IT-UNIVERSITY OF COPENHAGEN

Master Thesis

Growing Neural Networks

NeuroEvolution Through Neural Cellular Automata

AUTHORED BY:

Lucas H. Petersen .... lupe@itu.dk
Mikkel Ditlevsen .... midi@itu.dk
Oliver E. Astrup .... olas@itu.dk

31st May 2022

KISPECI1SE

Abstract

The design of the topology of artificial neural networks has often been done by human hand. However, with recent progress in the field of neuroevolution, such as deep neuroevolution and NEAT, there now exists algorithms that can reliably evolve the topology of networks, resulting in well performing AIs. In biological systems, neural networks are grown using cell reproduction which happens based on cells' local information. The technique, neuroevolution through neural cellular automata, is a new neuroevolution algorithm that tries to mimic the function of biological cells by making use of neural cellular automatas. The idea behind NETNCA is to apply neuroevolution to neural cellular automatas capable of growing the topology and the parameters of a neural network. This is similar to how genes of cells have evolved over billions of years to be able to produce complex neural networks, such as the brain. NETNCA proves to be able to solve different control and classification problems using non-traditional network topologies. It even suggests the possibility of using the same NCA to generate solutions for similar problems. Although there are still many improvements that need to be made to the NETNCA algorithm, it provides an important first step for the research of using neural cellular automatas to build neural networks.

Contents

1 Introduction

The study of Machine Learning is still a heavily researched topic. It is building on the concept of neural networks inspired by the human brain and how it works by sending signals through nerves in order to send messages around. Many different types of Artificial Neural Networks (ANNs) have come from this like feed-forward neural networks, convolutional neural networks and recurrent neural networks. It has also become well known in recent years that well constructed and trained networks can perform better than humans on discrete topics. An example of this is the AlphaGo algorithm trained on the complex game of Go, capable of beating the world champion in 2016. However, even with these successes, the field of machine learning has not shown any signs of slowing down. New approaches keep being researched and developed. Couple this with the increasing performance of modern hardware, the field is still active and opens up for many possibilities of where to go next.

One element to note when constructing any ANN is that the architecture should often be defined beforehand and there exists many research papers that experiment with and discuss what kind of architectures works the best in which cases (Vrbančič et al., 2018) (Naitzat et al., 2020). The topology is usually designed by humans, though the weights and biases are computer generated through a variety of different techniques like backpropagation and genetic algorithms. While there does exists some general guidelines for designing the topology of neural networks it can often end up feeling more like an art form than a science. As a result, different topologies are often iterated upon multiple times without ever finding a perfect solution that provides the absolute maximum value for the specified problem at hand. Biological neural networks in nature have not been designed by human hand or with any other kind of global control. They are the result of billions of cells working together and have evolved over million of years, to be able to produce the complex systems that biological neural networks are. In biology, cell specialisation is the process of a cell becoming specialized for a task, such as when cells are specialized to repair damaged tissue on the skin (Lakna, 2018). This process happens through signaling and the state of the cell. Hence, how a cell can be specialised is dependent on its own state and its local environment. In fact this differentiation of cells is in broad strokes how an embryo grows into a human being, and by extension the human brain which is the inspiration for ANNs. Therefore, it stands to reason that modelling the process of creating artificial neural networks after the biological process might yield interesting results. While the approach presented in this thesis is heavily inspired by biological processes, the authors have no in-depth knowledge of biology and the algorithm presented is not related to any biological equations as to how cells reproduce. Instead the algorithm is inspired by the abstract concepts of cell reproduction based on state and local information.

This thesis will try to leverage this concept in creating a new evolutionary algorithm called NeuroEvolution Through Neural Cellular Automata (NETNCA). Taking inspiration from natural biological processes, the NETNCA algorithm uses an Neural Cellular Automata (NCA) to mimic the concept of cell reproduction and cell specialization in order to grow ANNs. In this thesis each step of the algorithm will be explained, as it consists of different subsystems all working together in order to solve different tasks. Each of these tasks pose different and increasingly harder challenges in order to see how far it is possible to push the NETNCA algorithm and demonstrate what it can achieve. The algorithm contains four subsystems that has been created for the project. Firstly, a neuroevolution algorithm, used to evolve and train networks. This allows many networks to be trained at the same time and should allow for finding solutions given enough time and computer resources. The second system was, as mentioned previously, the NCA. This was the subsystem of the algorithm trained by the neuroevolution in an attempt to make it build the neural networks. It would allow for each neuron in the network to be processed individually conforming to the concept of each cell making independent decisions only based on the states of the cells surrounding it. This is also one of the focus points of the project, to study if this is even possible and if it can achieve good results. The third subsystem is the intermediate state before the neural network is usable, this is referred to as the graph, since a graph structure is very similar to the structure of a neural network with node and edges corresponding to neurons and connections. This intermediate stage holds all of the information and structure needed to generate the neural network. The last subsystem converts the graph to a working neural network and tests the performance of it.

This thesis will explain the process and ideas behind this approach and provide details in each of the follow sections. Section 2 will introduce the background of the technologies and theories used in this project. This is followed by section 3 where other approaches with similar characteristics are presented. Then section 4 will explain, in depth, each of the previously mentioned subsystems and convey the ideas behind their implementation, and how they interact. Section 5 briefly explains each of the different environments and problems that are used to test the performance of the NETNCA algorithm, to provide a better understanding of the experiments performed using these. This is followed by section 6 that analyzes the results of the hyperparameter experiments conducted. These experiments were conducted in order to find the best configurations of the algorithm. It also contains an analysis of using the NETNCA algorithm to solve a classification problem. The purpose of this was to test whether the NETNCA algorithm could solve different types of problems. Section 7 analyzes the results of running the NETNCA algorithm on each environment. The section also analyzes the networks produced by NETNCA. Finally it compares the results of NETNCA to other approaches. Following this in section 8 a discussion is presented containing reflections upon the algorithm and how well it performed from several different aspects. It also contains an in-depth step-by-step analysis of how the algorithm manages to build a neural network. Then finally section 10 contains the conclusion of this project as a whole.

2 Background

2.1 Artificial Neural Networks

Artificial neural networks (ANN) are computational systems which are inspired by the processes of biological neural networks. The purpose of an ANN is to try to approximate any complex non-linear function using non-linear regression. The building blocks of an ANN are neurons and synapses. More precisely a neural network is built up of neurons that are connected in some manner. There exists many different kinds of neural networks. The ones that will be the focus of this thesis are known as feed-forward neural networks, in which the neurons and the connections between them form an acyclic directed graph. In feed-forward neural networks, the neurons are usually layered. In this case, the network consists of an input layer, an output layer and a number of hidden layers. Each layer is connected only to the layer ahead of it. The input layer is connected to the first hidden layer, the first hidden layer is connected to the second hidden layer and so on. Finally the last hidden layer is connected to the output layer. Two connected layers can have any configuration of their connections. Often fully connected networks are used, in which all the neurons in the previous layer have a connection to every neuron in the next layer (Han et al., 2012).

A neural network takes a number of inputs and for each of these there exists a corresponding neuron in the input layer. When a neuron in the hidden or output layers receives its inputs, it sums them, adds a bias to the sum and applies an activation function to the final value, before sending it forwards to the next layer. Activation functions are usually non-linear functions that represent the activation level of each neuron. This is what enables a neural network to do non-linear regression. In layered networks each layer, except the input layer, has a single activation function that all neurons in that layer uses. Then the neuron propagates the output of the activation function through each of its connections. Each connection has a weight associated with it, which is used to scale the value passing through it. Finally, once the input has passed through the entire network, the values of the neurons in the output layer are the output of the network. Another way to view this is that the input to each neuron, except for the input neurons, is a weighted sum of the outputs of the previous layer and the neuron's bias. This allows these values to be computed by matrix multiplication which is highly optimizable. The topology of the network, being the number of neurons in each layer, the number of hidden layers, and the connections between them, is often up to the AI engineer to decide. It should be noted that there exists techniques which develops the topology of the ANN (Han et al., 2012).

It is important to state that a feed-forward neural network does not have to be layered. Layering is a good abstraction for the developer of the network to more easily define the topology of the network without having to take every single neuron in the network into account. Layering has shown to perform quite well on many machine learning problems. The networks that the NETNCA algorithm produces are not layered networks, as this seems to be an unnecessary limitation, since the topology is not designed by a human. When a neural network is non-layered, its activation functions are defined per neuron, as it is no longer possible to define per layer.

There exists many ways to optimize the weights and biases of a feed-forward neural network such that it gives a desired output on some input. The most common of these is probably backpropagation. Backpropagation is an algorithm that tries to minimize the mean squared error of the network by using stochastic gradient descent. Therefore it only works when an expected output is known for some inputs as otherwise it is not possible to calculate the error. Backpropagation calculates how each weight and bias in the network should be changed to minimize the error between the actual output of the network for a specific input, and the expected output for that input. This is done using the chain rule to find the derivatives of the network with respect to each weight and bias. If this is repeated for all inputs for which an expected output is known, these changes to the weights and biases can be averaged, and gradient descent performed by applying the changes. This is usually not done, as computing this for all the examples before applying the changes is very slow. Therefore stochastic gradient descent is done instead, in which only a subset of the inputs are backpropagated before updating the weights and biases. Another way to optimize the weights and biases, and sometimes even the topology, is using neuroevolution, which is evolutionary algorithms applied to neural networks. This will be explained in depth in section 2.3 (Han et al., 2012).

2.2 Cellular Automata & Neural Cellular Automata

A Cellular Automata (CA) is a model of computation that works on a regular grid of cells (Sarkar, 2000). Each cell can be in a finite number of states, and the CA has a set of transition rules that dictates how to update the state of each cell based on the state of the surrounding cells and the cell itself. This set of states is called the neighborhood of the cell. The usefulness of CAs can be found in their ability to model complex systems from a set of simple localized rules. A famous example of this is Conway's Game of Life (Gardner, 1970), which is a turing-complete, zero-player game that uses a CA to create complex patterns on a grid from a set of simple rules.

The two most common definitions of the neighborhood of a cell are the Von Neumann neighborhood and the Moore neighborhood (Sarkar, 2000). The Von Neumann neighborhood consists of the cells immediately above, below, left and right of the current cell. the More neighborhood also consists of these but include the the four diagonal directions as well, this is also the neighborhood used in the Game of Life. A visualization of these neighborhoods can be seen in Figure 2.1.

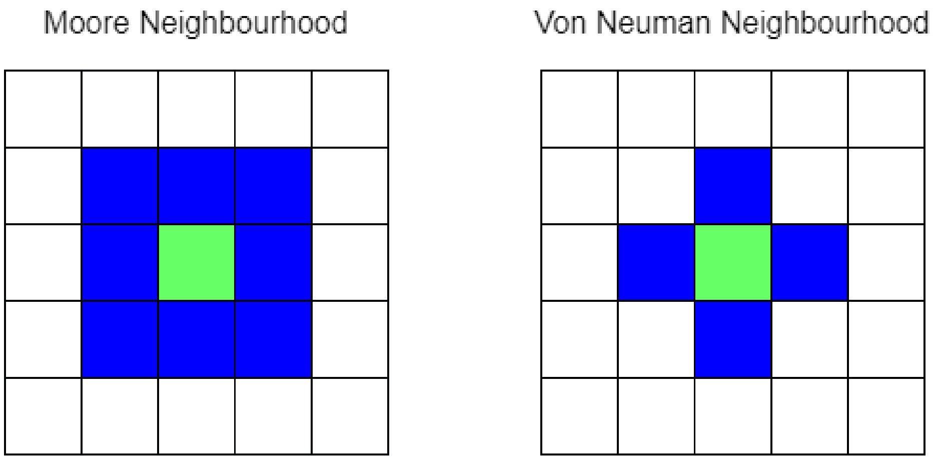


Figure 2.1: *The two most common CA neighborhoods. A Moore neighborhood to the left, and a Von Neumann neighborhood to the right. The blue cells are included in the neighborhood of the green cell. The white cells are not included.*

A CA works by iteratively updating the states of the cells of the grid based on their neighborhoods according to some given transition rules (Toffoli and Margolus, 1987). This is known as the gather-update formulation which can be seen in the equation below for a 1-dimensional CA where the neighborhood consists of the cells immediately to the left and right of the current cell.

$$s(c, t+1) = T(s(c-1, t), s(c, t), s(c+1, t)) \tag{1}$$

Here, c is the index of the current cell, t is the current timestep, s is a function that calculates the state of the cell with index c at timestep t. Finally T is the transition function that implements the transition rules. As can be seen the states of the neighborhood are first gathered by the cell, and the cell is then updated by T.

A Neural Cellular Automata (NCA) is an extension to the normal CA where an ANN, is used to implement the transition rules, as opposed to having a human defining the ruleset. This is analogous to using an ANN to approximate a function, except here an ANN is used to approximate a transition ruleset instead. With respect to the gather-update formulation seen in equation 1, an ANN is used in place of the function T. In the paper by (Mordvintsev et al., 2020) the term NCA was introduced. They train a Convolutional Neural Network (CNN), a special type of ANN useful for image recognition, to learn the transition rules of a 2-dimensional grid, which they interpreted as an image. More specifically, they trained the CNN to learn specific transition rules such that the image would converge to a given target image. In that paper the term NCA refers explicitly to CAs where the transition rules are learned by a CNN, but in this thesis, the term is expanded to let any ANN implement the transition rules. This approach was later expanded to 3D by (Sudhakaran et al., 2021), where the train an NCA to build structures in a game called Minecraft. In (Risi, 2021) it is argued that self-assembling, which CAs and NCAs are an example of, is the future of the AI field. They show multiple examples of how NCAs are able to produce functional and self-repairing machines in 3D and 2D. Further proving the strength of self-assembling and localized control as opposed to global control.

2.3 Neuroevolution

Traditionally, ANNs have been trained using backpropagation. With advances in computing power, new techniques such as neuroevolution has become a viable alternative (Such et al., 2017), especially as it has been shown that neuroevolution techniques are possible to run in parallel. This means that even though the evolution might be slow in CPU time, the wall clock time of the approach can still be lowered, if parallelization is properly applied to the process. Neuroevolution is artificial evolution known from genetic algorithms, but applied to neural networks. An evolutionary algorithm works by making small changes to some genes, called the genotype, which encode the actual structure that is evolved called the phenotype. Neuroevolution opens up for more changes to the network than just the weights and biases of the neurons and connections, as it can also be used to evolve the topology of the network itself (Stanley et al., 2019). In evolutionary algorithms there are two types of ways to encode the genes of a phenotype. Direct encoding and indirect encoding. When using direct encoding the evolution is performed directly on the network and as such the phenotype and the genotype are equivalent. In indirect encoding the evolution instead makes changes to an encoding of the phenotype. This is often done to make a smaller and more concise representation of the network, or to change the structure to more easily make the changes.

The basic form of evolution has been displayed in figure 2.2. It starts with the initialisation of a population of randomly parameterized individuals, where each receive a score according to how well it performs a given task. In the case of neuroevolution each individual corresponds to an ANN. The function for calculating this score is commonly referred to as a fitness function. This fitness is meant to illustrate the quality of each individual, so that over multiple generations this value should grow in order to show improvement. There is not a common fitness function that will work in multiple different problems, so instead they can be created on a problem to problem basis, as each fitness function should express how to achieve the best possible result for a problem. The neuroevolution algorithm will then have to choose what individuals in the current population should be used to generate offspring for the next generation. This is what is know as parent selection. There are different options on how to best perform parent selection, mostly they are based on the previously defined fitness score, as this should be the best indicator on what individual would breed the best offspring. Another indicator other than the fitness score is the age of an individual. In this case, age is usually defined by how many generations an individual has been alive. This is also a good value to use when doing parent selection, as it can help with the removal of stagnant individuals that have stopped improving. This happens most often because they have hit a local optimum and are therefore no longer as relevant as other individuals that are still showing improvement (Eiben and Smith, 2013). The actual selection process, can be done in several different ways. Simple approaches might just choose the parents solely based on who is performing the best in a generation, but other approaches would also give individuals with lower fitness a chance to be selected as well, since they might still show potential and help reducing stagnation in the population. Though these individuals would normally have a lower chance of getting picked as parents, since a better fitness is still the best indicator for the future possibilities of an individual.

The next step in the process is to create variations using the parents selected from the population. These variations could be mutations, where one or more parameters of the individual have been randomly changed by adding for example gaussian noise to it. Gaussian noise is also referred to as a normal distribution, or a bell-curve distribution. This means that when sampling from a gaussian distribution, values closer to 0 are sampled more frequently and values far from zero less frequently. When applying gaussian noise to a network parameter a value is sampled from this distribution and added to the parameter. Another variation is crossover, where the genes (parameters) of two individuals are being combined to create a new individual. The idea behind crossover is that combining two fit parents should be able to create an even fitter offspring (Eiben and Smith, 2013). After this the survivor selection happens where the worst performing individuals are culled from the population and then the process can repeat itself until a terminating condition has been met. This searches the space of possible individuals, and works based on the assumption that small changes to a well performing individual, could lead to better performing individuals.

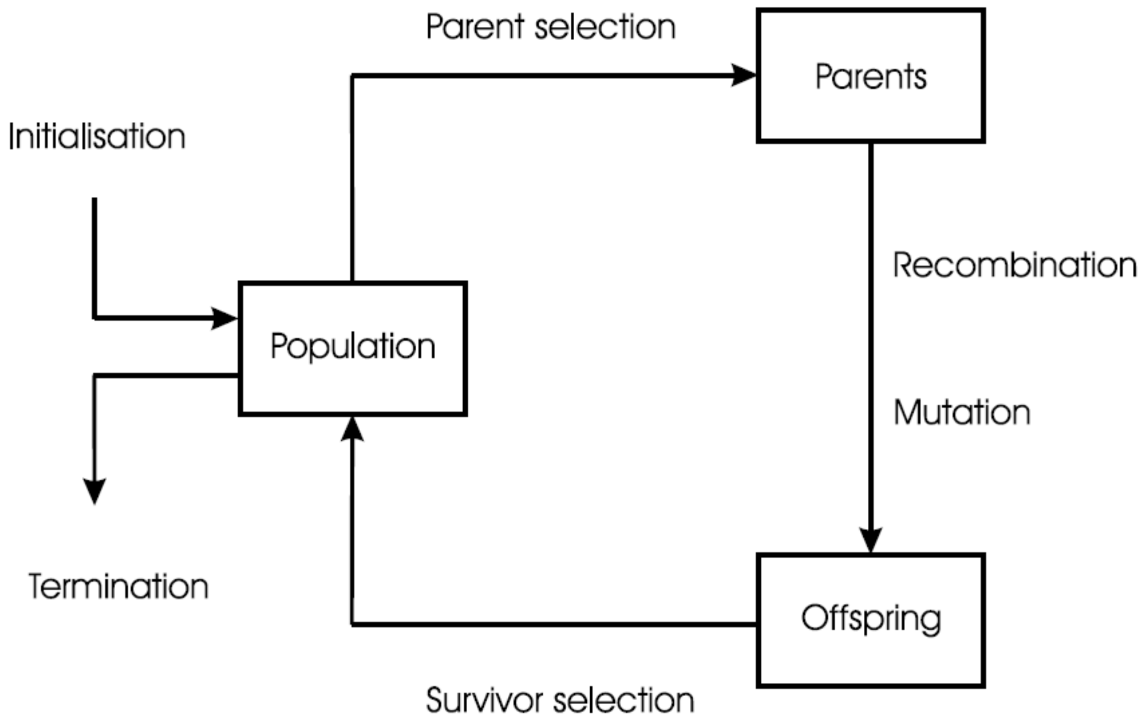


Figure 2.2: *A general overview of evolution from Eiben and Smith, 2013*

3 Related Work

The NETNCA algorithm was inspired by and builds upon many previous foundational works in regards to CAs, NCAs, Neuroevolution and other techniques to generate ANNs. These will explained in this section.

3.1 NEAT

A previous approach to generate both the parameters and the topology of neural networks as NETNCA does is NeuroEvolution of Augmented Topologies (NEAT) (Stanley and Miikkulainen, 2002). Like NETNCA it produces non-layered networks. It has shown great promise in not only evolving the parameters of the network, but also evolving the topology of the network itself. How does NEAT accomplish this? By using a special encoding for the genes of the network which solves the competing conventions problem, and also allows for crossover between the genes. The competing conventions problem results when the same neural network can be represented by different encodings of the genome. When this is present it makes crossover likely to produce worse offspring (Stanley and Miikkulainen, 2002). NEAT's encoding solves this by using historical markings, also know as innovation numbers. The network is encoded in NEAT as a list of the networks connectivity. Each connection between two neurons in the network makes up a single gene in the genome. Each gene contains information on which two nodes it connects, its weight, whether the connection is enabled or disabled, and an innovation number. The innovation number represents when the connection was made, and is thus simply just an incrementing integer. Whenever a new connection is created, it is assigned the current innovation number, and then the innovation number is incremented. This allows the genes to be matched up by their innovation numbers when doing crossover. Genes with the same innovation numbers are called matching genes. Genes that only exists in one of the genomes are either disjoint or excess genes, depending where they lie in the list of genes. When doing crossover, the offspring is constructed by randomly choosing between the matching genes of the parent, and the disjoint and excess genes are included from the parent with the best fitness. The possibility to line up the matching genes based on their historical markings solves the competing conventions problem, because the same genes in both parents can be matched.

Mutations in NEAT can be split up into three different types. Weight mutations, add connection mutations and add neuron mutations. Mutations of the weights are done as normal when using neuroevolution which is explained in section 2.3. Adding a connection, adds a new connection between two previously unconnected neurons. It adds a single new gene to the list. Adding a neuron always happens on an existing connection. The existing connection is disabled, and two new connections are added. One from the first neuron of the previous connection to the new neuron, and one from the new neuron to the second neuron of the previous connection. The first connection is given a weight of 1, and the second connection is given the same weight as the previous connection. This is done to minimize the impact of the non-linearity that the new neuron introduces to the network.

NEAT also solves another problem that results from evolving the topology of the networks. When a new node is added to the network, this will usually result in dip in fitness because the network has not yet had time to optimize the weights of the new connections. This means that these new networks might not survive the next generation even though their new topology might actually lead to better results given more optimized weights. NEAT solves this through speciation. The idea is to split the different networks into species depending on their similarity. This can be done by utilizing the historical markings. Since the more excess and disjoint genes a genome has compared to another, the more different they are. Therefore the following equation can be used to measure two genomes compatibility.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{2}$$

Where E is the number of excess genes, D is the number of disjoint genes, N is the number of genes in the larger of the two genomes, $\overline{W}$ is the average weight difference between the matching genes, and finally c_1, c_2 and c_3 are weights that can be adjusted to control the impact of each of the three factors. δ is then a number representing the difference of the two genomes. Using that equation, NEAT assigns the genomes to their species defining some threshold value δ_t. Each genome of a new generation is then assigned to the first species where its δ is less than δ_t, or becomes its own species if it never happens. To calculate δ a random network of the previous generation of the species is used for comparison. Having the networks speciated allows NEAT to calculate the fitness using explicit fitness sharing (Stanley and Miikkulainen, 2002). This is done by taking the fitness of each genome and dividing it by the number of genomes in its own species. The sum of each of these adjusted fitness scores in each specie is then used to determine the number of offspring that each specie is allowed to produce. To generate the next generation, the lowest performing genomes of each specie is removed, the offspring is created and the offspring then replaces the population.

3.2 Expanding CA and NCA to operate on graphs

As was stated in section 2, CA and NCA, are usually applied to a regular grid structure wherein the neighborhoods are easily defined, such as Moore or Von Neumann neighborhoods (Sarkar, 2000). Unfortunately for the NETNCA algorithm there is not straightforward way to represent an ANN as a grid. However, there exists several methods of extending this standard behaviour to be applicable to graphs, which are much closer in structure to an ANN. One of these methods, is known as the relational Graph Cellular Automata (r-GCA) (Małecki, 2017). It uses the structure of a directed and weighted graph instead of a grid to represent the environment, and the neighborhood for a cell is defined by the connections between this cell and other cells. The neighborhoods are dynamic, because new nodes might be added, nodes might be deleted, and connections might change. This expands classical CA to be able to model complex systems in which the number of objects, their interactions and relations are ever changing. In (Małecki, 2017), they use r-GCA to model the movement of vehicles in traffic.

Another approach known as Cellular Automota on Graphs (CAG), uses the collision-propagation update rule instead of the gather-update formulation, to extend CA to be applicable to any interconnection between cells (Chopard et al., 2008). The collision-propagation formulation for a 1-dimensonal CA can be seen in the equations below.

$$f(c,t) = (f_0(c,t), f_1(c,t), f_2(c,t))^T \tag{3}$$

$$f^{in}(t+1) = Pf^{out}(t) \tag{4}$$

$$f^{out}(t) = Cf^{in}(t) \tag{5}$$

$$f^{out}(t+1) = CPf^{out}(t) \tag{6}$$

Equation 3 shows that each cell's state is actually a vector of values. The number of values depends on the size of the neighborhood. There is one value for each cell in the neighborhood. $f_i(c,t)$ gives the value for cell c at time t corresponding to the cell at $c+i$. This alludes to the fact that it is possible for the states to be of different sizes, if the neighborhoods are of different sizes. This would be the case if the neighborhood were defined by the connections of a node in a graph. Equation 4 shows the propagate step. For each cell you take its current state $f^{out}(t)$ and apply the propagation operator P which is aware of the topology of the neighborhood. The propagation operator takes these states, and propagates the values to the corresponding cells producing a new intermediate state for each cell $f^{in}(t+1)$. Equation 5 shows the collision step, where the intermediate state for each cell $f^{in}(t)$ is evaluated by the collision operator C, which transform the intermediate state into the new state based on some transition rules. Finally equation 6 shows the two steps combined, where the current state is first propagated and then collided, which produces the new state in the next timestep. Having a multi-valued state for each cell, where each value corresponds to a cell in the neighborhood, and a propagation operator that is aware of the topology of the neighborhoods, allows you to extend to a graph structure easily. The neighborhoods are then the connectivity of each node, and the propagation operator is defined to take this into account. In the paper (Chopard et al., 2008) they also show that the collision-propagation update rule is mathematically equivalent to the classical gather-update formulation for CAs.

Neither of the previous approaches were designed for use with neural cellular automata. They both allow the number of states of the neighborhoods to be of differing sizes, which is not feasible when using a ANN, as the the size of the input vector cannot be dynamic. Otherwise the topology of the network itself would also have to be dynamic. An approach that is specifically targeted for NCAs is presented by (Grattarola et al., 2021). They propose an approach called Graph Neural Celluar Automata (GNCA), in which they use a Graph Neural Network (GNN) to learn the transition rules of a GCA, by taking advantage of the message passing scheme of GNNs (Zhou et al., 2020).

3.3 Neuroevolution on NCA

In the paper by (Elmenreich and Fehérvári, 2011) they use an evolutionary algorithm to train ANNs to learn the transition rules of a CA, such that it could reproduce small target images. This is known as morphogenesis. They used a small ANN for each cell in the grid. Each of these ANNs were connected to its neighbouring cells and produced an output for the color of the cell it belonged to. Each ANN had identical topology and were initialized with identical weights and biases. Then an evolutionary algorithm would search for the optimal weights and biases to produce the chosen target image. Although they do not call the ANNs NCAs, by the definition in this thesis they would be. In the paper they showed that optimizing NCAs with the use of neuroevolution was entirely possible.

4 Approach

To explain the approach used by the NETNCA algorithm, firstly a brief overview of how the different subsystems interact with one another is provided. The approach consists of 4 different subsystems that are connected in order to produce and build neural networks. An evolutionary algorithm, the NCAs, the intermediary representation of the output network, from now on referred to as the graph, and finally the conversion from the graph to the actual output network. The approach can be explained in genetic algorithm terms as follows. The population consists of a number of individuals. The genotype of these individuals is an NCA and it is this NCA that is then mutated by the genetic algorithm in order to evolve and produce better results. The phenotype of the individuals are the final output network. To convert from the genotype to the phenotype the NCA is used to grow a graph, which then in turn can be converted to a neural network. The fitness of an individual is calculated by using its NCA to grow a graph, convert the graph to a neural network and use its performance on the problem as the fitness score achieved by that individual. Figure 4.1 displays a visual overview of how all of these subsystems interact. The following sections will be an in depth explanation of the each of these subsystems.

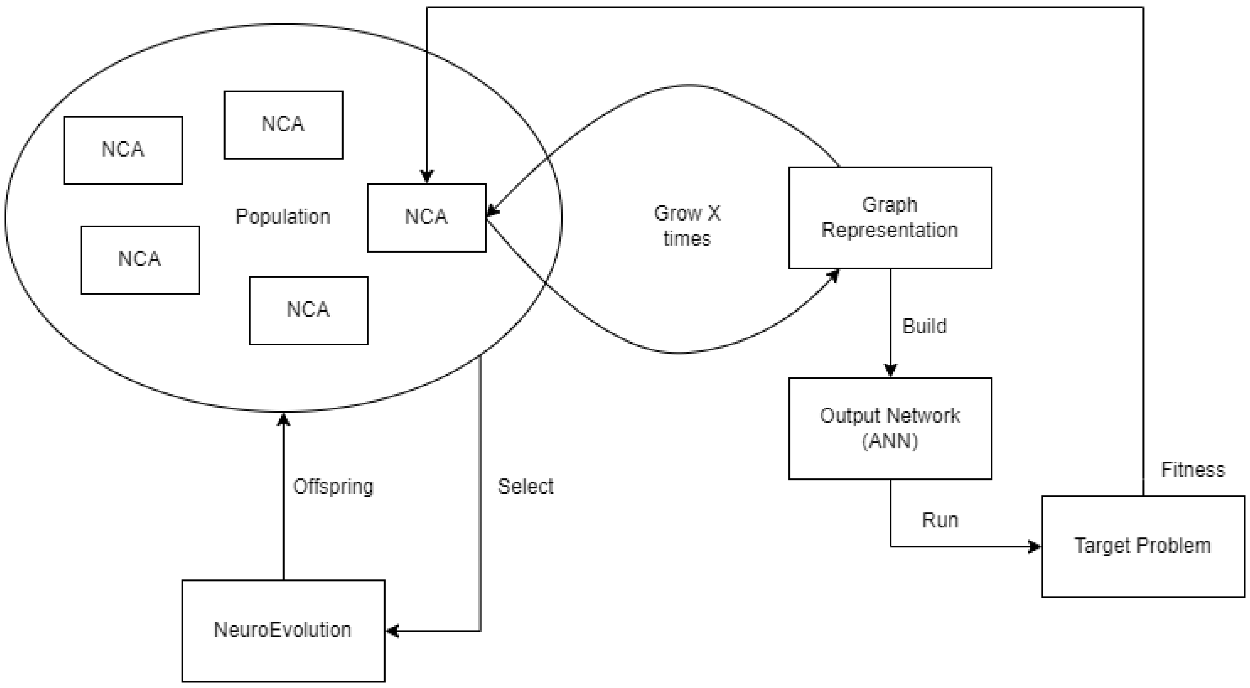


Figure 4.1: *A visual overview of the approach.*

4.1 Neuroevolution

In order to train the NCAs to produce well performing neural networks, they are evolved using a neuroevolution algorithm. The algorithm is used to mutate the weights and biases of the NCAs using gaussian noise, and should therefore be able to produce slightly better networks through each generation of the artificial evolution. This method has been chosen since it has been shown that with the processing power of modern computers this type of training generally produces good results (Such et al., 2017), (Stanley et al., 2019), (Risi and Stanley, 2019). Evolutionary algorithms are also suited to apply crossover to networks as a way to achieve better results, however, the authors found that using a crossover method on the weights and biases of a neural network would produce very poor results, and therefore was not used in this project. When selecting parents for mutations, the best individuals are chosen based on their individual fitness score, though the algorithm does have a chance of selecting a random individual instead, in order to try and counteract a local optimum overtaking the whole population. When a parent has been chosen for mutation, it will produce one or more offspring. Each one of these will be very similar to their parent, with only having a few weights or biases modified. All of the resultant offspring are then added back into the population followed by an application of truncation selection to bring the population back to its original size. Truncation selection is a breeding technique not unique to genetic algorithms, where individuals whose phenotypic expression is above or below a certain threshold. In this case the certain threshold is whether the individuals are in top X of scoring individuals where X is the size of the population. This process is repeated over multiple generations, until the resulting offspring stops providing a significant increase in fitness for a set number of generations.

In order to use neuroevolution a fitness function needs to be applied to select the best performing individuals for parent and truncation selection as described previously. Different approaches to fitness functions were tested and evaluated, in order to find a suitable candidate. The basic version of the fitness function was calculating the fitness as an average of scores produced by the individuals within a given environment. More information on the different environments and what problems they contain is described later in section 5. Another approach were to deduct a penalty from the fitness based on the topology and connectivity of the networks themselves. More information on this approach can be found in section 6. Using NEAT (Stanley and Miikkulainen, 2002) as the neuroevolution algorithm was also considered. This was ultimately not chosen due to NEAT performing poorly when the network becomes too large (Hunter, 2019), and as will be explained later in this thesis, the NCAs used in this approach were relatively large.

4.2 Graph

This section will go into depth about the graph data structure that plays a very central role in this project. The biggest challenge when designing the approach was figuring out how to transform a neural network into small discrete neighborhoods that could be used as input and output for the NCAs as typically NCAs and CAs are applied to regular grid structures. This presented numerous challenges to overcome. How should the neurons and the connections between them be encoded. The weights, biases and activation functions also needed to be encoded in this structure. To simplify this task, it can in its purest form be seen as trying to apply an NCA to a graph. This is because a graph has a very similar structure to an ANN, as nodes in a graph are equivalent to neurons in a neural network and edges represents the connections between neurons. There exists several variants of CAs and NCAs that can be applied to graphs, as mentioned in section 3. Common for all of these are that their neighborhoods are constructed by the connections between the nodes in the graph. This means that if two nodes are close together in space, with any definition of space for the graph, but are not connected, the NCA would not be able to take advantage of this information. The authors wanted this spatial information to be available to the NCA, as two spatially close nodes should have a higher chance of establishing a connection, than two distant nodes. Therefore a straight forward approach would be to somehow represent a neural network in a grid. A grid imposes spatial positioning on the graph that was not there previously. Therefore, the spatiality of the nodes in the graph have to be defined.

4.2.1 Horizontal ordering

A naive way to define the spatiality, would be to decide on an arbitrary starting length between the input and the output neurons, and then only spawn nodes in between these layers. This however puts an arbitrary depth restriction on the neural network. Therefore, strict partial ordering of the graph was used to determine the spatiality of the nodes. This removed the depth restriction, and made the distances between nodes dependent on the graph itself, and not on how it was built, as will be explained later in this section. For a strict partial ordering to exist in a graph, certain restrictions have to be placed on the graph. It has to be directed and acyclic. These restrictions are in accordance with a feed-forward neural network, as it is also directed and acyclic. Furthermore, a restriction such that no nodes could have a partial ordering less than the input nodes or larger than the output nodes was made. This makes intuitive sense in regards to a neural network. The strict partial ordering is used to determine the position in the grid on the x-axis. (The depth of the network). This means that the input nodes would lie in the first column of the grid. The second column would contain the nodes with the next immediate order, and finally the last column would contain the output nodes. Any ordering with no nodes assigned would be omitted and not take up a column. Finally if a node could be assigned more than one ordering it would always be assigned the lowest ordering to reduce the total number of orderings. With this definition, as the graph grows in depth and the partial ordering changes, the nodes can change their absolute positioning, but their relative position in regards to the other neurons will be dependent on the connectivity of the graph and not an arbitrary build order. This will allow the grid to grow without any restrictions regarding its depth, while maintaining a spatial relationship between the nodes. This ordering of the nodes will be referred to as the horizontal ordering.

An example of how horizontal ordering is applied for a simple graph, is displayed in figure 4.2 below. The figure shows a network with a single input node (A) and one output node (C). The new node (D) needs to be added to the graph and the horizontal ordering then needs to be recalculated. In this example, the new node will get an ordering of 2, since the node B, that currently has an ordering of 1, has a connection pointing to it. This in turn also moves the output node to order 3, since the new node has a connection to it. Though, since C is an output node, it would have been moved to order 3, regardless of the connection from D.

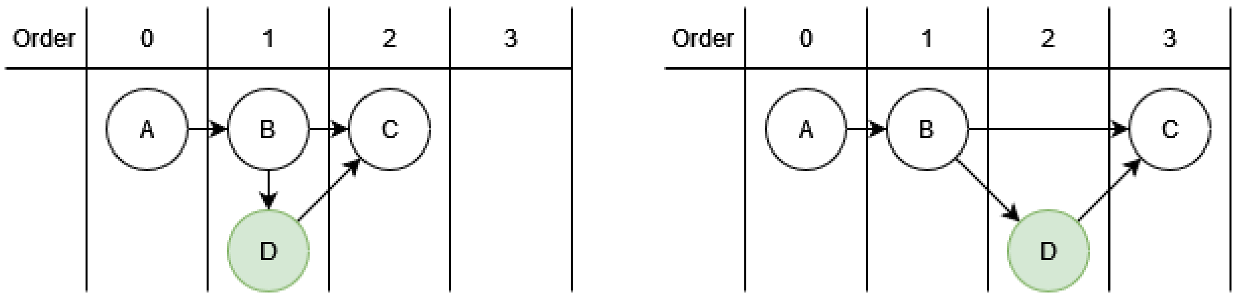


Figure 4.2: *Before and after horizontal ordering is applied to a graph.*

4.2.2 Vertical ordering

The horizontal ordering just described solves the problem of calculating the x-axis positioning of nodes in the graph, but it does not define a vertical position for a node in graph space also called the height of the network. Therefore vertical positioning rules were defined, that tries to accommodate the problem of relative and absolute positioning of the neurons, like the rules for horizontal ordering. To do this, the rules were dependant on the previous horizontal ordering of a node, and if that ordering had not been changed, then the vertical positioning of a node would not change either. This rule was implemented to keep the relative positioning stable, since if the horizontal ordering did not change, most of the previous nodes should not have moved either. As shown in figure 4.2 this is the case for node B. However, when the horizontal ordering of a node did change the absolute position of a node, it should have a minimal impact on the position of other nodes and ensure stability. So nodes that would change their horizontal ordering, would be moved to their respective positioning on the x-axis and then attempt to be inserted using the same vertical ordering.

From here, several rules would apply, the most simple of them being if the position is not being occupied by another node, the moving node will take this position. In figure 4.2 this is the case for node C, that originally had a horizontal ordering of 2, but got moved to horizontal ordering 3. However, since horizontal ordering 3 was empty, it was inserted with the same vertical position. Then, to ensure the protection of non-moving nodes, if the space is occupied by a node that should not be moved, due to not having changed horizontal ordering, the moving node will instead try to be inserted to the position below. This rule is displayed in figure 4.3 where the new node E, needs to be moved to horizontal ordering 2, but will then have to occupy the same position as node D. Since node D is a non-moving node, node E is inserted below it, as that position was empty.

The last rule of moving applies if the moving node encounters another moving node. In a case like this, to also ensure simplicity the current node will take over the occupied space, and the node that was occupying that space, would try to be inserted into a lower vertical position. It is important to acknowledge that this is probably not the best approach to keep the relative positioning of moving nodes. A more complex approach can be found in section 9. Rules are applied recursively at every attempt to insert a node. The rules will be applied, until a node has been inserted on an empty position. Figure 4.4 shows how these last rules work in practice. Here both node E and F are new nodes, that need to be moved in order to rebuild the horizontal ordering, because of their connection from node B. Node E is moved first like in figure 4.3, because D is a non-moving node, then node F is moved. However, this time unlike the previous figure, it tries to occupy the position of node E, and since node E is a moving node, node F will take over the position and node E will attempt to move a position down, where there is an empty space to occupy.

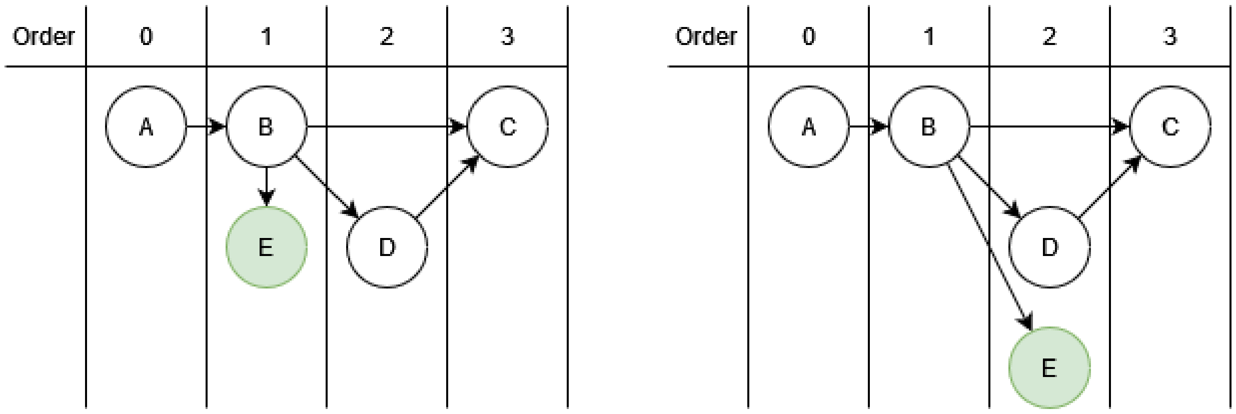


Figure 4.3: *Before and after ordering is applied without changing the vertical position of a non-moving node.*

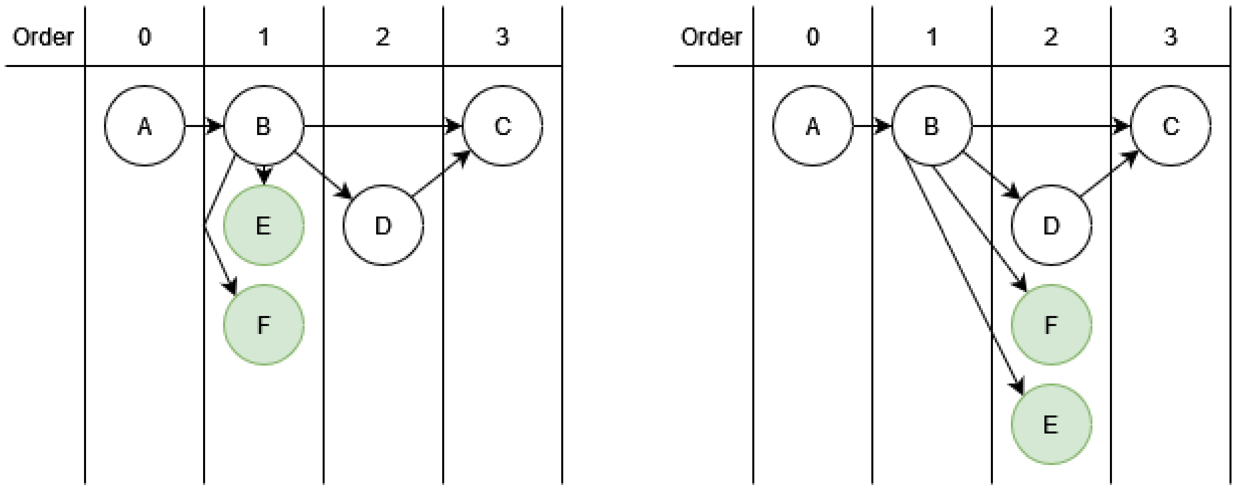


Figure 4.4: *Before and after ordering where moving nodes encounter each other.*

With these definitions of horizontal and vertical spatiality for node placements in the grid, it is possible to convert from the graph representation to the grid representation. This grid representation can then in turn be used to directly obtain and update the neighborhoods of each node, for use with the NCA. This achieves the goal of applying an NCA to a graph with spatial information.

4.3 Pruning

Once the graph has been fully grown, it might be the case that some nodes are not connected to either the input or to the output, directly or indirectly. If this is the case, the node can obviously not have any effect on the output of the network that the graph encodes. Therefore these nodes will be pruned, prior to converting the graph to the output network. Pruning also happens in between each application of the NCA to the graph. Nodes that cannot be given a strict partial ordering, cannot be placed in the grid, as they have no spatial positioning. Therefore these nodes are also pruned.

4.4 Neighborhoods

As described in section 4.2 the graph is placed in a grid structure. Because of this it is possible for the NETNCA algorithm to make use of typical CA neighborhoods, as nodes now have a spatial relation. Specifically, in this case Moore neighborhoods were utilized. Each cell in the grid were used to represent the state of the corresponding node in that position, the state would contain information about bias, the connections with other nodes, the weights associated with these connections and the activation function.

The encoding of all these properties except for the activation functions were as continuous values, since this makes intuitive sense for weights and biases, as these are already continuous values. However, for nodes and connections this is not true, as a node or connection is either there or not. By using continuous values for these two properties, it allows for the NCA to express how certain it was that a connection or node should be present or not. This concept is explained in more depth in section 4.5. The directions of the connections is always pointing to the right and down. This was done the reduce the amount of information needed to encode the connections. The activation functions are stored as a one-hot encoded array, as there is no continuous relation between the different activation functions. Figure 4.5 displays a 3x3 neighborhood as input and output for the NCA. The top of the figure shows how the part of the graph in the neighborhood is placed in the grid, and the bottom shows how it is encoded. The green node is the node whose neighborhood it is. The blue node is a newly created node. For the input, each of the cells in the 3x3 neighborhood contains three numbers. The first number represents whether a node is present in a cell. 1 meaning a node is present and 0 meaning no node is present. The second number represents if there exists a connection between the node in the cell, and the node whose neighborhood it is. The last number represents the weight, which is in the range between 0 and 1. This normalized weight can be transformed to any desired weight range later. The middle cell is omitted because a node cannot have a connection to itself and that node will always be present. For each neighborhood the bias is also included in the range from 0 to 1. A one-hot encoded array is also included to represent which activation function the node has.

The output is slightly different. First of all, the first number which represents the presence of a node is omitted. This is because the only way the NCA can remove a node, is to remove all connections to and from that node, and the only way it can create a node, is to create a connection to an empty cell, in which a node will then be created. Secondly the value representing the connection is revealed to be continuous. It is a value between 0 and 1. If the value is below some specified threshold, the connection will be removed, and if it is above some threshold it will be created. The output in this example has changed the weights, the bias and the activation function. It has also added a new connection, to an empty cell, which in turn has created a new node, namely the node that is colored blue in the figure.

In this simple example the output is very well behaved, meaning that if a connection is not present, its weight value is also 0. This is rarely the case in actual outputs, but the weight value is ignored, if the connection is removed or does not exist.

Though the neighborhood contains the weights of all the connections it is part of within the neighborhood, it only contains the bias and activation function of the node whose neighborhood it is. This was a choice made by the authors to keep the size of the neighborhood from becoming too large. This might not seem like a problem for a small neighborhood of size 3x3, but for larger neighborhoods it certainly makes a difference. It also makes intuitive sense, that an update to a node, should not change the biases and activation functions belonging to other nodes. Much like it only makes updates to connections that it itself is part of, and not other connections that might be within the neighborhood.

A problem that this definition of neighborhood exhibits is that connections can be pushed outside the range of a neighborhood. The authors call these dead connections. The way for dead connections to occur is when one of two connected nodes gets moved too far away from the other node by its horizontal or vertical ordering changing. An approach where a special NCA operated on the dead connections were experimented with, but simply leaving the dead connections usually led to good results. More about these results and dead connections can be found in sections 6 and 9.4.

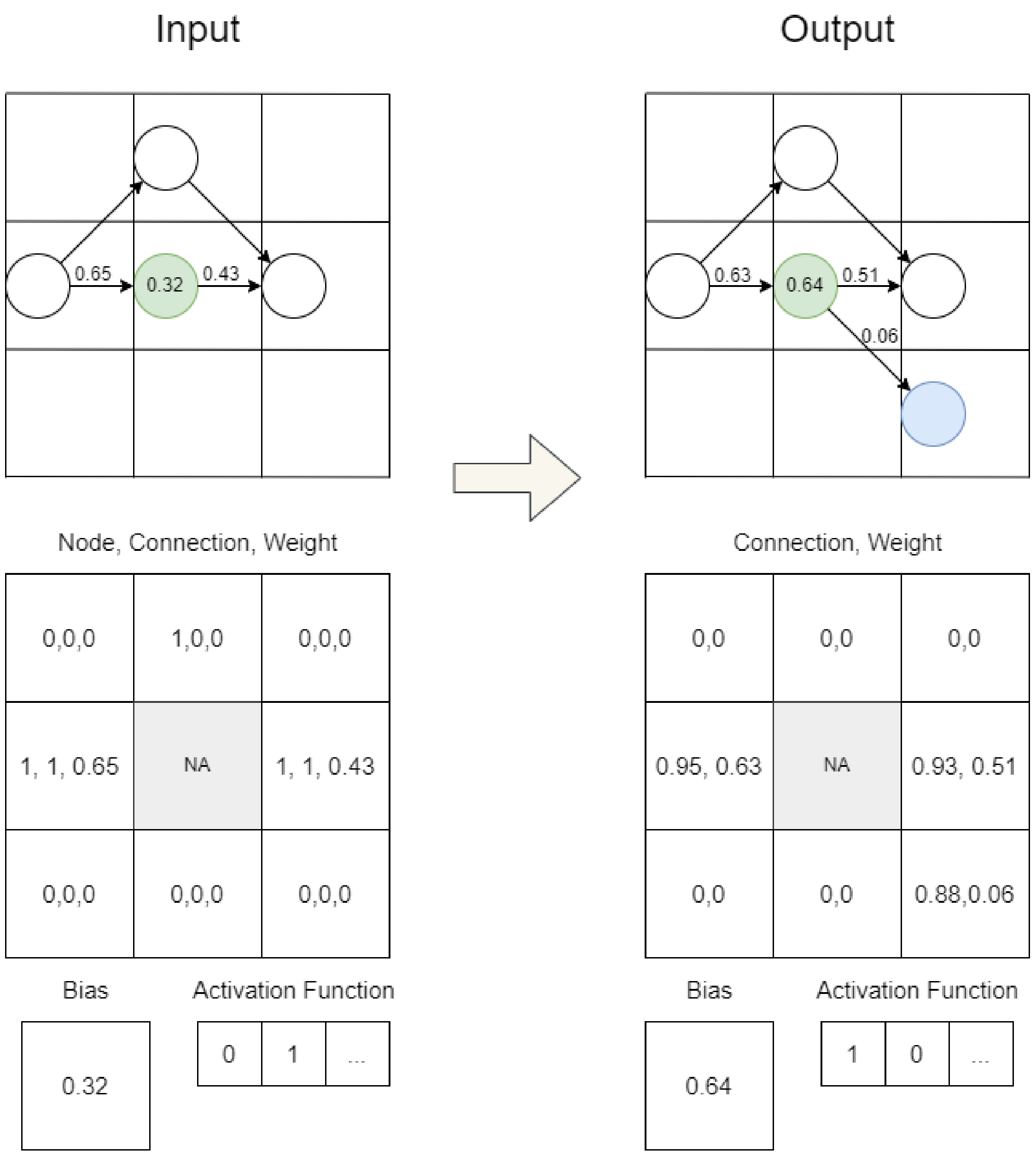


Figure 4.5: *3x3 neighborhood with corresponding encoding for input to and output from NCA.*

4.5 Neural Cellular Automata

The NCAs used in this project were fully connected feed-forward neural networks, using the neighborhood of a node as input vector and having the modified neighborhood as the output vector, as explained in the previous section. A single pass of the NCA applied on the graph involved collecting the neighborhood of each node in the graph and applying the NCA to them one at a time. The output vector with the modified neighborhood would then immediately be applied to the graph, with two minor exceptions. All new nodes that were required to be generated, would only be generated at the very end of an iteration, to ensure stable neighborhoods for the remaining nodes as otherwise a lot of movement of nodes could happen because of the horizontal and vertical ordering as described in section 4.2. It was also not up to a single node, if a connection from or to it should be removed, this was something both nodes in the connection should agree upon. So if both nodes wanted to remove a connection, then it was removed. However, if only one wanted to remove a connection, it was only removed if the other node was less certain to keep it. As the output of the NCA is continuous, the certainty of the decisions could be compared numerically, by comparing the values' distances to the thresholds. This approach was chosen in order to keep neighborhoods stable through each pass. It also decreases the runtime since in each pass the modifications would be applied instantaneously, instead of only in batch at the end of an iteration, so nodes could be updated immediately, instead of waiting for a full pass. This would also cause fewer situations where nodes wanted to take conflicting actions, like two nodes wanting to change the weight of the same connection or if a specific connection should be removed. In the example of creating a new node from figure 4.3, the node B wanted to create and connect to node E, however this will only happen when both node C and D have been modified, as it can have quite the effect on their neighborhoods and possible actions. In this case it is important since the possible actions for node D in relation to node E are different before and after this recalculation, and the recalculation process can be quite expensive depending on the size of the graph. However, if node B wants to change the weight of its connection to node D, then that change would happen immediately, since no new computations are needed and node D would have the modifications from B already in its neighborhood.

5 Environments

For evaluating the implementation OpenAI's Gym environments[1] was used. The environments are all control problems, that are usually used as a test-bed for reinforcement learning algorithms. The problems were chosen because of the lack of research on genetic approaches to these types of control problems, as well as an abundance of research on supervised problems. A few examples include: (Mordvintsev et al., 2020), (Palm et al., 2022), (Ruiz et al., 2020). The following is a list of the problems.

- **CartPole**, a control problem where the AI has to balance a pole on top of a cart by only moving the cart left and right. Maximum score is attained by keeping the pole balanced for 500 timesteps.
- **Acrobot**, a control problem where the AI has to swing a 2-joint pole over a certain threshold. The faster the problem is solved the better the score.
- **MountainCar**, a control problem where the AI has to push a cart up a hill. The faster the problem is solved the better the score.
- **LunarLander**, a control problem where the AI has to land a flying machine on a designated target in a procedurally generated terrain. The AI is penalized for crashing (letting anything other than the legs collide with the ground), and for the amount of time it takes for it to land on the designated target.
- **Pendulum**, a continuous control problem where the AI has to swing a pendulum into an upright position and then keep it there by applying torque to the Pendulum. The more of an upright position the Pendulum is in, and the less torque that is applied per timestep, the less of a penalty is given to the AI at each timestep.
- **BipedalWalker**, a continuous control problem where the AI controls a 2-dimensional 2-legged robot and has to walk rightwards as fast as possible. Reaching the end before the time runs out provides the maximum score. Falling (having the head collide with the ground) incurs a strong penalty.

The first four environments; CartPole, Acrobot, MountainCar and LunarLander are all control problems with a discrete environment. Each output neuron corresponds to a single action. The action taken is decided by which output neuron is most active. The two last environments; Pendulum and BipedalWalker are both continuous environments. An example of what this means is Pendulum, which has a single output neuron that controls the amount of torque to apply to the Pendulum. This value ranges in the continuous space between -2 and 2. All of the used Gym environments have a stochastic element to them as they all have multiple different starting states. For example the Pendulum environment may have the pendulum starting in an upright position at one run of the environment, and a downwards position in another run. This means that the networks evolved to solve this task cannot just memorize the assignment. This randomness means that the AI has to be evaluated multiple times, and its final score must be an average of all its scores in the environment.

5.1 Classification Environment

All of the environments described so far in this section has been different control problem environments as these have been the primary focus for testing the NETNCA algorithm. However, to test the capabilities of this approach a minor experiment were also carried out using a classification problem instead. This is different from the previously described problems, in that the algorithm should attempt to learn classification of static data, instead of deciding on one or more actions in an state that is in constant motion based on previous actions. In the case for this experiment the very simple Iris dataset from the sklearn library [2] was

[1]https://www.gymlibrary.ml
[2]https://scikit-learn.org

used, it is a small data set only containing 150 different data points to use for input and 3 different labels each corresponding to exactly 50 data points.

6 Experiments

The following section will showcase the experiments that were conducted in order to find the best tuning for various hyperparameters. Each experiment were run 25 times with a population of 100, generating 100 offspring for 100 generations. Then for each of the 25 runs the best individual in the population were selected as the result.

6.1 Experiment Overview

The focus of the experiments and the hyperparameters selected for testing were chosen due to their believed impact on the performance of the NETNCA algorithm by the authors. The following list is an overview of the experiments conducted, as well as an explanation as to why these experiment were thought to be important.

- **Different sizes of growth iterations**. Growth iteration is the parameter that controls how many times the NCA is applied to the graph. This parameter has, of all the parameters, the largest effect on the size of the output network. With too many growth iterations it may become computationally infeasible for the NCA to grow the network. However, a high value in this parameter could also allow for more complex and sophisticated networks. This parameter is very closely tied to the threshold parameters, which controls how high or low the output of the NCA has to be to remove and create nodes and connections. It is hypothesized by the authors that if the growth iteration parameter and threshold parameters are well balanced the networks are less likely to grow infinitely and will instead reach a stable state. Meaning a state in which no further changes to the graph will be done. It should be noted that Conway observed that predicting whether a Cellular Automata would reach a stable state or grow infinitely is impossible (Gardner, 1970). This test was conducted in order to see whether networks with 7 growth iterations would perform better or worse than the more simple networks using only 3 growth iterations.
- **Different NCA topologies**. The idea of using a neural network to grow the topology of a network is a powerful idea, but it does leave the engineer with the new challenge of deciding the topology of the network producing the topology. Since it would be impossible to test all the different topology strategies as well as activation functions available to the authors a more restricted approach was chosen. In this experiment a simple NCA with no hidden layers is compared to an NCA with 3 hidden layers with the same size as the input layer. In theory, a smaller NCA would mean a smaller ruleset for the underlying CA, which would mean a less complex system. However, as has been proven with CA models such as Conway's Game of Life (Gardner, 1970), a very simple set of rules can result in very complex patterns and behaviours. Therefore, an experiment testing no layers vs a few hidden layers was deemed as an appropriate test into whether the complexity of the ruleset had an importance on the NETNCA algorithm.
- **Dead connection NCA compared to one NCA**. In section 4.4 the problem of dead connections was explained. A proposed solution to this problem is the idea of the Dead Connection Neural Cellular Automata (DCNCA), which would run every growth iteration sequentially with the regular NCA. The DCNCA takes in dead connections and their neighborhoods as input, and outputs whether the connection should be removed, and if not, how its weight should be updated. The neighborhoods are differently defined than the regular NCA. Since the DCNCA runs on connections instead of nodes its neighborhood is also defined in terms of connections. It takes in the normalised amount of incoming and outgoing connections for each node as well as the weight of the connection, and the percentage of the network the connection spans. In order to properly test whether the DCNCA had a positive or an adverse effect on the network this was chosen as a topic of experimentation.

- **Penalizing graph for how many nodes are pruned compared to not penalizing**. As mentioned in section 4.3, any node that is not connected to either the output or the input at the end of growing the graph will be discarded. It is very common for graphs to have over 50% of its nodes pruned before it is converted to an output network. It was hypothesized by the authors that if for example 80% of the graph produced by the NCA had to be pruned, the phenotype would be so far removed from the actual genotype that it would severely worsen the process of evolution. To explain this, in section 4.1 it was presented that the NCA functions as the genotype and the output network functions as the phenotype. If 80% of the effects of the NCAs ruleset are ignored it could theoretically make some of the rules non-deterministic, since they would be victims of arbitrary pruning. Therefore, an experiment where a penalty was added to the fitness function based on how much of the graph was pruned at the last step was conducted.

- **Penalizing trivial networks compared to no penalty**. One of the drawbacks of evolution is its proclivity to get stuck in local optima. For some of the environments that the NETNCA algorithm has been tested on it was found that the evolution would get stuck in a local optimum with either no hidden nodes or some input neuron unconnected to the graph. These networks would perform very well, and therefore their genes would be propagated throughout the population, leading the evolution to getting stuck in this optimum. It was hypothesized by the authors, that by penalizing trivial networks these local optima would be avoided, and as such an experiment was set up to test this hypothesis.

6.2 Experiment Results

This section goes through the results of the experiments outlined in the previous section. The results can be observed in table 6.1

Name	Best	Average	Standard Deviation	T-test
Trivial network penalty vs no penalty (Acrobot)	No penalty: -69.52 Penalty: -68.68	No penalty: -73.40 Penalty: -73.92	No penalty: 3.15 Penalty: 4.03	0.62
Trivial network penalty vs no penalty (Pendulum)	No penalty: -503.51 Penalty: -169.91	No penalty: -746.90 Penalty: -672.19	No penalty: -73.34 Penalty: -225.49	0.12
Prune penalty vs no penalty (Acrobot)	No penalty: -65.80 Penalty: -66.20	No penalty: -71.41 Penalty: -70.26	No penalty: 4.32 Penalty: 3.10	0.29
NCA with 3 hidden layers vs NCA with no hidden layers (Pendulum)	3 Layers: -321.93 No Layers: -250.76	3 Layers: -845.20 No Layers: -518.79	3 Layers: 392.62 No Layers: 205.64	0.007
7 Growth iterations vs 3 growth iterations (Acrobot)	7: -68.96 9: -69.80	7: -248.24 9: -378.58	7: 333.88 9: 433.970	0.25
NETNCA vs NETNCA and DCNCA (Acrobot)	Single: -65.80 Double: -65.10	Single: -71.41 Double: -72.98	Single: 4.32 Double: 5.30	0.26

Table 6.1: *This table displays the results of all the hyperparameter experiments conducted. Each experiment contained two populations of size 100 (one for the baseline and one for the hyperparameter change), which were run for 100 generations. This process was repeated 25 times, each time saving the best member of each population at the end of the run into two different sets. The best, average and standard deviation of these two sets are showcased in this table, along with a t-test comparison of the two sets. Due to the long run time of these experiments the scores are an average of 25 iterations through the environments as opposed to the recommended 100 iterations. The significance value chosen for all p-values in this table is 0.05. All the data used to extrapolate these values can be found in appendix B.2.*

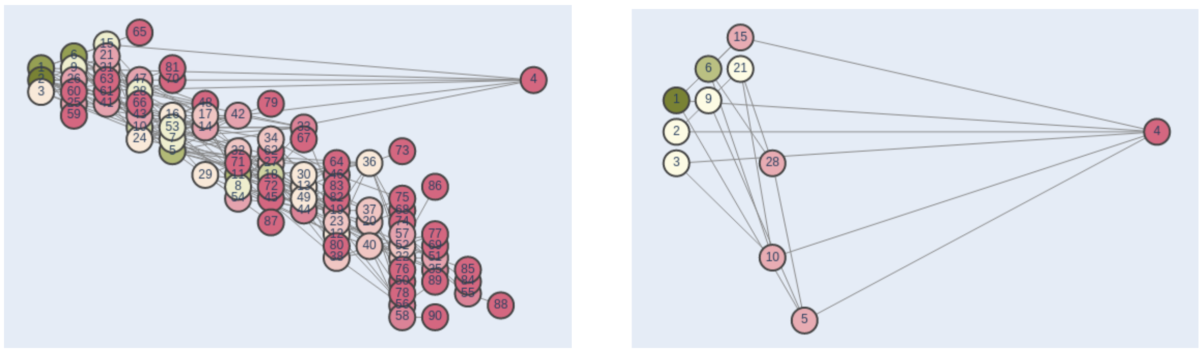
6.2.1 Penalizing Trivial Networks

One of the challenges in evolving a well performing Pendulum network is that Pendulum has very well performing local optima which can be achieved with trivial networks. For example, the average score of a randomly generated Pendulum network is about -1500. However, a trivial network which has, for example no hidden neurons and with one input neuron unconnected to the network can easily achieve a score of -800. This behaviour was observed many times during development and it was decided to add a configuration that controls whether the engineer developing an AI wants to penalize trivial networks or not. The reason that it was added as an option is due to the fact, that not all trivial networks have an adverse effect. A great example is CartPole which can be solved without any non-linearity. If a penalty was added to CartPole for trivial networks it would hinder its evolution, and while it would ultimately still solve the problem, it would lead to an over engineered network. For this exact reason this experiment was run two times. One on the Acrobot environment, where the authors believed that trivial networks would not hinder the evolution, and one on the Pendulum environment, where there could be an issue with the evolution getting stuck.

For the Acrobot environment there is almost no difference in the results between the no penalty approach and the penalty approach. In fact with a p-value of 0.62 there is no real difference between these two sets. However, the same experiment for the Pendulum environment produced results much closer resembling the hypothesis that Pendulum gets stuck in local optima when not penalizing trivial networks. While it should be noted that a p-value of 0.12 is not a statistically significant difference, looking at the Best-column the difference is staggering. Not only does the penalty approach achieve 350 points more than the non-penalty approach, but it has 6 members of its set that outperforms the best from the non-penalty approach. This supports the idea that because of the trivial networks in the non-penalty version, no network above score -500 is ever found. This makes intuitive sense. When there is a trivial pendulum network that dominates the population by scoring -500, this network may be evolved for generations but it is never going to score higher than -500. This is because the network has half of its input neurons unconnected. The problem is that the evolution is not going to try to connect these disconnected input neurons, since it will at first probably lead to a lower score. On the other hand the penalty approach produces a lower average, because even though it has some well-performing networks, it still has a lot of networks that never even gets close to a -500 score. In fact the worst network from the penalty set scored less than -1000. To sum up, the penalty approach has a set with much larger extremes than the no penalty approach, scoring both much higher and much lower. Given that when evolving networks the only thing that really matters is the absolute best network evolved, the penalty option seems to be an appropriate option when dealing with environments like Pendulum. A drawback with this option is that it only really has an effect on some environments, and the engineer using the NETNCA algorithm need to know which environments in advance.

6.2.2 Penalizing Pruned Graphs

As stated in the overview, the authors hypothesized that if a large portion of the generated network was pruned there may be an asymmetry between the genotype and the phenotype that would have an adverse effect on the evolution.



(a) *Network pre-pruned* **(b)** *Network post-pruned*

Figure 6.1: *Shows a Pendulum network before and after pruning. Exemplifies the problem with large portions of the network being pruned.*

Figure 6.1 shows this problem in action. In this NCA the ruleset seems to favor expanding the graph downwards while growing to the right. This makes it so that a large portion of the network grows so far out, that the final neurons can no longer see the one output neuron in its neighborhood, making it impossible to connect it back into the graph. This may seem like a rare example but this is actually very regular behaviour for the NETNCA algorithm when the networks developed are not penalized for being pruned.

When looking at the result of the pruning experiment conducted there can be observed no significant difference between the no penalty and the penalty approach. It appears that NETNCA does not suffer from a disjoint relation between its phenotype and genotype, despite the fact that large portions of its output is removed.

6.2.3 Amount of Hidden Layers in NCA

The experiment testing 3 hidden layers against no hidden layers is the only experiment that actually shows a p-value below the 0.05 threshold. In this case, it is the version with no hidden layers that performed significantly better. Both in the best and average the no hidden approach outperformed the 3 hidden by a lot. This was an interesting observation and seems to support Conway's argument that even a simple rule set can give rise to complex patterns and behaviours. In fact, here it can be observed that having a too complex ruleset (too many hidden layers and neurons) actually has an adverse effect on the network.

When conducting this experiment it was observed that having no hidden layers significantly increases the activity of the network. To explain, the less hidden layers in the network, the higher values would be observed in the output neurons. This leads to the generation of more neurons and connections in the graph when running iterations through the NCA.

6.2.4 Different Growth Iterations

The growth iteration hyperparameter is one of the more interesting parameters that is very unique to NCAs. An NCA that is evolved to create networks using, for example, 3 growth iterations, would not be able to create a network using 7 growth iterations. Its ruleset leading to a beneficial outcome in producing well performing networks is predicated on the growth iterations being the same number as the one the NCA was evolved on. This is due to the unpredictability of a how a CA will grow. A higher number of growth iterations allows for the NCA to create larger networks and to iterate more upon parameters already created, leading to the possibility of a more precise tuning.

In this experiment 7 growth iterations was compared to 3 growth iterations on the Acrobot environment. Once again there is no statistically significant difference between these two approaches. Unfortunately, there is a problem with this experiment. There is no significant difference between the two sets, but there is an external factor that could result in this. The external factor being the environment Acrobot. Acrobot is an extremely easy environment to solve and solutions can easily be found with 3 growth iterations. A more difficult environment such as Pendulum or BipedalWalker should have been used for this experiment, since 3 growth iterations should be insufficient to reach a network complex enough to be considered as a solution network for these environments.

6.2.5 Dead Connection Neural Cellular Automata

The DCNCA experiment provided inconclusive results. There can be observed no difference between the two approaches, which indicates that the DCNCA did not have an adverse nor a positive effect on the performance of the NETNCA algorithm. It did, however, negatively impact the speed at which it was able to tune a network inside a local optimum. This is due to the simple fact that the DCNCA increases the total number of parameters which can be chosen for mutations in the evolutionary loop. This means it will have a lower chance to randomly pick the weight it needs to pick in order to better tune the network. While the DCNCA did not work in this experiment, it provides a first step in trying to address the issue of dead connections. More on dead connections can be found in section 9.4.

6.3 Classification

As expressed in section 5.1 the classification environment does not contain a large amount of data points, but the whole point of this experiment was to test if it was even possible to use the NETNCA algorithm, for something else than control problems. It was not aimed at testing complicated classification problems with lots of multidimensional data and large amount of labels. The testing also involved an attempt to minimize overfitting of the network, as this is a commonly occurring problem where the training of the network tries to mimic a function too well, so that it can classify test data perfectly, but will not be able to classify new data points (Han et al., 2012). In order to do this the full Iris dataset was split into two, a set used for training the network and another set to validate the accuracy of the results following a 90/10 split between the two. During the training phase of the network only 135 data points were used corresponding to 45 points for each of the three labels. The remaining 15 data points would then never be used during training but instead used to calculate the accuracy of the network between each generation.

Another element needed in order to use classification with the NETNCA algorithm was a way to calculate the fitness of each individual in the population. This value is what the evolutionary algorithm uses for selections and unlike with the OpenAI environments, no default calculation was provided for this. Since this was not supposed to be a complex testing environment, and the previously described environments seem to respond well to simple fitness calculations, the *mean squared error* was used as shown in formula 7. This is a common formula to use for the calculation of loss during training, if this was treated as supervised learning using backpropagation instead of neuroevolution. The formula uses two vectors of values to use for its calculation, one vector contains the expected results (Y) and the other is a vector containing the actual observed values ($\hat{Y}$). In this example the variable n in the formula corresponds to the amount of possible labels in this case the Iris dataset contains 3 labels.

$$\frac{1}{n}\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2 \tag{7}$$

Within the constraints of this project and the Iris experiments, the value of the mean squared error function when applied, will always return a value between 0 and 1. These values can then be used to indicate how well the NETNCA generated network is at predicting the samples. The better the network becomes at predicting the correct labels for each sample and not indicating the other labels, the closer the mean squared error will be to 0. On the other hand, badly performing networks will have a score closer to 1. Since the fitness of an individual needs to be high in order to show good performance, as opposed to the result of the mean squared error. The fitness score was implemented with a total of 135 as the highest possible score.

This was equal to the test sample size. Then the sum of the mean squared error over all of the samples was deducted from this. This approach to fitness calculation, should help the NETNCA algorithm to not only learn the right result of the classification, but also minimize the wrong indications in the output. The final results of training the algorithm on the Iris dataset is displayed in table 6.2. Like with the previous tests, the results on this problem have been collected over a series of 25 independent runs of the algorithm on the exact same problem.

Test Accuracy			Validation Accuracy		
Average	Maximum	StDev	Average	Maximum	StDev
59%	98%	0.24	61%	100%	0.26

Table 6.2: *The test results of training the NETNCA algorithm on the Iris dataset. The full results can be found in appendix E.2.*

Studying the results of the classification as shown in table 6.2, it displays that it is indeed possible for the NETNCA algorithm to work within the classification environment, though with some very mixed results. The biggest problem in this case seems to fall on consistency, as shown by the standard deviation. The results between each run are drastically different with very little consistency between them. However, it is also interesting to see that while the results overall are very low, a few outliers have also been found. Out of the 25 runs, 3 of them had a test accuracy of over 95% with all of them also having a corresponding validation accuracy of 100%. From all of these tests the best performing individual from this experiment was one that ended up having a 98% test accuracy with 100% validation accuracy, meaning that it only classified 3 of out 150 samples incorrectly.

7 Results

This section describes the scores of the best performing output networks. In table 7.1 the different environments and their scores using different approaches are shown. CartPole, MountainCar, LunarLander and BipedalWalker are all solved environments, meaning there is a perfect solution to these environments that can reach a theoretical highest score. These solution scores are all gathered from the respective environments' GitHub repositories. For the unsolved environments Acrobot and Pendulum, there is an OpenAI Gym ladder[3] where people can submit their Deep Reinforcement Learning solutions to the environments, which will be used as substitution for the solution score. Any score within the range described here, which are the scores from best to worst on the rankings, will be considered a solution score. The table also displays several baselines made for the purpose of comparing the performance of the NETNCA algorithm with different approaches. Random values, which are values generated at random as the replacement for an output vector of a neural network. This technique is non-deterministic. This should provide a good baseline to compare if the NETNCA algorithm is learning, evolving and building to improvement or if it is just taking random actions. There is the random search baseline, which uses the NETNCA algorithm to generate networks, but skips the step of using neuroevolution to enforce learning on the NCAs. This should provide a good comparison to study if the learning step of the NETNCA algorithm provides any value to the results. The neuroevolution baseline is implemented with the neuroevolution subsystem of the NETNCA algorithm on a fully connected feed-forward neural network instead of an NCA. It uses this network to test fitness instead of growing networks. This should provide a comparison of whether the graph building subsystem of the NETNCA algorithm provides increased performance. Finally the last baseline is using NEAT, which as the state-of-the-art technique for evolving topology of neural networks works as an ideal goal for NETNCA to reach in terms of performance. Though it should be noted that NEAT also contains multiple different hyperparameters that should be tuned to the environment in use. This might explain why some of the NEAT results are lower then expected as shown in table 7.1.

Name	Solution	Ladder	NETNCA	Random values	Random search	Neuroevolution	NEAT
CartPole	500	-	500	23.34	500	500	500
Acrobot	Unsolved	[-42, -113]	-73.35	-496.56	-77.9	69.80	-95.38
MountainCar	-110	-	-100.44	-200	-120.43	-101.81	-191.93
LunarLander	200	-	148.29	-66.73	29.68	283.69	115.91
Pendulum	Unsolved	[-106, -152]	-178.85	-1164.31	-864.05	-141.54	-260.35
BipedalWalker	300	-	145.91	-22.44	12.57	18.90	-

Table 7.1: *Best achieved scores over 25 runs using different approaches. A run is different for the different approaches. For random value it is a single iteration, for NETNCA, NEAT and Neuroevolution it is until no improvement has been found for 40 generations and for Random Search it is 100 generations. The solution score was provided by OpenAI Gyms documentation and Ladder was provided by the Gym Leaderboard. The remaining values in this table are derived from the data in appendix D.2.*

7.1 CartPole

The first environment CartPole is also the easiest environment, as it requires no non-linearity to solve. To put it very simply, the AI only needs to evolve to learn, that when the pole is tilting to the right it has to move the cart to the left, and when the pole is tilting to the left it has to move the cart to the right. This problem can be solved sometimes in a single generation, and as such does not require evolutionary computation to be solved. This becomes evident as it can be observed that all techniques solves the environment, with the exception of random values.

[3] https://github-wiki-see.page/m/openai/gym/wiki/Leaderboard

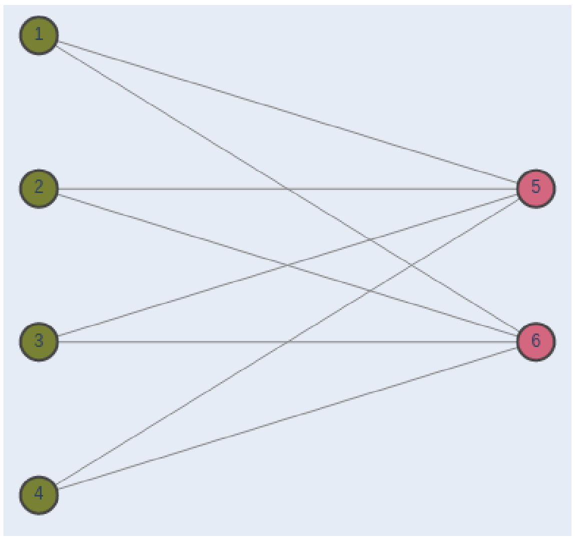


Figure 7.1: *Graph visualisation of a solution (500 score) CartPole network developed by the NETNCA algorithm.*

Figure 7.1 displays a very simple CartPole network developed using the NETNCA algorithm. This is a trivial network eg. there are no hidden nodes therefore the task can be solved linearly. As explained, all CartPole needs to do is move right when the pole is falling to the left and move left when the pole is falling to the right. However, it seems that this network has found an even simpler solution. Using the absolute value as the activation function in all the input neurons the network is unable to distinguish between whether the pole is falling to the left and when the pole is falling to the right. Instead it simply deploys a tactic of moving right when the pole has a high angular speed and moving left when the pole has a low angular speed. This effectively makes the cart oscillate, and when the output of the network is inspected a consistent pattern of alternating between right and left is observed.

7.2 MountainCar

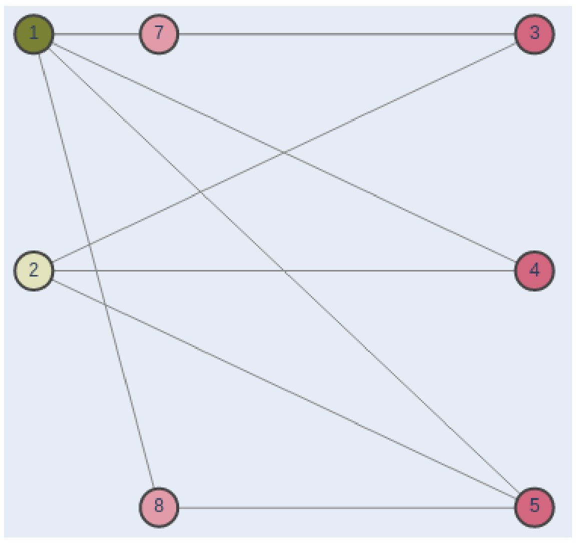
MountainCar is an environment that is hard to solve with just evolution. The reason for this is that no reward incurs before a solution has been found. This means that evolution will be no better than random search until the point that a network which can reach the top of the hill in the MountainCar environment is found and added to the population. Looking at table 7.1 it is shown that this was one of the environments wherein the NETNCA algorithm performed slightly better than neuroevolution.

	Best	Worst	Average	Standard Deviation	T-test with NETNCA
NETNCA	-100.44	-200	-151.48	41.04	-
Neuroevolution	-101.81	-200	-188.28	31.74	0.001
NEAT	-191.93	-200	-199.58	1.62	0.0000007

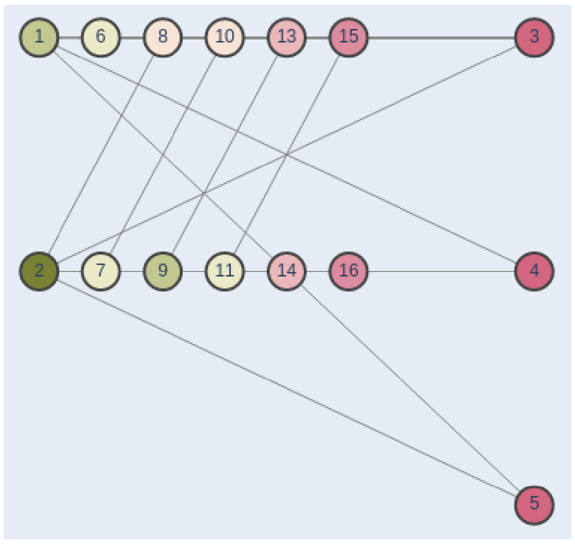
Table 7.2: *Performance measure for a comparison between NETNCA, Neuroevolution and NEAT on the MountainCar environment.*

Table 7.2 shows the performance of the different techniques on the MountainCar environment. While neuroevolution and NETNCA has an almost identical best network, it should be noted that only 3 out of 25 of the neuroevolution runs had a score above the minimum -200. Whereas NETNCA performed better in 17 out of 25 runs as can be seen in appendix D.2. This is evident from both the means and standard deviations of these two approaches. Furthermore, the NETNCA performs very well providing a solution with 10 scores above the required threshold to be considered a solution. Even the random search achieves some very impressive results in table 7.1. This indicates that it is not the evolutionary approach per say that allows NETNCA to solve the environment so efficiently, though it does bring it above that threshold of

-110, but instead the nature of the networks created by NETNCA and random search compared to networks generated with neuroevolution. Neuroevolution uses a traditional fully connected feed-forward neural network, whereas the models produced by NETNCA and random search are more akin to the networks created by a technique like NEAT. However, this theory seems lacking when viewing the comparison between NEAT and NETNCA. NEAT, much like neuroevolution, only had 3 runs that provided a better score than -200 with a top score of -191.93. This means NEAT was just able to reach the flag within the maximum 200 timesteps. This alludes to NEAT actually getting stuck in local optima, which is a surprising observation given the fact that NEAT has techniques like speciation to help with this. As mentioned a run for NETNCA and NEAT is defined as: when no improvement has been found in 40 generations. This means that even after finding a way to the top of the hill with the cart, NEAT was unable to improve upon this network, whereas NETNCA were able to improve it all the way to a solution. When looking at the p-value gained from running a t-test between the scores of NETNCA and NEAT it seems highly unlikely that the differences observed is due to chance. It should be noted that there is an unequal variance between the NETNCA and the NEAT set, therefore making the t-test less reliable. Another important aspect to take into consideration, is the fact that NEAT is highly configurable. This test used the default settings acquired from the NEAT library used to create the baseline. It may be that NEAT can perform much better if it is configured differently.



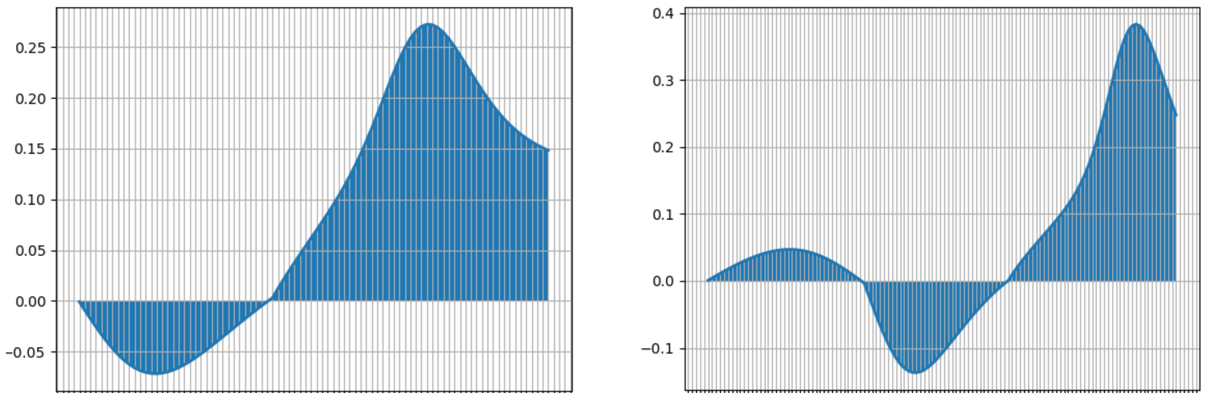
(a) *Gaussian network (score -100.44). All input and hidden neurons use a gaussian activation function. Output neurons use tanh.*



(b) *Sine network (score -118). All input and hidden neurons use a sine activation function. Output neurons use tanh.*

Figure 7.2: *Graph visualisation of two different solutions for the MountainCar environment generated using the NETNCA approach. The first input neuron is the carts position along the x-axis. The second input neuron is the velocity of the cart (negative being leftwards velocity and positive being rightwards). The three output neurons 3, 4 and 5 corresponds to the actions accelerate to the left, do nothing and accelerate to the right respectively.*

By studying the two well performing MountainCar networks shown in figure 7.2, it may be possible to understand what makes these networks unique. First of all it should be noted that for the purpose of this analysis the fourth neuron which corresponds to the action do nothing will be completely ignored, since any serious solution network would disregard this action. Starting with the well performing gaussian network, which uses the gaussian activation function in its input and hidden neurons. The second neuron, that is the one that takes the x velocity as input, is very strongly negatively connected to the third neuron and positively connected to the fifth neuron. This means that the AI wants to move the cart along the direction of its velocity. This by itself is actually almost enough to solve the environment. If the first input neuron were to be removed the network would still score an average of -113 over 100 runs. The first input neuron is more complexly connected but it basically boils down to the logic that when the cart is on the right the network want to move to the left, and when it is on the left the network wants to move to the right. This seems perfectly reasonable, since when the network has moved the cart enough up the hill to the left there is enough runway to create momentum to start moving to the right and vice versa. What makes this powerful is the fact, that as the velocity approximates 0 this rule will dominate the network. This means that it is not necessary for velocity to completely reach 0 to swap the direction of the velocity. All that needs to be done is for the velocity to approach 0. As such there is no wasted time waiting for the cart to completely stop before the network starts moving the cart in the other direction. This detail is what brings this network from a well-performing MountainCar AI to a MountainCar solution. What gives it the extra 10 score reaching a total of -100.44 when the solution is -110 is the fact that this rule of moving to the opposite side is so perfectly tuned that on runs where the cart start slightly to the right it is possible to reach a score of over -80. To explain, the cart starts around position -0.5 which is the center. Due to the bias in the first, seventh and eighth neurons, the network believes that numbers below -0.42 is right of the center and numbers above -0.42 is left of the center. Therefore, if the cart starts in -0.41, which is possible, the cart believes it is to the right and will start by moving left instead. This allows the network to fully take advantage of a lucky starting position and solve the environment in 30 timesteps faster than the average solution.



(a) *A run of the gaussian network where the cart starts in position -0.41.*

(b) *An average run of the gaussian network*

Figure 7.3: *The two graphs showcase the actions of the gaussian network in figure 7.2 given an optimal starting position and an average starting position. Positive values indicates moving to the right and negative values indicates moving to the left. The higher the value is the greater the difference between left and right was.*

By looking at the plots generated by the actions taken by the network in figure 7.3 it shows a smooth transitioning between moving to the left and the right as a result of the position of the cart having a larger effect and the velocity having a smaller effect as a result of deceleration when moving up a hill. Another interesting observation is that the graph generated by the actions of this network seems to take on the pattern of a wave whose frequency decreases and amplitude increases over time. Imagine for a moment that the height of the hills in MountainCar were infinite and that there was no cap on the velocity reachable with the cart. the graph would theoretically continue in this pattern indefinitely. This would mean that this network would be able to solve the MountainCar problem no matter the height of the hills as long as the environment is still physically possible.

In which way the gaussian activation functions of the network plays a role in its success is hard to know, but an attempt to change one neuron's activation at a time to a linear function had an adverse effect on the performance of the network. The first and second change saw a penalty of 5, whereas the third change dropped the score by an additional 50. One possible theory as to why the gaussian activation function is important could be due to the noise making the network more robust when dealing with the stochastic nature of the random initial cart position.

The second network shown in figure 7.2 is the sinus network. This network has evolved to only use the sinus activation function with the exception of the output neurons. This network displays characteristics that is common for a network created with NETNCA. The nature of the lower vertical positioning of neurons feeding into the higher positioned neurons shows that the nodes and connections between them are commanded by a CA ruleset generated by the NCA. This concept is explored further in section 8.1. As the network has approached a point of being incomprehensible for humans, there will be no attempt at human analysis here. However, it is interesting to see that this networks compounds sine functions on top of each other, which in theory should create a wave with very flat peaks and valleys. As have been observed before it seems that the graph generated from the actions taken by a solution network seems to resemble that of a wave. Therefore, the actions of the sine network should be able to be characterized as such as well.

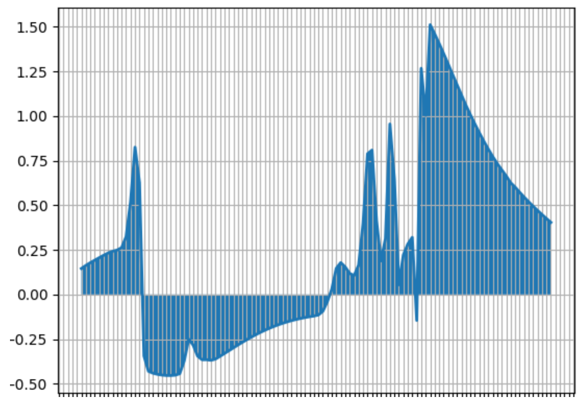


Figure 7.4: *The graph shows the actions taken by a run throughout the MountainCar environment with the sinus network. Positive values indicates moving to the right and negative values indicates moving to the left.*

Figure 7.4 displays the actions taken by the sinus network as a graph. The graph still resembles graph (b) in figure 7.3 but is much less smooth and much less sure in its actions. It can even be seen that on its second attempt up the right hill it took a single step to the left before it proceeded up to the right. This instability that was not seen in the gaussian network may very well be because of the lack of gaussian noise to give it that robustness. However, when looking at the very sharp incline right as the actions change from right to left the first time, it seems much more likely that this network has simply learned to just move in the direction of the velocity, without the finesse of the positional complexity. The weird swings observed going up and down within the one valley and two peaks could just be noise created by unnecessary computations within the network.

7.3 Pendulum

Pendulum is one of the hardest environments that there has been conducted experiments on in this thesis. As was mentioned in section 5, the continuous nature of its input makes it very hard to tune properly. Furthermore, the challenge of networks getting stuck in local optima is ever present. With this said the NETNCA algorithm performed extremely well on this environment.

	Best	Worst	Average	Standard Deviation	T-test with NETNCA
NETNCA	-178.85	-1008.35	-737.8348	199.5264	-
Neuroevolution	-141.54	-1080.2356	-693.6638	293.6994	0.545
NEAT	-260.35	-764.39	-616.44	133.85	0.02

Table 7.3: *Performance measure for a comparison between NETNCA, Neuroevolution and NEAT on the Pendulum environment.*

When viewing the comparison between NETNCA and neuroevolution in table 7.3, it is shown that NETNCA was slightly outperformed by the more simple neuroevolution approach though the t-test shows that this difference is statistically insignificant. Neuroevolution seemed to have higher extremes and therefore also a larger variance, whereas NETNCA kept the scores closer to the center of the distribution. This alludes to the possibility that NETNCA provides more consistent results than neuroevolution, though it is hard to say anything conclusive without more tests and perhaps a larger sample size.

Interestingly, NEAT seems to be outperformed by both neuroevolution and NETNCA. While the NEAT score seems to be much more consistent given that its worst score is close to the average of NETNCA, its peak performance is much lower than that of NETNCA. The best network achieved by NEAT in this experiment is insufficient to be considered a solution network since its score is outside the ladder range referenced in table 7.1. The p-value indicates that there is a statistically significant difference between the network populations of NETNCA and NEAT, which would allude to this performance difference not being a matter of chance.

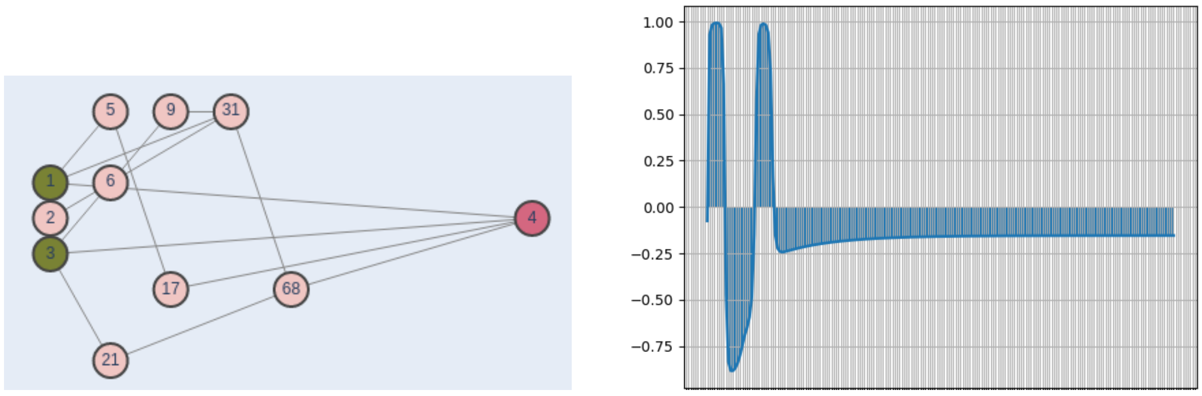


Figure 7.5: *A showcase of a Pendulum network and the actions taken by it in an average run (scoring -137). The first and second input neuron corresponds to the x and y position of the free end of the pendulum, or more precisely cosine and sine of the angle of the pendulum. The third input neuron corresponds to the angular velocity. The fourth neuron (the output neuron) indicates the amount of torque that should be applied to the Pendulum in the continuous range between [-2, 2]. Positive values in the action graph indicates counter-clockwise torque and negative values indicates clockwise torque.*

Figure 7.5 shows a Pendulum network developed with NETNCA. Pendulum is a more advanced environment, and with 7 hidden nodes it is quite tricky for humans to analyze exactly what this network achieves. By looking at the graph of the actions the behaviour was very similar to the one seen in the MountainCar action graph. In the run this graph portrays, the pendulum started in a downright position, applied torque to move to the right then to the left. Finally it used the momentum it had gained climbing up the left side to swing all the way from 9 o'clock in a counter clock-wise fashion all the way to 12 o'clock. It then proceeded to apply a small amount of torque to keep it in equilibrium. So the network must learn that when it is in an upright position it should apply a small amount of torque, and when it is not in the right position it should swing in a wave-like pattern. In the graph it is shown that all three neurons are connected, in some way, to neuron 68, which has a cosine activation function, which could be what produces the wave-like behaviour. Another property of the network is the gaussian activation function in neuron 3, 5, 6 and 17. For the MountainCar network it was hypothesized that the gaussian activation function may help smoothing the curve of a function. By changing the activation function of neurons which use gaussian noise to linear it is possible to observe the impact of certain functions on a particular network.

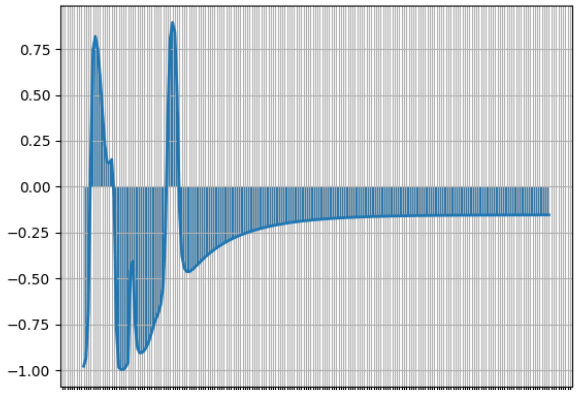


Figure 7.6: *Action graph from pendulum network in figure 7.5 where neuron 5 and 6 has been changed from using a gaussian activation function to using a linear activation function.*

Figure 7.6 shows the action graph from changing neuron 5 and 6 to using a linear activation function. The changes leads to a decrease in overall performance of the network by a total of 20 points on average. This indicates that the gaussian noise helps smoothing the network, which in turn improves its performance. What leads this network to succeed is: The combination of using a cosine activation function, which the inputs are fed into, to create a wave like behaviour, the gaussian noise to create a smooth curve, and a topology that is hard to achieve with other approaches.

7.4 Acrobot

Acrobot is one of the environments that does not have a known solution yet. What this means is that the maximum score attainable in this environment is still unknown. However, in the GitHub repository of Acrobot, it is noted that the environment is considered to be solved with a score above -100. As such it should be possible to test if NETNCA, NEAT, random search and neuroevolution are able to solve this environment with a score above -100 as seen in table 7.1. In this case the NETNCA algorithm has been outperformed by the more simple neuroevolution approach with its fully connected feed-forward network, though both neuroevolution, NEAT and NETNCA are all able to solve the environment.

	Best	Worst	Average	Standard Deviation	T-test with NETNCA
NETNCA	-73.35	-92.04	-78.73	4.27	-
Neuroevolution	-69.8	-76.56	-73.99	1.56	0.0000047
NEAT	-95.38	-101.21	-99.12	1.44	8.04e-27

Table 7.4: *Performance measure for a comparison between NETNCA, Neuroevolution and NEAT on the Acrobot environment.*

Studying the more detailed view of the Acrobot result data in 7.4, it can be observed that with a p-value of 0.0000047 allows one to almost conclusively reject the possibility that neuroevolution has performed better by chance. However, it should be noted that while neuroevolution objectively performed better than NETNCA, NETNCA is still far into the -100 threshold. Another thing to note is that Acrobot is an extremely easy environment to solve. In table 7.1 it is shown that even random search performed well within the threshold of solving the environment. Even more impressively, random values managed to complete the environment by getting a score higher than -500. Meaning it managed to solve the task of swinging the joint above the line, albeit very slowly, therefore the low score. Once again, the NETNCA algorithm seems to outperform NEAT scoring much higher much more consistently. Even the worst performing individual of the NETNCA population scores a higher value than the best performing individual of the NEAT population.

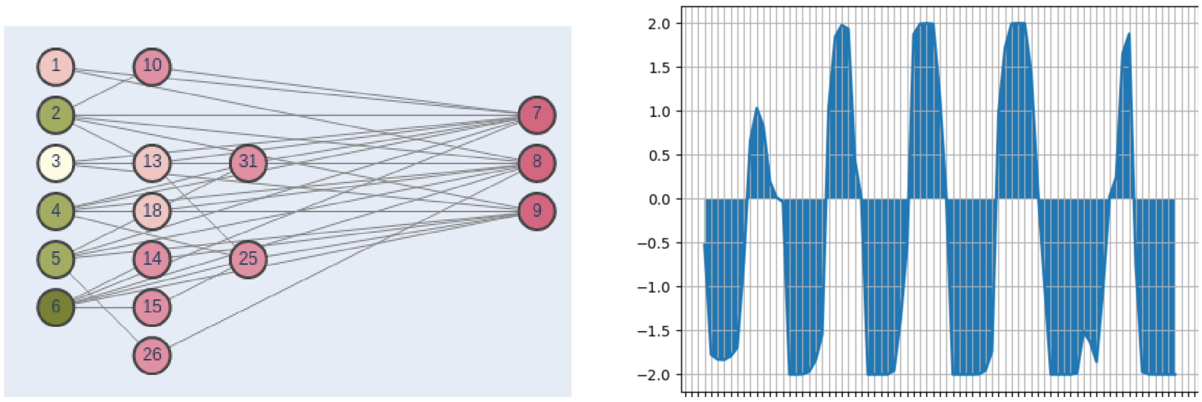


Figure 7.7: *A showcase of an Acrobot network and the actions taken by it in an average run (scoring -72). The first and second input neuron corresponds to the x and y position of the first joint, or more precisely cosine and sine of the angle of the first joint. The third and fourth input neurons corresponds to the x and y position of the second joint, or more precisely cosine and sine of the angle of the second joint. The fifth and sixth neuron corresponds to the angular velocity of the first and second joint respectively. The three output neurons corresponds, in order, to the following actions: Negative torque, do nothing and positive torque. Positive values in the action graph indicates counter-clockwise torque and negative values indicates clockwise torque.*

Figure 7.7 showcases the best performing Acrobot network developed by the NETNCA algorithm. Much like in both the MountainCar and Pendulum environments a wave pattern was observable in the action graph. Acrobot is a very similar though simpler environment than Pendulum. To solve the problem the AI just needs to swing the joint left and right until it has gathered enough momentum to swing the outer joint above the threshold line. This is the exact same task as Pendulum, however, the Pendulum task does not end after getting the joint above the threshold, but instead it also has to balance it there. Interestingly, neuron 1, 2, 3, 4, 5, 6, 14, 15 and 26 from 7.7 all make use of the cosine activation function, which much like the Pendulum network and the sinus MountainCar network previously discussed, is an activation function that generates a wave pattern as output.

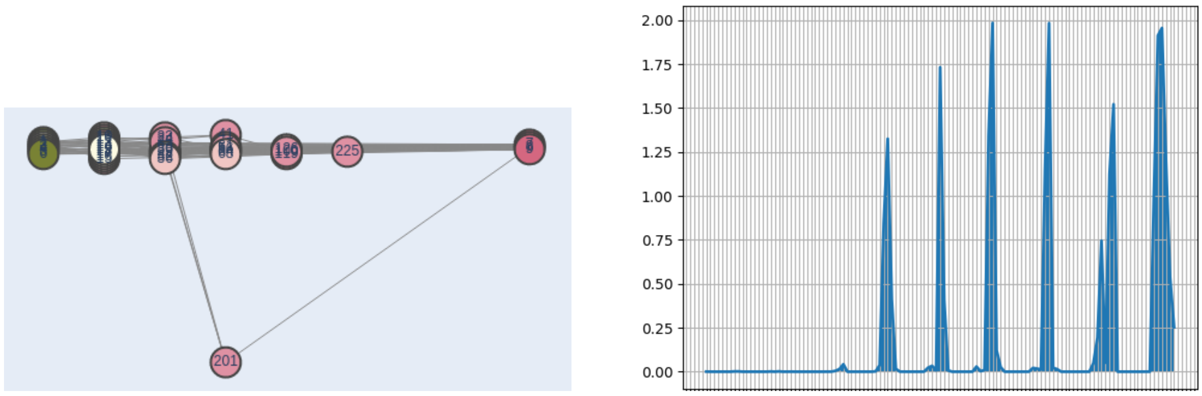


Figure 7.8: *An Acrobot network and its corresponding action graph on an average run (scoring -116). This network was created by using the high performing Pendulum NCA which created the network seen in figure 7.5.*

In fact, the similarity between Pendulum and Acrobot is so strong, that when using the best performing Pendulum NCA to produce an Acrobot network, the produced network manged to reach an impressive score of -116. Intriguingly, the network created using this method does not make use of the cosine activation function like the one in the network produced for the Pendulum environment. Instead it uses the sine activation function in 3 of its neurons. When studying the generated action graph for this network it is possible to see why it does not reach a score above -100. Instead of applying negative torque when swinging in the other direction, it just applies no torque at all and lets the gravity do the work of swinging joints in the opposite direction. While definitely a tactic that allows you to reach enough speed to get above the targeted threshold, it does slow down the network considerably since it does not use all the tools granted to it. In this case the ability to swing left. It should be noted that this does not mean that the NETNCA algorithm has somehow evolved to solve two different tasks, as it was never trained to solve two different environments to begin with, but rather that one of two things must hold true. Either the environments are so similar, that the ruleset created by NETNCA works for both environments, or Acrobot is such a simple environment that it can be solved by the Pendulum NCA randomly. While further experimentation would be required in order to test if these ideas holds, the authors of this thesis hypothesize that it may be a mix of both.

7.5 LunarLander

	Best	Worst	Average	Standard Deviation	T-test with NETNCA
NETNCA	148.30	-11.53	62.53	44.37	
Neuroevolution	283.69	-110.93	125.49	130.02	0.03
NEAT	115.92	23.60	64.91	22.30	0.81

Table 7.5: *Performance measure for a comparison between NETNCA, Neuroevolution and NEAT on the LunarLander environment.*

LunarLander is one of the environments that NETNCA was unfortunately unable to solve completely. When viewing the data from table 7.5 NETNCA is outperformed by neuroevolution. It is exceedingly hard to try to make a guess as to why LunarLander was not solvable with NETNCA. The environment is seemingly easier than something like Pendulum that has a continuous output space. One explanation could be the large input space. LunarLander has 8 input neurons and the test was run with a 9x9 neighborhood. This means that the neighborhood of something like neuron 8 would not include neuron 1, 2 and 3. This alludes to the fact that NETNCA may have issues when the input and output space increases to higher numbers. In order for the bottom input neuron to have the top input neuron in its neighborhood and vice versa, a neighborhood size of 15x15 would be required. This would mean the NCA would need 683 input neurons and 459 output neurons. Such a large neighborhood would exponentially increase the amount of neurons. Each neuron passed through the NCA has the potential of generating 225 neurons. In case of 7 growth iterations the maximum amount of neurons that could be generated from the initial LunarLander network would be $8 * 225^7$ leading to the creation of computationally infeasible networks. The networks may simply become too big to run the environments in a reasonable time. The danger of overflowing errors is also present in such large networks. As such the NETNCA algorithm may experience similar limitations as NEAT when it comes to the input space. In fact as presented in 7.5, the NETNCA approach actually performed better than NEAT, though not in any significantly meaningful way.

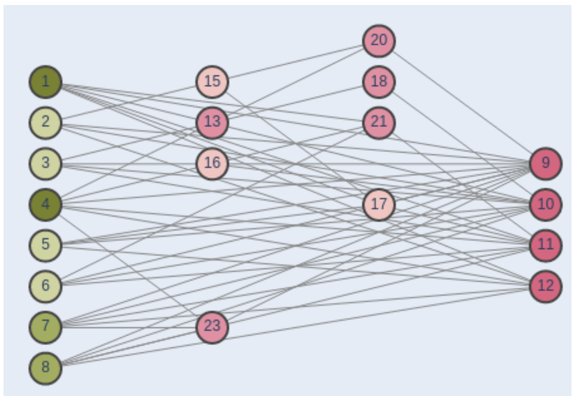


Figure 7.9: *The graph of a LunarLander network produced using NETNCA (-147 score). First and second neuron observes the x and y position. Third and fourth observes the linear velocity (x and y). The fifth neuron observes the angle, whereas the sixth neuron observes the angular velocity. The seventh and eighth neuron observes the contact with the ground for each leg. The four output neurons corresponds to the following actions: do nothing, fire left orientation engine, fire main engine and fire right orientation engine, respectively.*

When studying the topology of a LunarLander network produced by the NETNCA algorithm as shown in figure 7.9 it becomes evident that a human analysis of how this network works is very hard. There are 42 connections with different weights and 16 neurons with their own biases leaving a total of 58 parameters to consider when doing an analysis. Furthermore, since the output of LunarLander is 4-dimensional it becomes hard to try and pick up on a pattern using something like an action graph as done in previous experiments. Instead this network was analyzed by observing the visualisation of running the network on the LunarLander environment. From that it became evident, that while the network was indeed able to land the flying machine on the designated area without much fault, it had an issue whenever it was landing on an angle. When just about to land with an improper angle it would use a side engine to correct this, but often it would over correct and crash the side into the ground. The parameters that help steadying the craft must have been overtuned in such a manner that it overcorrected the problem, and also underprioritized since it only seemed to correct the angular problems right as it was about to land. Fixing the angle of the craft in the air, while theoretically not being a more optimal way of doing it, would allow overcorrection to not end in a crash. Another issue is that in about 1/3 times when the LunarLander successfully lands it will keep firing its side engines. The 7th and 8th input neurons which fire when the left and right legs touch the ground should be strongly connected to the 9th neuron. This is the output neuron, which controls the do nothing action. Continuously firing the engines after landing makes the run persists and as such a time penalty is incurred as a result. On further inspection, the 7th and 8th neurons are both connected to the 23rd neuron which is connected to the 9th neuron. The connections are weighted in such a way that when 7 and 8 fire, 23 will fire very strongly for the 9th neuron. While this seems correct, the tuning may still be lacking, since this is not the behaviour that is observed.

7.6 BipedalWalker

BipedalWalker is the environment used with the largest initial network. BipedalWalker has 24 input neurons and 4 output neurons each representing a continuous motor speed value for one of its 4 joints. For this reason, just like with LunarLander a neighborhood size of 9x9 was used instead. However, this neighborhood size may have been proven to be insufficient as well.

	Best	Worst	Average	Standard Deviation	T-test with NETNCA
NETNCA	145.91	1.54	23.26	35.69	
Neuroevolution	18.90	6.43	8.74	3.29	0.20
NEAT	-	-	-	-	-

Table 7.6: *Performance measure for a comparison between NETNCA, Neurovolution and NEAT on the BipedalWalker environment. The data used for this is highly unreliable. The Neuroevolution set only contains 11 networks, meaning the p-value is prone to error due to an insufficient sample size. NEAT was unable to finish a single run in the environment over a period of 3 days and the data collection for NEAT was therefore cancelled.*

While being unable to solve the environment, it can be seen from table 7.6 that something did go right. Reaching a score of 145.91, while only being half of the 300 solution score, is actually quite impressive. The NETNCA algorithm was able to grow an AI capable of walking the BipedalWalker, which is not that easy to do. As mentioned Pendulum was an exceedingly hard environment to solve. As an example NEAT was unable to reach as high a score as neuroevolution and NETNCA with its default settings within the parameters of the test. This is attributed to the fact that Pendulum outputs continuous values, making precise tuning of the network extremely important. However, Pendulum only had one output neuron, BipedalWalker has 4 making the task harder to solve.

As can be seen in table 7.6 the data for NEAT and neuroevolution is inconclusive. NEAT and Neuroevolution might be able to solve BipedalWalker, but there was an issue when running this environment that was not solvable. Therefore, the neuroevolution set only has 11 networks, which is insufficient for a proper t-test. NEAT on the other hand completely failed to run a single run on BipedalWalker and there is therefore no data to compare with.

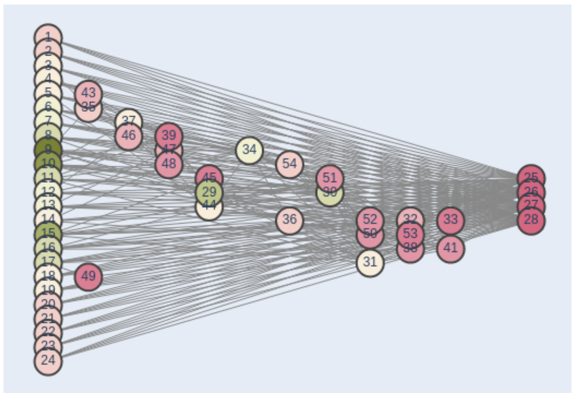


Figure 7.10: *The graph of a BipedalWalker network produced using NETNCA (145.91 score).*

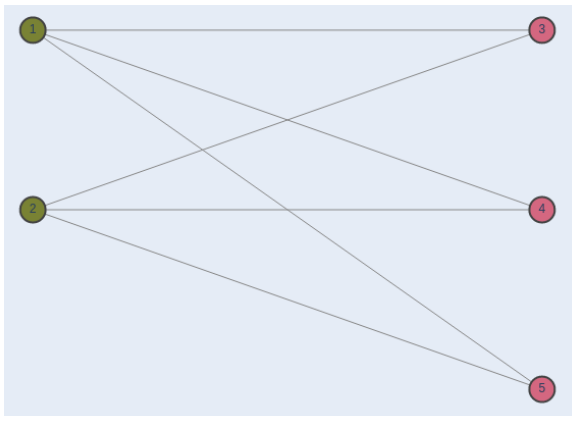
Figure 7.10 shows the network produced by the 145.91 scoring BipedalWalker NCA. The problems with the size of the input neurons compared to the neighborhood size discussed in this section as well as section 7.5 becomes very clear. For this experiment a 9x9 neighborhood was used. However, as can be seen there are 24 input neurons. This means that for the lowest input neuron (24) to be able to see the highest input neuron (1) in its neighborhood the size of the neighborhood would have to be 46x46. The highest amount of neurons that can be grown by such a network amounts to $24 * 2116^5$. While a very large number of these neurons would be duplicates and therefore not counted, the number of neurons would still, like with the example for the LunarLander AI, make the network computationally infeasible to run. Another important thing to note is the connections between input and output. The default and initial weight is always 0, meaning all connections are in practice inactive at the beginning. Neuron 24 does not have an output neuron in its neighborhood meaning its connection to the output is actually a dead connection from the very start of the growth. When looking at the generated BipedalWalker network in figure 7.10 it can be seen that the network is very sparse on neurons, and many of the input neurons in the extreme ends which have 0 weighted connections to the output are not connected to any hidden neurons. This means that some of the input neuron values are actually being ignored in this network, which definitely has an adverse effect on the network's performance. This makes it apparent that BipedalWalker needs a larger neighborhood size than 9x9 to succeed.

Despite large amounts of input data being ignored by the BipedalWalker network, it was still able to walk the AI quite impressively. So while the large input size presents a challenge to the NETNCA algorithm, the promising initial results of NETNCA on BipedalWalker warrants further experimentation into the configurations, to see if there is a way for NETNCA to overcome this problem.

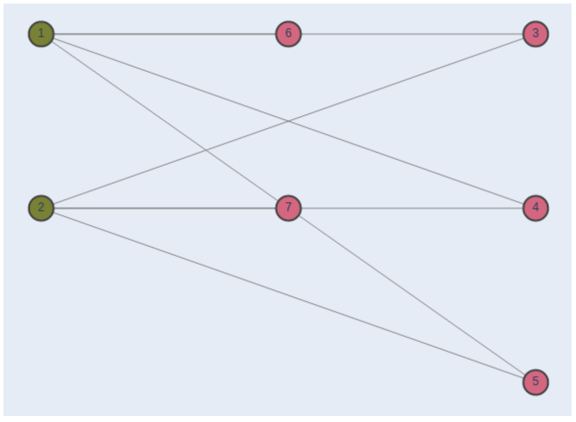
8 Discussion

8.1 Analysis of the growth iterations of a MountainCar network

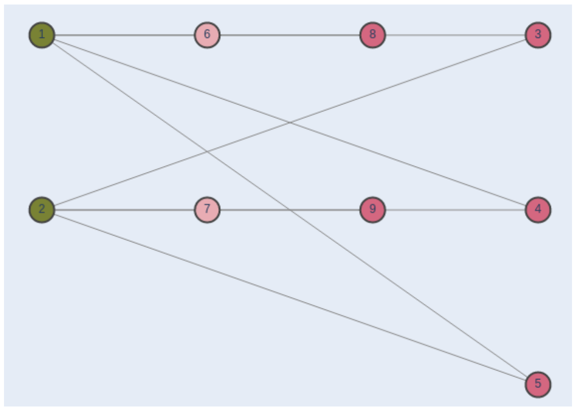
Throughout this thesis it has been demonstrated that the NETNCA algorithm is capable of creating neural networks that can solve classification and complex control problems. It has been shown that the networks produced perform well in some of the environments described in section 5. The outcome of the NETNCA algorithm is not in question as the results shown in section 7 speak for themselves. However, another important question does need do be answered: Does the NETNCA algorithm actually leverage the power of locality with the use NCAs? Figure 7.2 (b) presented the sinus MountainCar network. In that section it was argued that this network showed characteristics that was common for an object generated by using an NCA. To further examine this phenomena this section will present a walk through of the growth iterations of this MountainCar network as it was being generated by the NCA.



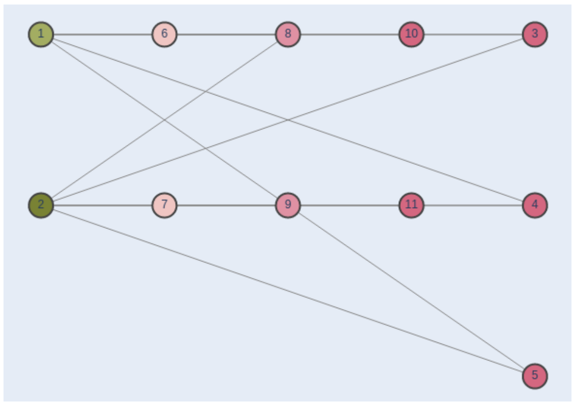
(a) *0th growth iteration*



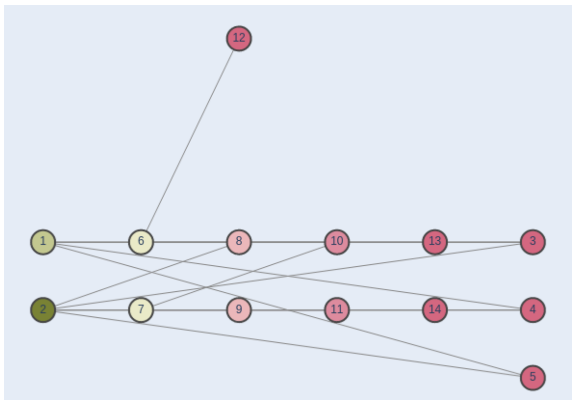
(b) *1st growth iteration*



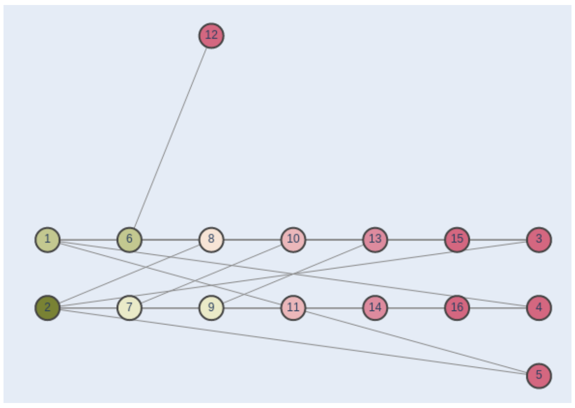
(c) *2nd growth iteration*



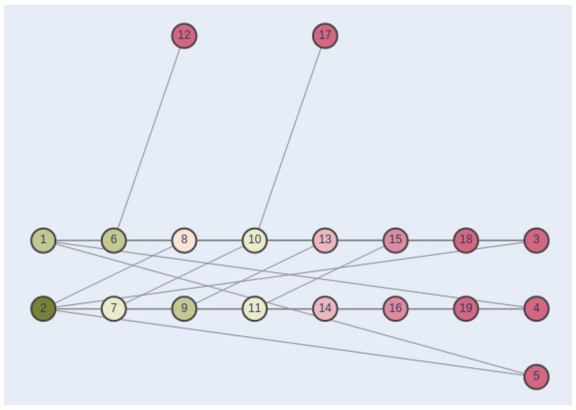
(d) *3rd growth iteration*



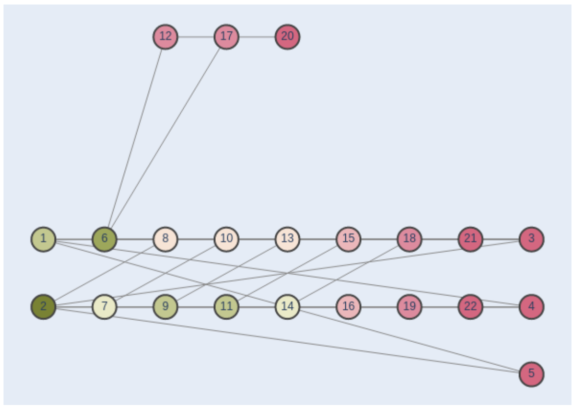
(e) *4th growth iteration*



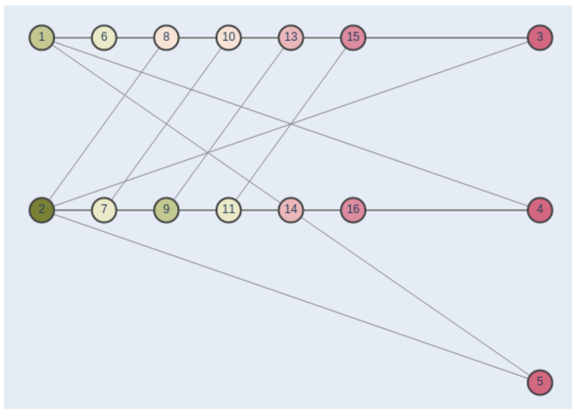
(f) *5th growth iteration*



(g) *6th growth iteration*



(h) *7th growth iteration, final step before prune*



(i) *Final network post-pruned*

Figure 8.1: *The figure shows the MountainCar sinus network from figure 7.2 (b). The first step (a) shows the initial graph, only having the input and output neurons fully connected with weights set to 0. Activation functions in input are linear by default and the output activation functions are tanh by default. Each following figure shows the network as all the neurons and their neighborhoods have been passed in batch through the NCA. The final image shows the final network post-pruned. The network was grown with a 7x7 neighborhood size.*

Looking at figure 8.1 the growth of a MountainCar network can be observed in each step. The network starts with its initial input and output neurons fully connected. The input neurons at this point both have the default linear activation functions. The output activation function is tanh and will never change since control over the range of the values outputted by the output neurons is required by the environments. In the first growth iteration the first hidden neurons can be observed. Neuron 6 and 7 both seem to have been created from the same rule since they appear in the same pattern. It is hard to say at this point exactly what this rule is, but it could have been simple like [1, 1, 0] -> [1, 1, 1]. (Meaning state [1, 1, 0] transitions to state [1, 1, 1]). Since the node is in-between the input and the output neuron, the new neuron must have been placed behind the output neuron on the same horizontal positioning. This would in turn push the output to the right of the new neuron, such that the output neurons keep their positions in the last horizontal ordering layer. Furthermore, neuron 1's activation function was changed to sine and neuron 2's activation function was changed to sigmoid.

The second growth iteration (c) seems to repeat this pattern. Here it is possible to see the two new hidden neurons. Neuron 8 and neuron 9. It is important to note here that neurons 1 and 2 are not connected to 8 and 9. This means that neurons 6 and 7 must have been the ones to generate these new hidden nodes, meaning they might have used the same CA rule that neurons 1 and 2 used to generate themselves in the previous iteration. It should also be noted that neurons 6 and 7's activation functions were also changed to the sine activation function.

The third growth iteration (d) again sees a repetition of the same rule that generated neurons 6, 7, 8 and 9 by having neurons 8 and 9 generate neurons 10 and 11. Once again neither neuron 1 nor neuron 6 is connected to 10. Following the same logic neither neuron 2 nor 7 is connected to neuron 11. However, neuron 1 is now connected to neuron 8 and neuron 2 is now connected to neuron 9. This means that while 1 and 2 were not able to produce neurons 8 and 9, they are able to connect to them. This is only possible due to the configuration having a different threshold for when a new neuron can be created and for when a connection can be established. The threshold for generating a new neuron is in this case higher than the one for connecting to an existing neuron. The third growth iteration also introduces one diagonal connection with neuron 2 connecting to neuron 8. In this iteration neurons 8 and 9 had their activation functions changed to sine.

The fourth growth iteration (e) continues the pattern with neurons 13 and 14. An interesting observation is that these rows of neurons first connect to output when there are 2 neurons between them and the output. For example, in this iteration neuron 10 connected to neuron 3, however neuron 13 did not. In the next iteration neuron 13 is going to connect to neuron 3 as well. This rule would look something like [... self 1 1 1] -> connect self with rightmost 1. In this iteration the activation function with sine continues in neurons 13 and 14. Interestingly in this iteration neuron 2 changed its activation function from sigmoid to tanh. This iteration also introduces the first new neuron created by another rule than the one that created all the other neurons. Finally, it can also be seen that neuron 7 connected to 10 by, seemingly, the same rule as the one that connected neuron 2 to 8.

The fifth growth iteration (f) repeats the same rules as before. Creating neurons 15 and 16, connecting 10 and 11 to their respective outputs as well as a new diagonal from 9 to 13.

The sixth growth iteration (g) yet again repeats all the patterns that has been observed so far. Diagonal connection from 11 to 15. 18 and 19 created in the forward going pattern. 17 is created with, seemingly, the same rule as 12 was. 13 and 14 both connect to their respective outputs following the [... self 1 1 1] -> connect self with rightmost 1 rule. It should be noted that this rule is not only used to connect to the output neurons. The same rule has connected 7 to 15, 6 to 13 etc.

The seventh and final iteration (h) shows the network before pruning is applied. It has repeated all the same patterns as before, however something completely new has happened as well. Neuron 12 has connected to neuron 17. At the same time the connection between 17 and 10 has been cut, meaning neuron 17 now has a new partial ordering. On top of that neuron 17 has generated neuron 20 in the position where it itself was the previous iteration.

Finally as shown in the final post-pruned network (i), since the neurons of 12-17-20 was disconnected from the output they were pruned from the network, as they would have no effect on any results. Also since there are not 2 neurons between the output neurons and neurons 18, 19, 21 and 22 the neurons were never able to use the, [... self 1 1 1] -> connect self with rightmost 1, connection rule, therefore they are also unconnected to the output and has been pruned.

An important observation to note here, is that even though the partial re-ordering of the graph at each iteration keeps changing positions of neurons as described in section 4.2, the pattern created remains consistent. Hence, the human intervention in form of algorithms enforcing some global rules does not interfere with the growth of the network. Neurons 6, 7, 8, 9, 10, 11, etc. are all neurons created by creating a neuron to the right of the output neuron, and then having the output neurons moved as the partial ordering has changed. So even though the node was moved it does not stop the pattern from repeating over and over again, proving that the global control does not destroy the local control that has given rise to these patterns.

From this analysis there is no doubt that the networks are not just randomly put together, but carefully crafted using the CA ruleset created by the NCA. Though the NCA that created this network may be too complex for human analysis, being a fully connected feed-forward network with 1282 neurons, it is very clear to see these patterns emerge from the growing of the graph. In fact when applying more growth iterations it further solidifies the presence of these patterns.

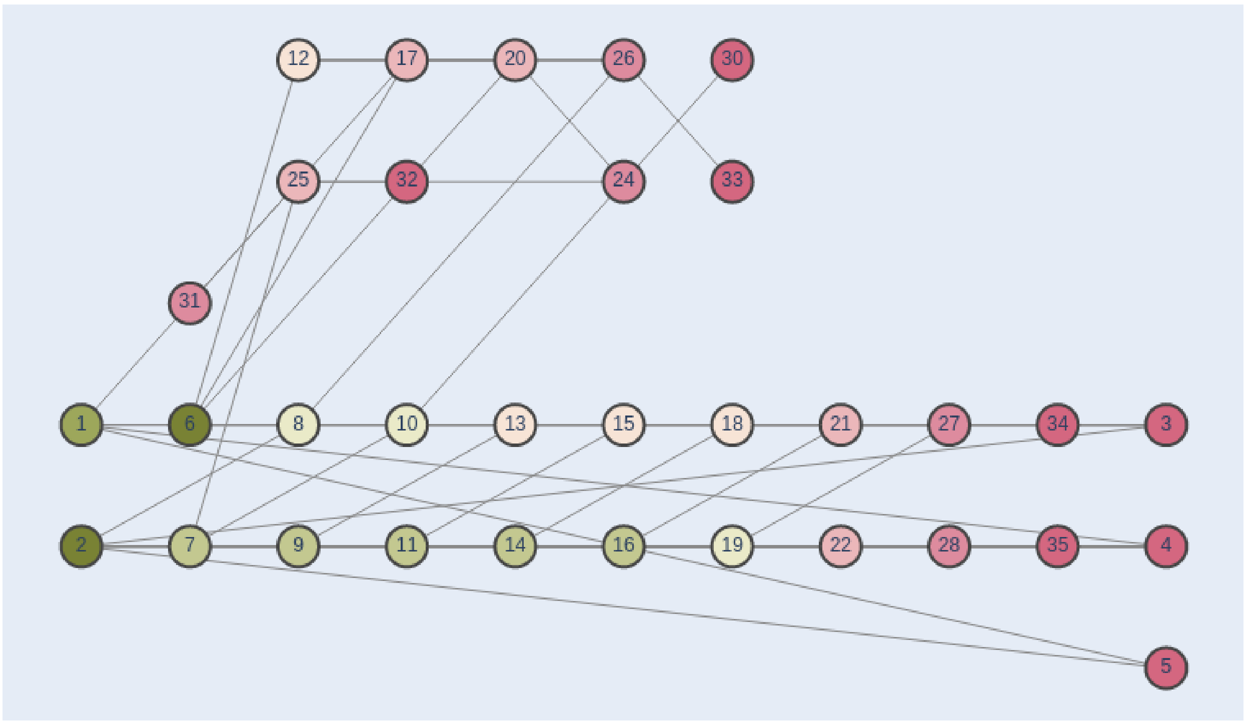


Figure 8.2: *The 9th growth iteration for the network presented in figure 8.1.*

In figure 8.2 the network has been grown for 9 iterations instead of the 7 it was evolved to do. The interesting part of this network is how a variation of the same pattern that grew the original network is starting to form in the 12-17-20 subgraph. This is not an exact replication of the original pattern, but that is well within expectations. With 1282 neurons the ruleset that governs this CA is incredibly complex and there are many more replication rules in this network than the 5 patterns found in this analysis. Furthermore, the original pattern is still just repeating. Moving the graph further and further right pushing a line of neurons ahead. The same diagonal connection from 7 to 10 can now be observed from 19 to 27.

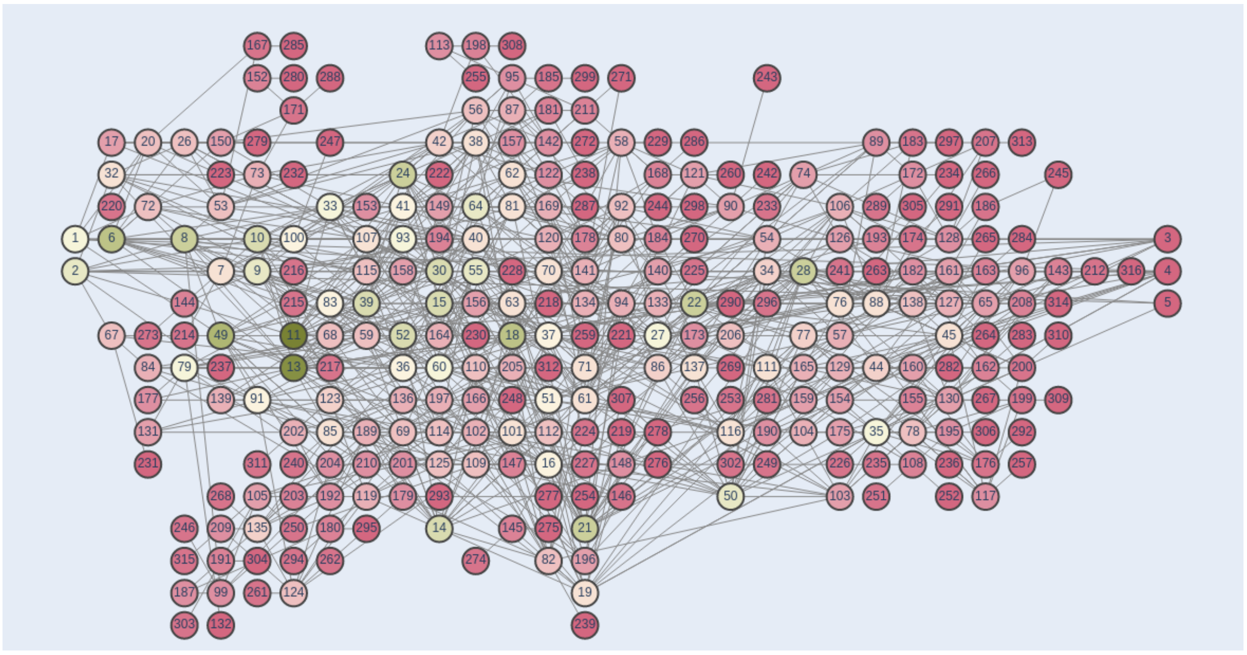


Figure 8.3: *The 15th growth iteration for the network presented in figure 8.1.*

One could be led to believe that this pattern would continue infinitely. However, while that would certainly have been possible, it is another trait of CAs that they are undecidable. Figure 8.3 shows the network after 15 growth iterations. The patterns observed up until now has been completely broken and the network looks random. The reason this happened is that the newly created pattern seen in figure 8.2 eventually fed back into the original pattern, completely changing the neighborhood for all the neurons leading to a completely different part of the ruleset generated by the NCA to be used.

8.2 NETNCA performance

Throughout section 7 the results of running tests on the environments was presented. This was done alongside several different baselines in order to compare the technique with already existing methods. These included the solution score, the best OpenAI leaderboard, random values, random search, neuroevolution and NEAT. For the control environments, NETNCA was able to reach a solution/leaderboard score in 4 out of 6 cases. The environments NETNCA was unable to solve, LunarLander and BipedalWalker, both had a larger input space than the neighborhood size that was used could cover. It should be noted that while NETNCA did not reach a solution for these environments the algorithm still performed very well. The BipedalWalker AI was able to walk, though not fast enough to reach the end, and the LunarLander AI could land on the designated target without much fault. Nevertheless, NETNCA seems to have some limitations that are very akin to the ones experienced with NEAT. Very large input and/or output spaces are hard to handle, since connecting two input neurons in a coherent way becomes harder if their spatial positioning is far apart. This can be alleviated by a larger neighborhood size, but this has an exponential growth to the size of the network produced by the NCA. Section 8.1 displayed how the NCAs used in NETNCA makes use of spatiality and the concept of locality. It is very clear to see that rules follows from a spatial pattern inside the local neighborhoods. However, how is an NCA supposed to govern reproduction rules that makes sense if the neighborhoods are too small to gather the relevant information? As such output networks with large input spaces have an adverse effect on the performance of the NETNCA algorithm. Future research into the NETNCA algorithm could try find how to arrange the input neurons in the grid in such a way that it compresses the information more efficiently. Though this would not fix the problem per say, it may be advantageous to allow input neurons to have other horizontal orderings than the first, allowing them to be placed to the right and left of each other. This would make it easier for NETNCA to solve something like

LunarLander.

When looking at the 4 control environments that was solved by the NETNCA algorithm some impressive results can be observed. NETNCA performed statistically significantly better than NEAT on MountainCar, Acrobot and Pendulum. Furthermore, NETNCA outperformed neuroevolution in MountainCar and BipedalWalker, and was only slightly worse than neuroevolution in Pendulum and Acrobot environments. Though NETNCA is still a novel technique it has shown great promise in the area of control problems. While a standard neuroevolution algorithm still seems to outperform the NETNCA algorithm in most control problems, albeit only very slightly, NETNCA consistently achieved better results than NEAT. It is worth also noting here that due to NEAT using things like innovation numbers and speciation it is much more durable when it comes to genetic algorithm issues such as local optima and the competing conventions problem. This is evident from the data in section 7 as NEAT consistently has a lower standard deviation, hence more consistent results. Despite this very powerful idea, which is not used by NETNCA, NETNCA still manages to outperform the maximum scores of NEAT. It can be argued that consistent results are a good metric for the performance of an algorithm, but the authors of this thesis would argue that when it comes to developing AIs to solve specific problems, the only thing that matters is the absolutely best AI developed. It is also important to note here, that in order to say anything conclusive about the performance between NETNCA and NEAT further experimentation would be required. Both larger sample sizes and experimentation with the hyperparameters of the NEAT algorithm, would be needed to make a fair assessment of each algorithms strengths and weaknesses.

Another interesting aspect of NETNCA observed during the experiments is that there seemingly is a connection between the ruleset created by the NCA and the environment for which the NCA was evolved. While it may seem obvious that there is a relation between the NCA and the environment it attempts to solve, since it is specifically evolved to grow networks capable of solving a specific environment, this relation seems to go a bit deeper. In section 7.4 it was found that the best performing Pendulum NCA was able to grow incredibly well performing Acrobot networks. This alludes to the fact that the environments of Pendulum and Acrobot may be so similar that the ruleset evolved for Pendulum works for the Acrobot environment. This makes intuitive sense, Pendulum and Acrobot in many way presents the same problem to solve, Pendulum was just a bit more complex. Even the input vectors of Pendulum and Acrobot are very similar. Both have x and y positions as well as angular velocity, Acrobot just has twice as many values for each property since it has two joints instead of one. The output is also similar, mapping to torque in either direction. The difference here is that Pendulum uses continuous values where Acrobot uses discrete torque. While it is possible that this happened by chance, the authors of this thesis urges for further research into the potential of generating NCAs with rulesets that are general enough to solve multiple different environments that have similar components. This could potentially be an important step in the development of AIs capable of solving more than one very narrow problem to instead being able to solve multiple problems with similar traits. The prospects of such an AI is exciting and would be a great stride forward in the development of AIs.

Another observation about NETNCA seen throughout section 7 is its proclivity towards non-traditional activation functions. Traditionally, neural networks use activation functions like tanh, step, sigmoid and relu. While cosine and sine are differentiable and therefore valid when used in backpropagation they are very rarely used for activation functions. NETNCA, on the other hand, almost always uses either sine or cosine in its solution networks. It was observed that many environments require a wave pattern-like behaviour to solve its task, which makes sine and cosine powerful activation functions to use. The framework used for NETNCA had a total of 10 activation functions available to it. Future research could look into expanding this list and allow for even more non-traditional networks. Since backpropagation are not used in this technique, there is no reason that non-differentiable functions could not be used as activation functions in NETNCA.

It was also fascinating to examine that the NETNCA algorithm could not only do control problems, but also a simple classification problem as shown in section 6.3. Even though the results of the experiments were very varied, it did mange to achieve quite a high accuracy within the environment. This did show that it was possible to do classification, but it should be tested further with more complex environments and with different hyperparameters. Much of the experimentation of classification was done with parameters similar to those used with the control problem, however, these are very different environments and could need very different parameters to solve.

8.3 Runtime

NETNCA, just like neuroevolution, is highly parallelizable. No attempts were made to try and gauge how fast the NETNCA evolution loop runs, since it is highly dependent on the amount of cores available and the complexity of the environment. A potential bottleneck that NETNCA has which neuroevolution does not have is the growing of the network. However, since networks are grown in batches for each iteration this process is also parallelizable. Therefore, there is no noticeable difference in runtime between traditional neuroevolution and the NETNCA algorithm.

8.4 Crossover

While evolution is a very important part of NETNCA, the technique presented in this thesis fails to take advantage of the idea of crossover. NETNCA has the same problem of mixing neural networks as the standard neuroevolution approach. There is no obvious way to take two good networks and combine them in any way that makes sense without the use of something like NEAT's innovation numbers. While there are different ways to create offspring using standard neuroevolution, the results often resembles that of creating random networks like in random search. Even something as simple as taking one network and then substituting a single neuron in that network, and then swapping them for a single neuron in another network leads to nonsense results. This approach also requires that the networks in the population have the same topology, and while that is true for the approach used in this thesis, it is not a requirement for the NETNCA algorithm to work.

9 Future Work

The current work on the NETNCA algorithm has shown some promising results as discussed in the previous sections of this thesis. However, there are still more work that can be done in order to both optimize the algorithm itself and expand upon it. This section will express some interesting areas where it could be possible to do this with the current implementation. Though it should be noted that this is of course not an exhaustive list as there are many cogs in this machine that could be changed for better or worse.

9.1 Different types of ANNs used for the NCA

In the time of writing NETNCA only makes use of a fully connected feed-forward neural network for its NCA. While this has showed promising results, it might be interesting to experiment with using other kinds of neural networks for the NCA. For instance, in (Mordvintsev et al., 2020), they use a CNN when they train NCAs with backpropagation to produce some target images. This approach allows the NCA to convolve and pool the structure of the output network. This might for example lead the NCA to discover more translation invariant rules about the networks, such that slight differences between neighborhoods with essentially the same structure, might still produce the same changes, which could be an improvement.

Using Recurrent Neural Networks (RNN) for the NCA, could also potentially lead to improvements. Allowing the NCA to be aware of the sequence of changes to a neighborhood to reach the current state might allow it to build more complex structures more easily. There are many different sequences of steps you can take to go from one state to another. Without a RNN, once the NCA reaches a state, it wont know how it got there, and would therefore always take the same actions next. An RNN however would allow the NCA to be aware of the sequence of actions, and thus it might take different actions based on the previous actions. It is not hard to imagine this information allowing the NCA to more easily build structures that might take several actions to realize.

Along the same line of having some knowledge of the specific neighborhoods, having many small interconnected NCAs, one for each cell in the grid, as they did in (Elmenreich and Fehérvári, 2011), would also be interesting to experiment with. This approach is similar to using a CNN, as each small NCA would correspond to a single kernel. The difference is, that while the kernel will have the same weights for every part of the grid, the NCAs would only start with the same weights, but could then evolve differently. The problem with this approach, is that the size of the grid is not known beforehand, so the number of NCAs needed is unknown. Solving this by letting the number of NCAs be dynamic depending on the changing size of the grid, would make it difficult to evolve.

9.2 Specialized NCAs

The current implementation of the NETNCA algorithm only makes use of a single NCA to train and learn all the nuances and possibilities of the output network. As expressed earlier in section 4, the NCA has to learn removal and addition of connections, modification of the bias for nodes, modifications of weights for connections and also attempt to find the best activation function. While all of these parameters are related with one another, it might overwhelm the NCA with the sheer amount of different actions, where some of them have vastly different results on the output network. Instead it could be interesting to study the effect of moving each part of the building responsibility to separate NCAs, so there would be one NCA responsible for learning about bias modifications, another trained just for activation functions and one that only focuses on connections. Even the NCA for connections, could be split into separate ones as well, one to add connections, one to remove connections and one to change weights. It does become a bit overwhelming with the amount of NCAs, but it could have a positive effect, where the NCAs might perform faster, since there is less for each NCA to learn. It also opens up for the possibility of switching the specific NCAs between the different individuals in the population, i.e. crossover. All of this should be manageable within the framework already presented in this implementation.

9.3 Better placement of vertical positions

In section 4.2.2 it was described how the vertical ordering of nodes were calculated. The different rules used to determine where a node should be placed, if conflicts would arrive with other nodes in the graph, were explained. However, the chosen method was quite simple and therefore might not have been the best choice, as in some cases it would end up sending nodes far away from their starting positions. This occurred if an already moving node kept getting conflicts with other nodes. Another slightly more complex rule could have been to swap moving nodes depending on the distance they have moved from their original position in the graph. This would help equalize the moved distance for nodes in a single ordering. A few nodes might end up moving a bit more, but it would help with the nodes that might have moved out of the neighborhood range and creating dead connections.

Another change would be to also allow nodes to move upwards instead of only downwards. So in cases where a node might have to move down several positions, possibly exiting the neighborhood range of other nodes, it could instead have moved up and kept the graph more stable. This is displayed in figure 9.1 where node F following the current rules would have a position below node E, but by using the upwards rule, it can stay close to node B. This can also have large impacts on the other neighborhoods. Depending on the size of the neighborhood, node F might have entered node C's neighborhood. So a new rule could be defined as moving vertically to the position that would be the closest to the original position, though this would increase the computation time, since it would need to check the result of both directions of movement instead of only one.

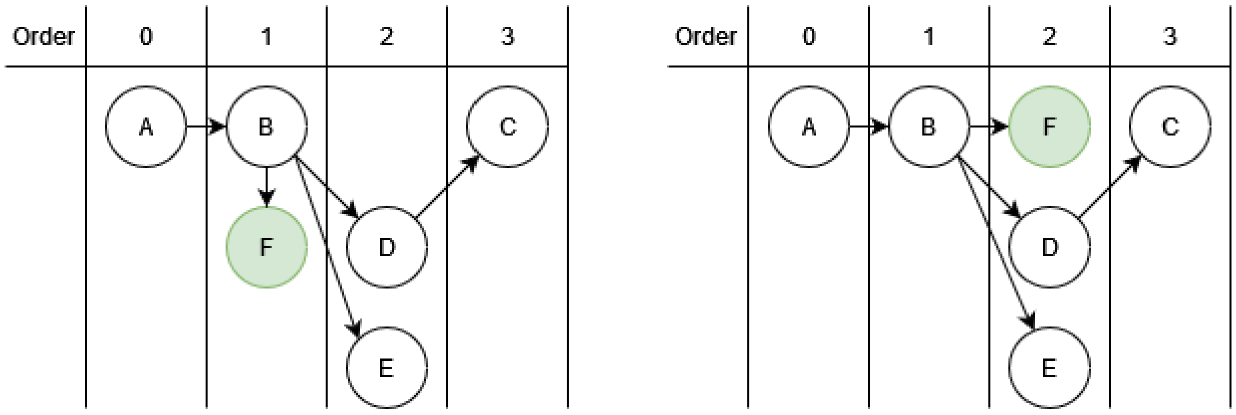


Figure 9.1: *An example of inserting a node that moves upwards.*

This should be tested more, to see if there is actually a problem with the current implementation. A few early tests showed that conflicts in moving nodes did not happen all that often and it is only in very specific cases that nodes actually move outside of the neighborhoods. But of course, every node that moves outside of a neighborhood, can have a negative effect on the graph as it have been discussed before on the concept of dead connections.

9.4 Dead connections

The problem of dead connections remains unsolved. As was observed in section 7.6, the problem of connections not having a chance to be optimized can have a detrimental effect on the grown network. In the case of BipedalWalker the input neurons placed in the extremes, such as neuron 1 or 24, would have inactive connections to the output, since its initial connections are dead from the beginning due to the neighborhood size being too small. An approach where a secondary NCA was added to the algorithm that operated directly on the dead connections was devised. This approach was tested and is described in section 6. The addition of a DCNCA had no statistically significant difference on the performance of the algorithm. However, different ideas to approach the problem could still prove fruitful. For example, maybe the connection-based neighborhood used for the DCNCA did not provide enough information to make a good decision about the weight. Another idea is that all connections including dead connections should be handled completely separately by a different NCA such as the idea with the specialized NCAs from section 9.2. There is also the possibility that a global rule enforcing removal of all dead connections as mentioned in section 6 could prove to be useful. This approach leaves new problems that need to be addressed such as what happens when the only connections that lead to the output neurons are dead connections? The point still stands, that there are many ways to deal with dead connections that have yet to be explored, which could be the subject of future experimentation and development into the NETNCA algorithm.

9.5 Other applications

While it was interesting to observe in section 6.3 that it was possible to use the NETNCA algorithm to solve a classification problem the amount of data was insufficient. The experiments were only tested using the simple Iris dataset that contains a very limited amount of data, making especially the validation set very small. It would be very interesting to see how this would work on a bigger and more complicated dataset and test the use of different hyperparameters to study if they could have an even better effect on the final results. As the results showed, the problem was almost solved by some of the experiments while others showed very low accuracy, and needs further testing.

The NETNCA algorithm seems to have the tools to solve different problems, so the question then becomes, what level of complexity in problems is it able to solve? Currently the solution using the graph implementation only supports building unlayered feed-forward neural networks. However, CNNs have also shown great promise in the use of deep learning (O'Shea and Nash, 2015) (Gu et al., 2015) (Mordvintsev et al., 2020), so an extension to the current NETNCA implementation would be to also allow for building CNNs, to see if it would produce prominent networks for deep learning as well.

10 Conclusion

In section 1 it was proposed that since trained networks often make better decisions than humans, and since the act of designing ANN topologies is often more art than science, it should follow that a neural network should be able to create better performing topologies than humans. Furthermore, since biological neural networks are constructed through cellular replication, which is without any form of global control, it stands to reason that using cellular replication to construct ANNs should be a prudent approach. Therefore, this thesis set out to develop the NETNCA algorithm which uses neuroevolution on NCAs to grow ANNs in much the same way as cells would grow a biological neural network. From the experiments run in control problem areas in section 7, the NETNCA algorithm performed very well and even outperformed algorithms such as neuroevolution and NEAT on some environments. It also performed competitively compared with deep reinforcement learning which is traditionally used for control problems. In the areas of classification the NETNCA algorithm also performed reasonably well, though more testing on larger samples of data would be needed to say anything conclusive. The networks grown from the NETNCA algorithm have interesting characteristics. This includes attributes such as frequent use of sine, cosine and gaussian activation functions as well as non-traditional topologies, which results in interesting solutions to different control problems.

One of the more impressive results generated from testing the NETNCA algorithm, which warrants further research, is the prospect that the rulesets evolved in the NCAs may be very general. This might allow the NCAs to grow networks capable of solving different environments as long as the solutions for the environments are similar enough, as seen in section 7.4.

This thesis laid the foundation of using local control and self-assembly to grow neural networks and was successful in this endeavor. The algorithm is still quite new and has a lot of hyperparameters that affects its performance. While attempts were made to test the different hyperparameters in section 6, there are still many ways to further increase the performance of the algorithm. The NETNCA algorithm presented in this thesis should be viewed as a first draft in future research of using NCAs to develop neural networks. While this thesis does not present enough evidence to conclusively prove that the NETNCA algorithm is better than current state of the art ANN producing algorithms, it does show that NCAs are capable of growing ANNs and should be researched further.

Acknowledgements

The completion of this thesis would not have been possible without the help and support from our supervisor, prof. Sebastian Risi, who was instrumental in both defining the project and providing relevant research material as well as discussing the viability of our core concepts.

Many thanks to Elias Najarro for his contributions and ideas for practical problems.

We would like to thank our university ITU for providing work spaces as well as access to high performance computers for running experiments.

We would also like to recognize the assistance of Aske Sylvester Krogh Nielsen in regards to giving us a surface level understanding of biological concepts.

Literature

Chopard, B., Falcone, J.-L., Razakanirina, R., Hoekstra, A., & Caiazzo, A. (2008). On the collision-propagation and gather-update formulations of a cellular automata rule. *5191*, 144–151. https://doi.org/10.1007/978-3-540-79992-4_19

Eiben, A. E., & Smith, J. E. (2013). Introduction to evolutionary computing. Springer.

Elmenreich, W., & Fehérvári, I. (2011). Evolving self-organizing cellular automata based on neural network genotypes. *6557*, 16–25. https://doi.org/10.1007/978-3-642-19167-1_2

Gardner, M. (1970). Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, *223*, 120–123. https://doi.org/10.1038/scientificamerican1070-120

Grattarola, D., Livi, L., & Alippi, C. (2021). Learning graph cellular automata. https://doi.org/10.48550/ARXIV.2110.14237

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., & Wang, G. (2015). Recent advances in convolutional neural networks. *Pattern Recognition*, *77*. https://doi.org/10.1016/j.patcog.2017.10.013

Han, J., Kamber, M., & Pei, J. (2012). Data mining concepts and techniques (third edition), 398–408.

Hunter, H. (2019). Hyperneat: Powerful, indirect neural network evolution. https://towardsdatascience.com/hyperneat-powerful-indirect-neural-network-evolution-fba5c7c43b7b

Lakna. (2018). How do cells become specialized. https://pediaa.com/how-do-cells-become-specialized/

Małecki, K. (2017). Graph cellular automata with relation-based neighbourhoods of cells for complex systems modelling: A case of traffic simulation. *Symmetry*, *9*, 322. https://doi.org/10.3390/sym9120322

Mordvintsev, A., Randazzo, E., Niklasson, E., & Levin, M. (2020). Growing neural cellular automata. https://distill.pub/2020/growing-ca/

Naitzat, G., Zhitnikov, A., & Lim, L.-H. (2020). Topology of deep neural networks.

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. https://doi.org/10.48550/ARXIV.1511.08458

Palm, R. B., González-Duque, M., Sudhakaran, S., & Risi, S. (2022). Variational neural cellular automata. https://doi.org/10.48550/ARXIV.2201.12360

Risi, S. (2021). The future of artificial intelligence is self-organizing and self-assembling. *sebastianrisi.com*. https://sebastianrisi.com/self_assembling_ai

Risi, S., & Stanley, K. (2019). Deep neuroevolution of recurrent and discrete world models, 456–462. https://doi.org/10.1145/3321707.3321817

Ruiz, A. H., Vilalta, A., & Moreno-Noguer, F. (2020). Neural cellular automata manifold. https://doi.org/10.48550/ARXIV.2006.12155

Sarkar, P. (2000). A brief history of cellular automata. *ACM Comput. Surv.*, *32*, 80–107. https://doi.org/10.1145/349194.349202

Stanley, K., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*, 24–35. https://doi.org/https://doi.org/10.1038/s42256-018-0006-z

Stanley, K., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*, 99–127. https://doi.org/10.1162/106365602320169811

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. https://doi.org/10.48550/ARXIV.1712.06567

Sudhakaran, S., Grbic, D., Li, S., Katona, A., Najarro, E., Glanois, C., & Risi, S. (2021). Growing 3d artefacts and functional machines with neural cellular automata. https://doi.org/10.48550/ARXIV.2103.08737

Toffoli, T., & Margolus, N. (1987). *Cellular automata machines: A new environment for modeling*. https://doi.org/10.7551/mitpress/1763.001.0001

Vrbančič, G., Fister jr, I., & Podgorelec, V. (2018). Designing deep neural network topologies with population-based metaheuristics.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, *1*, 57–81. https://doi.org/https://doi.org/10.1016/j.aiopen.2021.01.001

A How to run the implementation

This appendix will be used to explain each of the parameters used within this implementation of the NETNCA algorithm, what possible choices there are for different experiments and how they are linked together. At the end of the section is a short explanation how to setup new runs from scratch.

A.1 Configurations and parameters

The NETNCA algorithm implementation from this project contains 39 different parameters used for configuring the different subsystems of the algorithm. They are grouped into 4 different subcategories, in order to keep the configurations for the different subsystems of the implementation separate.

A.1.1 Individual

The individual configuration express the general parameters used for each individual in the population used by the genetic algorithm.

Name: kind
Possible values: normal, dc and nn
Description: This parameters is used to differentiate between the 3 different types of individuals that it is possible to generate using the current implementation. The *normal* value, makes the implementation use only a single NCA that handles everything with connections, positions, weights and activation functions. The *dc* value stands for dead connections, and builds two NCAs one is the same as the normal NCA, but the other is an NCA that will make decisions on removal and update of dead connections. The last value *nn* stands for neural network and removes the use of NCAs all together and builds a feed-forward neural network instead.

Name: nca_topolgy.input_size
Possible values: Constricted
Description: This value determines the input size of the main network used in NETNCA, though it is very dependant on the neighbourhood size that is used. It can be calculated using the formula 8, where n_{size} is the size of the neighbourhood and a_{size} is the amount of possible activation functions, in this project there are only 10 possible functions to make use of. The +1 at the end, is for the bias.

$$n_{size} * n_{size} * 3 + a_{size} + 1 \tag{8}$$

Name: nca_topology.output_size
Possible values: Constricted
Description: Like with the input_size parameter, this value is also dependant on other parameters for its calculation. The formula is similar, but a little different as seen by formula 9

$$n_{size} * n_{size} * 2 + a_{size} + 1 \tag{9}$$

Name: nca_topology.hidden_layer_sizes
Possible values: List of positive integers
Description: The length of the given list, corresponds to the amount of hidden layers used by the main network, while each value in the list expresses the amount of neurons in each layer. It is important that the

size of the list corresponds to the amount of activation functions given -1 for the output layer. So if there are 3 activation functions, there have to be 2 hidden layers.

Name: nca_topology.acts
Possible values: List with names of possible allowed activation functions
Description: This list should contain an activation function for each hidden layer and the output layer. The order of the list is the same as the order of the hidden layers in the main network. Allowed activation functions are: linear, sigmoid, relu and tanh.

Name: dc_topology.input_size
Possible values: constant of 12
Description: This value should not be changed, it determines the number of parameters used as input for the dead connections NCA.

Name: dc_topology.output_size
Possible values: constant of 2
Description: This value should not be changed, I have no idea why it is even changeable. The two neurons are used to determine if a connection should be removed and just how much the network wants to remove it.

Name: dc_topology.hidden_layer_sizes
Possible values: List of positive integers
Description: The length of the given list, corresponds to the amount of hidden layers used by the dead connections network, while each value in the list expresses the amount of neurons in each layer. It is important that the size of the list corresponds to the amount of activation functions given -1 for the output layer. So if there are 3 activation functions, there have to be 2 hidden layers.

Name: dc_topology.acts
Possible values: List with names of possible allowed activation functions
Description: This list should contain an activation function for each hidden layer and the output layer. The order of the list is the same as the order of the hidden layers in the main network. Allowed activation functions are: linear, sigmoid, relu and tanh.

Name: growth_iterations
Possible values: A single positive integer
Description: This value determines how many times the NCA is applied to the graph. Note that with 0 iterations, the graph will only contain fully connected input and output neurons. Be aware that high values for this parameter can have a high negative effect on the performance of the NETNCA algorithm, so the recommended range is between 5 for complex environments and up to 9 for simple environments.

A.1.2 Evolution

The evolution parameters and configuration have direct importance on the neuroevolution algorithm used to train the generated NCAs.

Name: pop_size
Possible values: A single positive integer above 0
Description: The total size of the pool containing all individuals. This is the amount of random individuals generated at the start of the process and the size to which the pool is returned after culling the population

after generating and testing the offspring. While the range of possible values is infinite, high values will have a negative effect of the running time of the algorithm. The suggested pool size is between 100 and 300 individuals for a reasonable running time.

Name: number_of_parents
Possible values: A single positive integer above 0 but below the pop_size.
Description: The amount of parents selected by the genetic algorithm, in order to generate offspring.

Name: random_parent_prob
Possible values: A single float value between 0.0 and 1.0
Description: The chance for the genetic algorithm to pick a totally random parent from the pool of individuals, instead of only picking the highest scoring individuals.

Name: improvement_threshold
Possible values: A single positive float
Description: This value determines how much improvement is needed for the score of an individual to be counted as still improving. This is used for the callback variable of iterations without improvements.

Name: fitness.func
Possible values: fitness_nca, fitness_penalize_trivial or fitness_nn
Description: This parameter determines the function used for calculating the fitness score. *fitness_nca* is the standard score, that calculates the average of all tests on an individual. *fitness_penalize_trivial* will apply a penalty to the fitness score if any input or output nodes in the graph are not connected to the rest of the network. *fitness_nn* is a fitness function to be used when using the *nn* value for the kind, it is a function specialised in testing torch networks.

Name: fitness.vars.fitness_iterations
Possible values: A single positive integer above 0
Description: This parameters expresses the amount of times the test environment is run, in order to calculate the fitness score. The fitness score will always be an average across all of the fitness runs. Note that higher numbers will result in significantly longer runtime, though the results are more precise.

Name: fitness.vars.penalty_scale
Possible values: A single positive integer above 0
Description: A penalty modifier that will be multiplied with the amount of penalised neurons in the network. This should be changed to give a proper score according to the environment in use.

Name: fitness.vars.env_kind
Possible values: gym or supervised
Description: A value describing the library where the algorithm should find the environment. *gym* uses the OpenAI library environments described in section 5 and *supervised* uses the sklearn library described in section 5.1.

Name: fitness.vars.env_name
Possible values: CartPole-v1, Acrobot-v1, Pendulum-v1, MountainCar-v0, BipedalWalker-v3, LunarLander-v2 and Iris.
Description: A specification of the previous parameter, describing what environment should be used from the selected library. Note that only some environments are implemented.

Name: mutation.func
Possible values: mutate and gauss_mutate
Description: This value selects the between the implemented mutation functions in the algorithm. *Mutate* uses the random function for mutations and *gauss_mutate* uses the gauss function instead.

Name: mutation.vars.num_of_mutations
Possible values: A single positive integer
Description: This value determines the number of offspring created per parent selected for mutation. For example, the value 4 would indicate that each parent would have 4 offspring with different mutations applied.

Name: mutation.vars.num_of_changes
Possible values: A single positive integer
Description: This value determines the amount of parameters changed per mutation.

Name: mutation.vars.gauss_scale
Possible values: A single positive float
Description: Standard deviation (spread) of the gaussian noise

Name: mutation.vars.weight_mutation_range
Possible values: A single positive float
Description: Maximum change that can be made to a weight in case of non-gaussian mutation

Name: mutation.vars.bias_mutation_range
Possible values: A single positive float
Description: Maximum change that can be made to a weight in case of non-gaussian mutation

Name: mutation.vars.bias_proc_chance
Possible values: A single positive float
Description: The chance that a bias is changed as opposed to a weight in case of non-gaussian mutation

A.1.3 Graph

Name: center_output_around_input
Possible values: Boolean
Description: This value determines where in the vertical output layer the output neurons should be placed. *False* will place the output neurons at the top of the layer while *true* will place the output neurons to match the center of the input neuron layer.

Name: neighbourhood_size
Possible values: Positive odd integer
Description: This value decides on the distance each node in the graph looks around itself and uses as input for the NCA. Since a neuron always have to be at the center position, this value always have to be an odd integer. As an example, a neighbourhood size of 5 means that each grid is 5x5 and the neuron looks 2 steps in each spatial direction. Note that this value is heavily connected to the input_size and output_size

from the individual configuration.

Name: default_act
Possible values: Integer between 0 and 9
Description: This value determines the starting activation function used for newly generated nodes in the graph. Each number references a specific activation function from the following list: Linear, Unsigned Step Function, Sin, Gausian, Tanh, Sigmoid, Inverse, Absolute Value, Relu and Cosine.

Name: default_weight
Possible values: Any float
Description: This value determines the starting weight of newly generated connections in the graph.

Name: default_bias
Possible values: Any float
Description: This value determines the starting bias of newly generated nodes in the graph.

Name: output_activation
Possible values: Integer between 0 and 9
Description: This value determines the activation function used for all output neurons in the graph. Each number references a specific activation function from the following list: Linear, Unsigned Step Function, Sin, Gausian, Tanh, Sigmoid, Inverse, Absolute Value, Relu and Cosine.

Name: add_conn_threshold
Possible values: Float between 0.0 and 1.0
Description: The threshold that needs to be passed, before the graph allows new connections to be added.

Name: add_node_threshold
Possible values: Float between 0.0 and 1.0
Description: The threshold that needs to be passed, before the graph allows for new nodes to be added.

Name: remove_conn_threshold
Possible values: Float between 0.0 and 1.0
Description: The lower threshold that needs to be passed, before the graph allows for removal of connections between nodes in the graph.

Name: remove_dead_conn_threshold
Possible values: Float between 0.0 and 1.0
Description: The lower threshold that needs to be passed, before the graph allows for removal of dead connections in the graph, if the kind has been set to *dc*.

A.1.4 neat

Note that this configuration file is very mislabeled and has little to nothing to do with neat. The naming comes from the use of the prettyNEAT library to covert the graph to a working neural network, but it does not use the NEAT algorithm to do this!

Name: weight_multiplier
Possible values: Positive float
Description: The value all weights are multiplied by during conversion from graph to neural network.

Name: bias_multiplier
Possible values: Positive float
Description: The value all bias are multiplied by during conversion from graph to neural network.

Name: parameter_range.min
Possible values: Any float
Description: The min of the range of values of the output activation function. Needed to convert from activation function range to expected output range in the environment used.

Name: parameter_range.max
Possible values: Any float
Description: The max of the range of values of the output activation function. Needed to convert from activation function range to expected output range in the environment used.

A.2 How to start from scratch

The implementation uses many different generators to simplify the setup and relations between different subsystems of the algorithm. The minimum implementation needed in order to run the program is listed below.

```python
from config import config_reader
from evolution import Evolution, IndividualGenerator
from graph import GraphGenerator
from network import NetworkGenerator
import hedwig


# Setting up proper and configurable logging
hedwig.init("logging_config.json")

# Getting configurations from the provided config file
evo_config, ind_config, graph_config, neat_config = config_reader.read_config("config.json")

# Setting up different generatos
network_generator = NetworkGenerator()
network_generator.set_parameters(**neat_config)
graph_generator = GraphGenerator()
graph_generator.set_parameters(**graph_config)
individual_generator = IndividualGenerator(graph_generator, network_generator)
individual_generator.set_parameters(**ind_config)
evo = Evolution(individual_generator)
evo.set_parameters(**evo_config)

# Callback used between each generation
def callback(iterations : int, iterations_wo_improvement : int, pop):
    hedwig.info(f'Iteration number {iterations}')
    hedwig.info(f'Iterations without any significat improvements: {iterations_wo_improvement}')
    hedwig.info(f'Best score: {pop[0].get_score()}')

# Start the algorithm
evo.run_evolution(callback)
```

B Experiment configurations and results

The following appendix contains all the raw data from the experiments conducted in section 6.

B.1 Configuration

```
"individual": {
    "kind": "normal",
    "nca_topology": {
        "input_size": 155,
        "output_size": 107,
        "hidden_layer_sizes": [255,255,255,255],
        "acts":["tanh", "tanh", "tanh", "tanh", "sigmoid"]
    },
    "dc_topology": {
        "input_size": 12,
        "output_size": 2,
        "hidden_layer_sizes": [64, 32]
        "acts": ["tanh", "tanh", "sigmoid"]
    },
    "growth_iterations": 7
}
```

Listing 1: *Base individual configurations for all NETNCA experiments*

```
"evolution": {
    "pop_size": 300,
    "num_of_parents": 10,
    "random_parent_prob": 0.20,
    "improvement_threshold": 2.0,
    "fitness": {
        "func": "fitness_nca"
        "vars": {
            "fitness_iterations": 25,
            "penalty_scale": 1000
            n/a
        }
    }
}
```

Listing 2: *Base evolution configurations for all NETNCA experiments*

```
"graph": {
    "center_output_around_input": true,
    "neighbourhood_size": 7,
    "default_act": 0,
    "default_weight": 0.5,
    "default_bias": 0.5,
    "output_activation": 4,
    "add_conn_threshold": 0.52,
    "add_node_threshold": 0.53,
    "remove_conn_threshold": 0.48,
    "remove_dead_conn_threshold": 0.48
}
```

Listing 3: *Base graph configurations for all NETNCA experiments*

Listing 3, 2 and 1 shows the base configuration used for all experiments. Each experiment changed specific hyperparameters for exact purpose of testing those parameters. The following list contains all changes to the configuration by each experiment:

- **Different sizes of growth iterations**. For the purpose of this test it was important that connection adding and node adding thresholds didn't have any human bias. Therefore the threshold for *add_conn_threshold, remove_conn_threshold and add_node_threshold* were all set to 0.5. In the baseline *growth_iterations* was set to 7 and in the experiment it was set to 3.

- **Different NCA topologies**. Since the experiment with no hidden layers created so many more nodes a change in the thresholds had to be made. *add_conn_threshold, remove_conn_threshold and add_node_threshold* were all set to 0.5. In the baseline *growth_iterations* was changed to 0.53, 0.47 and 0.54 respectively. The baseline used the following topology:

```
"nca_topology": {
        "input_size": 155,
        "output_size": 107,
        "hidden_layer_sizes": [155,155,155],
        "acts":["tanh", "tanh", "tanh", "sigmoid"]
    }
```

  And the experiment topology used was the following:

```
"nca_topology": {
        "input_size": 155,
        "output_size": 107,
        "hidden_layer_sizes": [],
        "acts":["sigmoid"]
    }
```

- **Dead Connection NCA compared to one NCA**. The DCNCA approach used the following dead connection topology defined in the individual configuration 1.

- **Penalizing graph for how many nodes are pruned compared to not penalizing**. For this experiment both baseline and experiment used the same configuration and the penalty was added directly in the code for the experiment.

- **Penalizing trivial networks compared to no penalty** For this experiment both baseline and experiment used the same configuration, except the experiment used the fitness function "fitness_penalize_trivial".

B.2 Results

Trivial Penalty		Prune Penalty		Growth iterations		Dead Connections		Different Topologies	
Penalty	No Penalty	Penalty	No Penalty	3	7	With DCNCA	No DCNCA	No Hidden	3 Hidden
-411.89	-423.84	-67.8	-71.5	-175.36	-1074.92	-73.0	-71.5	-825.55	-810.45
-785.91	-733.95	-70.4	-66.6	-126.88	-69.36	-65.1	-66.6	-638.06	-899.39
-669.20	-758.29	-71.7	-70.3	-74.2	-71.24	-76.1	-70.3	-319.79	-1744.56
-645.69	-782.66	-68.8	-69.5	-75.36	-115.24	-68.7	-69.5	-808.92	-1741.94
-966.97	-760.35	-68.3	-76.0	-1070.8	-72.12	-76.2	-76.0	-817.59	-1016.04
-907.35	-869.97	-70.6	-68.1	-1072.68	-75.12	-66.5	-68.1	-810.21	-916.38
-743.39	-658.16	-69.4	-67.7	-150.08	-499.0	-70.1	-67.7	-383.69	-995.08
-817.34	-717.12	-80.5	-69.2	-96.12	-74.6	-71.9	-69.2	-259.63	-488.12
-802.08	-707.64	-70.1	-80.9	-75.32	-85.28	-68.7	-80.9	-642.32	-742.72
-542.39	-704.48	-68.8	-72.4	-189.04	-90.36	-71.5	-72.4	-577.75	-1038.06
-207.44	-724.45	-70.4	-68.1	-73.24	-72.48	-72.6	-68.1	-329.74	-724.11
-677.50	-757.92	-76.8	-71.6	-184.68	-72.84	-71.6	-71.6	-278.58	-847.58
-957.22	-727.18	-66.2	-75.4	-74.24	-1073.12	-67.6	-75.4	-426.73	-600.52
-896.60	-709.86	-74.2	-66.8	-81.84	-74.76	-72.0	-66.8	-718.28	-662.09
-397.40	-723.34	-69.3	-69.0	-1072.28	-93.08	-86.6	-69.0	-539.81	-359.05
-795.07	-800.32	-68.2	-73.6	-1071.2	-499.0	-74.2	-73.6	-385.58	-479.21
-169.91	-743.19	-68.8	-68.4	-72.96	-76.16	-89.1	-68.4	-307.89	-647.39
-682.71	-726.34	-72.4	-69.3	-81.16	-75.92	-67.8	-69.3	-422.98	-766.31
-742.31	-808.14	-67.2	-75.8	-69.8	-68.96	-77.0	-75.8	-673.59	-385.94
-802.41	-728.51	-67.2	-77.7	-138.92	-76.8	-72.1	-77.7	-415.95	-552.94
-870.79	-857.40	-69.9	-82.1	-73.72	-71.4	-73.0	-82.1	-250.76	-1798.93
-394.17	-713.19	-70.5	-71.0	-1073.56	-69.8	-71.2	-71.0	-444.00	-870.69
-815.23	-750.24	-69.3	-65.8	-147.16	-499.0	-73.1	-65.8	-426.44	-321.93
-318.45	-853.91	-72.2	-71.6	-1072.36	-80.92	-74.1	-71.6	-327.56	-854.82
-785.50	-698.32	-67.4	-66.9	-1071.6	-1074.64	-74.6]	-66.9	-938.44	-865.84

Table B.1: *Result of all experiments run 25 times for 100 generations. Each score of the experiments is an average of 25 iterations through their respective gym environments*

C Baseline configurations and results

This appendix will contain all the configurations used for the baseline experiments, in order to allow for retesting to allow for validity of the test results. The appendix also contains all of the results from each experiment, as most parts of this paper only makes use of a few of them or the aggregated results.

There are two configurations that have been shared between all of the baselines experiments:

- All baseline tests where run independently a total of 25 times.
- For each calculation of fitness the gym environment is tested 100 times and the fitness is the average of the result.

C.1 Random

C.1.1 Configurations

There are no special configurations associated with the random baseline.

C.1.2 Results

CartPole	Acrobot	Pendulum	MountianCar	BipedalWalker	LunarLander
21.28	-495.13	-1220.63	-199.0	-31.59	-91.04
21.1	-496.06	-1205.10	-199.0	-33.17	-72.67
20.38	-498.78	-1201.00	-199.0	-33.08	-87.23
20.23	-499.0	-1232.14	-199.0	-29.20	-81.50
22.61	-499.0	-1206.74	-199.0	-28.95	-80.64
21.6	-496.12	-1214.72	-199.0	-36.55	-91.21
22.11	-494.87	-1211.52	-199.0	-36.65	-80.10
20.29	-498.82	-1228.25	-199.0	-33.12	-80.25
20.9	-499.0	-1272.27	-199.0	-35.51	-87.49
19.06	-498.01	-1249.43	-199.0	-30.45	-94.85
20.72	-498.52	-1251.37	-199.0	-37.06	-88.89
21.42	-498.69	-1213.14	-199.0	-37.23	-84.25
20.82	-497.57	-1200.74	-199.0	-33.57	-79.51
22.53	-497.91	-1201.27	-199.0	-40.21	-73.74
22.36	-494.56	-1164.31	-199.0	-39.16	-85.97
20.84	-499.0	-1206.52	-199.0	-33.23	-69.50
20.2	-498.52	-1267.79	-199.0	-30.53	-93.01
22.47	-499.0	-1218.42	-199.0	-35.54	-85.40
22.82	-497.75	-1283.72	-199.0	-32.76	-86.60
20.29	-498.78	-1235.22	-199.0	-34.75	-66.73
22.38	-497.72	-1208.44	-199.0	-35.01	-94.95
23.16	-496.96	-1195.53	-199.0	-28.85	-71.40
23.34	-497.7	-1211.14	-199.0	-37.69	-85.98
21.6	-497.62	-1217.79	-199.0	-22.44	-74.19
22.65	-498.56	-1233.46	-199.0	-30.97	-72.75

Table C.1: *The best result from each test using the random baseline*

C.1.3 Aggregated results

Name:	Avg:	Max:	Std:
CartPole	21.49	23.34	1.09
Acrobot	-497.75	-494.56	1.35
Pendulum	-1222.03	-1164.31	26.22
MountainCar	-199.0	-199.0	0.0
BipedalWalker	-33.48	-22.44	3.82
LunarLander	-82.39	-66.73	8.02

Table C.2: *The aggregated results from the random baseline*

C.2 Random Search

C.2.1 Configurations

The following section will express the configurations used for the random search baseline tests, while many of the experiments had an overlap in configurations, there were a few changes between them. Specially with the harsher environments as LunarLander and BipedalWalker.

For the individual configurations they are generally the same, except for the input_size and output_size that are higher (251 instead of 155) in the Bipedal and LunarLander environment, because of the larger neighbourhood used in those experiments. Other then that the growth_iterations has been lowered to 5 only for the BipedalWalker environment.

```
"individual": {
    "kind": "normal",
    "nca_topology": {
        "input_size": 155 (251),
        "output_size": 107 (171),
        "hidden_layer_sizes": [255,255,255,255],
        "acts":["tanh", "tanh", "tanh", "tanh", "sigmoid"]
    },
    "dc_topology": { n/a },
    "growth_iterations": 7 (5)
}
```

Listing 4: *Individual configurations for the random search baseline*

Since this baseline does not make use of evolution, there is only a single value used in the implementation. This is the pop_size of 300, that determines the amount of random individuals in a single search attempt.

```
"evolution": {
    "pop_size": 300,
    n/a
}
```

Listing 5: *Evolution configurations for the random search baseline*

Not many changes where in the graph configurations part of the baseline, except for a slightly higher neighbourhood_size for the BipedalWalker and LunarLander environments, that both used a size of 9 instead.

```
"graph": {
    "center_output_around_input": true,
    "neighbourhood_size": 7 (9),
    "default_act": 0,
    "default_weight": 0.5,
    "default_bias": 0.5,
    "output_activation": 4,
    "add_conn_threshold": 0.52,
    "add_node_threshold": 0.53,
    "remove_conn_threshold": 0.48,
    "remove_dead_conn_threshold": 0.48
}
```

Listing 6: *Graph configurations for the random search baseline*

The configurations used for the conversation to the neural network was also not changed between different environments and was the following:

```
"neat": {
    "weight_multiplier": 100,
    "bias_multiplier": 100,
    "parameter_range": {
        "min": 0,
        "max": 1
    }
}
```

Listing 7: *NEAT configurations for the random search baseline*

C.2.2 Results

CartPole		Acrobot		Pendulum		MountainCar		BipedalWalker		LunarLander	
AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
42.37	397.60	-423.84	-85.50	-1522.81	-1018.13	-198.89	-182.60	-38.84	0.21	-404.01	-18.01
42.47	499.00	-442.25	-88.30	-1527.83	-1076.42	-198.70	-157.80	-33.57	1.86	-392.13	-11.10
41.95	364.10	-431.17	-85.10	-1523.30	-1007.45	-198.86	-160.90	-38.99	12.57	-396.40	6.52
45.78	440.00	-427.97	-81.60	-1523.58	-1043.14	-198.13	-130.40	-37.41	1.44	-414.59	-5.15
52.18	458.60	-425.86	-84.90	-1532.63	-994.63	-198.78	-164.10	-33.84	0.36	-404.99	-6.93
48.25	355.60	-427.24	-86.00	-1525.77	-962.31	-198.76	-157.70	-40.53	4.15	-397.82	-15.26
48.98	447.00	-416.66	-79.60	-1533.57	-1009.01	-198.79	-147.20	-35.79	0.09	-399.93	-17.70
42.52	499.00	-424.10	-85.30	-1540.44	-990.78	-198.84	-176.80	-32.71	-0.17	-398.57	-19.55
51.02	495.30	-440.54	-77.00	-1533.89	-1020.33	-198.76	-134.50	-37.33	0.50	-385.71	-14.69
46.04	398.80	-414.52	-81.70	-1524.65	-992.74	-199.00	-199.00	-36.39	0.64	-405.24	-18.22
42.93	367.50	-428.26	-85.00	-1526.69	-943.86	-198.64	-128.60	-35.11	5.73	-412.63	-23.47
47.54	431.80	-432.36	-81.30	-1511.94	-864.05	-198.75	-155.00	-33.35	-0.80	-422.13	-13.57
46.91	444.80	-442.08	-84.50	-1522.91	-1032.72	-198.69	-156.30	-34.49	2.78	-388.31	-23.60
46.07	331.20	-429.55	-82.70	-1527.56	-1076.17	-199.00	-199.00	-40.88	1.21	-398.42	-21.97
49.83	430.00	-414.59	-83.30	-1528.51	-1015.79	-198.74	-122.20	-32.72	2.36	-409.36	-18.51
46.37	399.90	-436.11	-81.70	-1530.25	-1031.94	-198.59	-157.00	-39.73	1.86	-396.51	-19.65
47.90	499.00	-424.01	-82.90	-1539.33	-1100.18	-198.91	-170.60	-36.02	0.67	-422.31	-25.27
48.86	484.30	-434.78	-91.30	-1533.04	-1028.30	-198.95	-185.00	-35.76	11.58	-390.18	4.01
47.84	499.00	-425.45	-80.30	-1530.43	-1064.10	-198.89	-180.90	-37.23	4.28	-383.97	-3.66
57.62	499.00	-421.11	-85.90	-1530.09	-1041.57	-198.98	-194.20	-41.42	-0.56	-409.41	29.68
56.70	498.80	-418.50	-81.00	-1530.46	-1001.41	-198.80	-167.90	-34.63	0.64	-376.96	-16.07
46.70	318.80	-440.08	-87.00	-1521.15	-1031.83	-198.49	-120.40	-34.14	0.94	-426.36	-19.45
54.72	499.00	-436.20	-82.90	-1555.36	-1023.98	-198.40	-131.70	-33.65	1.25	-378.74	-20.50
45.19	499.00	-419.24	-84.90	-1532.04	-1034.29	-198.20	-120.40	-35.84	0.51	-394.56	1.11
48.77	494.00	-445.80	-90.00	-1521.85	-1018.59	-198.81	-143.00	-39.19	1.01	-390.31	-7.38

Table C.3: *The best result from each test using the random search baseline*

C.2.3 Aggregated results

	Results of AVG			Results of MAX		
	AVG	MAX	STD	AVG	MAX	STD
CartPole	47.82	57.62	4.13	442.12	499.0	58.63
Acrobot	-428.89	-414.52	8.89	-84.07	-77.9	3.11
Pendulum	-1529.20	-1511.94	8.01	-1016.95	-864.05	45.85
MountainCar	-198.74	-198.13	0.22	-157.80	-120.43	24.08
BipedalWalker	-36.38	-32.71	2.61	2.20	12.57	3.28
LunarLander	-399.98	-376.95	12.96	-11.94	29.68	12.07

Table C.4: *The aggregated results from the random search baseline*

C.3 Neuroevolution

C.3.1 Configurations

This section shows all the configurations used for the neuroevolution baseline experiments. They are very similar between the different environments, with the only changes being focused on the topology of the neural network, since input and output sizes are dependant on the amount of inputs and outputs on the environment.

The topology is handled by the individual configuration as shown below. The input and output sizes should be set depending on the corresponding size of the vectors in the environments. The hidden layer size has been calculated using formula 10 as shown below, so the hidden layers are just a bit smaller then the input_sizes. It should be noted, that the value is rounded to the nearest whole number, since it is a requirement for a layer to not have partial neurons. The only environment were this formula was not used was the MountainCar environment, as this rounded to only having 1 neuron in each layer and preformed abysmal, so instead it was forced to have 2 neurons in each layer preformed better because of it.

$$\lfloor input\_size * 0.66 + 0.5 \rfloor \tag{10}$$

```
"individual": {
    "kind": "nn",
    "nca_topology": {
        "input_size": input vector size,
        "output_size": output vector size,
        "hidden_layer_sizes": [hidden_layer_size, hidden_layer_size, hidden_layer_size],
        "acts": ["tanh", "tanh", "tanh", "tanh"]
    },
    "dc_topology": { n/a },
    "growth_iterations": n/a
}
```

Listing 8: *Individual configurations for the neuroevolution baseline*

The evolution parameters used for this baseline is shown below, and was not changed between the different environments.

```
    "evolution": {
        "pop_size": 300,
        "num_of_parents": 10,
        "random_parent_prob": 0.20,
        "fitness": {
            "func": "fitness_nn",
            "vars": {
                "fitness_iterations": 100,
                "penalty_scale": n/a,
                "env_kind": "gym",
                "env_name": environment name
            }
        },
        "mutation": {
            "func": "gauss_mutate",
            "vars": {
                "num_of_mutations": 10,
                "num_of_changes": 1,
                "gauss_scale": 1.0,
                "weight_mutation_range": 2.5,
                "bias_mutation_range": 2.5,
                "bias_proc_chance": 0.25
            }
        }
    }
```

Listing 9: *Evolution configurations for the neuroevolution baseline*

C.3.2 Results

CartPole		Acrobot		Pendulum		MountainCar		LunarLander		BipedalWalker	
AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
500.00	500.00	-77.19	-73.70	-913.57	-839.37	-200.00	-200.00	-83.76	-72.34	-9.22	18.90
500.00	500.00	-79.34	-76.50	-819.94	-770.59	-105.33	-103.00	-41.41	-22.56	7.45	7.60
500.00	500.00	-77.90	-75.10	-660.58	-609.58	-200.00	-200.00	271.65	277.82	7.33	7.48
500.00	500.00	-77.97	-74.90	-867.74	-810.50	-200.00	-200.00	277.29	283.70	8.60	9.11
500.00	500.00	-78.38	-75.20	-168.95	-141.54	-200.00	-200.00	160.08	200.95	7.75	8.29
500.00	500.00	-77.96	-74.60	-925.48	-872.11	-105.29	-102.10	250.53	264.81	6.91	7.21
500.00	500.00	-77.54	-73.10	-816.01	-773.17	-200.00	-200.00	-5.71	22.50	7.33	7.87
500.00	500.00	-77.50	-74.70	-972.52	-915.68	-200.00	-200.00	71.13	106.18	6.00	6.44
500.00	500.00	-78.63	-75.00	-897.59	-843.87	-200.00	-200.00	-84.60	-71.78	7.20	7.43
500.00	500.00	-73.09	-69.00	-857.50	-816.82	-200.00	-200.00	-37.99	-17.40	6.99	7.17
500.00	500.00	-77.96	-75.10	-1112.41	-1080.24	-200.00	-200.00	271.25	278.70	7.74	8.68
500.00	500.00	-78.59	-75.00	-168.14	-147.58	-200.00	-200.00	32.80	67.48	–	–
500.00	500.00	-77.06	-74.20	-684.08	-647.03	-200.00	-200.00	270.64	279.07	–	–
500.00	500.00	-75.53	-73.00	-928.78	-876.52	-200.00	-200.00	227.62	252.48	–	–
500.00	500.00	-79.87	-76.50	-833.32	-792.88	-200.00	-200.00	62.22	97.06	–	–
500.00	500.00	-77.03	-71.80	-248.57	-207.93	-200.00	-200.00	-116.47	-110.93	–	–
500.00	500.00	-77.66	-75.50	-220.60	-190.51	-200.00	-200.00	274.58	279.74	–	–
500.00	500.00	-75.92	-72.80	-173.37	-147.17	-200.00	-200.00	-41.88	-25.12	–	–
500.00	500.00	-76.19	-73.30	-1109.27	-1076.87	-200.00	-200.00	159.48	186.56	–	–
500.00	500.00	-74.97	-71.00	-882.10	-830.68	-200.00	-200.00	-31.29	-3.29	–	–
500.00	500.00	-76.93	-74.00	-511.63	-478.53	-104.48	-101.80	-7.54	21.18	–	–
500.00	500.00	-75.54	-72.60	-940.16	-870.79	-200.00	-200.00	89.97	119.10	–	–
500.00	500.00	-76.97	-74.10	-1117.50	-1073.24	-200.00	-200.00	181.14	211.15	–	–
500.00	500.00	-77.22	-73.00	-831.27	-778.69	-200.00	-200.00	76.02	108.31	–	–
500.00	500.00	-75.07	-72.00	-793.42	-749.72	-200.00	-200.00	270.65	278.47	–	–

Table C.5: *The best result from each test using the neuroevolution baseline, note that the experiments for BipedalWalker was not completed, because of technical problems.*

C.3.3 Aggregated results

	Results of AVG			Results of MAX		
Name	AVG	MAX	STD	AVG	MAX	STD
CartPole	500.0	500.0	0.0	500.0	500.0	0.0
Acrobot	-77.12	-73.09	1.47	-73.99	-69.80	1.56
Pendulum	-738.18	-168.14	301.07	-693.66	-141.54	292.70
MountainCar	-188.60	-104.48	30.86	-188.28	-101.81	31.74
LunarLander	99.86	277.29	134.60	120.47	283.70	132.07

Table C.6: *The aggregated results from the neuroevolution baseline*

C.4 NEAT

C.4.1 Configurations

Unlike the previous experiments, the NEAT tests have been run using an external library, also used for the conversion of graph to network in the NETNCA algorithm. The library is called *prettyNEAT* and contains a list of their own configurations, that needs to be set in order to use the library. For the experiments used in this paper, the a set of default parameters were used as provided by the library itself. The full list is provided below in listing 10.

```
{
    "task": environment name,
    "maxGen": n/a,
    "popSize": 300,
    "alg_nReps": 2,
    "alg_speciate": "neat",
    "alg_probMoo": 0.0,
    "alg_act": 5,
    "prob_addConn": 0.08,
    "prob_addNode": 0.05,
    "prob_crossover": 0.8,
    "prob_enable": 0.01,
    "prob_mutAct": 0.1,
    "prob_mutConn": 0.8,
    "prob_initEnable": 1.0,
    "select_cullRatio": 0.1,
    "select_eliteRatio": 0.1,
    "select_rankWeight": "exp",
    "select_tournSize": 2,
    "spec_compatMod": 0.25,
    "spec_dropOffAge": 64,
    "spec_target": 4,
    "spec_thresh": 2.0,
    "spec_threshMin": 2.0,
    "spec_geneCoef": 1,
    "spec_weightCoef": 0.5,
    "save_mod": 8,
    "bestReps": 10
}
```

Listing 10: *Default configurations for NEAT used for the baseline testing*

So unlike the NETNCA algorithm, where all the configurations are saved into a single file, the prettyNEAT library, also have separate files for the configurations and parameters that are specific to that environment, prettyNEAT calls these games. So each of the 6 environment files is provided below.

```
env_name='CartPole-v1',
time_factor=0,
actionSelect='prob',
input_size=4,
output_size=2,
layers=[1],
i_act=np.full(4, 1),
h_act=np.full(1, 1),
o_act=np.full(2, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False, False, False],
max_episode_length=500,
in_out_labels=['Cart_Position', 'Cart_Velocity', 'Pole_Angle',
    'Pole_Angular_Velocity', 'Force_left', 'Force_right']
```

Listing 11: *Game configurations for using NEAT on the CartPole Environment*

```
env_name='Acrobot-v1',
time_factor=0,
actionSelect='prob',
input_size=6,
output_size=3,
layers=[1],
i_act=np.full(6, 1),
h_act=np.full(1, 1),
o_act=np.full(3, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False, False, False],
max_episode_length=500,
in_out_labels=['Cos(theta1)', 'Sin(theta1)', 'Cos(theta2)', 'Sin(theta2)',
    'Vel(theta1)', 'Vel(theta2)', '-1 torque', '0 torque', '1 torque']
```

Listing 12: *Game configurations for using NEAT on the Acrobot Environment*

```
env_name='Pendulum-v1',
time_factor=0,
actionSelect='all',
input_size=3,
output_size=1,
layers=[1],
i_act=np.full(3, 1),
h_act=np.full(1, 1),
o_act=np.full(1, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False],
max_episode_length=200,
in_out_labels=['x', 'y', 'velocity', 'torque']
```

Listing 13: *Game configurations for using NEAT on the Pendulum Environment*

```
env_name='MountainCar-v0',
time_factor=0,
actionSelect='prob',
input_size=2,
output_size=3,
layers=[1],
i_act=np.full(2, 1),
h_act=np.full(1, 1),
o_act=np.full(3, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False, False, False],
max_episode_length=200,
in_out_labels=['Cart_Position', 'Cart_Velocity', 'Accelerate Left',
    'No Accelerate', 'Accelerate Right']
```

Listing 14: *Game configurations for using NEAT on the MountainCar Environment*

```
env_name='BipedalWalker-v3',
time_factor=0,
actionSelect='all',
input_size=24,
output_size=4,
layers=[1],
i_act=np.full(24, 1),
h_act=np.full(1, 1),
o_act=np.full(4, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False],
max_episode_length=2000,
in_out_labels=['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11',
    '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22',
    '23', '24', '25', '26', '27', '28']
```

Listing 15: *Game configurations for using NEAT on the BipedalWalker Environment*

```
env_name='LunarLander-v2',
time_factor=0,
actionSelect='prob',
input_size=8,
output_size=4,
layers=[1],
i_act=np.full(8, 1),
h_act=np.full(1, 1),
o_act=np.full(4, 1),
weightCap=4.0,
noise_bias=0.0,
output_noise=[False],
max_episode_length=1000,
in_out_labels=['x_coord', 'y_coord', 'x_velocity', 'y_velocity',
    'angle', 'angle_velocity', 'contact1', 'contact2', 'nothing',
    'left', 'main', 'right']
```

Listing 16: *Game configurations for using NEAT on the LunarLander Environment*

C.4.2 Results

CartPole		Acrobot		Pendulum		MountainCar		LunarLander	
AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
361.66	500.00	-115.18	-101.10	-1128.54	-705.93	-200.00	-200.00	-82.64	41.93
337.63	500.00	-116.56	-99.50	-1210.73	-764.39	-200.00	-200.00	-85.71	49.75
327.44	500.00	-111.06	-97.30	-973.66	-376.53	-199.96	-191.90	-93.04	66.31
371.53	500.00	-114.14	-98.00	-1094.86	-266.14	-200.00	-200.00	-57.79	83.41
393.50	500.00	-114.79	-96.40	-1185.29	-679.20	-200.00	-200.00	-68.85	30.26
310.84	500.00	-114.37	-95.30	-1155.68	-756.70	-200.00	-200.00	-57.80	60.62
324.67	500.00	-117.16	-101.20	-1121.07	-678.87	-200.00	-200.00	-84.42	87.68
374.96	500.00	-115.10	-100.20	-1187.37	-594.00	-200.00	-200.00	-83.32	58.49
337.55	500.00	-112.30	-99.10	-1135.88	-544.87	-199.99	-197.70	-83.61	96.35
339.19	500.00	-114.90	-96.80	-1180.70	-586.13	-200.00	-200.00	-93.97	23.60
369.63	500.00	-114.14	-100.20	-1156.00	-678.42	-200.00	-200.00	-88.66	69.85
426.19	500.00	-115.69	-100.00	-1153.30	-690.24	-200.00	-200.00	-61.84	76.78
293.10	500.00	-112.88	-98.20	-1098.23	-680.40	-200.00	-200.00	-97.40	71.92
341.85	500.00	-121.16	-99.00	-1152.34	-679.97	-200.00	-200.00	-80.71	64.21
366.41	500.00	-113.98	-98.30	-1058.13	-563.26	-200.00	-200.00	-76.23	85.37
357.11	500.00	-114.75	-99.20	-1202.52	-616.65	-200.00	-200.00	-75.91	100.71
382.50	500.00	-114.45	-100.40	-1154.69	-584.35	-200.00	-200.00	-66.17	59.93
292.70	500.00	-114.99	-99.00	-1142.17	-709.35	-200.00	-200.00	-96.08	52.45
419.42	500.00	-113.05	-100.50	-1143.30	-681.58	-200.00	-200.00	-66.44	47.92
396.89	500.00	-114.94	-99.80	-1095.83	-507.93	-200.00	-200.00	-89.00	46.48
401.29	500.00	-113.57	-99.20	-1191.81	-698.19	-200.00	-200.00	-91.30	44.95
394.91	500.00	-115.06	-99.20	-1108.55	-678.78	-200.00	-200.00	-79.82	84.94
305.24	500.00	-115.03	-98.00	-1180.75	-690.38	-200.00	-199.70	-65.09	37.79
294.19	500.00	-114.43	-99.00	-1207.39	-738.46	-200.00	-200.00	-73.30	115.92
431.83	500.00	-112.94	-100.00	-1053.80	-260.35	-200.00	-200.00	-59.23	65.25

Table C.7: *The best result from each test using the NEAT baseline*

C.4.3 Aggregated results

	Results of AVG			Results of MAX		
Environment	AVG	MAX	STD	AVG	MAX	STD
CartPole	358.09	431.83	41.10	500.0	500.0	0.0
Acrobot	-114.66	-111.06	1.83	-99.00	-95.3	1.42
Pendulum	-1138.90	-973.66	54.34	-616.44	-260.35	133.85
MountainCar	-199.99	-199.99	0.01	-199.58	-191.93	1.62
LunarLander	-78.33	-57.79	12.28	64.91	115.92	22.30

Table C.8: *The aggregated results from the NEAT baseline*

D NETNCA configurations and results

This appendix contains all the configurations used for the NETNCA displayed in section 7. This is done for full transparency of the scores of all the networks belonging to the experiments. All tests were run 25 times independently to create a large enough sample for statistical tests such as the student's t test. Each network's final score is an average of running the NETNCA network 100 times in its respective environment.

D.1 Configurations

In order to properly compare NETNCA using evolution and NETNCA using random search it was chosen to use the same configurations for NETNCA as Random Search. Therefore the configurations for all the NETNCA networks described in section 7 can be found in section C.1.1

D.2 Results

CartPole	Acrobot	Pendulum	MontainCar	BipedalWalker	LunarLander
500	-151.84	-568.18	-117.57	1.54	102.36
500	-73.26	-1008.35	-200	1.65	-6.97
500	-85.66	-940.85	-114.23	2.42	117.74
500	-78.89	-864.08	-199.0	2.52	100.08
500	-75.11	-794.10	-120.77	2.87	74.39
500	-74.04	-585.17	-200	3.5	46.72
500	-76.17	-688.55	-114.23	3.09	87.19
500	-76.02	-904.44	-165.81	3.11	94.93
500	-84.33	-553.73	-118.78	3.7	-9.80
500	-78.96	-178.86	-199.0	5.64	59.42
500	-80.52	-879.38	-139.76	6.22	90.49
500	-80.9	-545.93	-101.27	6.34	107.79
500	-73.42	-931.38	-200	9.14	-9.14
500	-77.71	-875.47	-114.39	9.72	-6.24
500	-72.83	-901.62	-101.04	10.83	61.98
500	-82.53	-730.02	-198.79	11.05	53.27
500	-84.34	-544.66	-131.35	11.49	84.86
500	-77.8	-377.60	-200	13.00	57.96
500	-91.65	-594.08	-110.03	15.29	36.56
500	-75.79	-831.02	-114.34	17.08	-8.21
500	-81.21	-843.20	-200	52.02	128.47
500	-74.21	-650.23	-200	71.13	70.15
500	-74.78	-892.31	-200	80.81	149.23
500	-76.77	-961.13	-117.93	91.48	82.65
500	-77.11	-801.55	-114.63	145.91	55.39

Table D.1: *The maximum score from each NETNCA run of each test through every environment*

D.3 Aggregated results

	AVG	MAX	STD
CartPole	500	500	0
Acrobot	-78.72	-73.35	4.27
Pendulum	-737.83	-178.85	199.53
MountainCar	-151.80	-100.44	41.04
BipedalWalker	23.26	145.91	35.69
LunarLander	62.53	148.30	44.37

Table D.2: *The aggregated results from the NETNCA runs*

E Classification configurations and results

This appendix will contain configurations and result data for classification experiment conducted for this paper. Only a single environment was used, the Iris dataset from the sklearn library.

E.1 Configurations

The configurations used for this experiment were quite similar to the ones used for the baseline experiment as well. Only the size of the NCA is significantly smaller and each individual was only tested once on the environment, since it was static. The improvement threshold should also be changed to 0.01, as the classification experiment improve with smaller scores.

```
"individual": {
    "kind": "normal",
    "nca_topology": {
        "input_size": 155,
        "output_size": 107,
        "hidden_layer_sizes": [102,102],
        "acts":["tanh", "tanh", "sigmoid"]
    },
    "dc_topology": { n/a },
    "growth_iterations": 6
}
```

Listing 17: *Individual configurations for the classification experiment*

```
"fitness": {
    "func": "fitness_nca",
    "vars": {
        "fitness_iterations": 1,
        "penalty_scale": n/a,
        "env_kind": "supervised",
        "env_name": "Iris"
    }
}
```

Listing 18: *Fitness configurations for the classification experiment*

The remaining configurations not shown here are identical to what was previously shown in listings 6, 7 and 9.

E.2 Results

Score	Test Accuracy	Validation Accuracy
73.50	0.33	0.33
86.84	0.33	0.33
90.16	0.33	0.33
91.67	0.39	0.40
95.07	0.33	0.33
96.07	0.33	0.33
96.13	0.33	0.33
99.34	0.81	0.93
99.88	0.66	0.66
101.57	0.24	0.26
102.12	0.73	0.8
103.36	0.45	0.33
104.96	0.66	0.66
105.19	0.36	0.33
106.2	0.65	0.66
106.88	0.53	0.66
109.34	0.65	0.66
110.00	0.66	0.66
113.31	0.88	1.00
113.81	0.87	0.87
114.17	0.66	0.66
115.78	0.64	0.66
122.75	0.96	1.00
125.35	0.96	1.00
126.12	0.98	1.00

Table E.1: *The best result from each test sorted by score and using classification*

E.3 Aggregated results

	Test Accuracy			Validation Accuracy		
Dataset	AVG	MAX	STD	AVG	MAX	STD
Iris	59%	98%	0.24	61%	100%	0.26

Table E.2: *The aggregated results from the classification experiments*