Conventional Neuroevolution for Handwritten Digit Classification

Nathan Smith

Student number: 20105878

December 12, 2021

Problem Description

The brain is one of the most complex devices known to us as humans at this point in time allowing us to do all the things we've accomplished today. We've attempted to mimic the power of the brain through the use of neural networks, a simple approximation of how we think, allowing the computers we've built to do the same. In this project, we take a look at the training of such a network to see if a more successful approach to learning can be found, as opposed to the typical implementation of stochastic gradient descent. I decided to take on the task of implementing conventional neuroevolution to the network to see how well it performs overall.

This project is the implementation of an evolutionary algorithm applied to neural networks of fixed topology, in order to train their weight and bias values to categorize an image set of handwritten digits. The fixed topology or 'conventional' design was selected not only due to ease of implementation, but also to allow for a more fair comparison of the neuroevolutionary approach against backpropagation as both networks are of the same structure, and only their weight and bias values differ. The main idea for this project was to compare an evolutionary exploration approach for neural network training to the more popular method of backpropagation and gradient descent.

EA Design

The evolutionary algorithm has a population of neural network objects, each containing their respective weight and bias values. We'll discuss how each main section of the evolutionary algorithm was implemented and the design choices behind why we did it.

Main evolutionary algorithm

Population initialization

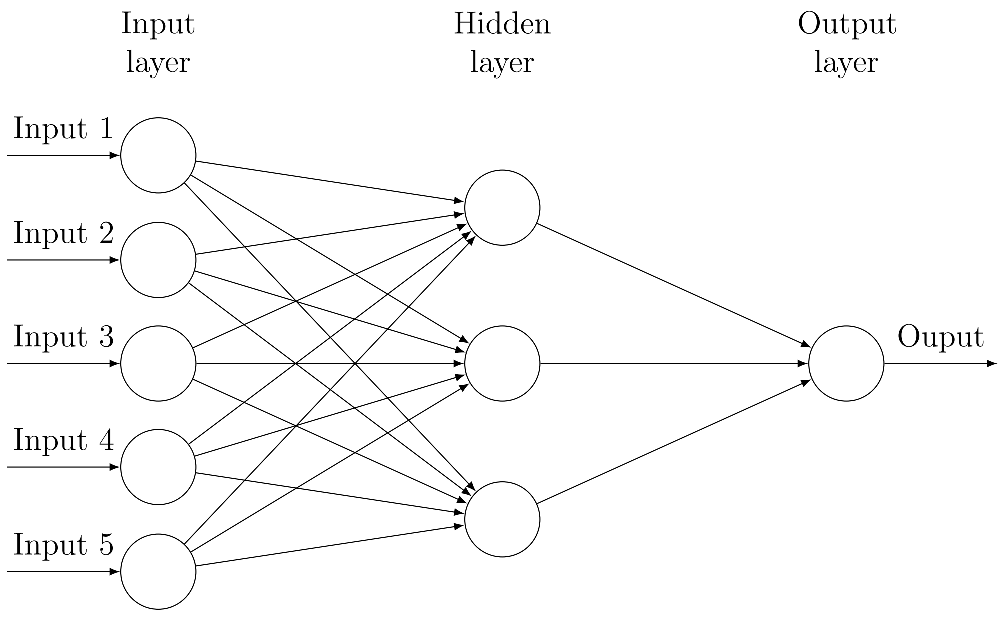
Population initialization is quite simple thanks to the implementation of a neural network object we created. This is simply a matter of appending to the total population list a new neural network object initialized with random values about a mean of 0. The only value we use to initialize the neural network object is a structure tuple, which dictates the structure the neural network will take on - it's important all entities within the population are identical in this respect.

Selection/Replacement

Due to the computational power required to find essentially random values that allow the network to function better, it's important that we maintain genetic diversity as we continue. With this in mind, we use a tournament to select the two parents and use them to produce two offspring, this occurs repeatedly until the entire pool is replaced - except for two. This removes all networks that performed poorly, while essentially doing a search in the direction that seems most likely to bear better networks. The two most fit individuals of the current population make it onto the next generation as well, ensuring there is always an optimal candidate within the population and to ensure we don't devolve.

Crossover

There are two main methods for crossover that were used within this project. The first is where each child starts off as an exact copy of each parent. We then iterate through each weight value of each child and have some chance to swap their values. The other method used is akin to one point crossover, except applied to each layer of neurons separately. We iterate through each layer, and randomly swap entire neurons beyond a specific random point between the children. For example, using the diagram below, this one point crossover would first choose a random value between 0 and 3, and swaps each neuron in the hidden layer between the two children, along with all their weights; it then moves on to the next layer, and does this repeatedly until it reaches the end.



Mutation

This function was applied directly within the neural network object as opposed to within the main evolutionary program because mutation was easier as a property of the network, allowing it to manipulate its own weights. How we go about mutation is similar to crossover; we iterate over every weight and bias value, and with an equally random chance we do one of four mutations: replace whatever value is there with a newly generated one, effectively resetting it; we multiply the weight value by some random value distributed around 1; we add/subtract a small value from the weight, or lastly, we change the sign of the weight. One of these four mutations occur with a random chance on a random amount of weights, dictated by the mutation chance parameter.

This method was implemented due to it being the most ideal for this situation. Originally, only replacement was implemented, however in order to nudge more sensitive networks later on in the training process, resetting weights provided too much of a shift, so manipulations were added to allow nudging to be done, instead of a pure random approach to weight mutation.

Due to the way this method was implemented it's by far the most computationally taxing part of the program, with its computational cost being directly proportional to the mutation rate. A lower mutation means a faster execution time due to fewer weights being changed, and consequently, heavy mutation results in large time increase per loop.

Lastly, we implemented a decay rate to the mutation rate. This allows for more mutation to occur at the start of the program and taper off at larger epochs. This was done because a constant mutation rate made tweaking the network at later epochs much more volatile than at the beginning. To find better weight values for the network, making mutations at the same rate as at the beginning of the program reduces our chance to find an improvement, since such dramatic changes are being made. The decaying mutation allows for less and less exploration of weight values once we get a better network base. The decaying mutation rate also helps to speed up the simulation as time progresses.

Fitness

The implementation of this within the program is combined with evaluation into one function. It was unnecessary to split evaluation and fitness into two separate functions, especially considering fitness relies on the results on a per-image basis - it would be costly to go back through each image to calculate the fitness after evaluation; hence they are done in the same loop.

The fitness used is that of the cost for the backpropagation approach, that being the sum of the squared errors. Using the accuracy of each neural network would be the most obvious choice, however utilizing the cost allows for networks overall closer to the actual values to succeed more. Using cost as opposed to accuracy lends us additionally the ability to more easily compare this neuroevolution approach to backpropagation. Calculating the cost function is simply taking the sum of the squared differences between the network output and the desired output, an example of this is provided below. Where $\vec{i}$ contains the values the network predicts, and $\vec{x}$ is the vector containing the correct digit.

$$C(\vec{i}) = \begin{cases} (i_0 - x_0)^2 + \\ (i_1 - x_1)^2 + \\ (i_2 - x_2)^2 + \\ (i_3 - x_3)^2 + \\ (i_4 - x_4)^2 + \\ (i_5 - x_5)^2 + \\ (i_6 - x_6)^2 + \\ (i_7 - x_7)^2 + \\ (i_8 - x_8)^2 + \\ (i_9 - x_9)^2 \end{cases} \quad \text{Example case} \rightarrow \begin{cases} (0.8 - 0)^2 + \\ (0.2 - 0)^2 + \\ (0.9 - 0)^2 + \\ (0.1 - 0)^2 + \\ (0.1 - 0)^2 + \\ (0.7 - 1)^2 + \\ (0.3 - 0)^2 + \\ (0.2 - 0)^2 + \\ (0.9 - 0)^2 + \\ (0.6 - 0)^2 \end{cases} \quad = 2.9$$

When evaluating the network, we do basic matrix multiplication using the weights with the previous layers values added with the biases. This is then passed through an activation function, in this case ReLU, before repeating the process over again. On the final layer, we pass it through softmax instead of ReLU to get fairly distributed choice amongst the outputs. A simple diagram of this process can be seen below.

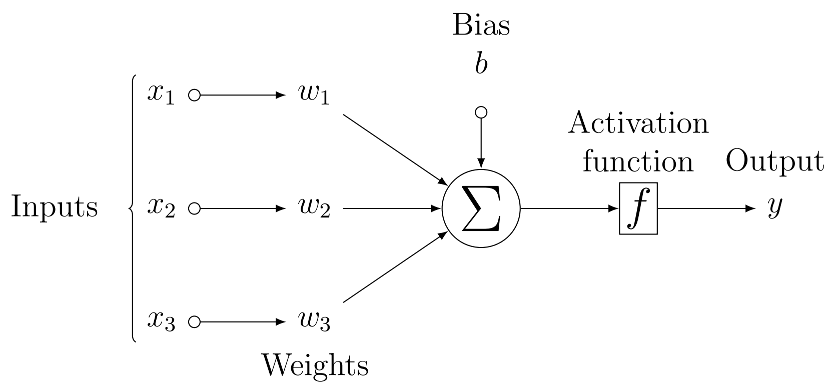


Figure 1: The overall process of what goes on in the neural network

The time cost of evaluating the network is high - with time growing linearly with the amount of images to evaluate. The training set we use has over 50k images, however, only about 100 are used for training mainly due to the limiting factor of Python being too slow to evaluate through that many images. Using more provides little benefit and increases computation times.

Before evaluation can occur, we normalize the data by dividing each pixel value by 255, turning it into a number from 0 to 1. This reduces high pixel values from biasing the network, and allows all data to be treated fairly.

EA Results

Despite being essentially a pure random search, it produced somewhat surprising results. The absolute highest accuracy I was able to achieve within 1000 epochs of training was about 81%. It should be noted all data was used on the same set of data it was trained with, no test sets were used, so the results may not apply well to more diverse settings and overfitting may be present.

These are the accuracy results of one of the first implementations of the algorithm. This used a constant mutation rate, with no crossover or breeding, only mutation. As you can see, the results peak at about 45% accuracy after 1000 epochs. Since mutation is by far the most computational heavy task, this wasn't much faster than later implementations, taking about an hour and a half to train.

Upon implementing crossover we see an increase in performance. The parent selection for this was based on selecting the parents at the start of the epoch and producing the children thereafter. A respectable performance gain from the above trial.

This was when it broke through 70% accuracy. A large milestone. This was thanks to the decaying mutation rate, allowing for more fine-tuning of weight and bias values as the simulation continued.

This was the best performance achieved. After implementing unique parents per child to replace each member of the population based on a tournament; not based on a per-epoch, but a per child method. Including all the above changes in the previous graphs above we produced a neuroevolutionary algorithm that was able to achieve a maximum of 81% accuracy in about an hour and half of training time. For completion's sake, cost is included within the graph as well alongside accuracy - and as you can see, they are essentially identical.

These graphs are related to the above in that they are from the same simulation. The graph on the left shows the time in seconds it took to calculate each epoch. As you can see, the first roughly 100 epochs are by far the most computationally taxing due to the high initial mutation rate. Once the mutation rate decays, the performance stops decreasing - staying consistent at just under 6 seconds per epoch.

For the other graph, it details the amount of genes that are identical between two parents when they mate. It showcases the lack of genetic diversity per epoch. Again, for the first roughly 100 epochs when the mutation rate is high we have high genetic diversity, however when the mutation rate subsides we stop mutating as much and are left with similar individuals within the population. This is by design, as the further we make it into the simulation, the less dramatic changes we wish to make.

Comparison

A comparison between a highly optimized library and a crude implementation of neuroevolution is a bit of an unfair comparison, so to make this comparison slightly more fair, we are going to compare it to a fairly bare-bones raw python implementation of backpropagation taken from [3]. With this we can more accurately compare the two algorithms. All these graphs are from using a similar dataset size to the evolutionary algorithm.

The two graphs are of the same data point, that being the accuracy of the network per epoch. As you can see, it varies, however it is consistently bad, which is a surprising discovery, as the evolutionary algorithm beats in it in terms of accuracy when using the same amount of data, and epoch count.

However, with that said, the backpropagation algorithm crushes the neuroevolutionary approach in terms of speed. As can be seen from the graph above, to evaluate 1000 epochs takes only about 150 seconds, or 2 and a half minutes total. This is a massive time save in comparison to the neuroevolutionary algorithm at about 2 hours total.

However, all these values were with the same reduced dataset size that the evolutionary algorithm used, which may be unfair, as the evolutionary algorithm needs to evaluate upwards of 50 networks, whereas this algorithm need only do one. Increasing the dataset count the algorithm uses to the full 60k, we obtain values such as these:

Only 50 epochs were evaluated due to time, however it can be seen comparably, that backpropagation takes the lead in terms of performance, but it lags behind in total time by a lot. Backpropagation additionally has the strength in its consistency in learning. If the above tests were to run for longer, backprop would most likely outperform neuroevolution after a certain point, as gradient descent would be more optimal than random mutations. The reason this was not done was mainly due to time needed to accomplish this. To evaluate a full 1000 epoch backpropagation with these times, it would take about 12 hours to train - compared to the 2 and a half for neuroevolution.

Overall, when working with a small dataset, backprop is worse in accuracy, but significantly faster at evaluation. Conversely, with a large dataset backprop achieves a better accuracy, at the cost of taking more time.

Initially when creating this project I thought backprop would be better overall consistently, however that does not appear to be the case, and both algorithms have specific strengths and weaknesses when compared to one another.

Discussion

So the question remains, what can be done from here? This was a very introductory approach, and many changes could be made to enhance it. The program was able to get around 81% accuracy through what was essentially random changes progressively being made. A strictly random approach as was done here is just not the best for this type of task of tweaking over 70k weights.

It was interesting to see how the neuroevolutionary approach compared to the backpropagation approach, noting how the evolutionary algorithm achieved its results significantly faster than backpropagation, and with small datasets, performing better in terms of accuracy.

Increasing the overall complexity of this approach could be done. One could abandon the idea of enforcing similarity between the two networks and their competing algorithms and instead go for the most optimal approach to both the neuroevolutionary approach and compare it to the best backpropagation. This would allow neuroevolution to flourish a bit more, through the application of NEAT, or similar algorithms. I'm confident the neuroevolutionary approach could do significantly better than what was accomplished here.

On another note, it would be very interesting to see an implementation that unifies the two approaches, implementing a neuroevolutionary algorithm that perhaps instead finds the best starting weight and bias values for optimal gradient descent learning, or applying gradient descent learning somewhere within the mutation or crossover phase allowing for the process overall to be less random based.

Bibliography

[1] MNIST database, https://en.wikipedia.org/wiki/MNIST_database, Wikipedia, Wikimedia Foundation, 2021, Aug.

[2] MNIST Dataset, https://deepai.org/dataset/mnist, DeepAI.

[3] Neural Network From Scratch with NumPy and MNIST, https://mlfromscratch.com/neural-network-tutorial/#/, Machine Learning From Scratch, Hansen, Casper, 2020, Oct.

[4] Linear Classification, https://cs231n.github.io/linear-classify/#softmax, Convolutional Neural Networks for Visual Recognition, Stanford.

[5] An overview of gradient descent optimization algorithms, https://ruder.io/optimizing-gradient-descent/, Sebastian Ruder, Sebastian Ruder, Sebastian Ruder, 2020, Mar.

[6] Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com/, Neural networks and deep learning, Determination Press, Nielsen, Michael A., 1970, Jan.