Evolving Neural Networks:

Using an evolutionary algorithm to create multi-class classification networks

Author:
Friso Verweij

Supervisor and first assessor:
Dr. G.A.W. Vreeswijk

Second assessor:
Dr. D. Nguyen

Bachelor Artificial Intelligence

June 26, 2020
7.5 ECTS

Abstract:
Designing the topology of a neural network can be a difficult task, as there is no reliable method of producing such a network. Studies have shown that the topology and connection weights of a neural network can be obtained through evolutionary algorithms. However, research has focused primarily on networks that generate real-valued output. In this thesis, we ask the question if evolutionary algorithms can be used to produce neural networks suited for multi-class classification problems. This is non-trivial, as these kinds of networks generally feature more nodes and therefore are more complex and harder to generate. We present an algorithm that generates and adapts such neural networks using evolutionary algorithms. Results from testing the algorithm on four different datasets show that the algorithm is capable of producing networks for relatively simple datasets. The algorithm was, however, unable to produce useful networks for the most complex dataset.

Keywords: Artificial Intelligence; Neural networks; multi-class classification; evolutionary algorithms; neuroevolution; network topology

Utrecht University

Faculty of Humanities
Utrecht University
Netherlands

Acknowledgements

I would like to take this opportunity to thank my supervisor Gerard Vreeswijk. Because of the many helpful comments, feedback and tips he provided, I was able to produce this thesis.

I would also like to thank the UCI Machine Learning Repository for providing useful datasets, which were used to test the performance of the algorithm presented in this thesis.

Contents

1 Introduction

Artificial Intelligence is an important part of our modern world. From recommendation systems to face detection, AI might not always be obviously present, but is mixed in our everyday life. One incredibly interesting subset of AI is the field of *deep learning*, where we use (artificial) neural networks to find answers to questions after training them on a lot of data. However, even though neural networks are part of a well-researched area, creating a neural network for a complex classification task can be a challenging objective.

1.1 The design problem

When we want to train a neural network with one of the well-known learning algorithms (for example *gradient descent*), we first have to design the topology of the network. This means that the number of layers, number of nodes within each layer and the way these nodes are connected to each other must be decided beforehand. For simple networks, this process might not be that difficult. However, when working with a complex set of data, this can become a very challenging task, as there is no reliable method of creating such topology. This process is therefore often accompanied by trial-and-error and as a result could lead to inefficient or suboptimal networks. Besides this initial design problem, existing neural network topologies can often not be reused for other datasets, meaning that new problems require new topologies. An algorithm that could reliably generate well-performing neural networks on its own would therefore be very useful.

One possible solution to this problem is the incorporation of *evolutionary algorithms*. These stochastic optimisation algorithms apply the principle of natural selection to a population of individuals and feature a set of operations to alter these individuals to better-performing versions.

One paper that shows that the combination of neural networks and evolutionary algorithms can work well is written by Miller et al. in 1989. They have shown that given a neural network, the way the nodes are connected with each other can be altered using evolutionary algorithms to increase the network's performance. They used mutations like *crossover* to alter which node was connected to which and trained the adapted network in the usual way. Their research also shows that the traditional, hierarchical way the nodes are connected, where each node in hidden layer x is connected with each node in layers x - *1* and x + *1*, is not necessary. This means that there might be less intuitive, but more efficient networks possible if any node could be connected with any other node.

Other research carried out by Montana and Davis in 1989 has shown that a different usage of evolutionary algorithms is possible. They showed numerous mutations that can be used to train an already existing network topology. Their results show that the usage of evolutionary algorithms can be an interesting alternative to the conventional learning algorithms like gradient descent, as these algorithms have the tendency to get stuck in local minima. As evolutionary algorithms are fundamentally stochastic, it gives rise to the opportunity to jump out of local minima.

These papers showed that both the creation of network topologies and the training process can be combined with evolutionary algorithms. Stanley and Miikkulainen (2002) created an algorithm, called *NEAT*, that is able to create complete neural networks on its own. Their algorithm proved capable of solving complex problems, without having to determine the network's topology themselves.

Research has mostly focused on neural networks that generate real-valued output and therefore the algorithms created are specialised in generating this kind of output. However, *multi-class classification* is another incredibly interesting use of neural networks. With multi-class classification, we do not want our answer to be continuous value, but one of multiple classes. Think for example about diagnosing a patient with a disease based on symptoms or identifying plants based on their properties. This approach requires the network to be different, for example by using different transfer functions and by the additional output nodes, adding to the complexity of the network.

1.2 Research question

In this thesis, we focus on finding an evolutionary algorithm that can generate multi-class neural networks and we will determine if the produced networks can be considered useful. In this context, we will call a neural network useful if it is able to fulfil its desired purpose. A network that performs slightly better than an algorithm that purely guesses, would for example not be classified as useful. The question we will try to answer is the following: Can an evolutionary algorithm be used to produce neural networks for multi-class classification?

1.3 Thesis overview

After this introduction, a brief overview of neuroevolution is presented in **chapter 2**, as well as an explanation about how neural networks for multi-class classification are different from other neural networks. **Chapter 3** presents the algorithm itself. Here, the representation of the individuals is described, as well as how the initial population is created and the set of operations it uses to adapt the networks. The way the algorithm will be tested is described in **chapter 4**. Here, the used datasets are explained and the chosen parameters for the algorithm are presented. **Chapter 5** shows the gathered results for each dataset, followed by the conclusion in **chapter 6**. **Chapter 7** presents related work in the field of neuroevolution and how our algorithm differs and shares features with other algorithms. The discussion can be found in **chapter 8**, where we will go over the limitations of this thesis and suggest future research. A guide on how to install and run the algorithm is presented in the appendix in **chapter 10**.

2 Background

2.1 Neuroevolution

Neuroevolution is the field in artificial intelligence that combines the creation or modification of neural networks with evolutionary algorithms. Many algorithms have been created to automatically generate and evolve neural networks, like NEAT, HyperNEAT (Stanley et al., 2009) and GNARL (Angeline et al., 1994). Each algorithm differs in its implementation, but all share a common structure.

Each algorithm specifies a representation to describe the networks. In *genetic algorithms*, individuals are described using 'genes'. These genes represent certain properties of the network, for example a connection or connection weight. Genetic algorithms often feature crossover, where a new individual is created by combining the genetic code of two other individuals. In contrast, in *evolutionary programming*, individuals are not constrained to be represented by genes. These algorithms do not use crossover and instead focus on mutations, where individual elements of the networks are altered. Besides the differences in representation, each algorithm has a way of creating an initial population, selection method and set of operations to change the networks.

Algorithms using neuroevolution have successfully generated neural networks that perform well on difficult problems, like the double pole balancing problem, where two poles are attached to a movable cart. The network is tasked with applying forces to keep the poles balanced for as long as possible. This is a very difficult task, as the poles differ in length and will both react differently to the applied forces.

Additionally, networks created using this technique have the potential to generalise better than conventionally designed networks. As Todd et al. showed, a specific layered topology is not necessary. When connections are allowed to skip layers, much smaller networks can be created. As smaller networks are less complex, they tend to generalise better.

2.2 Multi-class classification

Research in the field of neuroevolution has focused primarily on neural networks that generate real-valued output. Neural networks for multi-class classification differ from these kinds of networks in the number of output nodes and the way the final output is determined. Instead of a single output node, there are as many output nodes as there are possible classes. Each output node belongs to a specific class. When all output values are calculated, the class associated with the node with the highest output is chosen as the final answer.

According to Ou and Murphey (2007), two-class classification networks are well-understood. The extension from two-class to multi-class, however, often leads to increased complexity and decreased performance. A common way to create a multi-class classification network is by creating multiple two-class classification networks and combining these. In this thesis, however, we will focus on creating complete multi-class networks.

3 The evolutionary algorithm

3.1 Network representation and node structure

As with any evolutionary algorithm, the chosen representation of an individual is very important, because it determines how the individual can evolve. The representation this algorithm uses to describe a neural network is very similar to the Cellular Encoding used by Gruau (1994). Each node of the network is a separate object with references to incoming and outgoing connections. For each incoming connection, there is a reference to a weight value for this connection. This representation allows us to easily alter a network with the mutations discussed later in this chapter. Each node has a variable indicating the layer the node is in. Even though this network does not have a specific layer structure, as connections between all nodes are allowed (except between input nodes and between output nodes), we have to make sure no loops can exist in our network. Loops can occur when a node in a higher layer becomes an incoming connection for a node in a lower layer. To prevent this from happening, each node has a variable called *nodeClass*, indicating the depth of the node in the network. When creating new connections, this variable will be checked to prevent loops from being created.

In conventional neural networks, each node has a connection to one additional node called the 'bias' or 'threshold'. This node will always output a value of 1 and together with a weight value, directly influences the receiving node. In the representation used in this algorithm, these bias nodes are not necessary, as each node can contain one additional variable containing the weight of the bias. This simplifies the network immensely, as fewer nodes and connections are required.

3.2 Calculation

Given a certain node in our network, we can calculate the output of this node. First, the signal s is calculated, by taking the sum of the multiplication of the output of each incoming node (including the bias) and its corresponding weight, as shown in equation 1. Here, A is the set of all incoming nodes and b is the node we wish to calculate the output of.

$$s = \sum_{a \in A} output(a) \cdot weight_{a \to b} \tag{1}$$

The signal is then passed over to the hyperbolic tangent function (tanh), as shown by equation 2. The hyperbolic tangent returns a value closer to 1 if the input becomes larger and a value closer to -1 if the input becomes smaller, with a smooth transition through the origin. This function is very useful in multi-class classification, as the output of a node could be seen as how much the node is 'activated'. When selecting the final output of the network, we choose the

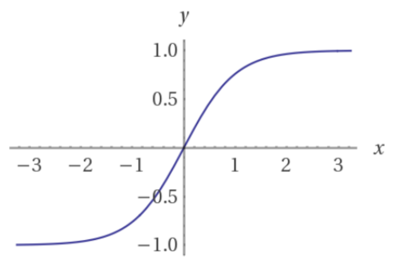

Figure 1: The hyperbolic tangent plotted

output node with the highest activation.

$$output(b) = \frac{e^s - e^{-s}}{e^s + e^{-s}} \tag{2}$$

When presented with a data point, the input values of the data point are loaded into the input nodes. After that, the output of each node is calculated recursively, starting at the output layer. Each node uses the output of the incoming nodes, which will in its turn use the output if its incoming nodes. When a node reaches an input node, the value loaded in from the data point will be the output of the input node.

3.3 Initialisation

Because we want our algorithm to work with datasets of different complexities, we must first analyse the dataset. While parsing, the number of input values, as well as all possible classes are noted. When constructing the initial network, the number of input nodes must be equal to the number of input values and the number of output nodes must be equal to the number of possible classes. This way, different datasets will produce different initial networks.

After all nodes are created, each input node will be connected to each output node. Each connection weight and the bias will start with a value of 0, meaning that every input does not contribute anything. This produces a very minimalistic network with very little complexity, but provides the opportunity to grow more complex. The initial population is then filled with copies of the initial network.

3.4 Iteration

An iteration starts with the evaluation of each individual network in the population. Using the train set, we calculate the output of the network for each data point and compare this to the given right answer. The percentage of correct answers is the *score* of the individual. To make sure the individuals do not grow overly complex, there is a very small penalty for each node and connection, as

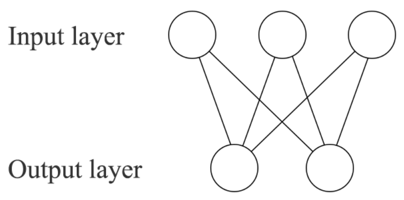

Figure 2: The initial network for a dataset with 3 input values and 2 possible classes

shown in equation 3.

$$Penalty = 0.03 \cdot NodeCount + 0.005 \cdot ConnectionCount \tag{3}$$

As described by Ceollo (2002), the usage of penalty functions is very common in the field of evolutionary algorithms. A correctly constructed penalty function can help find an optimal solution given certain constraints. In our case, we have a soft constraint, being that we do not want our networks to grow unnecessarily complex.

The penalty is subtracted from the score, resulting in the *fitness* of the individual. Using the penalty function, the less complex individuals are preferred over much more complex, but slightly higher scoring individuals. The specific values for the penalty were determined after extensive testing and result in less complex networks, while barely or not affecting the final score. As the calculation of the fitness of the individuals is completely independent of each other, it is trivial to multi-thread the code.

After the fitness of each individual is calculated, the individuals are sorted based on their fitness. Individuals with a higher fitness have a greater chance of being picked to be mutated and advance to the next population.

Zitzler et al. (2000) showed that using an elitist approach in a multi-objective evolutionary algorithm significantly increases the performance of the algorithm. We will therefore include this elitist approach in our algorithm as well. A certain number of best-performing networks will automatically advance to the next generation without a mutation applied to them. This will also ensure that the best-performing network of a given generation *x* is never worse than the best-performing network in generation *x - 1*.

3.5 Mutation

When a certain individual is picked to advance to the next population, it is first mutated. There are five possible mutations, namely changing the weight of a connection, adding a new connection, removing a connection, adding a node and removing a node. Each mutation has a set chance of being chosen to alter the individual. These five mutations are a minimal set of changes that allow any network to transform in any other network given a certain number of mutations.

The mutations have very distinct purposes. The mutations that change the network topology have the purpose of increasing and decreasing the network complexity. These mutations are not designed to change the network's performance. The mutation that changes the weight of a connection is made for that exact purpose, to exploit the current complexity and change the behaviour of the network. Even though it is not always possible to not change the behaviour of the network when changing its topology, we will attempt to minimise the effect.

3.5.1 Changing weights of connections

To change the weight of a connection, a random node that has at least one incoming connection is selected. Following the selection of the node, a random incoming connection or the bias weight is chosen. A random, but uniformly distributed, value between -10.0 and 10.0 is added to the weight of the chosen connection. This specific range is chosen after extensive testing, providing a good balance between not changing behaviour enough and changing behaviour too much to the extent that fewer improvements are found each iteration.

3.5.2 Adding a connection

To add a new connection, two random nodes are selected with three conditions: it is not allowed that both nodes are in the input layer, both nodes are in the output layer and that both nodes are the same node. Using the variable *nodeClass*, it is determined which node is placed deeper in the network. The deeper node will receive data from the shallower node. A connection between the nodes is made with a weight value between -3.0 and 3.0. We allow the new connection to start with a random weight, as setting its initial weight to 0 would require one additional mutation for it to take effect. The range is chosen in a way that a new connection does have a slight effect (so the weight should not be 0), but does not drastically change the network's behaviour.

3.5.3 Removing a connection

To remove a connection, a random node in the network is selected that has at least one incoming connection. After that, a random incoming connection is chosen. The connection is removed, as well as its associated weight.

3.5.4 Adding a node

To increase the complexity of the network, we can add a new node. The new node will be placed in between an existing connection. To make sure the network can grow, but not drastically change behaviour, the weights of the new connections are calculated in a way that minimises any effect on the behaviour of the network.

To do so, we first select a random node c from the network that has at least one incoming connection. After that, a random incoming connection to node a

is selected. This is the connection we will substitute. A new node b is created, as well as two new connections connecting a and b, and b and c. The weight of the connection between a and b will be chosen randomly. The weight of the connection between b and c, however, must be calculated in a way that will result in no changing input for node c. We can calculate this by dividing the old signal contribution from a to c by the new output b generates, as shown by equation 4.

$$newWeight_{b\to c} = \frac{output(a) \cdot weight_{a\to c}}{output(b)} \tag{4}$$

After the new weights are determined and applied, the old connection is removed.

3.5.5 Removing a node

When removing a node, we want to decrease the complexity of the network, while minimising the change of its behaviour. To do so, a random node b that is not an input node nor an output node is selected. This is the node that will be removed.

Let A be the set of all input nodes of b and let C be the set of nodes that receive input from b. Each shallower node $a \in A$ will have a new connection with every deeper node $c \in C$. Each deeper node c has a certain expected signal that it receives from b, which we should not change if we want the behaviour of the network to remain the same. The sum of all values contributed by the new connections should be the same as the value contributed by the old node. We can therefore divide the expected signal contribution from node b to a node c by the sum of all outputs from less deep nodes *a'*, as shown in equation 5.

$$newWeight_{a\to c} = \frac{output(b) \cdot weight_{b\to c}}{\sum_{a'\in A} output(a')} \tag{5}$$

The result is the new weight for all new connections that are connected to a certain node c. If, however, there already exists a connection between a and c, the new calculated weight will be added to the already existing connection weight. After all calculations are done, the node is removed. This will result in no or minimal change of behaviour, while decreasing the complexity of a network.

4 Method

To test the performance of the algorithm, the algorithm will be trained and tested on four different datasets. During the training phase, data is collected for each iteration. This data includes the iteration number, the current highest score of the population and the current average score of the population. After the training phase, a summary is produced, which shows the performance of the best-performing network on the test set.

4.1 Datasets

This section presents four different datasets, each representing different properties of possible datasets.

4.1.1 The XOR dataset

The XOR dataset is a dataset often used to represent simple problems that can be solved using neural networks. Because the data is not linearly separable, the network requires hidden nodes to solve the problem. The input consists of two real values, x and y. The output is either *true* or *false*.
This toy dataset has been created using equation 6 and 7:

$$0.95x - 2y \tag{6}$$

$$2.5x + 0.7y \tag{7}$$

If exactly one of these equations return a positive value, the answer is *true* and *false* otherwise.

Using these two equations, a train set and test set, both consisting of 10000 data points, will be generated.

4.1.2 The disease dataset

The disease dataset represents are more complex problem, as it has more input values and more possible output values. The input is made out of ten binary values representing symptoms of a disease. A value of *1* means that the patient has the symptom, while a value of *0* means the symptom is not present. Based on the symptoms, the network must determine which of five diseases the patient has or if the patient is healthy.

If a person is healthy, there are certain probabilities for each symptom. Each disease increases the probability for certain symptoms. Since every condition has probabilities for the same symptoms, it is possible that a person has all the signs pointing towards disease A, while the person actually has disease B. A score of 100% is therefore highly unlikely, even for a human expert.

The probabilities of the symptoms for each condition are described in table 1. These probabilities are not based on real data. To generate this dataset, first a random condition is chosen. For each symptom, a new random value is generated and it is determined if the patient has the symptom. This process will repeat until the train set and the test set both contain 10000 data points.

4.1.3 The Glass Identification dataset

The Glass Identification dataset is a dataset provided by the UCI Machine Learning Repository and is originally created by German in 1987. The objective of this dataset is to train networks to identify a type of glass, based on its chemical composition. Each data point has 9 input values and one of 7 classes.

Table 1: Probabilities of the symptoms for each condition. Note that no disease decreases any probability of a symptom.

Condition	Coughing	Sneezing	Headache	Loss of taste and smell	Vomiting	Fever	Nausea	Fatigue	Hair loss	Itching
Healthy	0.04	0.04	0.04	0.01	0.01	0	0.06	0.20	0	0.02
Disease A	1	0.08	0.4	0.01	0.01	0	0.06	0.20	0	0.02
Disease B	0.04	0.04	0.8	0.3	0.5	0.9	0.8	0.20	0	0.02
Disease C	0.04	0.04	0.3	0.5	0.01	0	0.06	0.9	0	0.02
Disease D	0.04	0.04	0.04	0.01	0.4	0.8	0.8	0.20	0.3	0.02
Disease E	0.6	0.7	0.04	0.01	0.01	0	0.06	0.20	0.6	0.9

This dataset is very small, only consisting of 214 data points. It is therefore used to test how the algorithm performs on very small datasets. Because there is a small number of data points, all data points will be used for training and none for testing. The results will, however, be compared to previously mentioned research performed by G. Ou and Y. Murphey, where they compared the performance of different multi-class classification techniques.

4.1.4 The Letter Recognition dataset

The Letter Recognition dataset is another dataset provided by the UCI Machine Learning repository and is created by Frey and Slate in 1991. This dataset can be used to train networks to identify letters of the English alphabet. The letters images were based on letters in 20 different fonts and were heavily distorted. Of the final images, 16 features were extracted, which form the input values of the data points. The dataset consists of 20000 data points, which were for this thesis split into a train set of 10000 data points and a test set of the same size.

This dataset is very complex compared to the other datasets in this thesis and will be used to test the algorithm on more complex problems. The results will we compared to the results found by G. Ou and Y. Murphey, who also used on this dataset.

4.2 Runs and variables

As evolutionary algorithms are fundamentally stochastic, it is important to train on each dataset multiple times. Therefore, each dataset will be trained on for ten times. Additionally, because the Letter Recognition dataset contains many input and output values, the algorithm will train once more on this dataset with a larger population and many more iterations. The exact variables are described in tables 2 and 3.

Table 2: Standard variables used in testing

Standard run	
Runs	10
Iterations	1000
Population size	100
Number of elite networks	10
Probability of weight mutation	0.8
Probability of adding a connection	0.06
Probability of removing a connection	0.06
Probability of adding a node	0.06
Probability of removing a node	0.02

Table 3: Variables for the additional run on the Letter Recognition dataset

Additional run	
Runs	1
Iterations	3000
Population size	1000
Number of elite networks	100
Probability of weight mutation	0.8
Probability of adding a connection	0.06
Probability of removing a connection	0.06
Probability of adding a node	0.06
Probability of removing a node	0.02

5 Results

5.1 The XOR dataset

After the algorithm trained on the XOR train set, the average best score of each run on the test set was 98.384 and the highest overall test score was 99.85, meaning the algorithm gave wrong answers to 15 out of 10000 questions. Even though a score of 100 was possible, it appears that the algorithm is capable of delivering a suitable network for this task.

The average best score of each run on the train set was 98.714 and is as expected higher than the average best score on the test set. Because of this small difference, the networks seem to generalise well.

As can be seen in figure 3, the best score quickly rises, after which progress slows down.

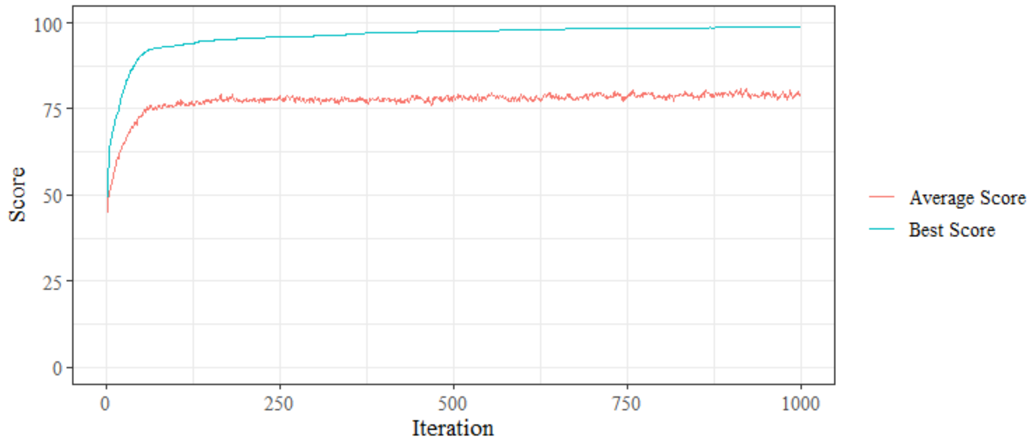

Figure 3: The average of both the average and best score of all runs plotted for each iteration during the training phase on the XOR dataset

5.2 The disease dataset

In a more realistic scenario, a problem is more complex than the XOR problem and can feature some uncertainty. After training on the disease train set, the average best score and highest overall score on the test set were 88.169 and 88.4 respectively. As this dataset is just a toy dataset and not based on actual medical data, we do not know the performance of a human expert and therefore we cannot compare our results to it. The results, however, do appear to suggest that the algorithm is sufficiently capable to generate an appropriate neural network for this kind of dataset.

The generated networks also seem to generalise well, as the average best score of the networks on the train set was 89.092.

Similar to the XOR dataset, the performance of the networks quickly rises initially and slows down afterwards as shown in figure 4.

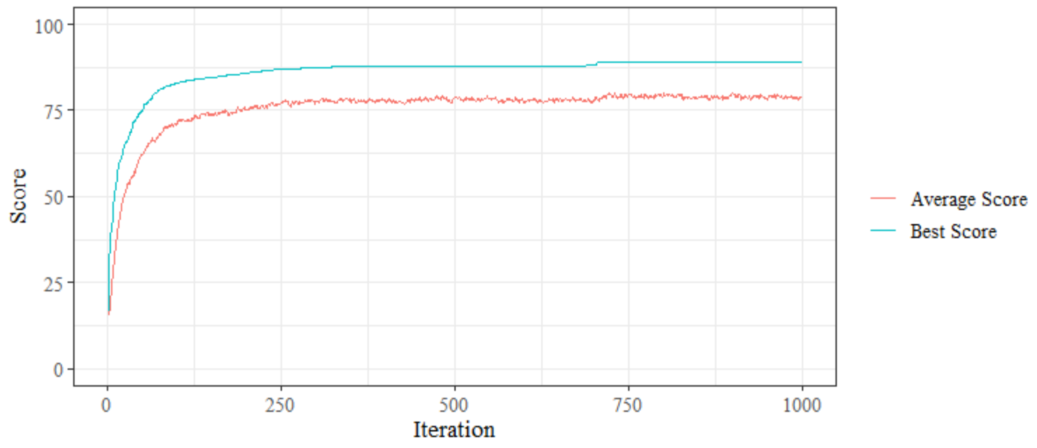

Figure 4: The average of both the average and best score of all runs plotted for each iteration during the training phase on the disease dataset

5.3 The Glass Identification dataset

The average best scores of the networks the algorithm produced on the Glass Identification dataset were much lower than the scores of the networks on the previous datasets. As this dataset is very small and does not feature a separate test set, we can only test the produced networks on the train set itself. The average best score and the highest overall score on the train set were 70.841 and 73.364 respectively.

We can, however, compare the performance on the train set to the results of Ou and Murphey. Their most similar network, of which topology was determined in the conventional 'trial-and-error' way, classified 70.56% of data points correctly, resulting in a score of 70.56. Even though our algorithm reached a higher score, a simple two-tailed t-test with a significance level of 0.05 shows that the results are not significantly different. Our algorithm therefore produced a network that seems to perform on the same level as the network produced by Ou and Murphey, without designing the network topology.

5.4 The Letter Recognition dataset

As the Letter Recognition dataset is the most complex dataset used in this thesis, the algorithm was expected to struggle most with this dataset. After training for 1000 iterations, an average best score and a highest overall score of the produced networks on the test set were 29.162 and 32.16 respectively. These results are disappointing, considering the network trained by Ou and Murphey

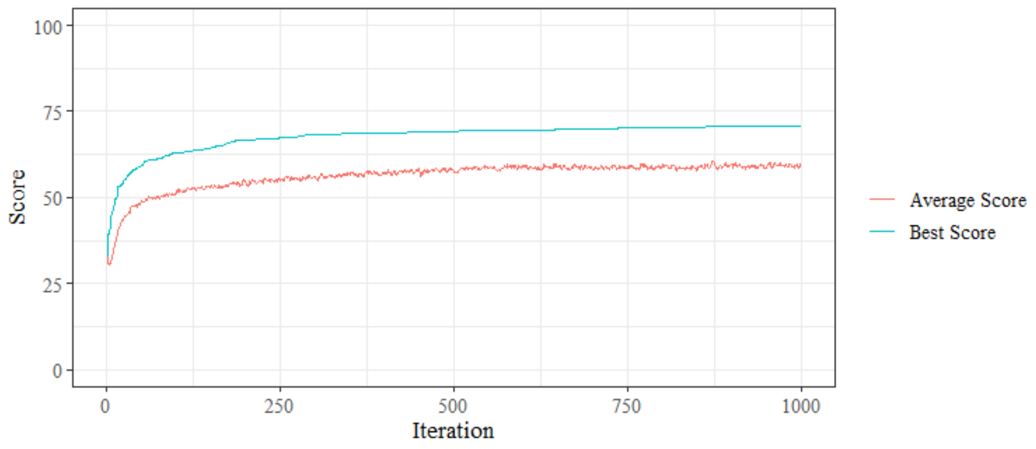

Figure 5: The average of both the average and best score of all runs plotted for each iteration during the training phase on the Glass Identification dataset

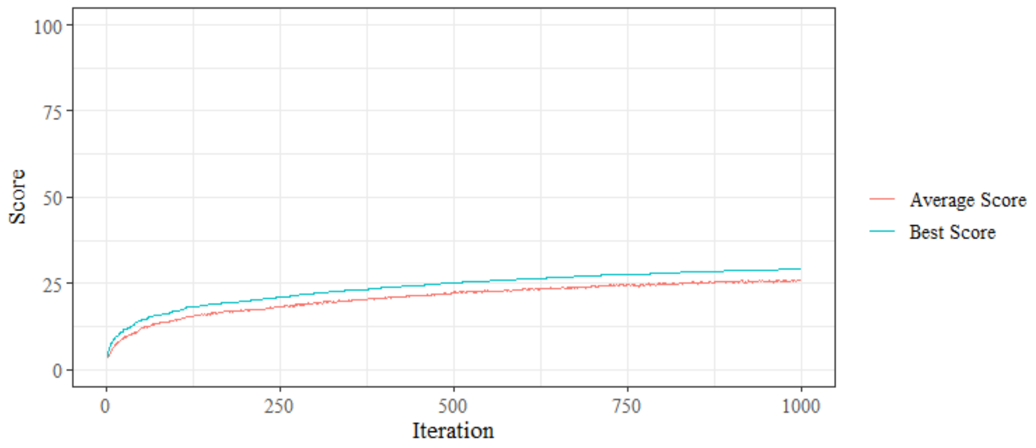

Figure 6: The average of both the average and best score of all runs plotted for each iteration during the training phase on the Letter Recognition dataset

classified 87.38% of data points correctly. Surprisingly, the average best score of the produced networks on the train set was lower than the scores on the test set, with a score of 29.157.

However, as this dataset features many input values and even more possible output values, the algorithm might benefit from more iterations and a larger population size. Figure 6 suggests this as well, as in contrast to figure 3, 4 and 5, the performance of the networks do not seem to plateau as much. After running the additional run consisting of 3000 iterations and a population size of 1000, the produced network reached a score of 34.72 on the test set and a score of

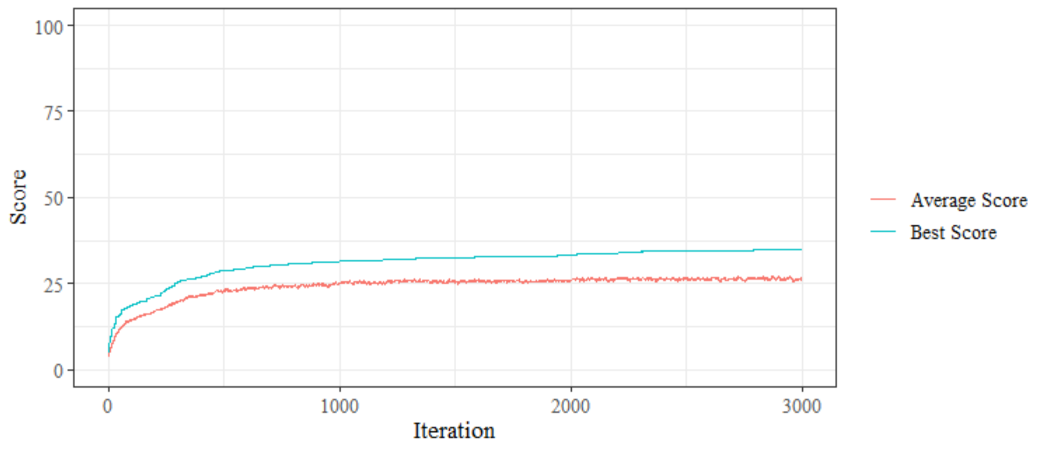

Figure 7: The average and best score plotted for each iteration during the training phase of the additional run on the Letter Recognition dataset.

34.72 on the train set, which are coincidentally the same. Figure 7 shows that the performance does plateau this time. The additional run suggests that the algorithm does benefit from a longer training time on complex datasets, but it still does not reach the same level of performance as the conventionally created neural network.

6 Conclusion

In this thesis, we asked the question whether evolutionary algorithms could be used to construct useful multi-class neural networks, without having to design the topology of the network ourselves. To answer this question, an evolutionary algorithm was constructed, which can generate and adapt neural networks, given a multi-class dataset. The algorithm was tested on four different datasets, of which two were created specifically for this thesis and two were created by other researchers.

The first results concerning the XOR dataset and the disease dataset were promising. The algorithm was capable of generating neural networks that could classify a high percentage of data points correctly, with top-performing networks reaching scores of 99.85 for the XOR dataset and 88.4 for the disease dataset.

The results concerning the Glass Identification dataset were promising as well. Even though the algorithm was not able to produce a network that could reach scores as high as the networks trained on the previous datasets, the produced networks did seem to perform just as well as neural networks of which the topology was constructed in the conventional way.

However, the results concerning the Letter Recognition dataset were disappointing. With top-performing networks reaching scores of 32,16 for the standard runs and a score of 34.72 for the additional run, the performance was much worse than the manually created topologies, which were able to reach a score of 87.38.

The algorithm therefore appears to be able to construct simple neural networks well. However, the algorithm does appear to be incapable of constructing useful networks for more complex datasets. To answer the research question explicitly, the algorithm appears to be able to produce useful multi-class neural networks on simple datasets, but it seems unable to produce useful networks on complex datasets.

This algorithm could therefore be used to create multi-class classification networks for, for example, people that would like to use a relatively simple multi-class neural network, but that do not have the knowledge to construct such network themselves. If there are many different neural networks needed, this algorithm can also help reduce the amount of labour needed. Generating a neural network may become trivial, as no effort is needed to create one, apart from gathering data. However, if a very complex network is required, this algorithm does not seem suitable.

7 Related work

7.1 NEAT

NEAT (NeuroEvolution through Augmenting Topologies) is a widely used genetic algorithm that can construct and adapt both network weights and topologies, created by Stanley and Miikkulainen in 2002. It uses a direct encoding, where each gene specifies a connection between two nodes, along with its weight. NEAT can adapt its networks using crossover, by changing a connection weight, adding a connection and adding a new node. This allows the networks to grow in complexity. However, decreasing the complexity is much more difficult, as this can only happen using crossover. Besides this, if there is a population of networks that share a specific subnetwork, it is impossible to reach a network smaller than this subnetwork. This means that given a certain network, not every other network might be able to be reached given a certain number of generations. NEAT therefore has a smaller search space than our algorithm, but the optimal solution might get excluded from the search space in this way. By providing our algorithm the mutations necessary to shrink in complexity, the exclusion of the optimal solution from the search space is prevented from happening.

As crossover as a means of reproduction frequently leads to networks that are infeasible, our algorithm does not use crossover. It instead uses a minimal set of basic mutations needed to alter the networks, which cannot create infeasible solutions. However, the use of crossover is an important aspect of NEAT. The solution Stanley and Miikkulainen came up with was the use of a *global innovation number*. This value is unique for every applied mutation and gets stored in the gene that was created by the mutation. If two individuals share some global innovation numbers, the associated genes can be lined up. When crossing over, these matching genes will be swapped randomly. When genes do not match, however, the gene from the parent with the highest fitness is chosen. This system can prevent infeasible solutions from being generated.

As pointed out by Stanley and Miikkulainen, changes to the topology of the network usually initially decrease the network's performance and therefore new topologies often do not survive long. Their solution is to divide the population into groups of similar networks, called species. The reproduction phase is then applied to each species separately, instead of to the entire population as a whole. This way, when a new network is created that is significantly different from existing networks, the network will form its own species. This method, however, does add more variables that have to be tuned and is quite complex, as they need to determine which species an individual belongs to. Our algorithm, on the other hand, minimises the change in behaviour that structural changes cause by adjusting the connection weights. Dividing the population into smaller groups is therefore not needed to protect new topologies.

7.2 Cellular Encoding

The encoding used in the presented algorithm is very similar to that of the Cellular Encoding, which was first presented by Gruau in 1994 and was extended and refined in the following years. In this encoding, a cell is a node that is part of a graph that represents the neural network. The cells are linked to each other to receive input or send output. Each cell has its own memory, containing for example weight values of the connections. This encoding is very useful, as new nodes and connections can easily be added or removed.

The network grows by means of several unique cell divisions. A mother cell gets replaced by two new child cells. Depending on the specific operation used, a number of properties of the mother cell are passed over to the child cells. One of these properties is for example the way the new cells are connected to the other cells or to each other. Even though the number of different cell divisions has increased in following research (Gruau et al., 1996), the result of all cell divisions is two new cells that are very similar to the mother cell. A single cell division can thereby add many new connections at once, changing the network in large steps. It is possible to remove those connections, but this can take several iterations, which might be enough the remove the network from the population completely. Besides this, there is no option to add a new link on its own.

7.3 GNARL

Our algorithm has many shared features with the algorithm named GNARL (GeNeralized Acquisition of Recurrent Links), created by Angeline et al. in 1994. Like our algorithm, GNARL does not use crossover to modify the network's topology. Rather, it uses mutations to alter the connection weights and to add and remove nodes and connections. Additionally, GNARL strives to change the behaviour as little as possible when mutating the topology of the network. However, as achieving this is not so simple when removing a node, the node is just removed, along with its connections. This differs from our algorithm, as we add and modify connections surrounding the removed node to minimise the change in behaviour of the network.

The initial population that GNARL creates consists of randomly generated networks. These networks share the same number of input and output nodes, but differ in the number of hidden nodes and how these nodes are connected. The number of hidden nodes and connections (as well as a number of other important parameters) are based on a value given by the user. This means that the user is at least partly responsible for the topology of the network and it might require trial-and-error to find a useful initial population, which is something we aim to remove. Additionally, when the user finds a working solution given certain input, chances are that the given input results in unnecessary complex networks. To avoid this, our algorithm starts with a population of very basic networks and allows it to grow more complex on its own.

8 Discussion

8.1 Limitations

In this thesis, the algorithm was tested using the same parameters for every run, apart from the additional run on the Letter Recognition dataset. The probabilities for each mutation did not vary. Because of this, we cannot conclude that, for example, no complex network can be found, as different mutation probabilities could produce very different results.

Besides this, only four datasets were used to test the performance of the algorithm, of which two were created specifically for this thesis. Each dataset represented a certain kind of problem (for example simple, uncertain, complex etc.). However, to better generalise the performance of the algorithm for each kind of problem, the algorithm should be tested on more datasets.

While constructing the algorithm, it was tested on the XOR dataset. However, this dataset was used to determine if the algorithm worked correctly and was not used to test if it produced pleasing results. Additionally, some parameters have been chosen based on testing the algorithm on all datasets. As a result, there might be some bias towards producing an algorithm that worked correctly on these specific datasets.

8.2 Future research

As the algorithm presented in this thesis is quite simple, many potential improvements could be made to increase its performance. For example, the probability of a mutation being selected is fixed in our algorithm. However, increasing or decreasing these probabilities based on the number of completed iterations or current score could be beneficial. Simulated Annealing is one such technique to alter the probability of a certain operation, based on a value referred to as the *temperature*. As the program iterates, the temperature decreases. Initially, the probabilities for changing the topology could be very high, to search widely in the search space. As the temperature decreases, we start to search more narrowly, optimising promising networks that were found.

The topology can be altered using our five mutations. However, the calculations within the nodes are fixed. The transfer function, which is in our case the hyperbolic tangent, cannot be changed. Compositional Pattern-Producing Networks, or CPPNs, (Stanley, 2007) is a type of neural network that allows the transfer function to be mutated. These transfer functions can, for example, be Gaussian functions, sine functions or linear functions. As these networks can therefore be seen as a combination of different functions, they are capable of creating patterns. Our algorithm might improve if the algorithm was allowed to mutate the transfer function along with the topology and weights of the network.

As the results showed that the algorithm had more difficulty creating useful networks for complex datasets, using a different method to produce an initial population might prove beneficial. Instead of starting with a population of very basic networks, the algorithm could start with a population of more complex individuals.

9 References

Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks, 5*(1), 54-65.

Coello, C. A. C. (2002). Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer methods in applied mechanics and engineering, 191*(11-12), 1245-1287.

Frey, P. W., & Slate, D. J. (1991). *Letter Recognition Data Set* [Data file]. Retrieved from: https://archive.ics.uci.edu/ml/datasets/Letter+Recognition

German, B. (1987) *Glass Identification Data Set* [Data file]. Retrieved from: https://archive.ics.uci.edu/ml/datasets/glass+identification

Gruau, F. (1994). *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm.* [Unpublished doctoral dissertation], École Normale Supérieure de Lyon.

Gruau, F., Whitley, D., & Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. *Proceedings of the 1st annual conference on genetic programming* 81-89.

Miller, G. F., Todd, P. M., & Hegde, S. U. (1989). Designing Neural Networks using Genetic Algorithms. *ICGA Vol. 89*, 379-384.

Montana, D. J., & Davis, L. (1989). Training Feedforward Neural Networks Using Genetic Algorithms. *IJCAI Vol. 89*, 762-767.

Ou, G., & Murphey, Y. L. (2007). Multi-class pattern classification using neural networks. *Pattern Recognition, 40*(1), 4-18.

Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines, 8*(2), 131-162.

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life, 15*(2), 185-212.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation, 10*(2), 99-127.

Zitzler, E., Deb, K., & Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation, 8*(2), 173-195.

10 Appendix

10.1 How to install the program

As the program uses Windows Forms to visualise the best-performing network of the population, this program can only be executed on hardware running Windows.

The algorithm and both the XOR dataset and the disease dataset can be downloaded from GitHub using the following url:
github.com/FrisoVerweij/Multi-class-classifying-neural-network-evolving-algorithm
Extract the contents of the zip file to a convenient location. The folder named *Datasets* contains both datasets. The folder *Program* contains the program itself.

10.2 Other datasets

The Glass Identification and Letter Recognition datasets can be downloaded from the UCI Machine Learning Repository. More interesting datasets can be found there. The algorithm does, however, require the dataset to be in a specific format. Each line in the dataset file should contain one data point. A data point is described in the following way:

$$x_1; x_2; x_3; x_4; ...; x_n; y$$

Here, the x values represent the input values, n is equal to the number of input values and y is the corresponding class. The input values must either be binary or continuous values. Depending on the desired dataset, the format might need to be modified.

10.3 How to run the program

To run the program, Microsoft Visual Studio needs to be installed. Start by opening the solution file in the folder named *Program.* Open the file Variables.cs in the solution manager. Change the variables PATHTOTRAINSET, PATHTOTESTSET and PATHTOREPORT with a path to the train set, a path to the test set and a path towards a folder where the collected data will be placed in. To change the probabilities of the mutations, modify the variables ADDNEURON, ADDLINK, REMOVENEURON and REMOVELINK. The probability of the mutation to change a connection weight will automatically be 1 subtracted by the sum of the above-named variables. To modify the number of iterations, the size of the population and the number of best-performing individuals that will advance without mutations, modify the variables ITERATIONCOUNT, POPULATIONSIZE and ELITECOUNT. Depending on the hardware that is used to run the program, the time needed for an iteration might decrease by changing the variable THREADCOUNT, which will determine how many threads will be used to calculate the fitness values of the individuals and to apply the mutations. To change the values of the penalty function, modify the variables

NEURONPENALTY and LINKPENALTY. After all variables are in place, the program can be compiled and run.

When the program has completed all iterations, two files are placed in the folder specified by PATHTOREPORT, containing the data collected for each iteration and a summary. There is an opportunity to pose questions to the best performing network. Simply follow the on-screen instructions. The program can be terminated at any time.