Computational Intelligence
MIA 2024-25
November - 11- 2024

Evolutionary Computation

Training Neural Networks with Evolutionary Algorithms

Marina Bermúdez Granados
marina.bermudez.granados@estudiantat.upc.edu

Asha Hosakote Narayana Swamy
asha.hosakote@estudiantat.upc.edu

Table of contents

1. Introduction to the problem

Evolutionary computation methods can be applied to a myriad of fields, especially optimization problems. In this project, we will study their application to the training of neural networks in comparison to the other usual techniques. We will implement a multi-layered perceptron and we will optimize its weights through a derivative and a genetic approach. Then, we will generate synthetic data sets to experiment with these networks. The ultimate goal is to compare the performances between these two approaches.

2. Synthetic Data

In `data/synthetic_data_generator.py`, we implemented a `SyntheticDataGenerator` class to create synthetic regression data with different configurations. We used `make_regression` from sklearn for generating data, `MinMaxScaler` for scaling, and `train_test_split` to divide the data into training, validation, and test sets. We created three datasets:

1. Base Dataset: Consists of 2000 samples with 4 features and a noise level of 10. This dataset is stored in `datasets/base_dataset`.
2. Larger Dataset: Contains 4000 samples with 8 features, double the size and features of the base dataset, with the same noise level of 10. This is saved in `datasets/larger_dataset`.
3. Noisy Dataset: Includes 2000 samples with 4 features but a higher noise level of 50, making it more challenging for models. It is saved in `datasets/noisy_dataset`.

For each dataset, we set the training size to 70%, with the rest split between validation and test sets. Each split is saved as a CSV in its respective directory.

3. Derivative ANN Implementation

An Artificial Neural Network (ANN) is a computational model inspired by the way human brains process information. It consists of layers of interconnected nodes, or neurons, that can learn patterns from data. The goal of this project is to implement an ANN for regression, where we predict continuous values based on input features. By training this ANN on a given dataset, we can observe how adjusting the network’s weights and hyperparameters impacts its performance, allowing us to gain a deeper understanding of neural networks.

In this project, we implemented two versions of the ANN: one using custom code (without Keras) and another using Keras. Implementing it without Keras gave us insight into the low-level operations of a neural network, while using Keras allowed for faster development and better performance tuning. This approach let us compare both versions and understand the efficiency trade-offs between custom and library-based neural network implementations.

We have defined ANN_keras class where the ANN model uses the Keras library. This class accepts training, validation, and test data, and it initializes a model with one hidden layer (150 neurons with ReLU activation) and an output layer for regression. The model is compiled with the Adam optimizer, and metrics like Mean Squared Error (MSE) and Mean Absolute Error (MAE) are used to evaluate performance. The class also includes methods for fitting (training), predicting, and evaluating the model, with the option to save results to a CSV file for analysis.

4. Genetic ANN Implementation

The genetic algorithm is an evolutionary approach to optimization problems, inspired by natural selection and genetics. These algorithms store different populations as subsets of individuals, candidate solutions for the problem at hand. Individuals are formed with chromosomes, vectors of variable values, also known as genes. At every generation, the genetic algorithm selects a solution, creates offspring through crossovers, mutates the new population, and assesses their fitness. The fitness function returns the probability of being selected for each individual.

We are implementing the genetic algorithms using PyGAD, an open-source Python library that will allow us to use genetic algorithms to optimize neural networks. This library has its own modules for working with neural networks, including support for other libraries too. We developed two classes to experiment with two different implementations: one using the genetic neural networks from PyGAD library and another combining the genetic algorithms with the Keras models. The comparisons between the derivative and evolutionary approaches will be done with the latter option, though. The final implementations of all the approaches can be found in the models directory.

During the optimization process, the genetic algorithm will include all the trainable parameters from the neural network into its individuals. The same way as before, all these parameters are selected, mixed, and mutated until the best configuration remains.

5. Experiments

All the discussed experiments can be found in the experiments folder, while the consequent results can be found in the results directory.

5.1. Problem setup

Taking the derivative and genetic implementations, we will experiment with three synthetic datasets in order to assess their performances. Parting from a base dataset of 2000 samples and 4 features in a regression problem, the second data set will double its dimensionality, while the third one will increment the amount of noise up to 50%. During the experiments, the data will be split into 70% for training, 15% for testing, and 15% for validation. From here, we will test different architectures for a Keras model for all the datasets. The best one will be fused with the genetic algorithm, and different configurations will be examined thoroughly.

5.2 Best ANN

Our aim was to implement an artificial neural network (ANN) without using Keras, allowing us to explore how manual adjustments of weights and hyperparameters affect network performance. We found that training the network manually was computationally intensive. For this initial implementation, we used a simple set of parameters, specifically learning rate and activation function, but it still took considerable time to train. Additionally, we set the model with one hidden layer containing 150 neurons and an output layer.

In our tests with the non-Keras ANN model, the best results came from using a learning rate of 0.01 and a sigmoid activation function. This setup gave us a low Mean Absolute Error (MAE) of 0.036, a very low Mean Squared Error (MSE) of 0.002, and a high R²-score of 0.991, meaning the model made accurate predictions. The training time for this setup was 24.66 seconds, and the testing time was very quick at 0.03 seconds, which shows it was efficient. On the other hand, the worst performance was with a learning rate of 0.04 and a ReLU activation function. This setup gave us a much higher MAE of 0.435, a much higher MSE of 0.282, and a negative R²-score (-0.163), indicating the model didn't predict well. It took 23.68 seconds to train and 0.03 seconds to test.

Dataset	Learning rate	Activation function	Neurons	Train time	Test time	Test time	MAE	MSE	MAX error	R²-score
Larger dataset	0.01	Sigmoid	150	24.6555 seconds	0.0321 seconds	0.120 seconds	0.0363	0.0020	0.1465	0.9915

Table: Best ANN Parameters and results

5.3 Best ANN Keras

To optimize our process, we then used Keras, which allowed us to test a wider range of hyperparameters, including learning rate, activation function, number of epochs (50), and batch size (32), with the Adam optimizer (learning rate set to 0.01). The results demonstrated that Keras significantly improved processing speed and performance. The analysis results for the Keras-based model, including various parameter combinations, can be found in `results/ANN_derivative_keras/ann_derivative_keras_analyse.csv`, while the results for the non-Keras ANN implementation are documented in `results/ANN_derivative/ann_derivative_analyse.csv`.

For the Keras-based ANN model, the best performance was on a larger dataset with a learning rate of 0.01 and a sigmoid activation function. The model achieved an MAE of 0.033, an MSE of 0.002, and a very high R²-score of 0.991. The training time here was 8.68 seconds, which is pretty fast for a larger dataset. However, the model struggled with noisy data when using a learning rate of 0.01 and ReLU activation. This setup gave us the highest MAE of 0.176, MSE of 0.048, and a much lower R²-score of 0.810, showing the impact of noisy data on performance. The training time for this was 8.69 seconds, similar to the other tests, but the errors were higher.

Overall, we found that the best results came from using a sigmoid activation function with a learning rate of 0.01, and ReLU struggled, especially with noisy data. Training times were fairly quick, but the type of data and the choice of hyperparameters made a big difference in the model's performance.

Parameter	Value
Datset	Larger dataset
Learning rate	0.01
Activation function	Sigmoid
Optimizer	Adam
Epochs	50
Neurons	150
Batch size	32
Train time	8.69 seconds
Test time	0.12
MAE	0.0327
MSE	0.0016
Max error	0.1448
R^2-score	0.9910

Table: Best ANNKeras parameters and results

5.4. Best Genetic ANN

We need to test different configurations of the genetic algorithm using the best neural network architecture. Specifically, we will explore the options of five parameters: selection criteria, mutation methods, and crossover policies. For all experiments, we will set the number of generations to 50, mutations set to the default 10%, and keeping all the parents for the next generation.

There were many selection criteria, so we considered most of the available options. The first one is the steady set selection, which tends to converge faster while sacrificing the exploration. Instead of picking the roulette wheel selection, we opted for the stochastic universal sampling selection to reduce the risk of premature convergence. The last one is the tournament selection; the next generations are defined by the individuals with the best fitness from random comparisons. We will leave all parameters related to the selection criteria to their defaults.

We have decided to follow two methods to generate the new solutions from the parents: the default single-point crossover and a uniform approach. When it comes to mutations, there were also a few options to choose from. We decided to leave the default random mutations and add the scrambled mutations, where the values of a consecutive set of chromosomes are randomly shuffled.

From our results and prioritizing the mean absolute error, we find that the best configurations highly depend on the data set used. For the base data set, stochastic uniform selection, single point crossover, and random mutations were the highest-ranking parameters among the best results. The noisier dataset still has the best results when single point crossover is used, but using the tournament selection with scrambled mutations. The larger dataset mixed the results even further; the top configurations usually used stochastic uniform selection and single point crossover like in the first dataset, but adopted the scrambled mutations from the second one. Overall, the best metrics occur in the base data set, the noisier one and the larger one, respectively. The measured times are inversely faster, though. From slowest to quickest, the ordered datasets would also be the basic, noisy, and large. However, a stopping criteria would have greatly shortened the measured times. Since our objective is to study and compare approaches, we decided not to implement this behavior so as to avoid early convergences. Either way, we can infer the possible stopping points with the following plots.

In the following table, we visualize for each dataset the fitness against all the generations. These plots were extracted while using the best configuration for each case. The parameters may vary from the ones described previously because we were focusing on the commonalities of the best results, not the best combinations of parameters. We can see how there are multiple times in all plots where the fitness seems to converge. The large dataset seems to fall into a suboptimal configuration before finding the next most optimal one. We can see how important it is to avoid premature convergence and run the same experiments multiple times to better exploit the randomness inherent in evolutionary algorithms.

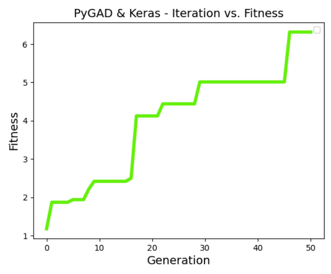
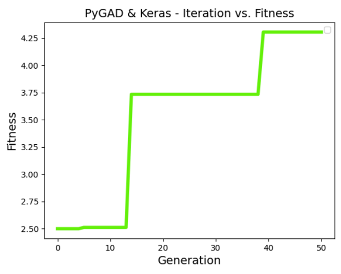
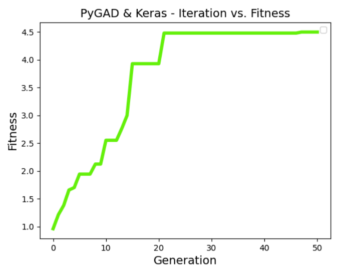
Base dataset	**Large dataset**	**Noise dataset**
		
Stochastic uniform selection Single point crossover Random mutation	Tournament Single point crossover Scrambled mutations	Stochastic uniform selection Single point crossover Scrambled mutations.

Figure Fitness vs Generation per data set

The worst performances are also highly dependent on the application. The base data set does not seem to show any specific pattern. There is no discernible pattern for the noisy dataset either. On the other hand, all the lowest rankings within the large dataset have a uniform crossover combined with random mutations. We can conclude that too much randomness does not benefit the final results.

6. Analysis

For every experiment, we have measured the mean absolute error, the mean squared error, the maximum error, the training times, and inference times. We will be comparing the performances with the respective best and worst configurations for each Keras implementation per dataset. All the data will be organized into the three following tables with the most relevant metrics. The green values represent the best values, while the red ones indicate the worst ones.

Starting from the base dataset, we can see that the approach with the genetic algorithm does not quite match the results from backpropagation. However, all metrics indicate that both approaches indicate that the predictions are good approximations to the actual values. Note that the training times from the genetic algorithm should not be taken into account due to the lack of stopping criterion.

Base dataset	MAE	MSE	Max_error	Training times	Inference times
ANN with Keras	**0.03908**	**0.0025**	**0.2144**	**6.072431**	**0.11287**
	0.0738	**0.0072**	**0.2188**	**5.8488**	**0.1121**
GANN with Keras	**0.1570**	**0.0333**	**0.5290**	**210.7649**	**0.1656**
	0.3130	**0.1563**	**1.5331**	**308.0828**	**0.1600**

Similarly to the previous case, the genetic approach does not reach the metrics from the derivative one. The metrics still indicate good predictions, though. Most importantly, the metrics seem to remain within the same ranges. This means that an increase in dimensionality does not affect the optimisation process. As expected, the backpropagation method spends more time training parameters. Conversely, the genetic approach takes less time than before in both training and inference. With better parameters and a stopping criteria, the genetic approach may be able to obtain the best configurations quicker.

Large dataset	MAE	MSE	Max_error	Training times	Inference times
ANN with Keras	**0.0327**	**0.0016**	**0.1448**	**8.6826**	**0.1208**
	0.07634	**0.0075**	**0.2061**	**13.0537**	**0.1625**

GANN with Keras	**0.1634**	**0.0438**	**0.8408**	**172.3163**	**0.0958**
	0.3922	**0.2479**	**1.6402**	**171.3692**	**0.0945**

There are some notable differences compared to the previous cases. It appears that the noisier synthetic data affects both techniques equally, doubling the performance metrics, minus the times. No method appears to be better than the other. In spite of this, even with 50% of noise, both techniques show adequate predictions can still be executed. Surprisingly, the inference times are better in the genetic case.

Noisy dataset	**MAE**	**MSE**	**Max_error**	**Training times**	**Inference times**
ANN with Keras	**0.1624**	**0.0411**	**0.5574**	**7.2295**	**0.1175**
	0.1759	**0.0478**	**0.7134**	**8.6898**	**0.1430**
GANN with Keras	**0.2234**	**0.0790**	**0.7409**	**117.5965**	**0.0752**
	0.2602	**0.1112**	**1.5979**	**116.8764**	**0.0817**

7. Conclusion

We compared the performance of artificial neural networks with derivative and genetic optimisations across three different datasets in a regression problem. As we have seen with all the previous metrics, in all cases, the backpropagation-trained networks performed better than the genetic-trained ones. This suggests that the classic derivative approach was sufficient for these datasets. The additional complexity of GANNs did not lead to significant improvements over standard ANNs.

However, it is important to note that the performance of GANN is highly dependent on the chosen parameters. A different set of parameters than the ones chosen in this report could yield better results. There are many more options in the PyGAD library, from elitism techniques to even more selections, crossovers, and mutations. Note again that the obtained results indicate good predictions, despite not completely matching the classical methods.

The training times are not comparable since the genetic neural networks did not have a stopping criteria. Regardless, we have been able to take a closer look at the fitness evolution across generations without early convergences. The mutations caused random improvements at specific moments of time, as well as stagnant sections.

In terms of more specific conclusions, we can make the following claims for our problem:

- **Accuracy:** We noticed ANN model performed better than GANN, ANN training with Keras yields more accurate predictions than the genetic algorithm-based approach.
- **Inference time:** Surprisingly, GANN takes comparatively less time than ANN model for predicting.

For future work, we could explore even more configurations for the genetic algorithm. We had to limit the scope of the work to just experimenting with some of the selection, crossover, and mutation types offered. In theory, we could try implementing even more strategies or add alternative policies, like elitism among others. In addition, we could vary the percentage of mutations and the amount of parents to be kept per generation. The genetic algorithms are very flexible evolutionary-based techniques with many study possibilities.