Automating Optimization of Artificial Neural Network Architecture

Nikhil Iyer

Fair of Science and Engineering

# Automating Optimization of Artificial Neural Network Architecture

# Abstract

Artificial neural networks are biologically inspired computational structures that are capable of “learning” to perform a task heuristically when presented with examples as opposed to being programmed directly by the developer. Unfortunately, due to their “black box”-type nature it is difficult to optimize them for properties such as speed of execution or accuracy. This study seeks to produce a software that is capable of optimizing the architecture of a neural network to fit the researcher’s needs. I created my own implementation of a neural network that allows for a wide variety of network architectures to be constructed and validated quickly. Each network is trained and scored; the algorithm assigns each of the networks a fitness score. The bulk of the logic occurs in an evolutionary algorithm that tunes structural elements such as connections and network parameters by maximizing the fitness function. Using this strategy, I was able to find the simplest network that can achieve at least 90% accuracy recognizing handwritten numbers. It was also able to produce a 4-node 5-connection network that is able to achieve 100% evaluating the XOR function. These results indicate that my software can be successful in autonomously developing a neural network that closely fits the requirements set forth by the developer. Neural networks are used for everything from identifying malignant tumors to facial recognition.

# Background

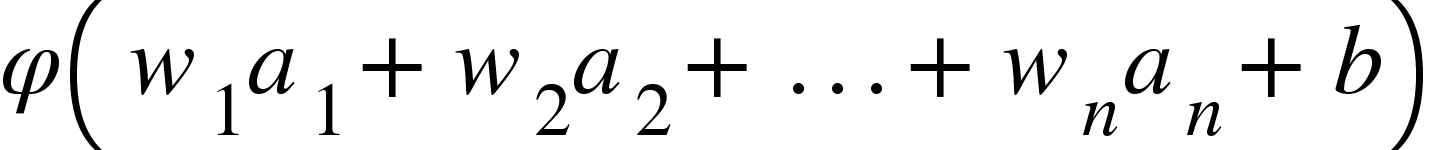
## Neural Networks

In its most essential form, an artificial neural network is a style of coding that is able to tackle complex problems by mimicking the structure and thereby the capabilities of

*Fig.1: Visualization of an XOR network with higher values being lighter (True=1=white; False=0=black). The first two nodes on the left are the input layer. The middle two are the ‘hidden layer’. The last node on the right is the output layer.*

a biological brain. They are characterized by a set of virtual nodes with weighted connections between them. Each node holds a value which it passes along each of the connections leading out of it. When a value is passed through a connection it is multiplied by the weight of that connection, added to all the other weighted values being passed to the receiving node, and transformed by any nonlinear activation function (so that the network may be capable of learning non linear patterns).

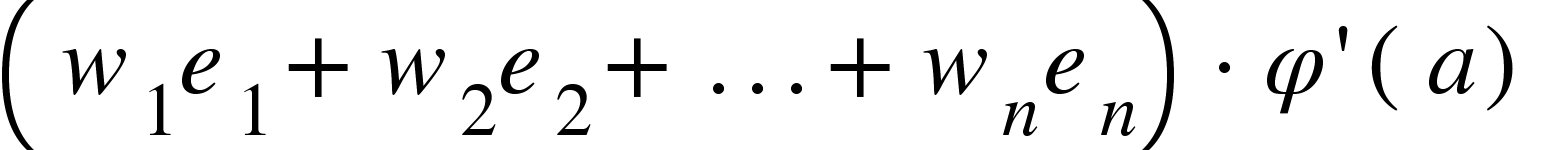
***Eq. 1: Activation of a Node***

*Activation of a node with n incoming connections where wn is the weight of the nth connection, an is the activation of the nth connecting node, b is an optional bias and φ is any nonlinear function.*

This structure allows the network to learn by encoding different weights on its connections.

Traditionally, the nodes are organized into layers. The first layer is the input layer; this is where inputs are passed in as an ordered list of values (traditionally from 0 to 1). Next there are the “hidden layers”; these layers are where the neural network performs the majority of its processing and learning. Finally, there is the output layer; this is the layer that returns the information the developer is hoping to extract, in the form of an ordered list of values.

The way an artificial neural network learns is by finding the weights such that inputs are correctly mapped to their outputs. The current most reliable method for finding these magic weights is a long and time-consuming process known as backpropagation. Backpropagation performs gradient descent on the neural network using pairs of inputs, their intended outputs, and a loss function. First it initializes the input nodes using an input and performs a forward pass to determine the output currently produced from that input. Using the loss function, the expected output, and the actual output it calculates an error signal for each of the nodes in the output layer. Then it propagates the error signal backwards using a process that is similar to the forward pass albeit in reverse, this is where it gets the name ‘back-propagation’.

***Eq. 2: Error of a Node***

*Error of a node with n outgoing connections where wn is the weight of the nth connection, an is the error of the nth connecting node, and φ’ is the derivative of φ.*

Each of the weights and biases is then adjusted by the product of a small arbitrary constant (the learning rate), the error of the receiving node, and the activation of the transmitting node (for bias this is set to be one).

When constructing a neural network, there are a multitude of factors that influence its ability to learn. Generally speaking, the more complex a problem is the more nodes and connections are required to meet accuracy requirements, but this is not the only thing that influences the maximum performance. Increasing the number of connections requires more iteration over training data to achieve similar results and an increased amount of time to process a prediction. While these are important things to keep in mind, they barely scratch the surface of variables that must be optimized. The sheer variety and volume of options almost ensure that there exists a more functional design for the particular problem in question. To combat this problem a variety of methods have been proposed for the purpose of tailoring a neural network to the specific design requirements.

### Manual Optimization

There are more in-depth strategies that help a developer perform optimization manually. For example, the developer may initialize a large number of networks with varying specifications. Then graph a measure of each of their performances in relation to their hyper parameters (loss function, activation function, node count, epochs, batch size). The developer would then visually identify trends that lead to better networks and iteratively reduce the number of variables until they are left with one that is close to optimum. Unfortunately it still remains tedious to perform such optimization and it remains difficult to optimize structural elements such as connectivity.

**Genetic Algorithms**

The most commonly proposed solution is to implement a type of evolutionary algorithm known as a genetic algorithm. “Evolutionary algorithm” is the umbrella term for any algorithm that employs Charles Darwin’s Theory of Evolution to perform optimization. Genetic algorithms, in particular, operate on strings of bits much like cells operate on strings of DNA and make use of the genetic operations of crossover and mutation to imitate reproduction. Then it assigns a fitness score to each of the ‘individuals’ using a task-specific fitness function to determine their likelihood to produce offspring. Finally, it simulates competition by replacing individuals with lower fitness scores with higher scoring individuals. In the case of neural networks, the fitness function is typically some measure of the network’s accuracy.[19]

However, genetic algorithms rely on a method for converting networks (the phenotype) into fixed length binary strings (the genotype) upon which genetic operations may take place. Since there is currently no function capable of mapping the entire output space of neural networks a fixed length binary string, it means there is no deterministic implementation of a genetic algorithm that is capable of searching the entire solution space and therefore any output of a such a search cannot be said to be the globally optimal network.[11]

# Method

## Materials

All computations were performed on the researcher’s personal computer with an Intel(R) Core™ i5-6600k CPU @ 3.50GHz processor, 16.0 GB RAM, and an NVIDIA GeForce GTX 950 GPU running Windows 10.

Code was written using Visual Studio Code, an open source Integrated Development Environment built by Microsoft. Version control was accomplished with Git through GitHub Desktop. The following extensions were installed into Visual Studio Code:

* Anaconda Extension Pack
* Excel Viewer
* GitLens—Git supercharged
* Python
* Visual Studio IntelliCode
* YAML

The project was completed using the Python programming language and the following libraries encapsulated in an Anaconda virtual environment:

* Numpy[18]
* Numba[15]
* Matplotlib[8]
* NetworkX

To test the network construction algorithm the following publicly available datasets were employed:

*Modified National Institute of Standards and Technology Database (MNIST)[12]*

The MNIST dataset is a bank of sixty thousand 28x28 grayscale images of handwritten numbers that have been normalized and labeled.

*Diagnostic Wisconsin Breast Cancer Database (DWBC)[6]*

The DWBC dataset is a bank of 569 sets of 30 numerical measurements of tumors that are labeled as malignant or benign.

## Assessments and Measures

The performance of the algorithm was measured by the accuracy, connections, and node count of the network it produced. Greater accuracy, fewer connections, and fewer nodes characterize a successful run of the algorithm. The algorithm was run on each of the datasets as many times as possible to ensure results were significant. Each dataset had its own unique properties and pitfalls however the algorithm ran on top of each of the datasets with little to no implementation modifications. The algorithm went through more exhaustive testing using the XOR dataset because of its speed.

## Framing a Solution

Considering the volume of studies pursuing genetic algorithms, I believe that option has been or will be sufficiently explored. What my study will focus on is the broader realm of evolutionary algorithms. Since there exists no function to map the infinite space of potential networks to a fixed length binary string, why try? Why not apply evolutionary operations directly to the neural networks themselves. This reduces the overhead of converting between binary strings and trainable neural networks. It also exposes the entire space of possible networks to the search space limited only by hardware capabilities. To make an evolutionary algorithm capable of optimizing a population of networks it is necessary that the implicit conditions for an evolutionary algorithm are met:

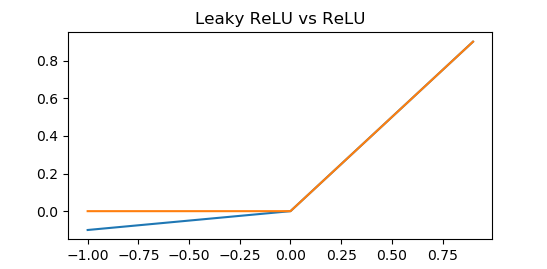
* The child of two networks must resemble its parents
* There must be some fitness function that takes in a network and determines its suitability for the task and therefore its potential to produce offspring similar to itself.

The fitness function is task specific and should be defined by the developer employing this software. They would still be required to numerically describe the desirability of a network.

Although it is possible to have the evolutionary algorithm optimize node count as well, this study opted for a simple more controlled approach. A single number is selected from between the maximum developer defined node cap and the dataset defined node floor (input size + output size) and is then used to initialize a population with that node count. Depending on how well that population scored the range was adjusted.

**Creating a Neural Network**

In order to facilitate my neural network optimization algorithm I needed to create a bare bones implementation of a neural network. For the sake of simplicity my neural network doesn’t implement loss functions, various activation functions, batch training, dropout, momentum, or multi-threading. Its activation function is currently fixed on a variety of the rectified linear unit (ReLU) known as the ‘leaky ReLU’ which was introduced to solve certain issues associated with the ReLU. The loss function is simply raw error by output node. This is not a study of optimization of hyper parameters but may be extended into this field in the future. The performance of this network in a fully connected unoptimized setting is slightly lower than standard.



*Fig.2: A graph of the ReLU (orange) vs. the ‘leaky’ ReLU (blue). A simple piecewise function that mimics the thresholded activation of a biological neuron.*

However, it does have some unique features that made constructing my algorithm easier. It forces all connections to be feedforward while also allowing connections to exist between any two nodes. This allows networks to be not strictly layered in order and thereby significantly expands the search space. The implementation also forces uniqueness of connections without requiring any expensive computations so the same two nodes don’t get connected multiple times. On top of that, it is easier to debug, add features to, and understand because I wrote it myself.The downside is that the network itself doesn’t benefit from many of the newer more complex features available on other neural network providers. One such limitation would be implementing effective hardware acceleration techniques such as multithreading and GPU computing become much more difficult. With fully connected strictly layered networks the passing of one layer’s activation to the next can be done in parallel as nodes within the same layer do not influence each other. The only certainty in my framework is that nodes with a higher index may not influence nodes with a lower index. There most likely remains an elegant way to parallelize this structure but that is not the intent of this study. (See Appendix A)

**Creating an Evolutionary Algorithm**

With my adapted neural network design, implementing the evolutionary algorithm was rather straightforward. The only part that would require extra consideration would be the recombination step. The recombination step is the most important part of the evolutionary algorithm as it preserves the continuity of the output space of neural networks and allows the fitness function to be optimized smoothly and quickly. If the recombination step does not produce networks similar to then the ‘evolutionary algorithm’ will be no more effective than a random search algorithm. A random search while not ideal may still produce viable results.

The first step of any evolutionary algorithm is to spawn an initial population upon which further operations will take place. A population may be spawned on the basis of 4 parameters, the number of individuals in the population, the dimension of the input vectors, the dimension of the output vectors, and the number of nodes to use. During spawning each network is initialized with *n* nodes that are completely unconnected. Connections are added in two steps. The first step ensures that there exists a path between every output node and at least one input node. The path may take a minimum of 2 and a maximum of 10 nodes to connect an output to an input. In the second step *n2 ÷ 2* random connections are proposed to be added and the implementation of connectivity takes care of removing duplicates, loops, and other network artefacts.

The next step of the algorithm is evaluation. Each network is trained over one epoch (full set) of the training data provided by the developer. Then they are scored based on the metrics and fitness function also supplied by the developer. The supplied metrics are sum square error over 1 epoch, validation accuracy, and the networks properties (ie. node count, connections, etc.). For example, a user searching for the most accurate network architecture would simply set the fitness function as the validation accuracy of the network whereas someone searching for the fastest learner would rely on loss. Most developers will create some combination thereof.

The next steps are the evolutionary operations of mutation, selection, and recombination. In this study mutation was performed before recombination in each loop. For mutation, a developer defined number of mutations *m* are randomly applied to *m* networks randomly selected from the population with replacement. Each mutation has the chance to add a random connection or adjust the learning rate by a small amount.

Then there is selection and recombination, to hopefully produce offspring similar to the highest performing individuals. Networks are selected on a rank basis meaning the exact fitness value is not particularly important. Each network is given a probability of being selected as a parent, then *2p* parents are chosen where *p* is the number offspring. A single individual may be selected to parent multiple offspring however adjacent parents are selected without replacement so a network may not reproduce with itself.

The recombination scheme for this study was relatively simple involving only two independent variables. First, the union of the parents sets is constructed, then a random number of connections are dropped. This number may be anywhere from all to none. No safeties are employed during this step. Then the average of the parent’s learning rates is calculated and the new network is constructed using the set of connections and learning rate mentioned above. Finally, each of the connections inherits the waits from the parent it came from with parent 0 taking precedence in the case of a conflict.

Any individuals that underwent a modification and also all the new individuals are evaluated according to the same standard as the original population. Individuals with lower fitness scores are replaced by those with higher fitness scores to prevent the population from growing and to accelerate the convergence.

An important step I overlooked initially was to save a deep copy (a copy in which the target of each reference is copied as opposed to the reference) of the highest scoring network so that it may be returned intact even if it undergoes mutation.

## Node Count Search

The search algorithm for the node count was a simple random search in which the range is limited according to a set of known rules about the search space. Initially a random integer is selected on the range between the minimum node count and a developer defined maximum node count. If a population produced a suitable result on node count all node counts greater than that value are removed from the range. If a population fails to produce a suitable network at that node count, the lower end of the range contracts. To prevent a series of unfortunate events from discluding the optimum by this process if the population at the lowest node count in the range produces a suitable network then the range is extended in that direction. Because the algorithm’s intent is to find the smallest possible network this will reduce the likelihood of premature convergence.

# Results

## XOR

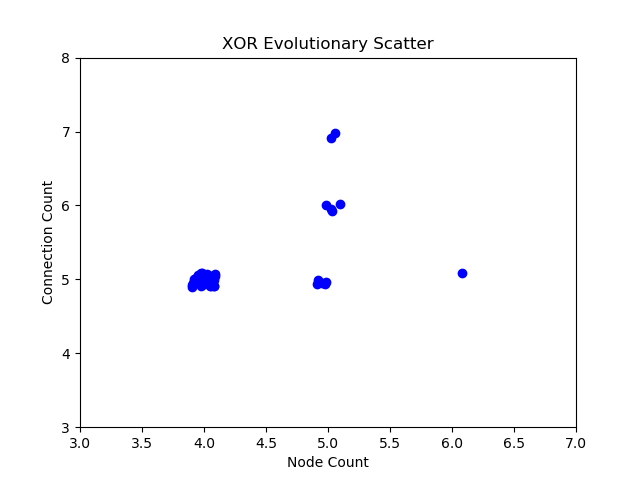
Because the XOR function is so simple yet its output is not linearly separable (able to be separated into only 0s and only 1s by a single line) it represents the perfect test problem for artificial neural networks as such a function, unlike AND or OR would require a hidden layer in order to complete.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | Inputs |
| 0 | 0 | 1 |  |
| 1 | 1 | 0 | Outputs |

*Fig.3: A table of the XOR function. The XOR boolean function is true when one and only one of its inputs is true. Any other case is false.*

Using the XOR function I generated a training set of 60,000 examples. First I ran a single simulation, in which the goal was to construct the smallest possible network capable of achieving 100% accuracy on the dataset generated from the XOR function. To my amazement the network returned a 4-node 5-connection network that was able to solve the XOR function. The reason this surprised me is because with a strictly layered network this configuration would be impossible. Even though there would be a hidden layer, because it contains only one node the information from the input layer would be compressed before it is able to be separated. The not strictly layered architecture gets around this by directly connecting the input nodes to both the output layer and hidden layer.

In order to more adequately judge the capabilities of the constructed search algorithm, I ran 50 iterations of it. Thirty-eight (76%) of the outputs were identical to each other and the theoretical optimum for my neural network implementation, 4-nodes 5-connections. Of the remaining twelve (24%), six (12%) were homeomorphic to (identical topologically but not element by element) the theoretical optimum and six did not converge completely

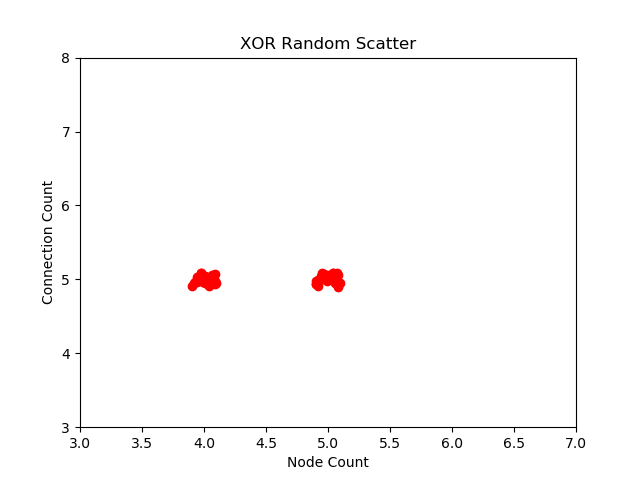


*Fig.4: Plot of each of the 50 results from running the evolutionary search.*

*[slight offsets were added for visualization]*

To further explore the effectiveness of the evolutionary algorithm, specifically the recombination step, I ran another 50 iterations but replaced the recombination step with a spawn population step. What this does is reveal whether the recombination scheme is producing offspring with similar fitness as the parents or if it is just producing random viable networks. It is worth noting that the search space for the optimal XOR network may be explored in its entirety which is simply not feasible for something such as the MNIST dataset or the DWBC dataset.

The random search produced twenty-seven (54%) networks identical to each other and the theoretical optimum; of the remaining twenty-three, eighteen (36%) were homeomorphic to the theoretical optimum and the remaining (10%) had converged to a local optimum requiring five nodes.

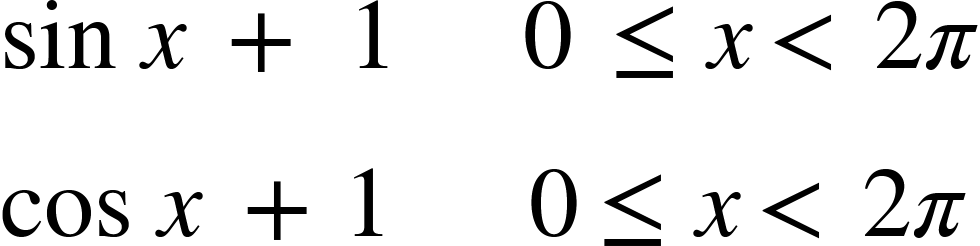


*Fig.5: Plot of each of the 50 results from running the random search.*

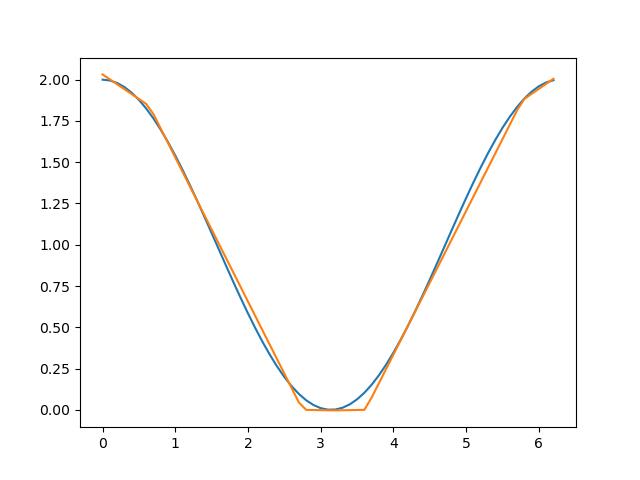
*[slight offsets were added for visualization]*

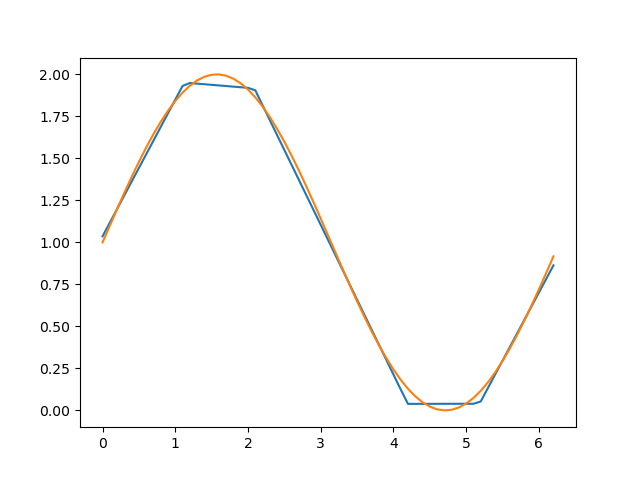
## SINE & COS

The algorithm was tested on datasets of fifty thousand generated ordered pairs using the trigonometric sine and cosine functions with a slight modification to keep their range inside the range of ReLU.



This dataset was used in order to assess the algorithm’s ability to create not only classification networks but also networks that produce real valued outputs. If the squared error between the actual output and expected output fell below 1.0 x 10-2 then the network was said to be accurate so the approximation isn’t perfect.

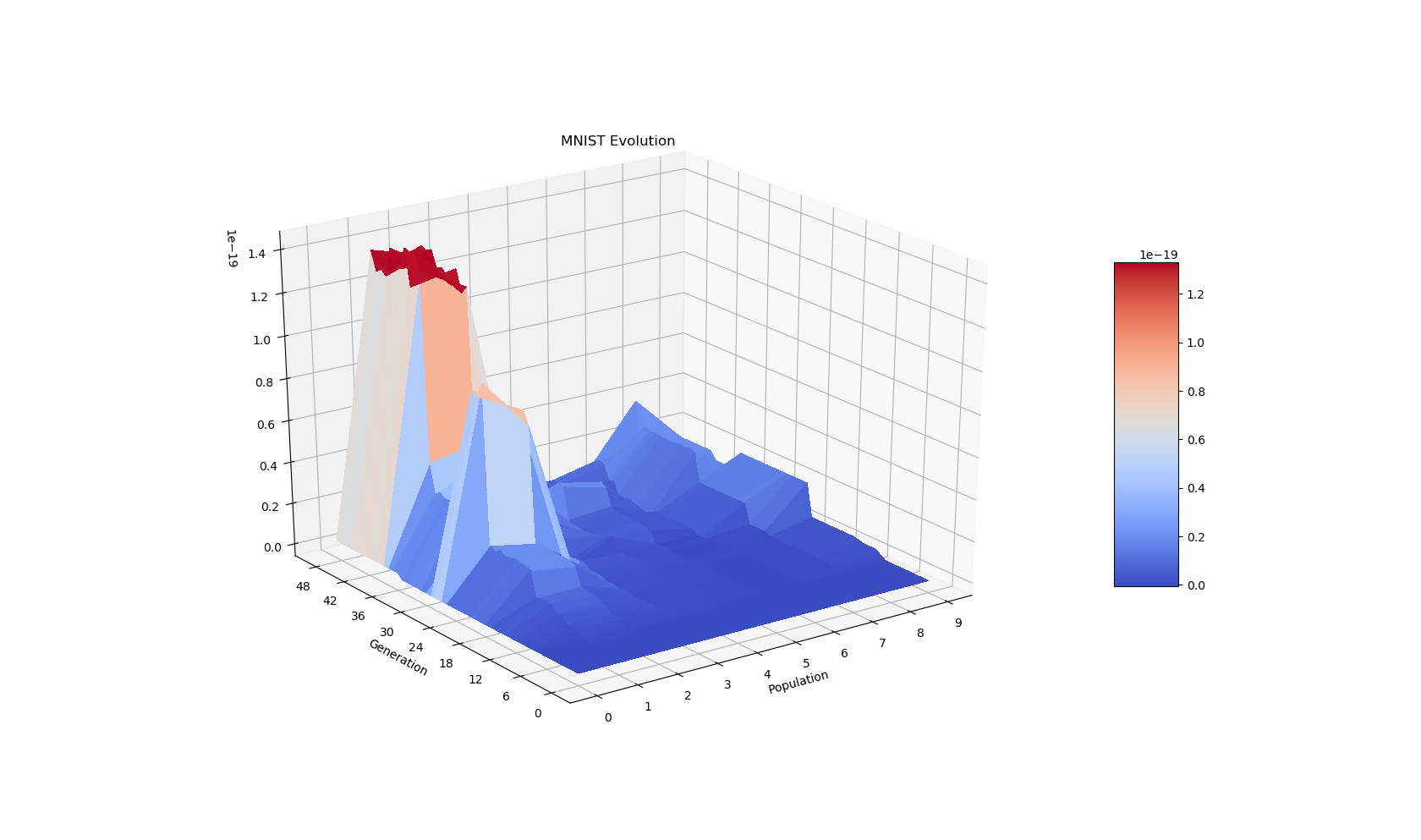


*Fig.6: A graph of the neural network’s approximation of a cosine wave from 0 to 2π (orange) overlayed on top of a true cosine wave (blue).*

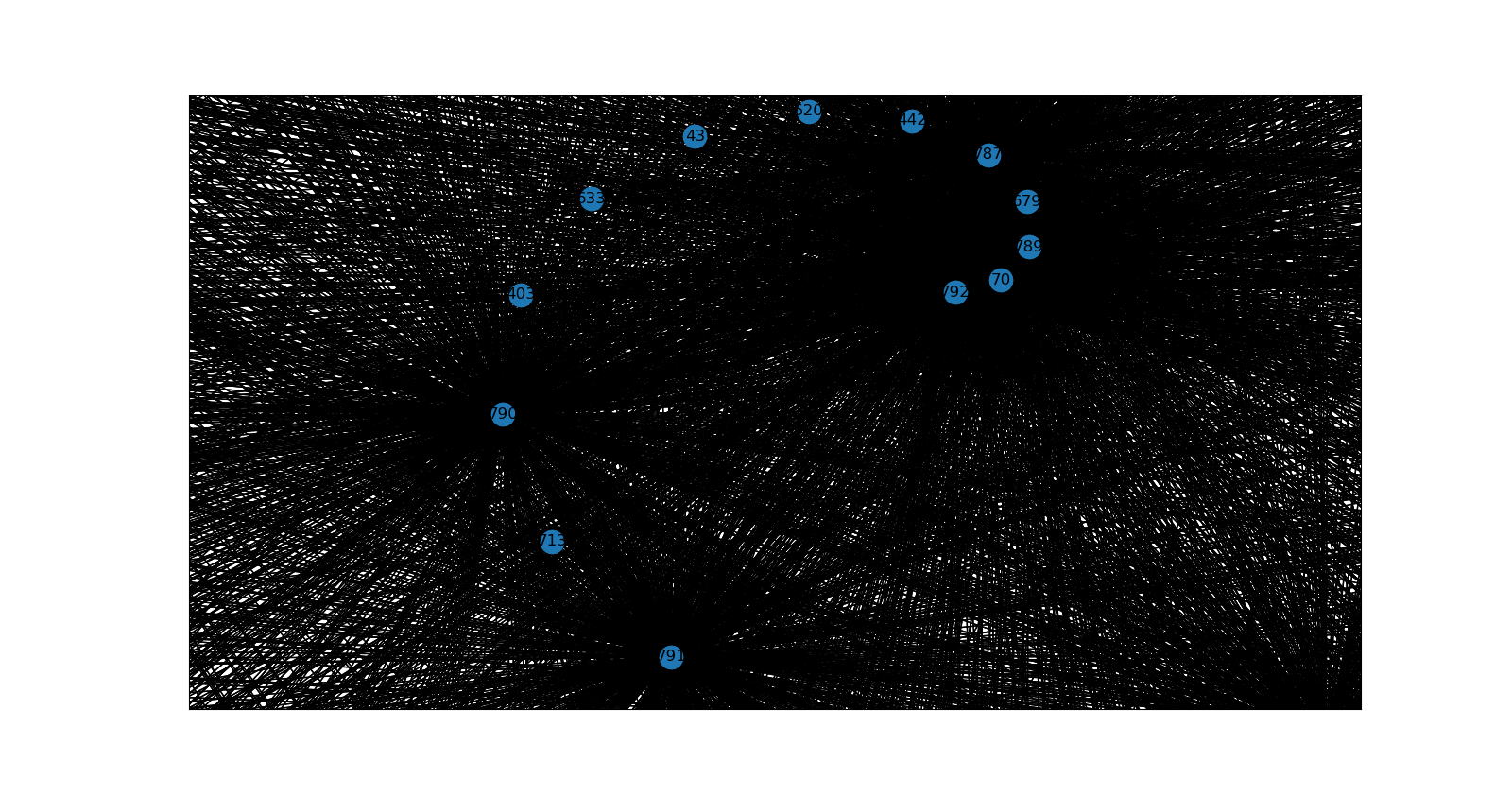
*Fig.6: A graph of the neural network’s approximation of a sine wave from 0 to 2π (blue) overlayed on top of a true sine wave (orange).*

## MNIST

The MNIST dataset is another classic problem for testing neural networks. It is both remarkably simple yet nuanced and complex and is capable of fully describing the problem of image recognition. Here is where the tunability of the evolutionary algorithm was tested. Unfortunately, due to resource limitations only one trial was run. In this trial the algorithm was instructed to find the smallest possible network capable of at least 90% accuracy. The trial was successful and within 10 populations of 50 generations the algorithm converged on a network of only 794 nodes (0 hidden nodes) and 5,367 connections (not fully connected) that was capable of just over 90% accuracy (90.54%).



*Fig.6: Graph of the max fitness vs number of generations vs the population number. Accuracy was deliberately sacrificed to reduce size. Red areas are more accurate.*

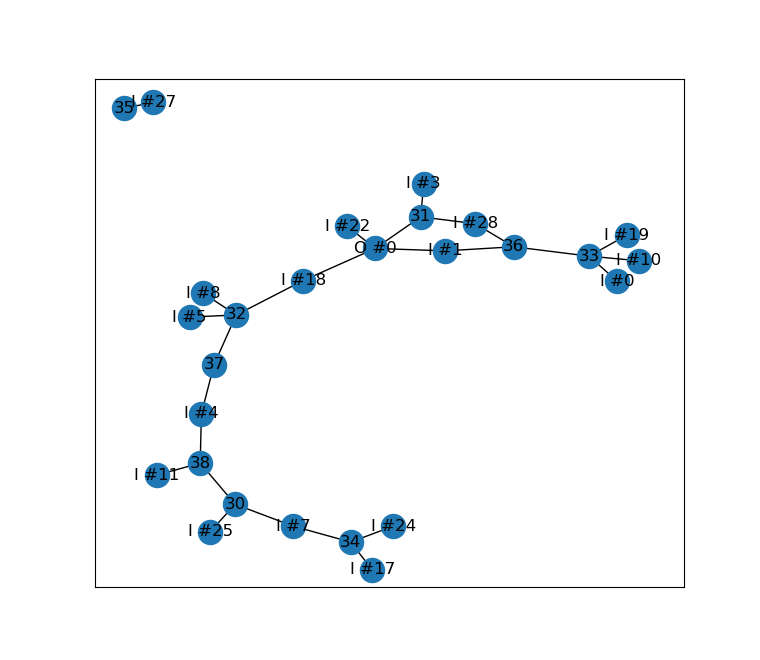


*Fig.7: Zoomed in render of the MNIST network where a black line represents a single connection and a blue dot represents a node. Nodes #790 and #791 are visibly more connected and are also output nodes.*

## DWBC

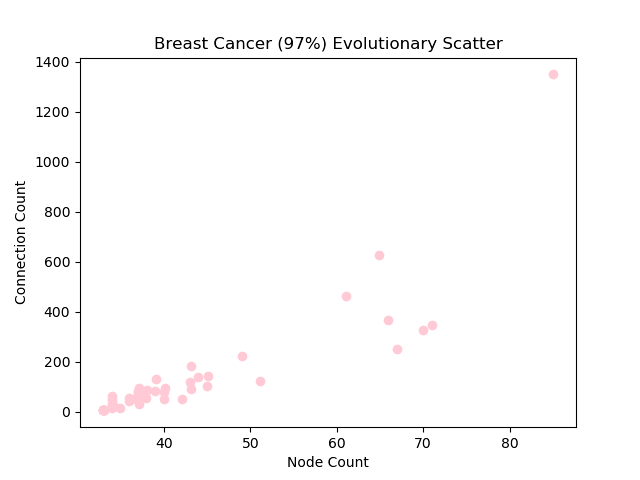
This dataset was introduced for the purpose of measuring the algorithm’s performance when faced with a practical and applicable problem and real experimental data. In each of the other sets data was very reliable and plentiful, this dataset however only supplied 569 patterns, 100 of which were set aside for validation. Also the relationship between the 30 numerical inputs and the malignance of the tumor is significantly more obscure than the relationship between the binary inputs and outputs of the XOR.

Despite these concerns the results were remarkable. When asked for the smallest possible network with 97% accuracy on the 100 tem validation set the algorithm returned a network of 40 nodes (9 hidden nodes) and 26 connections

**

*Fig.8: Illustration of 40 node cancer network. Notice the existence of many redundant nodes and connections. The search algorithm can be improved to remove these as well.*

with an accuracy of exactly 97% percent. Moreover, the algorithm returned much faster than anticipated which revealed I had the resources to compute many more simulations than planned. I ran 50 simulations of the optimizer with population size of 15, a max population count of 15, 50 generations per population, and a target accuracy of 97%. Every single iteration produced a network with upwards of 97% accuracy on the validation set and there was only one outlier. On top of that, most converged under 55 nodes with below 200 connections.



*Fig.9: Scatter plot of node count vs connections for 50 iterations of optimization*

*[slight offsets were added for visualization]*

After witnessing the results of this trial I was curious if I could push the accuracy even higher while maintaining such high rates of convergence. So I ran another 50 iterations with the same parameters except the target accuracy I bumped up the target accuracy to 99%. While still amazing the spread of convergence increased significantly as the algorithm fought two conflicting directives:

* become smaller
* increase accuracy

# Discussion

The results of this study indicate a great level of success in finding networks capable of accomplishing the tasks. However, there still exist a multitude of operations that must be undertaken to improve and finalize the software.

# 

References

[1] Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, *5*(1). https://doi.org/10.1109/72.265960

[2] Benardos, P., & Vosniakos, G.-C. (2007). Optimizing feedforward artificial neural network architecture. *Engineering Applications of Artificial Intelligence*. https://doi.org/10.1016/j.engappai.2006.06.005

[3] Branke, J. (n.d.). Evolutionary algorithms in neural network design and training- A review. *Research Gate*. Retrieved from https://www.researchgate.net/publication/246121926\_Evolutionary\_algorithms\_in\_neural\_network\_design\_and\_training-A\_review

[4] Brownlee, J. (2016, November 7). How to code a neural network with backpropagation in Python (from scratch) [Blog post]. Retrieved from https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

[5] Deb, K., Sindhya, K., & Okabe, T. (n.d.). Self-adaptive simulated binary crossover for real-parameter optimization. *DBLP Computer Science Bibliography*. https://doi.org/10.1145/1276958.1277190

[6] Dua, D., & Graff, C. (2017). {UCI} machine learning repository. Retrieved March 6, 2020, from UCI Machine Learning Repository website: https://archive.ics.uci.edu/ml/

[7] Gentile, C., & Sznaier, M. (2001). An improved voronoi-diagram-based neural net for pattern classification. *IEEE Transactions on Neural Networks*, *12*(5). https://doi.org/10.1109/72.950151

[8] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90-95. https://10.1109/MCSE.2007.55

[9] Karlik, B., & Olgac, A. V. (n.d.). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, *1*(4), 111-122. Retrieved from http://www.cscjournals.org/manuscript/Journals/IJAE/Volume1/Issue4/IJAE-26.pdf

[10] Klöckner, A. (n.d.). PyOpenCL [Computer software]. Retrieved from https://mathema.tician.de/software/pyopencl/

[11] Koehn, P. (1994). *Combining genetic algorithms and neural networks: The encoding problem* (Master's thesis, The University of Tennessee, Knoxville, TN). Retrieved from http://homepages.inf.ed.ac.uk/pkoehn/publications/gann94.pdf

[12] LeCun, Y., Cortes, C., & Burges, C. J.C. (n.d.). The MNIST database of handwritten digits. Retrieved March 6, 2020, from MNIST handwritten digit database: http://yann.lecun.com/exdb/mnist/

[13] Love, B. (2002). Comparing supervised and unsupervised category learning. *Psychonomic Bulletin & Review*, *9*(4), 829-835. https://doi.org/10.4758/BF03196342

[14] Mikko. (2018, May 15). Hyperparameter optimization with Keras. Retrieved February 6, 2020, from Towards Data Science website: https://towardsdatascience.com/hyperparameter-optimization-with-keras-b82e6364ca53

[15] Numba [Computer software]. (2019).

[16] Raptis, S. N., Tzafestas, S. G., & Karagianni, H. (n.d.). Optimal genetic representation of complete strictly-layered feedforward neural networks. *Research Gate*. https://doi.org/10.1007/3-540-45723-2\_15

[17] UFLDL tutorial. (n.d.). Retrieved December 14, 2019, from Stanford University website: http://ufldl.stanford.edu/tutorial/

[18] Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy Array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2). https://doi.org/10.1109/MCSE.2011.37

[19] Whitley, D., Starkweather, T., & Bogart, C. (1990). Genetic algorithms and neural networks: optimizing connections and connectivity. *Science Direct*, *14*(3). https://doi.org/10.1016/0167-8191(90)90086-O

# 

# Appendix A:

**import numpy as np**

**import numba**

**from numba import jitclass, njit**

**from numba import int32, int64, float32, float64, uint8**

**from numba import prange**

**from ast import literal\_eval**

***# Activation Functions and Derivatives***

**a = 0.01**

**@njit**

**def relu(x):**

**return x if x > 0.0 else 0.0**

**@njit**

**def leaky\_relu(x):**

**return x if x > 0.0 else x \* 0.01**

**@njit**

**def sigmoid(x):**

**return 1/(1+math.exp(-x))**

**@njit**

**def d\_relu(x):**

**return 1.0 if x > 0.0 else 0.0**

**@njit**

**def d\_leaky\_relu(x):**

**return 1 if x > 0.0 else 0.01**

**@njit**

**def d\_sigmoid(x):**

**y = sigmoid(x)**

**return y/(1-y)**

**network\_spec = [**

**('input\_shape', int32),**

**('output\_shape', int32),**

**('node\_count', int32),**

**('nodes', float64[:,:]),**

**('to', int32[:,:]),**

**('frm', int32[:,:]),**

**('deps', int32[:,:]),**

**('weights', float64[:,:]),**

**('connections', numba.typeof({(int32(0), int32(0))})),**

**('learning\_rate', float32),**

**('id', int32)**

**]**

**@jitclass(network\_spec)**

**class JIT\_Network:**

**def \_\_init\_\_(self, input\_shape: int, output\_shape: int, node\_count=0, learning\_rate=0.01, id\_num=-1):**

**np.random.seed(120)**

**self.input\_shape = input\_shape**

**self.output\_shape = output\_shape**

**self.node\_count = input\_shape + output\_shape if node\_count<input\_shape+output\_shape else node\_count**

**self.nodes = np.empty(shape=(self.node\_count, 3), dtype=float64)**

**self.nodes.fill(-1.0)**

**self.to = np.empty(shape=(self.node\_count, self.node\_count), dtype=int32)**

**self.frm = np.empty(shape=(self.node\_count, self.node\_count), dtype=int32)**

**for i in range(self.node\_count):**

**self.to[i, -1] = 0**

**self.frm[i, -1] = 0**

**self.deps = np.column\_stack((np.zeros(shape=self.node\_count, dtype=int32), np.full(shape=self.node\_count, fill\_value=self.node\_count, dtype=int32)))**

**self.weights = np.random.random((self.node\_count, self.node\_count)).astype(float64)/self.node\_count**

**self.connections = {(int32(0), int32(0))}**

**self.learning\_rate = learning\_rate**

**self.id = id\_num**

**def get\_activation(self, node\_i: int32) -> float64:**

**p\_activation = 0.0**

***# Sum the weighted activations of previous nodes***

**for from\_node\_i in self.frm[node\_i][:self.frm[node\_i, self.node\_count-1]]:**

**p\_activation += self.weights[node\_i, from\_node\_i] \* self.nodes[from\_node\_i, 0]**

***# Apply bias then activation function***

**self.nodes[node\_i, 0] = leaky\_relu(p\_activation + self.nodes[node\_i, 2])**

**return self.nodes[node\_i, 0]**

**def get\_backward\_delta(self, node\_i: int32) -> float64:**

**p\_error = 0.0**

***# Sum the weighted errors of previous nodes***

**for to\_node\_i in self.to[node\_i][:self.to[node\_i, self.node\_count-1]]:**

**p\_error += self.weights[to\_node\_i, node\_i] \* self.nodes[to\_node\_i, 1]**

***# Multiply error by derivative of the output***

**self.nodes[node\_i, 1] = p\_error \* d\_leaky\_relu(self.nodes[node\_i, 0])**

**return self.nodes[node\_i, 1]**

**def forward\_propagate(self, input\_layer):**

***# Initialize the input layer***

**for node\_i in range(self.input\_shape):**

**self.nodes[node\_i, 0] = input\_layer[node\_i]**

***# Calculate the activation of each subsequent node***

**for node\_i in range(self.input\_shape, self.node\_count):**

**self.get\_activation(node\_i)**

***# Return a view of the output\_layer***

**return self.nodes[-self.output\_shape:, 0]**

**def backward\_propagate\_error(self, output\_deltas):**

***# Initialize the output layer***

**for node\_i in range(self.output\_shape):**

**self.nodes[self.node\_count-self.output\_shape+node\_i, 1] = output\_deltas[node\_i]**

***# Calculate the error of each node***

**for node\_i in range(self.node\_count-self.output\_shape-1, -1, -1):**

**self.get\_backward\_delta(node\_i)**

**def update\_weights(self):**

**for connection in self.connections:**

***# Adjust the weight by a small step proportional to the error of the receiving node and the activation of the previous node***

**self.weights[connection] += self.learning\_rate \* self.nodes[connection[0], 1] \* self.nodes[connection[1], 0]**

**for node\_i in range(self.node\_count):**

***# Adjust the biases by a small step proportional to the error of the node***

**self.nodes[node\_i, 2] += self.learning\_rate \* self.nodes[node\_i, 1]**

**def predict(self, inputs):**

***# Detect init connection (removed before training)***

**if (0, 0) in self.connections:**

***# Warn that the network has not been trained***

**print("Network hasn't been trained")**

***# Calculate the output layer associated with each input and store it***

**outputs = np.zeros((len(inputs), self.output\_shape), dtype=float64)**

**for i in prange(len(inputs)):**

**outputs[i] = self.forward\_propagate(inputs[i])**

***# Return the array of outputs***

**return outputs**

**def validate(self, val\_x, val\_y, compare):**

***# Get the network's prediction for each val\_x***

**outputs = self.predict(val\_x)**

**correct = 0**

**for i in range(len(val\_x)):**

***# Use the compare function to determine whether an output is acceptable***

**if compare(outputs[i], val\_y[i]):**

**correct += 1**

***# Divide total correct by total inputs to calculate % accuracy***

**return correct/len(val\_x)**

**def train(self, x, y, n\_epoch=1, batch\_size=1000, target\_error=0.01):**

***# Limit the batch size to prevent index overflow***

**if batch\_size > len(x):**

**batch\_size = len(x)**

***# Remove init connection***

**if (0, 0) in self.connections:**

**self.remove\_connection((0, 0))**

***# Initialize the biases to random numbers***

**self.nodes[:, 2] = np.random.random(self.node\_count)**

**acc\_error = 0.0**

**set\_count = 0**

**break\_out = False**

***# Iterate over the training set n\_epoch times***

**for epoch in range(n\_epoch):**

***# sum\_error = sum error per epoch***

**sum\_error = 0.0**

***# Error batch (NOT a learning batch)***

**for b in range(0, len(x), batch\_size):**

***# error = sum squared error***

**error = 0.0**

**for index in range(batch\_size):**

**i = b + index**

**output\_layer = self.forward\_propagate(x[i])**

***# Loss function is just 1D signed distance***

**self.backward\_propagate\_error(np.array([(y[i, j] - output\_layer[j]) \* d\_leaky\_relu(output\_layer[j]) for j in range(self.output\_shape)], dtype=float64))**

**self.update\_weights()**

**for j in range(self.output\_shape):**

**error += (y[i, j] - output\_layer[j])\*\*2**

**sum\_error += error**

**set\_count += batch\_size**

***# If there is a number exception or sum squared error is below target, quit loop***

**if np.isnan(error) or error/batch\_size <= target\_error:**

**break\_out = True**

**break**

***# Accumulate error over epochs***

**acc\_error += sum\_error**

**if break\_out:**

**break\_out**

**return acc\_error**

***# Convenience method for setting a single weight given a connection***

**def set\_weight(self, connection, weight):**

**self.weights[connection] = weight**

***# Convenience method for getting a single weight given a connection***

**def get\_weight(self, connection):**

**return self.weights[connection]**

***# Method for setting up connections in a form that is efficient to access during propagation***

***# Returns whether the connection is valid and initialized***

**def setup\_deps(self, to, frm):**

***# If a connection is unsuitable return with code 0***

**if to <= frm or to < self.input\_shape or frm > self.node\_count-self.output\_shape:**

**return 0**

***# Add index of the 'to' node to the 'frm' node's list of recipients***

**self.to[frm, self.to[frm, self.node\_count-1]] = to**

**self.to[frm, self.node\_count-1] += 1**

**self.deps[to, 0] = max((self.deps[to, 0], frm)) *#DEPRECATED LAYER CONSTRUCTION***

***# Add index of the 'frm' node to the 'to' node's list of senders***

**self.frm[to, self.frm[to, self.node\_count-1]] = frm**

**self.frm[to, self.node\_count-1] += 1**

**self.deps[frm, 1] = min((to, self.deps[frm, 1])) *#DEPRECATED LAYER CONSTRUCTION***

***# iF a connection is successfully added return with code 1***

**return 1**

***# Add one unique feed forward connection to 'to' from 'from' where 'to' is not an input node and from is not an output node***

**def add\_connection(self, to, frm):**

***# If nodes are not suitable for a connection, do not connect***

**if not (to, frm) in self.connections and self.setup\_deps(to, frm):**

***# Add the connection to the set of connections***

**self.connections.add((int32(to), int32(frm)))**

***# Convenience method for adding large numbers of connections***

**def add\_connections(self, tos, frms):**

***# Reformat the two arrays as a set of tuples***

**new\_connections = set(map(lambda x, y: (x, y), tos, frms))**

***# Filter non-unique connections and add the two sets together***

**new\_connections.difference\_update(self.connections)**

**self.connections.update(new\_connections)**

***# Setup connection matrices***

**for connection in new\_connections:**

***# If a connection is not suitable, remove it***

**if not self.setup\_deps(\*connection):**

**self.connections.remove(connection)**

**continue**

***# Convenience method for overwriting connections with a new set***

**def set\_connections(self, new\_connections):**

***# Clear connections matrices***

**self.to[:,-1].fill(0)**

**self.frm[:,-1].fill(0)**

**self.connections = new\_connections**

***# Setup connection matrices***

**for connection in new\_connections:**

***# If a connection is not suitable, remove it***

**if not self.setup\_deps(\*connection):**

**self.connections.remove(connection)**

**continue**

***# Convenience method for removing a connection from both set and matrices***

**def remove\_connection(self, connection):**

**if connection in self.connections:**

**self.connections.remove(connection)**

**self.set\_connections(self.connections)**

***# Convenience method for removing a random connection from both set and matrices***

**def remove\_random\_connection(self):**

**if len(self.connections > 1):**

**self.remove\_connection(self.connections.pop())**

***# Method for creating DEEP copy of the given network***

**@njit**

**def clone(in\_network):**

**network = JIT\_Network(0, 0, 0, 0.0, -1)**

**network.input\_shape = in\_network.input\_shape**

**network.output\_shape = in\_network.output\_shape**

**network.node\_count = in\_network.node\_count**

**network.nodes = in\_network.nodes.copy()**

**network.to = in\_network.to.copy()**

**network.frm = in\_network.frm.copy()**

**network.deps = in\_network.deps.copy()**

**network.weights = in\_network.weights.copy()**

**network.connections = in\_network.connections.copy()**

**network.learning\_rate = in\_network.learning\_rate**

**network.id = in\_network.id**

**return network**

# Appendix B:

import utils

import numba

import datetime

import itertools

import numpy as np

from numba import njit, prange

from numba.typed import List

from homebrew.network import JIT\_Network, clone

*# Called whenever a new population is spawned*

@njit

def spawn\_population(population\_size, input\_shape, output\_shape, node\_count):

*# Creates a list in which to store members of the population*

population = List()

*# Loops {population size} number of times*

for j in range(population\_size):

*# Creates a new network with the specified input and output shapes and learning rate scaled based on node count*

new\_net = JIT\_Network(input\_shape, output\_shape, node\_count, np.random.random()/node\_count, j)

node\_count = new\_net.node\_count

*# Set the node count to display the actual node count because the network might make modifications to it*

*# For each output node*

for node in range(1, output\_shape+1):

to\_node = node\_count-node

*# Create connections util it is connected to an input node*

while input\_shape <= to\_node:

*# Find a random node infront of the current receiving node*

from\_node = to\_node - np.random.randint(low=node\_count/10, high=node\_count/2+1)

*# If it's out of bounds set it to a random input node*

if from\_node < 0:

from\_node = np.random.randint(low=0, high=input\_shape)

*# Add the connection*

new\_net.add\_connection(to\_node, from\_node)

*# Link the chain*

to\_node = from\_node

*# Randomly add more connnections on top of the constructed web*

connections = np.random.randint(0, high=node\_count, size=(2, (node\_count\*\*2)/2)).astype(numba.int32)

new\_net.add\_connections(connections[0], connections[1])

*# Place this network in the population*

population.append(new\_net)

return population

*# Called whenever a network needs to be evaluated or reevaluated*

@njit

def evaluation(x, y, val\_x, val\_y, compare, population):

*# Allocate an array for storing respective fitnesses*

fitnesses = np.empty(shape=len(population), dtype=numba.float64)

*# For each network in the population get a reference to the network and its index*

for j, network in enumerate(population):

*# Train the network*

traits = network.train(x, y, 1, 1000, 0.001)

*# Calculate and save its fitness*

if np.isnan(traits) or len(network.connections) == 0:

fitnesses[j] = -1

else:

val = network.validate(val\_x, val\_y, compare)

fitnesses[j] = (val\*\*int(network.node\_count/2+10))/len(network.connections)

*# Return the respective fitnesses*

return fitnesses

*# Called every generation to select parents*

@njit

def selection(num\_pairs, fitnesses, population):

population\_size = len(population)

*# Create a list of network indices where higher fitnesses are at the tail end of the list*

rank\_list = fitnesses.argsort()

*# Because Numba doesn't support irregluar distributions and selection without replacement*

*# I had to implement it myself*

choices = np.empty(0, dtype=np.int32)

*# For each index in the rank list*

for n, index in enumerate(rank\_list):

*# Populate the choice list with n (the index of index) instances of index*

choices = np.append(choices, np.full(n+1, index, dtype=np.int32))

*# Allocate an array to store the pairs of parents*

pairs = np.empty((num\_pairs, 2), dtype=np.int32)

*# For each pair*

for p in range(num\_pairs):

*# Randomly choose one parent*

pairs[p, 0] = np.random.choice(choices) *# This parent will tend to have higher fitness*

*# Remove instances of the first parent and andomly choose the second parent*

pairs[p, 1] = np.random.choice(choices[np.invert(choices==pairs[p, 0]), ...])

return pairs

@njit

def recombination(input\_shape, output\_shape, pairs, population):

new\_networks = List()

for i in range(len(pairs)): *# Using each of the pairs generated by the rank selection*

mom = population[pairs[i, 0]] *# Set one parent as the mom*

dad = population[pairs[i, 1]] *# And one as the dad*

*# Merge their connections*

connections = mom.connections | dad.connections

if len(connections) > 0: *# Redundant Safety check*

for j in range(np.random.randint(0, len(connections))):

connections.pop() *# And discard a random number of them*

*# Create a new network with the same shape as the parents*

*# and set the child's learning rate to be the average of the parents*

new\_net = JIT\_Network(input\_shape, output\_shape, mom.node\_count, 0.5\*(mom.learning\_rate+dad.learning\_rate), mom.id+dad.id)

*# Set the connections calculated above*

new\_net.set\_connections(connections)

*# Transfer the weights to the child (Lamarkian evolution)*

for connection in connections:

if connection in mom.connections:

new\_net.set\_weight(connection, mom.get\_weight(connection)) *# Make sure the weights are carried with each connections*

else:

new\_net.set\_weight(connection, dad.get\_weight(connection))

*# Add the child to the list of new networks*

new\_networks.append(new\_net)

return new\_networks

@njit

def mutation(population, mutations):

mutated = List()

*# {~mutations} number of mutations should occur per generation*

for rand in np.random.random(mutations):

*# Randomly choose a network to mutate*

mutated.append(np.random.randint(low=0, high=len(population)))

network = population[mutated[-1]]

*# 30% of the time it will undergo change to its connections*

if rand < 0.30:

network.add\_connection(np.random.randint(network.input\_shape, network.node\_count), np.random.randint(0, network.node\_count-network.output\_shape))

*# Sometime the learning rate will also change*

if rand < np.random.random():

network.learning\_rate += (rand-0.5)/network.node\_count

*# Return a list of which networks were mutated*

return mutated

*# Simulation of competition between individuals in a species*

@njit

def competition(fitnesses, population, new\_fitnesses, new\_networks):

*# Replace least fit network with more fit new\_network*

for i, new\_net in enumerate(new\_networks):

rank\_list = fitnesses.argsort()

if new\_fitnesses[i] > fitnesses[rank\_list[0]]:

fitnesses[rank\_list[0]] = new\_fitnesses[i]

population[rank\_list[0]] = new\_net

*# Combine the aforementioned sets*

@njit(parallel=True)

def evolution(x, y, val\_x, val\_y, compare, population\_size, node\_count, generations):

*# First spawn the initial population*

population = spawn\_population(population\_size, x.shape[1], y.shape[1], node\_count)

*# Then evalutate them*

fitnesses = evaluation(x, y, val\_x, y, compare, population)

*# and save the best one*

best\_fitness = fitnesses.max()

best\_network = clone(population[fitnesses.argsort()[-1]])

print(fitnesses.max(), best\_fitness)

*# Repeat this {generations} number of times*

for i in range(generations):

*# Apply mutations and reevaluate mutated networks*

mutated = mutation(population, int(population\_size\*0.075))

mutation\_fitnesses = evaluation(x, y, val\_x, val\_y, compare, [population[mutate] for mutate in mutated])

for j in prange(len(mutated)):

fitnesses[mutated[j]] = mutation\_fitnesses[j]

*# Pair up parents for offspring*

pairs = selection(int(population\_size\*0.15), fitnesses, population)

*# Breed the parent pairs and evaluate offspring*

new\_networks = recombination(x.shape[1], y.shape[1], pairs, population)

new\_fitnesses = evaluation(x, y, val\_x, val\_y, compare, new\_networks)

*# Remove networks with the least fitness*

competition(fitnesses, population, new\_fitnesses, new\_networks)

*# Save the best network*

print("Generation #", i, " Max Fitness:", fitnesses.max())

if fitnesses.max() > best\_fitness:

print("New Best Fitness")

best\_fitness = fitnesses.max()

best\_network = clone(population[fitnesses.argsort()[-1]])

*# Return the saved best network and validate it*

accuracy = best\_network.validate(val\_x, val\_y, compare)

print("Accuracy: ", accuracy\*100,"%")

return best\_network, best\_fitness, accuracy

*# Search funcation for optimizing node count as well+*

@njit

def evolve\_node\_count(x, y, val\_x, val\_y, compare, population\_count, population\_size, node\_cap, generations, target\_accuracy, r):

*# Initialize the psuedorandom number generator to allow for reprodceibility*

np.random.seed(r)

*# Allocate space to save the best network*

best\_fitness = 0

best\_accuracy = 0

best\_network = None

*# Set the node floor to be equal to the number inputs + number of outputs*

node\_floor = x.shape[1]+y.shape[1]+1

*# Repeat {population count} number of times*

for i in range(population\_count):

*# Select a random node count between the floor and cap*

node\_count = np.random.randint(node\_floor, node\_cap)

print("Population:" ,i, " Node Count:", node\_count)

*# Run the evolutionary search which returns a network, fitness, and accuracy*

network, fitness, accuracy = evolution(x, y, val\_x, val\_y, compare, population\_size, node\_count, generations)

*# If it meets minimum requirements*

if accuracy >= target\_accuracy:

*# Check if it is the very best*

if fitness > best\_fitness:

*# and save it*

print("Replaced ", best\_fitness, " with ", fitness)

node\_cap = min(node\_cap, node\_count)

best\_fitness = fitness

best\_network = network

best\_accuracy = accuracy

*# Also check whether the range must be expanded*

if node\_count == node\_floor:

node\_floor -= int(node\_count/4)

else:

*# If it does meet minimum requirements but its better than before*

if not best\_accuracy > target\_accuracy and accuracy > best\_accuracy:

*# Save it*

print("Replaced ", best\_fitness, " with ", fitness)

best\_fitness = fitness

best\_network = network

best\_accuracy = accuracy

*# Adjust the range*

node\_floor += int(0.5 \* (node\_count-node\_floor)) + 1

*# If the range converges early break the loop*

print(node\_floor, "<= node\_count <", node\_cap)

if node\_cap==node\_floor:

break

*# Return the result*

print("Node Count:", best\_network.node\_count)

print(best\_network.input\_shape)

print(best\_network.output\_shape)

print("Weighted Connections:")

for c in best\_network.connections:

print(c,':', best\_network.weights[c])

print("Biases:", list(best\_network.nodes[:,2]))

print("Best Accuracy:", best\_network.validate(val\_x, val\_y, compare)\*100,'%')

return best\_network