is its overall threshold of activation. If the neuron has a very negative bias, it will have to receive much stronger input signals for the neuron to produce a strong output. Similarly, if a neuron's bias is set very high, the neuron will remain active for all but the weakest inputs.

Two common types of artificial neurons are perceptrons and sigmoid neurons. These two neurons are differentiated by the functions that govern their activations.

Perceptrons are very much binary neurons, they can output a zero or a one with no values in between. Likewise, they can only receive inputs of a zero or a one, any value in between is an invalid input for a perceptron. The function that governs the activation of a perceptron is shown in Equation 1. As shown, a perceptron will only activate (output a one) if its inputs are especially strong or weights and bias especially high. These neurons are useful in basic binary systems due to their ability to imitate a number of digital logic gates when used in a small neural network. However, due to their binary activation function, performing more complex tasks can be very difficult. This behavior means that a small change in the weights or bias of a perceptron can cause its output to flip from a zero to a one or vice versa,

$$output = \begin{cases} 0 & if\ w \cdot x + b \leq 0 \\ 1 & if\ w \cdot x + b > 0 \end{cases}$$

Equation 1 - Perceptron Output Function (Nielsen, 2016)

resulting in unpredictable behavior in the layers that follow. In order to perform more complex functions, a neuron with a smoother output function is required.

Enter the sigmoid neuron. Rather than following the binary function described in Equation 1, sigmoid neurons follow the exponential function described in Equation 2. This function allows a small change in weights and inputs to cause a small change in output, permitting a greater learning ability. Sigmoid neurons can also accept inputs anywhere between zero and one. All of the networks discussed in this paper are constructed of sigmoid neurons.

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad z = \sum_j w_j x_j - b$$

Where $\sigma(z)$ represents the neuron's output and w_j and x_j represent the weights and activations of neuron *j* in the previous layer respectively.

Equation 2 - The Sigmoid Function (Nielsen, 2016)

Neural Network Structure

Most neural networks are made up of three main parts: an input layer, one or more hidden layers, and an output layer. In many neural networks, every neuron in each layer is connected to every neuron in the next layer, as is the case for all of the networks described in this paper. Although there are a number of different types of neural networks, this paper focuses on feedforward networks, the most basic neural network structure. If a neural network contains two or more hidden layers, it can be further classified as a deep neural network. Deeper neural networks are often able to perform more abstract processing and can work on tasks of greater difficulty.

The process by which a neural network receives and processes an input is known as activation. Activating a network and comparing its output with the desired output for a given input is a common method of evaluating the performance of neural networks.

Feedforward Networks

A feedforward neural network (shown in Figure 8) is one of the most basic types of network. This network consists of an input layer, one or more hidden layers, and an output layer. As

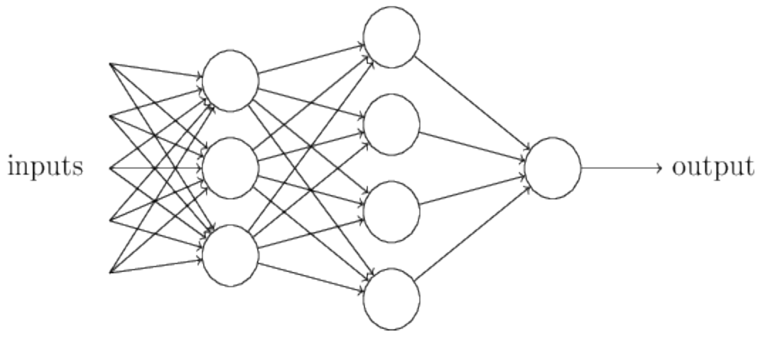


Figure 8 - A basic feedforward network (Image Credit: Michael Nielsen, 2016)

is the case in most ANNs, all of the outputs of one layer travel to every neuron in the next layer, creating what is called a fully connected layer.

Feedforward networks are generally trained using the Backpropagation and Stochastic Gradient Descent (SGD) algorithms. While the inner workings of these algorithms are beyond the scope of this paper, suffice it to say that this pair is one of the most common in neural network training. All of the networks described in this paper were trained by the Backpropogation and SGD algorithms.

Neural Network Learning Strategies

Before use, every neural network must be trained on a dataset. There are three main forms of training: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning makes use of a labeled dataset, where every element of the dataset contains the data a network would receive as an input as well as the desired output of the network. Such a learning technique can only be performed offline, as the dataset it requires is often a vast collection of inputs, where each element is independent of every other element. For example, a network could be trained to recognize playing cards from a deck using images of the faces of the cards and an example of a network output pertaining to each card.

Unsupervised learning uses an unlabeled dataset. A network being trained using unsupervised learning would have to determine for itself an appropriate way to classify the elements of a dataset. This type of learning can be performed online, as a network does not need a large predefined dataset to learn. An example of unsupervised learning would be a network given the task of sorting small blocks. The network could end up sorting the blocks by color, size, or weight. Without labels (desired network outputs), the network may sort the blocks unpredictably.

Finally, reinforcement learning (RL) works on a basis of positive or negative reinforcement. Reinforcement learning is a powerful technique and can only be performed online. RL works by having an agent (the network) interact with an environment, after which the agent receives the updated state of the environment and an often positive reinforcement in the form of a score, similar to the technique used to train animals. After an animal performs a trick or action correctly, the trainer rewards it. Otherwise, the trainer may scold the animal or withhold the reward. This is very similar to the way in which an agent using RL would learn in a computer environment. First, an RL agent interacts with its environment. If it does so correctly, it gets rewarded. If it fails, it receives a negative reward. A classic example is that of an agent designed to navigate a maze. Such an agent is rewarded for completing the maze and will receive negative rewards for failing to do so. After the training is complete, the agent will have learned to navigate a maze.

All of these training approaches share some common terminology, most notably the concepts of a training epoch and Mean Squared Error (MSE). A training epoch is one iteration of training. Over the course of an epoch, supervised and unsupervised learners will learn from their respective training datasets and a reinforcement learner will complete an interaction with its environment and receive reinforcement. After an epoch, the MSE of an agent is computed. This value represents the performance of the agent during training. A low score is indicative of a low error, thus a well-performing agent. All of the networks described in this paper were trained with a target MSE of 1.