Hyperparameter Optimization on a Feed-Forward Neural Network using Pruning and Genetic Algorithms

Atif Farooq[1]

Australian National University, Research School of Computer Science,
Canberra ACT 2601, Australia

u6104602@anu.edu.au

Abstract. This paper presents the results of a study in which the 'Ionosphere Data Set' from UC Irvine's Machine Learning Repository was taken and fed through a multi-layered feed-forward network designed to solve a binary classification task. A technique that removes neuronal connections on the basis of a metric known as 'sensitivity' was then applied to the data in order to reduce the size of the network. Finally, a genetic algorithm was applied to a 'population' of these networks to evolve high-performing hyperparameters. The results showed that both approaches resulted in networks that resulted in improved classification accuracy. The genetic approach, however, was more computationally expensive.

Keywords: Artificial Neural Networks, Network Reduction, Pruning, K-Folds Cross Validation, Binary Classification, Sensitivity Analysis, Evolutionary Algorithm, Hyperparameter Optimization, Crossover

1. Introduction

There has been a massive revival of interest in Artificial Neural Networks (ANNs) and their applications within the last few years. A lot of the modern learning techniques that are being developed and employed are based on the basic foundation built by researchers in the mid-80s, through to the late 90s. Some of this earlier work also focused on techniques to optimize the performance of the network, without compromising its ability to generalize and to make predictions on unseen testing data. These techniques can either focus on one specific aspect of the network or attempt to find global solutions to an array of network 'hyperparameters', such as by using *genetic algorithms*.

1.1. Context

The choice of deciding upon the number of hidden neurons in a neural network is not an easy one. Many researchers have noted that training a simple feed-forward neural network within a reasonable timescale often requires the addition of hidden neurons that are not strictly required. Some of these might merely be providing a 'supporting' role and can effectively be 'pruned' from the network without any major loss. Determining an optimal network topology is important, as bloated topologies are generally computationally expensive and resource hungry.

Over the years, several major 'pruning' techniques have been devised and used for experimentation. Gedeon and Harris have discussed these in detail and summarized them on the bases of specific neuronal characteristics and their relationship with the network's learning ability [2]. These include metrics such as the hidden unit's 'relevance', 'contribution', 'badness', 'sensitivity' and 'distinctiveness'. In this paper, I will focus on *sensitivity*.

In general, if we consider the number of hidden units to be just another network hyperparameter to be 'tuned', the problem can be generalized into a broader one; that of hyperparameter optimization. This process constitutes an important step in the development of a learning model. Instead of focusing only on one parameter, we can optimize a whole array. As pointed out by Orive et al., hyperparameter optimization used to be performed manually based on human experience due to the limited computational power available [6]. However, with the availability of GPU processors and vast computing clusters, this task has become easier.

There has been significant research that attempts to find techniques to tune hyperparameters such as the number of layers, number of neurons and activation functions etc. [7] and [8]. These techniques include grid search, random search and Bayesian optimization-based methods [9]. One way to achieve this is to employ a '*Genetic Algorithm*'. These are general purpose, global optimization techniques that are inspired by the biological process of evolution. This paper will use the

canonical genetic algorithm to determine whether it returns a set of high-performing hyperparameters for a neural network or not.

2. Dataset

The chosen dataset comes from the Space Physics Group in Johns Hopkins University's Applied Physics Laboratory. It consists of data collected from a radar system in Goose Bay, Labrador, Canada. The data documents 'radar returns', which are used to study the physics of the ionosphere. The radar targets free electrons in the ionosphere by emitting pulses. The 'returned' signals are then passed through what is known as an 'autocorrelation function', using the time of the pulse and the pulse number as input arguments. The output of this function is 17 pairs of numbers, each corresponding to the real and complex parts of the complex electromagnetic output signal. Instances of these output signals are then used to classify radar returns as either 'good' or 'bad'. These outputs tell us about the structure of the ionosphere; 'good' signals imply the existence of some structure, whereas 'bad' indicates that the signals have passed through the ionosphere, and therefore no significant structure exists [1].

This classification process historically required manual supervision by experts, which was considerably time consuming. This makes the dataset a suitable candidate to feed into a neural network and perform a binary classification task. There are a total of 351 instances of output signals in this dataset. There are 35 predictor attributes. 34 of these are continuous valued (corresponding to real and complex parts of the 17 pairs of numbers from the output signal). The 35^{th} attribute indicates whether the signal is 'good' or 'bad', as specified above. Since this a relatively small dataset, the resultant network generated will not take as long to train, and hence, experiments with genetic algorithms (which are traditionally considered to be computationally expensive) can realistically be conducted even without a GPU processor.

2.1. Data Pre-processing

The data is mostly in the form of real values. Rare instances of integer values were automatically converted to 'floating point' values for the ease of computation during the pre-processing stage. Mimicking the steps taken by the original authors of the paper associated with the dataset, each input in the training set was normalized to fall in the range [-1, 1]. Since none of the input features were of a 'categorical' nature, there was no need to encode data accordingly.

3. Methodology

The aims of my research were threefold: Firstly, training a network to conduct a binary classification task and comparing my results with that of Sigillito et al. (who originally used this dataset on a neural network). Secondly, experimenting with 'sensitivity' and determining if it has any effect on my network's prediction ability or not. Thirdly, using a genetic algorithm on a population of these networks to 'evolve' a set of chosen hyperparameters and determining their behaviour with respect to the percentage current classification accuracy. Unlike traditional implementations, the genetic algorithm was run on a traditional CPU.

For the first part, since the size of the dataset is relatively small, instead of using the traditional training set/testing set split, the stratified k-fold cross-validation technique was used with k = 10. This ensured that I could maximally utilize the input instances without having to 'lose' any data. Furthermore, 'stratified' cross-validation was used to ensure that the proportions of the dichotomous 'good' and 'bad' classifications roughly stay similar in each fold. At the end of each 'session', the average percentage correct classifications of the test sets over the 10 training runs was reported. This differs from Sigillito et al. in that they simply used a test/train split to report their results, and their training set consisted of 200 radar return instances (out of a total of 351). The network that was built was a feed-forward net with 34 input neurons (each corresponding to a predictor attribute), two output neurons (corresponding to a 'good' value and a 'bad' value) and one hidden layer. The number of neurons in the hidden layers and the learning rate were changed during the experimentation.

For the second task, the network's number of hidden nodes were set to 10, and it was first trained for n = 20,000 epochs. Sensitivity analysis was then used to determine which hidden node should be pruned. Based on this, the network was trained again, but starting from the point at which it had left off, and without the hidden unit with the lowest sensitivity measure.

For the third task, in order to keep the cost of computation low, the k-folds cross validation value was lowered to k=3. The evolutionary process was run iteratively until the hyperparameter values converged.

3.1 Sensitivity Analysis

The seminal work on sensitivity was conducted by Ehud Karnin. In his paper titled *'A Simple Procedure for Pruning Back-Propagation Trained Neural Networks'*, Karnin defines sensitivity as the 'sensitivity of the global error function to the inclusion/exclusion of each synapse in the artificial neural network' [3]. A higher sensitivity would imply that the error was high when the unit in question was removed. Conversely, a very low or zero sensitivity would mean that the connection (represented by its weight) can be removed without there being any difference in error. Karnin achieves this by keeping 'shadow arrays' in memory, which keep track of the changes in weights during each epoch during the training. At the end of the training process, we obtain 'sensitivity' measurements for each connection. The connection with the lowest sensitivity can then be pruned. This approach is preferable to many earlier approaches as it does not require the cost function to be explicitly modified and does all the required computations during the training process itself (without any interference).

Sensitivity is calculated using the following formula:

$$S_{ij} = \sum_{0}^{N-1} \left[\Delta w_{ij}(n)\right]^2 \frac{w_{ij}^f}{\eta(w_{ij}^f - w_{ij}^i)} \qquad (\mathbf{1})$$

The formula shows the sensitivity for individual connections between nodes i and j. The term on the right side of the equation is calculated once the training has been completed. For the purposes of our understanding, we can construe the weights to be in the form of tensors (since we will do computations at the same time (and maintain weight tensors in memory). Thus, w_{ij}^f represents the *final* weight tensor when the training has concluded. Similarly, w_{ij}^i is the *initial* tensor containing the 'small random weights' that we have initialized the network to and η represents the learning rate. The first half of the equation represents the squared changes in the weights, summed over all the training epochs.

However, there are two possible issues that arise during the implementation of this formula that Karnin does not consider in his paper. First, as pointed out by Thimm and Fiesler, the denominator of the equation can sometimes become zero [4]. Following their suggestion during my implementation, the whole fraction is set to zero whenever this happens. Secondly, Wilson has discussed the case were calculation of sensitivities results in some negative values [5]. Ideally, the sensitivity calculation should never be negative. According to Wilson's research, this happens whenever the initial weights were not small enough. I solved this problem by initializing to very small random values.

3.2 Genetic Algorithm

A 'canonical' genetic algorithm was implemented on a population of simple feed-forward neural networks. The first step involved the generation of a 'population' of *n* neural networks. This number could be varied. To keep things simple and computationally feasible, a total of three hyperparameters were chosen: these included the learning rate, the number of epochs to train the network for, and the number of hidden later neurons. A function was written that could generate randomized networks, represented by arrays of hyperparameters. The ranges of these hyperparameters were constrained to lie within certain values:

Hyperparameter	Range
Learning Rate	0.01 - 1.00
Number of Epochs	5000 - 20,000
Hidden Neurons	1-20

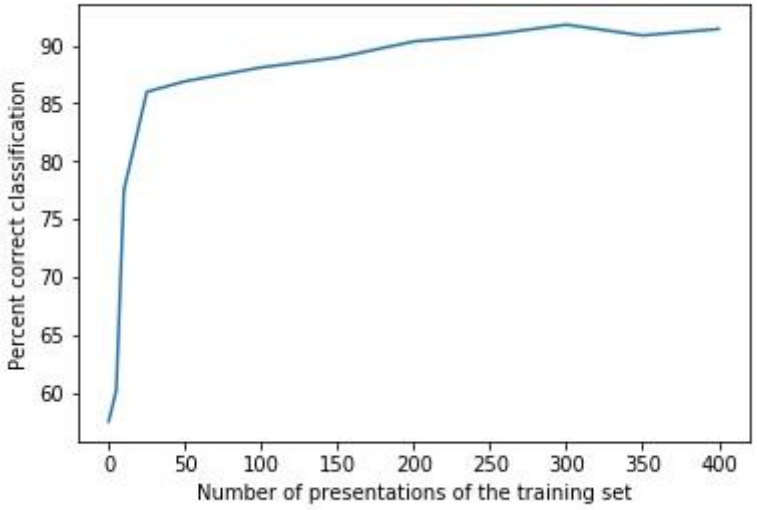
Each 'individual' in this population was then given a score of the basis of a fitness function. The percentage correct classification accuracy of that specific network after k-folds cross validation was used to determine its fitness score. The higher the percentage correct classification accuracy of an individual, the 'fitter' it was considered to be. Next, a list 'parents' was chosen. This list consisted of the top 20% fittest individuals, plus 15% randomly selected lesser performing individuals. This was done in order to promote 'genetic diversity', thereby reducing the chances of the algorithm getting stuck in local maxima. These parents were then used to 'breed' children using fix-point cross-over. The breeding involved the intermingling of 'chromosomal' information (represented as arrays). The population was then replenished. A small

random portion of chromosomal information from this new population was then mutated. This evolutionary process was then repeated until the hyperparameters converged to an array of high-performing values.

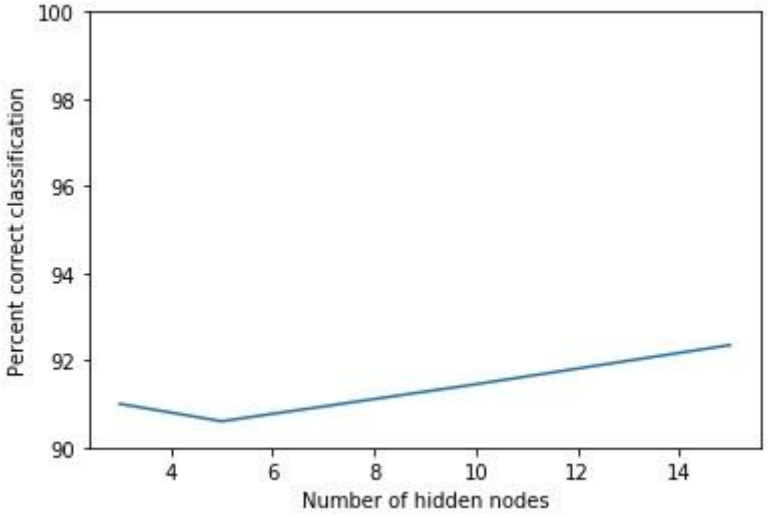
4. Results and Discussion

4.1. Binary Classification

Curiously enough, Sigillito et al. do not mention the learning rate they used to train their network in their paper. I could not match their given results in any way using the conventional learning rate of 0.01 since they have only reported results for the first 400 presentations of the training set. Using a low learning rate and the same number of hidden neurons (5) implied that I did eventually get accurate results, but the network had to be trained for a much longer period. I could, however, replicate their results when I drastically increased the learning rate to a value of 2. The figure below uses the same scales on the x and y-axes as used by the authors, and displays results that are roughly similar. The only significant difference would be that Sigillito et al. had a final convergence rate of between 99.5-100% correct classifications at the end of 400 presentations, whereas mine stopped at 91-92%.



Next, the researchers subjected their network to a few testing benchmarks, and reported the results. One of these included varying the number of hidden nodes without changing anything else, and documenting its effect on the percent correct classification. Their results indicated that the average percentage correct classification stayed above 95%, and rose to a slight local maxima when the hidden nodes were between 5-10 before evening out again. As before, my implementation stayed below 93% (assuming only 400 presentations of the training set) but rose slightly yet constantly in the 5-10 hidden node range. Some of these differences might be explained by my using a k-folds cross-validation, which evens out peculiarities, and the fact that my network still had not been fully trained within the 400 presentations of the training set.

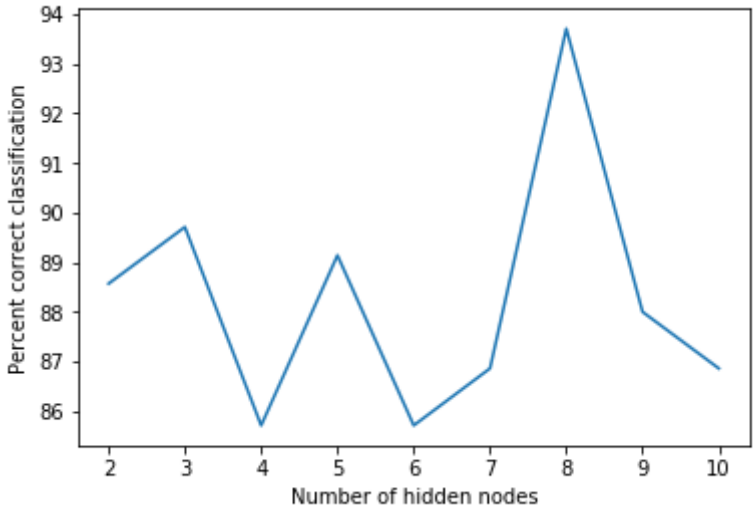


4.2 Sensitivity Analysis

A different script was used to implement the sensitivity analysis. This built on top of the basic network the functionality to calculate sensitivities for neural connections based on Karnin's equation. The sensitivities are presented in the form of an *m*n* matrix, where *n* = the number of hidden layer neurons and *m* = the number of output nodes. For the sake of convenience, I only focused on the connections from the hidden layer to the output later. An example run gave the results shown in the figure below:

1.00000e-02*	**hidden layer neuron**									
output node	0.4296	1.1389	0.3208	2.5225	0.5843	2.7452	2.6870	0.3652	0.1377	1.0058
	0.4296	1.1389	0.3207	2.5225	0.5843	2.7452	2.6869	0.3652	0.1376	1.0057

The script returns the index of the column where the lowest sensitivity value resides, and this can be used to remove that corresponding column in the final weight tensor (not shown here), and then reinitialize the network with that modified tensor and n – 1 hidden layer neurons. For instance, the example run returned a value of 0.1376 corresponding with [1, 8]. The removal of the column corresponds to 'pruning' the connection. The results of my experiments were very interesting: pruning seems to have a positive impact on the network's ability to correctly classify the input signals into the two output categories in the short term (starting with 10 hidden nodes).



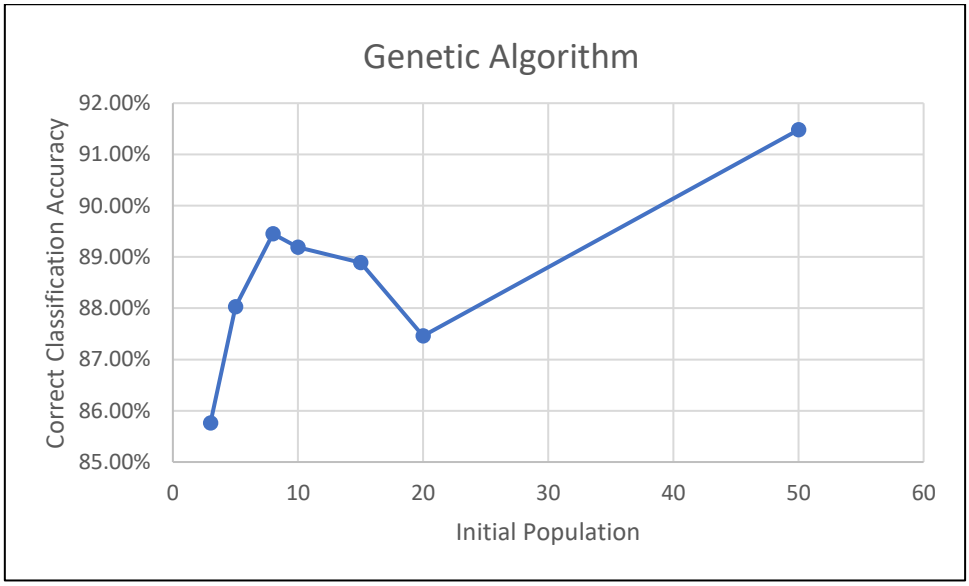
As shown in the graph above, the removal of the neuron for the first time increased the percent correct classification from 86.86% to 88.00%. Doing so for a second time resulted in a peak of 93.71%. This then diminished, before rising again to 85.71%. Surprisingly, the network was still performing quite well at only 3 hidden layer neurons, with a percentage correct classification rate of 89.71%.

4.3 Hyperparameter Optimization

I ran the genetic algorithm multiple times by initializing the number of individuals in the initial population to different values. Each run of the algorithm determined the set of three hyperparameter values that gave the 'best' result upon convergence. The results of the experiments are shown in the table below.

Population	Correct Classification Accuracy	Evolved Hyperparameters
3	85.76%	[0.52, 10000, 12]
5	88.03%	[0.07, 10000, 4]
8	89.45%	[0.46, 13000, 10]
10	89.19%	[0.16, 19000, 20]
15	88.89%	[0.83, 6000, 2]
20	87.46%	[0.04, 8000, 18]
50	91.48%	[0.28, 20000, 14]

Each array within the 'Evolved Hyperparameters' column represents the learning rate, the number of training epochs and the number of hidden layer neurons respectively of the network that performed the 'best' out of the initial population.



In general, the highest classification accuracy was given by a set of parameters that were evolved from the biggest population (n = 50). Conversely, the lowest was given by a very small population (n = 3). This is to be expected, as very small populations do not have enough genetic diversity to reach high-performing solutions in the limited search space. Larger populations, on the other hand, will contain enough diverse genetic material to gradually be able to converge to better solutions. It should be noted, however, that I constrained the limits within which my randomized hyperparameters could be chosen for the sake of computational efficiency. These results, therefore, are limited in that globally *better* solutions might exist if the algorithm is run for a long enough period of time for a broader range of hyperparameters.

Within this confined range, an evolutionary approach gave the best result of 91.48% correct classification accuracy as compared to the peak of 93.71% attained by using pruning and sensitivity analysis.

5. Conclusion and Future Work

My research reveals that 'sensitivity' is an important metric vis-à-vis the pruning of neural networks, and seems to accurately determine hidden units that can be discarded. Doing so does not result in any major loss in classification accuracy. In fact, my experimental runs provided slightly improved classification results in the short-term.

However, in order to maximize the benefits, the network topology should not be reduced to such as extent that it starts losing information and is not able to make accurate predictions on unseen data. Moreover, it does have certain problems in its current formulation. This includes the problem of negative sensitivity values in cases where network weights have not been initialized to small enough values, and the problem of occasionally having zero as the value of denominator in Karnin's original equation. Apart from work by Wilson and by Thimm and Fiesler, research into this area is relatively sparse. These problems need to be investigated further, so that a more efficient method of pruning based on 'sensitivity' can be devised and implemented in the future.

As far as the canonical genetic algorithm is concerned, it does converge on solutions that give better-performing sets of hyperparameters. However, this approach has its own set of drawbacks. Foremost would be the high computational resources required. Within very large search spaces, genetic algorithms clearly have a disadvantage if one does not have access to GPUs. On the flipside, they do enable us to determine near-optimal multiple hyperparameters all at once as opposed to focusing on any one attribute (as is the case with hidden layer node pruning). Further experimentation with the mutation rate and selection criteria for crossover can result it a better understanding of the way this algorithm operates, providing better heuristics for researchers who want to implement this evolutionary approach to their neural networks.

References

1. Sigillito, Vincent G., Wing, Simon P., Hutton, Larrie V, Baker, Kile B: Classification of Radar Returns from the Ionosphere using Neural Networks. Johns Hopkins APL Technical Digest (Applied Physics Laboratory). 10, 262—266 (1989)
2. Gedeon, T.D., Harris, D: Network Reduction Techniques. Proceedings International Conference on Neural Networks Methodologies and Applications. 1, 119—126 (1991)
3. Karnin, ED: A Simple Procedure for Pruning Back-Propagation Trained Neural Networks. IEEE TRANSACTIONS ON NEURAL NETWORKS. 1, 239—242 (1990)
4. Thimm, G., Fiesler, E. Pruning of Neural Networks. IDIAP Research Report 97-03 (1997)
5. Wilson, Matthew Robert. Comparison of Karnin Sensitivity and Principal Component Analysis in Reducing Input Dimensionality. Clemson University, All Theses, Paper 2357 (2016)
6. Orive, David., Yarritu, Gorka Sorrosal., Borges, Cruz Enrique., Andonegui, Cristina Martín., Vicario, Ainhoa Alonso. Evolutionary Algorithms for Hyperparameter Tuning on Neural Networks Models. Proceedings of the 26th European Modelling & Simulation Symposium. 402—409. (2014)
7. Murata, N., Yoshizawa, S., Amari, S., 1994. Network Information Criterion-Determining the Number of Hidden Units for an Artificial Neural Network Model. IEEE Transactions on Neural Networks, 5 (6), 865–872
8. Panchal, G., Ganatra, A., Kosta, Y.P., Panchal, D., 2010. Searching Most Efficient Neural Network Architecture Using Akaike 's Information Criterion (AIC). International Journal of Computer Applications, 1 (5), 41–44
9. Suganuma, M., Shirakawa, S., Nagao, T. A genetic programming approach to designing convolutional neural network architectures. *GECCO*. (2017).