University Of London
BSc Computer Science

CM3070 - Final Project
Evolving optimal neural networks with Genetic Algorithms

Janos Tamas Ruszo

2024

Contents

1 Introduction

This document reports the reviewed literature and work done while designing, implementing, evaluating and refining the approach used for evolving the architecture of neural networks with genetic algorithms.

1.1 Template Chosen

For this project the 2nd project of the **CM3020 - Artificial Intelligence** module was chosen: **Automated design using evolutionary computation**.

1.2 Github Repository

The code developed for the project and instructions on reproducing results, running resulting code is available at: https://github.com/jruszo/uol-cm3070-fp

1.3 Motivation

Neural networks were first proposed in 1944, and since that date, the possibilities of neural networks have evolved constantly. With the increase in available computational resources, we are seeing better-performing and bigger neural networks. Especially in the past few years, the competition to create more advanced and capable networks got to the point that the models can no longer be run on traditional hardware, as they require multiple, server-grade graphics processing units (GPUs).

While the bigger models are more capable, designing the architecture of the model is done by the developers and researchers based on their previous knowledge and intuitions. This can lead to suboptimal utilization of the resources, as there is a good chance, that a different, smaller neural network would perform just as well.

Genetic algorithms have already proven themselves in scenarios where multiple possible good approaches exist and the correlation of parameters, and change of parameters have a non-deterministic effect. Genetic algorithms are widely used for designing moving structures, such as robots and there were a few tries to use them for evolving neural networks, that resulted in unexpected, novel architectures. Due to the compute intensity of genetic algorithms combined with neural networks, the available research is very limited, however, due to the increase of available computational capabilities, more complex tasks can be experimented with.

1.4 Neural Networks

Artificial neural networks try to mimic the biology of a brain. In an artificial neural network, there are cells, like neurons in a brain. The neurons are usually organized into layers with variable amounts of connections between the layers. During the training of a neural network, the error (difference between the result and expected result) is back-propagated through the network, tuning the weights of the connections between layers and the biases of the neurons.[1]

1.5 Genetic Algorithms

Genetic algorithms are inspired by natural selection and are part of the greater group of evolutionary computing. Genetic algorithms are searching for an optimal solution using evolutionary techniques such as crossover, mutation and selection. In most cases, genetic algorithms work better than traditional neural networks in cases, when the parameter space has no gradient and multiple local optima present. Due to the lack of gradients, it is not possible to effectively back-propagate for fine-tuning, which is required for neural networks.[11]

2 Literature Review

2.1 Designing Neural Networks using Genetic Algorithms[8]

This is the first paper I am reviewing as part of the project. The authors of the paper theorised, that it is possible to solve the XOR problem, without programming the neural network manually but using genetic algorithms to create both the architecture as well as set the weights of the units.

Since the change in the architecture is not linearly correlated to the performance of the neural network, making the problem undifferentiable, which makes it impossible to use techniques such as gradient descent. Furthermore, due to the number of possible networks, the number of combinations is infinite, making a grid search or random probing impossible or inefficient. The researchers chose to use strong specification, which encodes the exact connectivity patterns in the genomes, eliminating human biasses.

During their testing, they found that the genetic algorithm could find a working architecture within 10 generations, however, they found that the architecture was different. The algorithms created an extra connection between the input and the output, which made it symmetry-breaking and resulted in faster learning of the task. The second task they tested is called the four-quadrant problem. It is a more advanced version of the XOR problem, where the inputs instead of 0 or 1, are real numbers ranging from 0.0 to 1.0 inclusive. Traditionally this problem is solvable with a one-hidden layer feedforward network with acceptable precision, given the hidden layer has enough units. With two hidden layers, a smaller solution is possible. The researchers trained the networks for 200 epochs, and after 10 generations they received good results from the genetic algorithm, however, the resulting networks tended to be asymmetric, unlayered and convoluted, making it very hard to understand its inner workings.

As their conclusion, further research is needed in this area, and their paper just represents the early stages of a long-term research program, however, the

results are promising, they were able to rapidly evolve network architectures that were capable of learning simple mapping problems.

2.2 Use of Genetic Algorithms with Back Propagation in Training of Feed-Forward Neural Networks[2]

The authors proposed a model selection methodology for feedforward neural networks using genetic algorithms. They used a genetic algorithm to evolve the type of inputs, the number of hidden units and the connection structure between the layers[2]. They also tested how the performance changes if they introduce a local elitist procedure.

The authors implemented a custom fitness function, which is based on the inverse of the mean squared error of the neural network. The lower the error rate of an individual the closer fitness gets to one. For the genetic algorithm, they implemented reproduction, crossover, mutation and selection operations, with an initial population size of 50. They encoded the initial weights and number of hidden units, number and type of inputs into a binary string as the individual network's phenotype.

According to their results, the genetic algorithm achieved lower values of mean squared prediction error(MSPE), but with a larger number of hidden units (more complex network) compared to other model selection methods, such as the SIC and the AIC [10]. They also concluded, that using the local elitist procedure, greatly reduced the time required for the genetic algorithm to converge.

2.3 Training Feedforward Neural Networks Using Genetic Algorithms[9]

In this article, the authors proposed to use a genetic algorithm to optimize the weights and biases of the feedforward network. They reason that the effectiveness of backpropagation diminishes as the complexity of the network increases, furthermore, backpropagation tends to stuck in local minima. The backpropagation can escape the local minima if the momentum is large enough, however, there are no guarantees, that the next local minima will be better. Therefore, they replaced the traditional backpropagation with a genetic algorithm and used evolution to adjust the weights and biases of the network. They defined a set of mutation methods such as crossover, mutate weakest node, and biased and unbiased mutate weights. These all adjust the weights differently. For the fitness function, they chose to use the sum of the squares of the errors after evaluation on a test dataset.

The authors conducted multiple experiments by limiting the available mutation techniques to determine which type of mutations provide faster and better convergence. As per their published results, the genetic algorithm reached a lower error rate than backpropagation in as little as 50 iterations and contrary to backpropagation the error rate continued to decline monotonically. Future work changing from fixed training data to a stream of continually changing training data would require modifying the genetic algorithm to handle a stochastic evaluation function.

2.4 Genetic algorithm-determined deep feedforward neural network architecture for predicting electricity consumption in real buildings [7]

The authors proposed to determine the architecture of the neural network using a genetic algorithm, due to the complexity and trial-and-error nature of manually figuring out the optimal number of hidden layers and the number of units on every given layer. They used a real-world problem as their dataset, based on the electricity consumption in the United Kingdom (UK). They implemented and tested different numbers and sizes of hidden layers, four different activation functions and four different optimization algorithms. They also conducted experiments to determine the optimal probability of retaining, selection and mutation. In their testing, 80%, 20% and 20% respectively yielded the best results. For determining the fitness of the individuals, the authors chose the mean absolute percentage error function.

During testing the genetic algorithm reached convergence in less than 50 generations and reached similar performance as the reference models. On the day-ahead hourly data, the genetic algorithm showed marginally worse results during the training phase but achieved better results on testing data compared to reference models. For week-ahead daily energy consumption data, the genetic algorithm achieved better results both during training and testing compared to the reference models.

3 Design

When evolving neural networks, there are two viable encoding methods present. The strong declaration, which explicitly describes which neuron is connected to which. The other declaration is called weak declaration. In this case, the architecture is encoded in the DNA of the individuals, however, the explicit connections are not described. For the project the weak encoding was chosen, which allows us to shorten the DNA sequence, by defining only the type of layer and its characteristics. Furthermore, the weak encoding allows us to make use of already existing layer types in pytorch.

3.1 DNA Scheme

For effective encoding, the DNA translation was defined first. For this project,we use only a subset of the available layer types. We chose the most common **Dense layer**, where the neurons are connected to all the neurons of the previous layer, hence forming a dense connection. This type of layer is commonly used for classification tasks.

The next layer we defined is a convolutional layer, namely **Conv2D**, which allows convolution on 2-dimensional inputs, offering great capabilities for vision tasks, such as object recognition.

We also included activation layers in the definition. The activation layer defines the output of the neurons of a given layer. Different activation layers have different characteristics, here we defined two commonly used layers. We included the **ReLU** activation layer, which is defined as the positive part of the input. This activation layer results in values from 0 to +inifinite. Since all the

layers have different amounts of required and optional parameters, we defined the encoding scheme as follows:

- Dense layer: D,<number_of_units >
  example: D,15
- Conv2D layer: C,<<channels>
  example: C,32

The activation layers are not encoded as separate chromosomes, as we only use a single type of activation layer as of now. After every layer we include a ReLU activation layer. Once a chromosome is defined, chromosomes are separated using semicolon(;) for easier processing and readability.

An example individual with two convolutional layers followed by 2 Dense layers is defined as `C,32;C,64;D,9216;D,128`

To make this definition task agnostic, the input layer (head of the individual) is not encoded in the DNA, but rather supplied by the task itself and is the same for all individuals in the population. The output layer (tail of the individual) is also supplied statically by the task on hand and is the same across the generations and the whole population.

3.2 Fitness function

For evaluating the fitness of the individual within the population, we use two performance factors from the individual. We rank the individuals within the population in ascending order based on their achieved accuracy on the test data. We then multiply the rank of the individual with a value X_1, which is a hyper-parameter of the fitness function.

Then we rank the individuals again in descending order based on the number of parameters the individual's neural network has (complexity). We then multiply this rank of the individual with a value of X_2, which is also a hyperparameter of the fitness function.

The X_1 is a multiplier to the rank of the individual, making the shorting by accuracy more prominent. Then we also multiply by the accuracy itself to get the fitness value. This multiplication differentiates the fitness of the individuals more, as individuals with low accuracy will have much lower fitness. The population size is denoted by n, therefore the individual i (i=1,n) has accuracy, denoted as A_i. We formulate the fitness function as follows:

$$F_i = rank(A_i) * X_1 * A_i$$

This defined fitness function is incentivising not only to be more accurate but also to evolve into smaller, more efficient networks at the same time, which is one of the goals of our experiment.

3.3 Evolution

Traditionally, genetic algorithms use mutation and crossover for evolution. On a crossover, the chromosomes of the parents are split up and re-combined creating one or more child individuals, which poses a mix of the 2 parents' chromosomes. We chose not to include crossover in our implementation for now, however, it is

an open possibility to further develop the experiment by implementing crossover as well. Mutation is when the chromosome of the individual's phenotype is changed. We defined 4 types of different mutations, all having a 25% chance to occur for every chromosome.

When mutating the phenotype of an individual we randomly select a chromosome for the mutation. The possible mutations that can happen with the chromosome are:

- Add a new layer that is larger than the layer defined by the chromosome, after the selected chromosome
- Add a new layer that is smaller than the layer defined by the chromosome, after the selected chromosome
- Change the size of the layer by multiplying its size with a random factor between 0 and 1
- Delete the selected chromosome

To preserve high-performing individuals from the population, we remove 25% of the worst-performing individuals and replace them with a possibly mutated version of one of the top-performing individuals as a parent. This motivates evolution to produce better new individuals based on the phenotype of a successful parent. If a child performs better than the parent, it pushes the parent lower in the ranking and causes it to be eliminated eventually.

Also for the sake of fairness between runs, we reset the weights of the neural networks and fully re-train them, to measure real improvements between generations.

4 Implementation

The algorithm is written in the Python language using Pytorch as the machine learning library. The final version of the algorithm uses image recognition, so the torchvision package is included as well, which provides functions to handle image processing and transformations. The main implementation of the algorithm consists of two classes. First I implemented the "ConstructNet" class, which takes numerous arguments, but most importantly a list of strings, the DNA sequence. From that DNA sequence, the class initializes a neural network which can be trained and evaluated using standard pytorch functions. The other class I implemented is called "GA". This class takes standard genetic algorithm parameters as well as parameters for the neural network, then synthetizes the configured amount of DNA structures, that gets converted into neural networks using the ConstructNet class. After every iteration, the GA class executes different mutation techniques, which produces new individuals for the population.

4.1 The GA class

The GA class is responsible for managing the population and its evolution. First, it receives parameters that describe the experiment we are doing. It receives instructions regarding how many generations need to be evaluated, how many individuals should be in a generation, what are the initial settings for the DNA

generation as well as instructions for the ConstructNet class, which are more-or-less standard neural network parameters, such as the used loss and optimizer functions, the number of epochs we run during training, train and test data loader and the input and output size of the neural networks. Once the class is initialized, the auto_evolve function orchestrates the process by first generating a population, then train and evaluating them, finally, it calculates the fitness of the individuals and prints out some statistics.

The generate population function makes sure that every generation has an equal amount of individuals. After an iteration, it deletes 33% of the individuals with the worst fitness and replaces it with one of the individuals from the top 5.

When the GA class is generating an individual it is using only "C" as the type of DNA, which corresponds to the Conv2D CNN layer. The second parameter of every chromosome is the size of the layer. To keep the layer size reasonable, the layer size is always a factor of 2. Furthermore, when a new individual is created we execute the mutation function as well to differentiate the new individual from its parent. The mutation function takes the genome of an individual and executes four different mutations, each with a 15% chance. First, it might add a larger layer by choosing a layer randomly inserting a new layer after the chosen one and increasing the factor used for the layer size by one. The second mutation does the same, but instead of creating a larger layer, it creates a new smaller layer. The third mutation selects a layer randomly and randomly either increases or decreases the factor of layer size by one. If the layer size becomes zero, then the layer is removed, however, if it would remove this last layer then the mutation leaves the layer untouched. The last mutation selects a layer randomly if there is more than one layer in the genome.

Once the population is created and mutations are performed, the GA class calls train and then evaluation population. These functions are just wrapper functions to the ConstructNet class's neural network training and evaluation functions.

Finally, we calculate the fitness. First, we rank the individuals by accuracy and then parameters in case two individuals have the same accuracy. Then we calculate the fitness of the individuals by taking their rank * 10 * accuracy. Since we put the best individuals first, this results in lower fitness for the best individuals essentially trying to minimize the fitness of the individuals.

4.2 The ConstructNet class

This class is responsible for the neural network itself by creating it from a DNA sequence and for the training and evaluation of the network. The class takes in inputs from the GA class. It takes a DNA parameter which is a list type, consisting of the DNA of the individual, the loss function and the input and output size. On initialisation, the algorithm iterates over the DNA and creates the required CNN layers after each other using ReLU activation layers. When the CNN layers are created a Flatten layer is appended, to make the output compatible with Dense layers. To determine the resulting size of the Flatten layer, a forward pass is executed using the input size parameter. Once the flatten layer's output size is acquired two dense layers are created. The first one has the output of 10* the final output size then the output layer itself with the specified output size.

The train and evaluation functions are standard implementations. The train function takes in a device a train data loader, the optimizer function and the number of epochs to train for. First, it moves the data and target to the device in case of CUDA or MPS device, then makes sure that the optimizer is zeroed by calling optimizer.zero_grad(), following that the data is fed into the network and the loss function is called against the ground truth and the result from the network. Finally, we call the backward method of the loss function and step the optimizer.

5 Evaluation

5.1 Reflection on aims

The first aim of the project was to evolve the architecture neural networks for a selected task without giving a good base solution. This task turned out to be way more complex than initially thought, due to the different implementations required for different layer types. Also, there are several limitations imposed by the different layer types, for example, it is not easily possible to connect a Conv2D layer after a Dense layer. Due to these challenge, the project had to be split into two phases. In the first phase, the prototype was using only Dense layers on the Iris Flower dataset[3], which is a simple toy dataset and was presented in the preliminary report. The original goal of the project was still to do image recognition on the MNIST[6] dataset, which was the focus of the final iteration. Due to the complexity of working with the CNN layers, the combination of layers and number of layer types had to be reduced, therefore only CNN layers were used for the MNIST classification, without Batchnormalization or Maxpooling layers, as originally planned.

The other challenge that I faced during the experimentations was the compute intensity of the evolution. To do 100 evolutions on a small population of 30 individuals for 15 epochs took five hours on M2 apple computer with the MPS backend, and took close to 7 hours on Google Colab using T4 GPU acceleration, which limited the number of experiments possible during the timeframe.

The second aim of the project was to achieve good performance using Dense layers on the toy dataset, to validate the hypothesis that genetic algorithms are capable to evolve more complex network than the XOR problem. This task was successfully completed in the prototype, which succeeded as the maximum accuracy reached was around 95% and the number of parameters showed a decrease over time, however as with prototypes the evolution was not stable and the accuracy declined after 70 generations as we can see on the graph below.

The third and last aim of the project was to achieve a comparable performance to existing solutions on Kaggle to the MNIST dataset. This dataset is much bigger than the previous, with more than 10.000 samples and 10 different output labels. Training these models required more computation extending the duration of experiments and more memory for the networks, limiting the size of the population. Despite the difficulties, it was possible to implement an algorithm using CNN layers that can perform adequately compared to manual solutions (see Table 1).

5.2 Self-reflection

The complexity of the task greatly exceeded my expectations and required me to deepen my programming knowledge in Python and required me to go beyond my knowledge in deep learning and pytorch. Computational requirements is a major limiting factor in further development and experimentation as it takes 5-8 hours to get results of a change and see whether it improved or harmed the solution in overall, furthermore this also limits the size of population, which makes mutations suboptimal.

If I were to start this same project again, I would focus on optimizations more from the begining, such as keeping only a single network initialized at the time and extend the experiments early to utilize Google TPU devices, due to their better scalability and higher performance.

5.3 Final Evaluation Results

The best-performing individual in the final experiment were in generation 54, individual 25. This network achieved 94.31% accuracy on the MNIST dataset using 314,806 parameters. With additional layers types or different stride and padding settings, this accuracy could be increased further. Below can see the architecture of the network, most of the parameters are used by the dense layer after flattening the output of the CNN layers.

Listing 1: Architecture of best-performing individual

```
----------------------------------------------------------------
       Layer (type)               Output Shape       Param #
================================================================
           Conv2d-1          [-1, 2, 28, 28]              20
             ReLU-2          [-1, 2, 28, 28]               0
           Conv2d-3          [-1, 4, 28, 28]              76
             ReLU-4          [-1, 4, 28, 28]               0
          Flatten-5               [-1, 3136]               0
           Linear-6                [-1, 100]         313,700
             ReLU-7                [-1, 100]               0
           Linear-8                 [-1, 10]           1,010
          Softmax-9                 [-1, 10]               0
================================================================
Total params: 314,806
Trainable params: 314,806
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.10
Params size (MB): 1.20
Estimated Total Size (MB): 1.30
----------------------------------------------------------------
```

The Genetic Algorithm also showed a good general tendency. As we can see on the graph below (See Figure 2), the mean accuracy showed a good increasing tendency in the first 10 generations, then it reaches an average of 70%, with sporadic decreases due to evolutionary randomness. The maximum accuracy hovers in the 90s as well after 10 generations and stays steady. The mean

accuracy could be tuned further with more experimentations with the fitness function and with the introduction of a more dynamic mutation algorithm. Furthermore, the complexity of the network (See Figure 1), after a steep initial decline stabilises between 150.000 and 300.000 parameters.

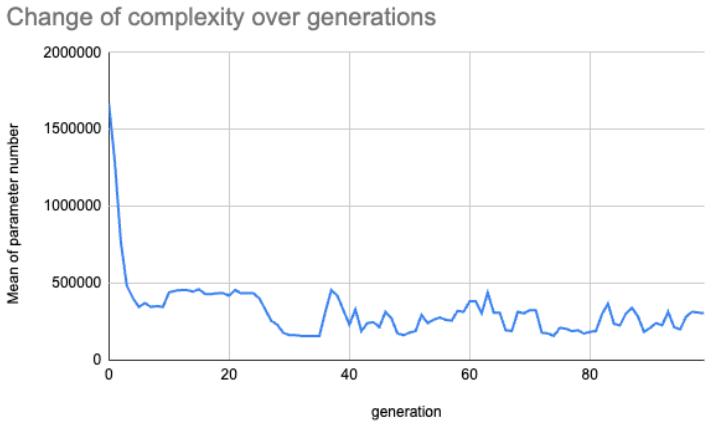

Figure 1: Average number of parameters

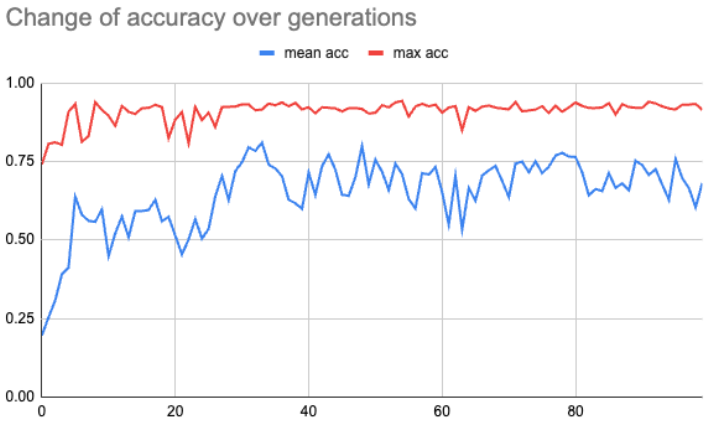

Figure 2: Accuracy of the networks

As we can see on the table below, the result is not far from the manual fine-tuned architectures, which shows that using genetic algorithms have a potential in every day use, where the resulting model or data size is not too large. With some further optimization, the accuracy could be increased higher and also the performance could be optimized for faster training.

Name of network	Accuracy	Parameters
SpinalNet(VGG-5)[5]	99.69%	3.646M
ResNet-9[4]	99.679%	unknown
ConstructNet	94.31%	364.000

6 Conclusion

The project successfully created a genetic algorithm that can evolve the architectures of neural networks for more complex problems than the XOR problem or other toy datasets. Furthermore, as of now, there are no research done on using Convolutional neural networks, the available research papers are only use dense layers or custom neurons.

The project intorduced the ConstructNet class, that can materialize the neural network from a DNA using Conv2D and Dense layers with RELU activation layers, providing a better possibility for further research. Furtheremore, the increase of computational capacity, allowed to revisit abandoned research area, as the genetic algorithms required too much computation, limiting the application to the XOR and other tiny examples. With the technological advancement the project was able to push out this limit and create networks for a more complex problem.

6.1 Future Work

Due to the time constrains during the project, the scope had to be limited, however the results are promissing and showing further development opportunities. First, the mutation algorithm could be reworked and researched further to possibly implement crossover, different types of mutations and a dynamically changing mutation possibility. Furthermore, the implementation of different layers and more option in the existing layers would allow the algorithm to find more precise solutions and tailor the network to the task better.

After the closure of the project I plan to continue experimenting and implement the possible improvements, as this research area could be revisited now with the new possibilities of modern hardware.

References

[1] Abdi, H. A neural network primer. *Journal of Biological Systems 2*, 03 (1994), 247–281.

[2] Arifovic, J., and Gencay, R. Using genetic algorithms to select architecture of a feedforward artificial neural network. *Physica A: Statistical mechanics and its applications 289*, 3-4 (2001), 574–594.

[3] Fisher, R. A. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[4] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[5] Kabir, H. M. D., Abdar, M., Khosravi, A., Jalali, S. M. J., Atiya, A. F., Nahavandi, S., and Srinivasan, D. Spinalnet: Deep neural network with gradual input. *IEEE Transactions on Artificial Intelligence 4*, 5 (2023), 1165–1177.

[6] LeCun, Y., Cortes, C., and Burges, C. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist 2* (2010).

[7] Luo, X., Oyedele, L. O., Ajayi, A. O., Akinade, O. O., Delgado, J. M. D., Owolabi, H. A., and Ahmed, A. Genetic algorithm-determined deep feedforward neural network architecture for predicting electricity consumption in real buildings. *Energy and AI 2* (2020), 100015.

[8] Miller, G. F., Todd, P. M., and Hegde, S. U. Designing neural networks using genetic algorithms. In *ICGA* (1989), vol. 89, pp. 379–384.

[9] Montana, D. J., Davis, L., et al. Training feedforward neural networks using genetic algorithms. In *IJCAI* (1989), vol. 89, pp. 762–767.

[10] Swanson, N. R., and White, H. A model-selection approach to assessing the information in the term structure using linear models and artificial neural networks. *Journal of Business & Economic Statistics 13*, 3 (1995), 265–275.

[11] Whitley, D. A genetic algorithm tutorial. *Statistics and computing 4* (1994), 65–85.