EEG signal classification using a genetic algorithm

Cameron Dunn
cameron.dunn@anu.edu.au
2021 ANU 2601
CECS Building

Abstract. There are many techniques available to researchers to train networks for a range of different tasks. In this paper the technique of a genetic algorithm is explored to test its effectiveness versus a neural network trained with Stochastic Gradient Descent to evaluate the accuracy of a binary classification network. The results found demonstrate a statistically significant increase in performance and decrease in training time when SGD and backpropagation learning is utilised instead of a genetic algorithm.

1 Introduction

A common problem relating to neural networks is the issue of learning which techniques are best suited to discerning the features of a given data set. The reason for anyone to choose to use an evolutionary algorithm varies between tasks, but they are best suited for use in tasks where there is little information known about the important features in a dataset or the general approach to approximating the correct function is unclear. In the case of EEG signals, there is so much complex activity taking place in the brain all the time, and in many parts simultaneously. This, combined with the fact that we still do not have a thorough idea of how the brain functions at more than a macroscopic level, it results that it can be very difficult to extract meaningful and useful features from EEG data. This is where a technique such an evolutionary algorithm can provide highly valuable insight.

2 Data Pre-processing

For this model, the dataset that was used is the UCI EEG dataset, which was taken from a trial conducted by the Neurodynamics Laboratory at the State University of New York[4]. The primary goal of the experiment was to present the subject with a stimulus image taken from the Snodgrass and Vanderwart picture set, and then record their brain activity through an EEG signal generated by 64 electrodes placed on the head. Each electrode sampled at a rate of 256 times per second, resulting in data of dimensions 64x256, or 16384 inputs, which if fed directly into a neural network would be immensely difficult to find meaningful features. As such, the solution to this was to perform a Fast Fourier Transform on each of the 64 time-series data and split the resulting frequency data into three bands: Theta (4-7Hz), Alpha (8-13Hz), and Beta (13-30Hz). This operation resulted in the creation of 192 (64x3) data points for each individual trial. For this research task, it was decided that we will only test the networks on cross-subject data. The reason for this is that there is not very much data available for each individual in the trial, and it would result in a much less generalised network with limited function, i.e. it is not practical to have to train a new network for each individual that wishes to have a scan. Therefore, all training and testing is using cross-subject data.

Firstly, the data was required to be split into two parts: the training data set, and the testing data set. Each of the ~11000 trails from the ~120 person experiment has been compiled into one large dataset. The purpose of splitting the data set into two groups is to allow some of the data to be used as a form of 'test' to assess the accuracy of the trained network. The networks are trained on a majority subset of the total data set, and then the unseen portion of the network is shown the data from the test section as an analogue for if this network was used in a real-life environment with unseen data. To achieve this, the first step of pre-processing is to randomise the training data and the training labels in the same way and split them apart by a ratio of 0.8 to 0.2 for training and testing data respectively.

The network architecture that was chosen for this paper was to use a simple feedforward neural network, and as such, the data requires further pre-processing and reshaping, which will be addressed in the methodology section. Due to this, decision, it was vital to get an understanding of how the data is distributed in order to select the appropriate hyperparameters and pre-processing method. This was achieved as follows:

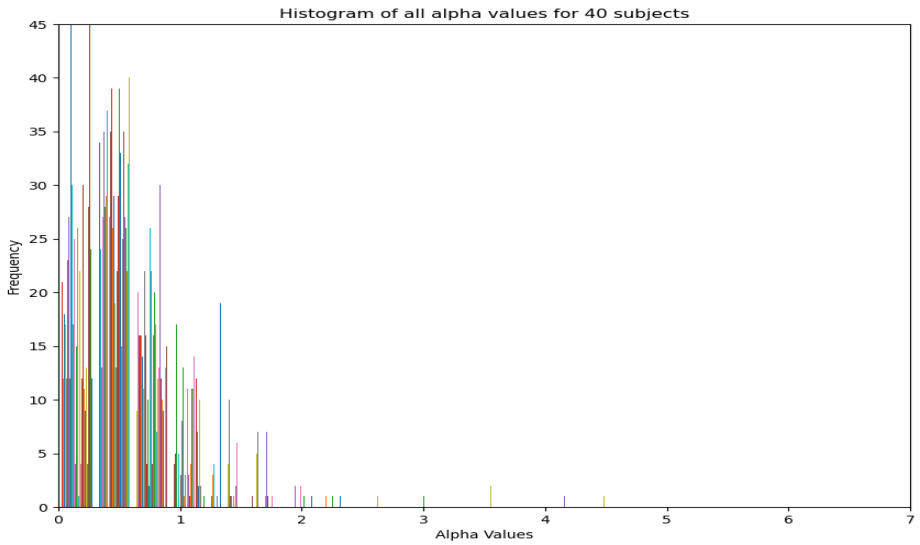

Figure 1. Frequency distribution for the alpha channel from a randomly sampled selection of 40 subjects

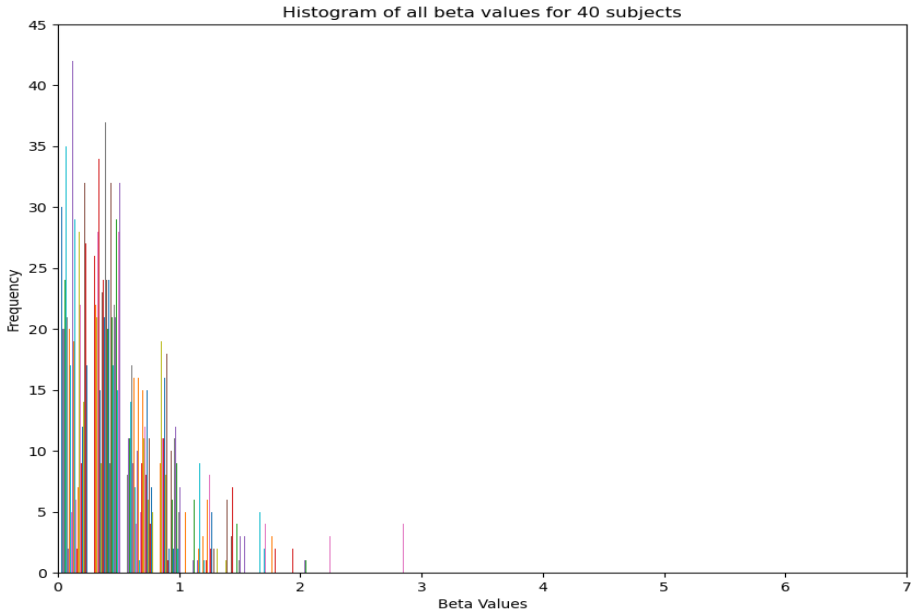

Figure 2. Frequency distribution for the Beta channel from a randomly sampled selection of 40 subjects

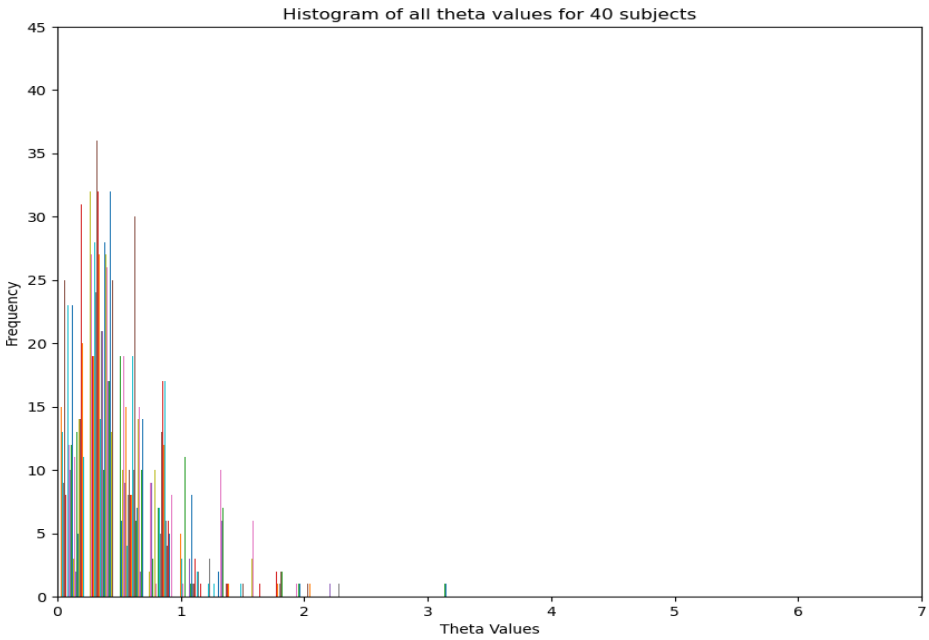

Figure 3. Frequency distribution for the Theta channel from a randomly sampled selection of 40 subjects

From this frequency analysis it was found that the majority of the values are distributed between zero and two for all channels, with only a few outliers larger than that. This data was therefore further normalised using what is called a Z-Score[3] according to the following equation:

$$Z_i = (X_i - X_{mean})/X_{Stdev} \qquad (1)$$

This is where each value of the dataset is normalised to be between -1 and 1, where its value is determined by how many standard deviations away each data point is from the average.

3 Methodology

The core idea of an evolutionary algorithm in this instance is to find the best possible solution to the problem by emulating the process of natural selection on a population of different neural networks. *Agents* within the population are continually culled and bred to pass the best 'traits' down through the generations to hopefully eventually produce an agent that combines all the best traits learned by the population.

3.1 Initialisation

Each NN is initialised according to the *Xavier Initialisation* scheme, which is a technique that randomly initialises the weights of each network that minimises the problem of disappearing gradients [8]. Even though disappearing gradients is not an issue for evolutionary algorithms as they do not learn using gradients, it is still best practice to use this scheme as it allows for better comparison to other training techniques that **do** use gradient descent.

3.2 Culling

Once a population of agents has been initialised, we must determine the *fitness* of each candidate. The *fitness* represents how accurately a given agent can correctly predict the label of either alcoholic or non-alcoholic. The function that was chosen is the *Mean Squared Error* loss, a commonly used loss function in SGD. Each agent is used to predict the labels of the entire training data set (~8000 cross-subject trials), and the average loss is returned as the *fitness* of the agent. Now that the agents have been assessed, the culling can begin. For this task, we chose to use a death rate of 0.5, meaning half the population die after each generation, with any agent falling below the median fitness being removed.

With the population reduced, we can now move to breeding the remaining candidates to produce more agents for the next generation.

3.3 Breeding / Crossover and Mutation

Now that half of the population has been removed, the remaining agents will need to *breed* to create new agents which can inherent the important and useful traits of its parents. Unlike in real life, any number of agents can be chosen to breed a new child, and as such, it was chosen that each child should be produced from three different parents. The reason for this is because it was found that such a design accelerated the learning process as more successful genetic material could be transferred to the next generation.

```
offspring = np.zeros(shape1)
for i in range(0, shape1[0]):
    for j in range(0, shape1[1]):
        rand = np.random.uniform(0, 1)
        noise = np.random.normal(0, rate)

        # Randomly choose a parent's 'genetic code' to be  passed on to the child
        if rand < 0.333:
            offspring[i][j] = m1[i][j] + noise
        elif (rand >= 0.333 and rand < 0.666):
            offspring[i][j] = m2[i][j] + noise
        else:
            offspring[i][j] = m3[i][j] + noise
```

Figure 4. Code example for breeding algorithm

As can be seen in Figure 4, the weights of the child are determined by three parents, *m1, m2,* and *m3,* which are chosen at random using the *Numpy.random.uniform*(). Additionally, there is also a randomly generated value for the amount of noise that should be added to the values of the child. This is a core aspect of any genetic algorithm, as it allows for some amount of randomness to be expressed through genetic mutation, which is extremely important for avoiding local minima and exploring as much of the feature space as possible. The specific values used for this will be discussed in a future section.

3.4 Behaviour through generations

Over time, the population will gradually converge to what it determines to be the 'global optimum' solution to the problem, meaning the genetic diversity will be greatly limited[2]. An example of this convergence has been provided:

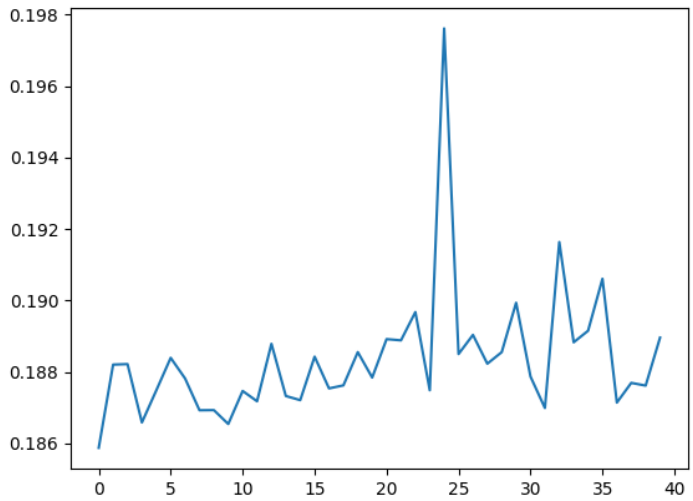

Figure 5. Performance of population of 40 agents after 75 generations

With some allowance for statistical outliers, it is clear how the performance of a population after many generations converges to a similar level performance, with the bottom half of the population exhibiting a slightly lower level of performance due to random mutations.

3.5 Hyperparameters

As with any neural network training algorithm, in a genetic algorithm there are a plethora of options for fine tuning the performance of the network through hyperparameters.

3.5.1 Neural Network Parameters

As previously mentioned, the chosen network architecture is a feedforward network with three layers, one input of a 192x1 vector, one binary output, and one hidden. Given that the input and output of this network are fixed, it is only the hidden layer that can be tuned. After much trial and error, it was found that the optimal number of hidden neurons was 38, so this is the value used in both the evolutionary algorithm, and the backpropagation control.

3.5.2 Population size

This is a hyperparameter that only really affects the results if it is sufficiently small. There are significant diminishing returns for increasing the size of the population past ~40 agents, and only adds to the already large amount of time taken to train the agents. Even though a larger population inherently means more traits that can be passed through the generations, it is not worth the trade off in time for the statistically insignificant increase in performance.

3.5.3 Mutation Rate

It was found that this hyperparameter had a significant impact on the overall performance of the network, and thus was subject to a lot of fine-tuning. Originally, the plan for the mutation rate was to have a constant value over the entirety of the training period, meaning it would mutate as much at generation zero as the final generation. However, this proved infeasible as the performance of the network severely suffered, sometimes falling even as low as a 50-50 guess of the label. The solution to this was to utilise a rate that was a function of time, which in this case is the number of generations that there have been so far, according to the following equation:

$$M_t = 1 / \sqrt{(t)} \qquad (1)$$

The reason that this equation is used is that it will gradually get smaller as the number of generations increases. This is an important feature as it is desirable to have a significant amount of noise and randomness present at the beginning. Having randomness present while the agents are unpredictable and have not yet learned very much information means that more possible solutions will be explored early on. The rate is then reduced over time to still allow for some exploration, but once a few optimal solutions have been learned, it is more effective to focus on exploiting them rather than try to search for a different solution.

4 Results

The following two figures are the results of the best network produced by a genetic algorithm and a neural network of the same dimensions that has been trained with backpropagation:

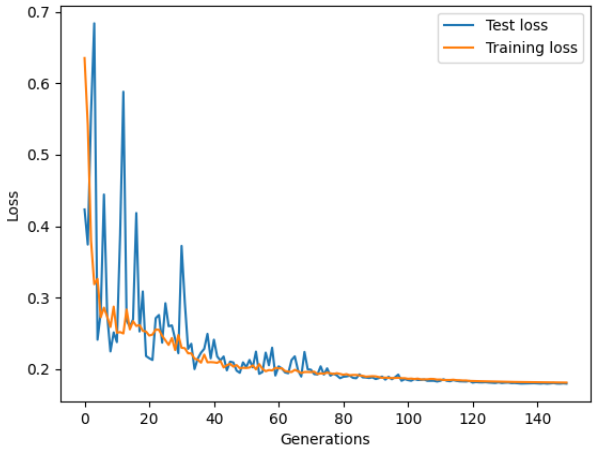

Figure 6. GA performance graph

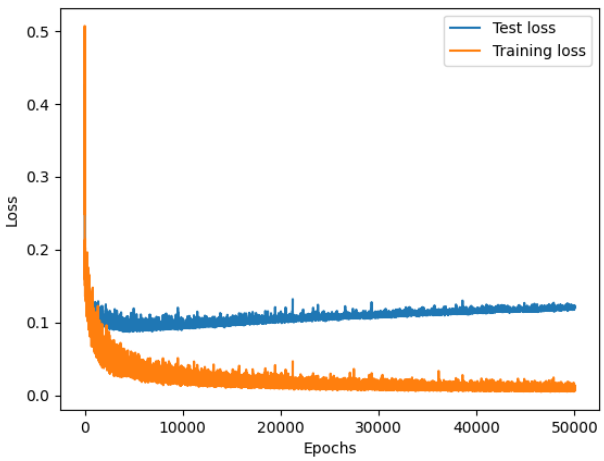

Figure 7. Backprop performance graph

For the GA performance, I took the best agent in each generation and used it to calculate the testing loss at each generation, and the training loss is the average loss across the whole population. The GA took approximately 5 minutes and 40 seconds to complete, whereas the backprop algorithm only took 1 minute and 50 seconds. This difference in time is a result of the functional complexity of each algorithm, with GA being $O(n^2)$, and backpropagation being O(n) in terms of its main Over all the trials of the GA, the minimum test loss I ever encountered was ~0.18, usually settling around a loss of 0.19. The minimum test loss for the backpropagation algorithm however is significantly less, siting at minimum of ~0.1, which increases as the number of epochs increases, indicating overfitting. These results demonstrate how even after only a few thousand epochs, which would only take a matter of seconds to compute, the performance of the backpropagation algorithm far surpasses that of the genetic algorithm.

5 Discussion

The results that have been gathered are sufficient to state that in the case of EEG signals, SGD and Backpropagation are the superior techniques for neural network learning. This is evident in the fact that not only does the SGD algorithm produce networks with far higher accuracy, but they are trained in a fraction of the time. One of the problems with genetic algorithms is that they are not well suited for approximating certain subclasses of functions, meaning that some problems are too irregular for a GA to produce exceptional results[5]. The class of problem pertaining to EEG signals seems to be too complex for the relatively insensitive GA algorithm to perform well.

In the future, there are a few aspects of the algorithm that should be changed for better results. The first improvement would be to implement weighting to each parent's genetic material in the breeding process based on their level of fitness. Theoretically this would mean that a more fit parent is more likely to pass on its genetic material than the other parent(s), and thus improve the performance of the child. The second improvement would be to use a different neural network design, such as a Convolutional Neural Network, instead of a simple feedforward network. As the data has been split into 3 channels, the data could be treated as if it were an image for which the best performing network would be a CNN. Finally, the last improvement would be to implement a more sophisticated fitness algorithm. The chosen fitness function for this task worked well enough, but a more finely tuned function such as Binary Cross Entropy Loss or similar could lead to a far superior model.

6 References

1. Yao Y, Plested J, Gedeon T. D, Deep feature learning and visualization for EEG recording using autoencoders, International Conference on Neural Information Processing, pages 544-566, 2018
2. R R Sharapov, A V Lapshin : Convergence of genetic algorithms, Patter Recognition and Image analysis, pages 392-397, July 2006
3. Gedeon T. D and Turner H. S.: Extracting Meaning from Neural Networks
4. Safavian S. R and Landgrebe D: A survey of Decision Tree Classifier Methodology, IEEE Transactions on Systems, Man, and Cybernetics, VOL. 21, May 1991
5. Forrest S, Mitchell M : What makes problems hard for A Genetic Algorithm?, Kluwer Academic Publishing, 1993
6. Prateek Joshi, Understanding Xavier Initialisation, https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/, March 2016