Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Erik Wiker

Reducing the Search Space of Neuroevolution using Monte Carlo Tree Search

Master's thesis in Informatics
Supervisor: Massimiliano Ruocco, Stefano Nichele
June 2019

NTNU
Norwegian University of
Science and Technology

Erik Wiker

Reducing the Search Space of Neuroevolution using Monte Carlo Tree Search

Master's thesis in Informatics
Supervisor: Massimiliano Ruocco, Stefano Nichele
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Preface

This is a master's thesis for Informatics at the Norwegian University of Science and Technology (NTNU) in the field Artificial Intelligence. The thesis is supervised by Massimiliano Ruocco and co-supervised by Stefano Nichele.

I would like to thank my supervisors for giving valuable feedback and guidance throughout the period.

Trondheim, June 13, 2019

Erik Wiker

Abstract

This thesis explores the possibility of using Monte-Carlo tree search to reduce the search space of the well-known machine learning algorithm Neuroevolution of Augmenting Topologies, with the goal of achieving shorter run-times and better solutions. How combination of the two algorithms can be done is researched through design, implementation and experimentation. Three main methods are proposed based on experimentation done in this thesis as well as the experience from previous work in the field. The algorithms were tested on environments from Open AI gym and compared to the original algorithm.

The results show that none of the proposed algorithms are able to outperform Neuroevolution of Augmenting Topologies, producing larger solution networks, longer run-times or worse fitnesses. However, as this is an early attempt at the strategy, the results show that the work proposed in this thesis may serve as a important foundation for further development in the area.

The source code for this project can be found at https://github.com/MrWe/NEAT_MCTS.

Sammendrag

Denne oppgaven undersøker muligheten for å bruke Monte-Carlo-tre-søk for å redusere søkeområdet til den velkjente maskinlæringsalgoritmen Neuroevolution of Augmenting Topologies, med sikte på å oppnå kortere kjøretider og bedre løsninger. Hvordan en kombinasjon av de to algoritmene kan konstrueres er undersøkt gjennom design, implementering og eksperimentering. Tre hovedmetoder er foreslått basert på eksperimentering utført i denne avhandlingen, samt erfaringen fra tidligere arbeid i feltet. Algoritmene ble testet på miljøer fra Open AI gym og sammenlignet med den opprinnelige algoritmen.

Resultatene viser at ingen av de foreslåtte algoritmene er i stand til å overgå Neuroevolution of Augmenting Topologies. De produserer ofte større nettverksløsninger, lengre løpstider eller dårligere treningsformer. Men da dette er et tidlig forsøk på strategien, viser resultatene at arbeidet som foreslås i denne oppgaven kan fungere som et viktig grunnlag for videreutvikling i området.

Koden for dette prosjektet er tilgjengelig på https://github.com/MrWe/NEAT_MCTS.

Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Network. 16, 22

CNN Convolutional Neural Network. 17

CoDeepNEAT Coevolution DeepNEAT. 17

CPPN Compositional pattern-producing network. 17

DNN Deep Neural Network. 20

HyperNEAT Hypercube-based NeuroEvolution of Augmenting Topologies. 16, 17

MCTS Monte Carlo Tree Search. 12, 14, 20, 31, 32, 35, 67, 68

NAS Neural Architecture Search with Reinforcement Learning. 17, 18

NEAT Neuroevolution of Augmenting Topologies. 12, 14–17, 27–29, 33, 35, 39, 46, 57–60, 63, 64, 66–68

RNN Recurrent Neural Network. 17

TWEANN Topology and Weight Evolving Artificial Neural Network. 12, 27

UCT Upper Confidence Bound applied to trees. 20

Chapter 1

Introduction

Artificial intelligence has quickly become a million dollar industry and is widely used in systems like processing customer data for stores, assisting in medical diagnosing, to using face identification to open phones, and as computing power increases so too does the number of possible use cases for such systems. The main component in many artificial intelligence systems are neural networks and large amounts of resources are used to improve this technology. One such improvement is the creation process of the neural network. In the creation of neural networks, much time is spent in finding the optimal architecture, topology, and weights. While this process is negligible for many problems, many cases require a significant effort to construct adequate architectures.

1.1 Background and motivation

The process of constructing and training neural networks is often time-consuming as well as requiring expert knowledge to be done effectively. To make learning through neural networks more attainable for the general populace, automating the process is vital. Existing methods are already able to construct expert level topologies, however, often with the use of days, or even months of GPU hours, as can be seen in [32]. Such computing power is unrealistic for the vast number of researchers or even cooperation's.

This thesis explores the possibilities of decreasing the time and computer power needed in the creation and training of neural networks so as to decrease the effort required to do ma-

chine learning with neural networks. The process, called Topology and Weight Evolving Artificial Neural Network algorithms, involves exploring the search space of possible architectures and weights, often through a process of evolution. The method explored will be an adaption of Neuroevolution of Augmenting Topologies (NEAT) where the architecture search will be replaced with Monte Carlo Tree Search (MCTS). MCTS have met success in various situations in later years [1], and a successful combination of Topology and Weight Evolving Artificial Neural Network (TWEANN)s and MCTS can greatly improve upon the state of the art.

1.2 Goal

Goal *Reducing the search space of the Neuroevolution Through Augmenting Topologies using Monte Carlo Tree Search.*

The goal of this thesis is to reduce the number of states the NEAT algorithm will have to search through to encounter valid and acceptable solutions. To achieve this goal this thesis will explore the effect of applying MCTS, firstly to a general evolutionary algorithm to document general observations, and then to the NEAT algorithm itself to document domain specific observations.

To initialize the process of solving this goal, multiple questions must be answered. The research questions below have been chosen so as to gradually build knowledge of the domain to a point where the main goal can be achieved.

RQ1 How can Monte Carlo Tree Search be adapted to replace an evolutionary search process?

RQ2 In which ways can Monte Carlo Tree Search be adapted to benefit Neuroevolution of Augmenting Topologies, and which results does the adaptions yield compared to the standard algorithm?

1.3 Research methods

The research method in this thesis will follow the same general path as that proposed in [19]. The paper proposes six main activities with the aim of producing and representing information.

The six activities are as follow:

1. Problem identification and motivation
2. Objectives of a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

Chapter 1 outlines both problem identification and the motivation for selecting this as the topic of choice, as well as the reasoning behind the proposed solution to the problem. Chapter 4 will detail the design and development phase, as well as early work. A demonstration of the results, as well as the evaluation of the proposed solutions, can be found in Chapter 6, while this thesis in its entirety represents the sixth and last point.

1.3.1 Research Protocol

The research protocol used in this thesis is the snowball method as described in [30]. A short summary of the method, as well as an explanation of why it is used in this thesis, will follow in this section while a utilization of the method can be found in section 2.

Finding Start Set

To begin the snowballing method, a good start point should be found. The start set will be the entry point to finding the rest of the literature and as such must accurately be identified.

The start set in this thesis was identified by a combination of finding articles as a basis for the research and using a search query on relevant search engines to find research tightly connected to the articles already found.

The basis of this thesis is [26] and [28], and as such these are the very beginning of the start set. Relevant articles were then selected from both the citation list of the papers in the start

set, as well as articles that cite the start set. Named as forward and backward snowballing. The process is then repeated as necessary with the new papers found until no new relevant papers occur.

Iterations

When the start set decided both forwards and backwards snowballing can begin. Backwards snowballing includes using the reference list of the paper being examined to find earlier papers than the one being examined. Forwards snowballing is looking at the papers referencing the paper being examined. This will result in newer papers than the current one. Papers that are decided to be relevant are added to a set of new papers to examine. This iterative process is repeated with the set of new papers until no new relevant papers can be found.

1.4 Contributions

The contributions for this thesis are:
C1: Examining ways of changing the search protocol of NEAT to a more structured tree search.
C2: An implementation of NEAT using MCTS as the search protocol.
C3: Comparisons between the standard NEAT algorithm, and two new methods proposed in this thesis.

Chapter 2

Related Works

This section will present relevant state of the art within the field of study of this thesis. The section will aim to give the reader an in-depth account of problems and solutions from early development within the field to recent improvements.

2.1 Network Architecture Construction

Various approaches to architecture search, explained in Section 3.1.1, have been proposed, especially in later years, and great strides in both accuracy and search speeds have been achieved from earlier algorithms. The search for algorithms which automatically creates neural networks have persisted since the 1980s and in [23] outlined some of the first combinations of genetic algorithms and neural networks. The paper outlined the problems regarding *competing conventions* as well as illustrating a direct genome encoding scheme using a connection matrix, showed in Figure 2.1, describing how such an approach will grow exponentially in the size of matrix needed with the number of nodes in the network.

NEAT, proposed by [26], addressed both the problem of competing conventions and the exponential growth of the genome representation. As this thesis builds upon NEAT, a more in-depth explanation of the algorithm can be found in Section 3.3, however, a short summarization will be given here. NEAT utilizes node- and connection-genes to describe a network, where node-genes specify the nodes in the network, and connection-genes describe the relationship between each node. If there is no connection-gene specifying a connection between

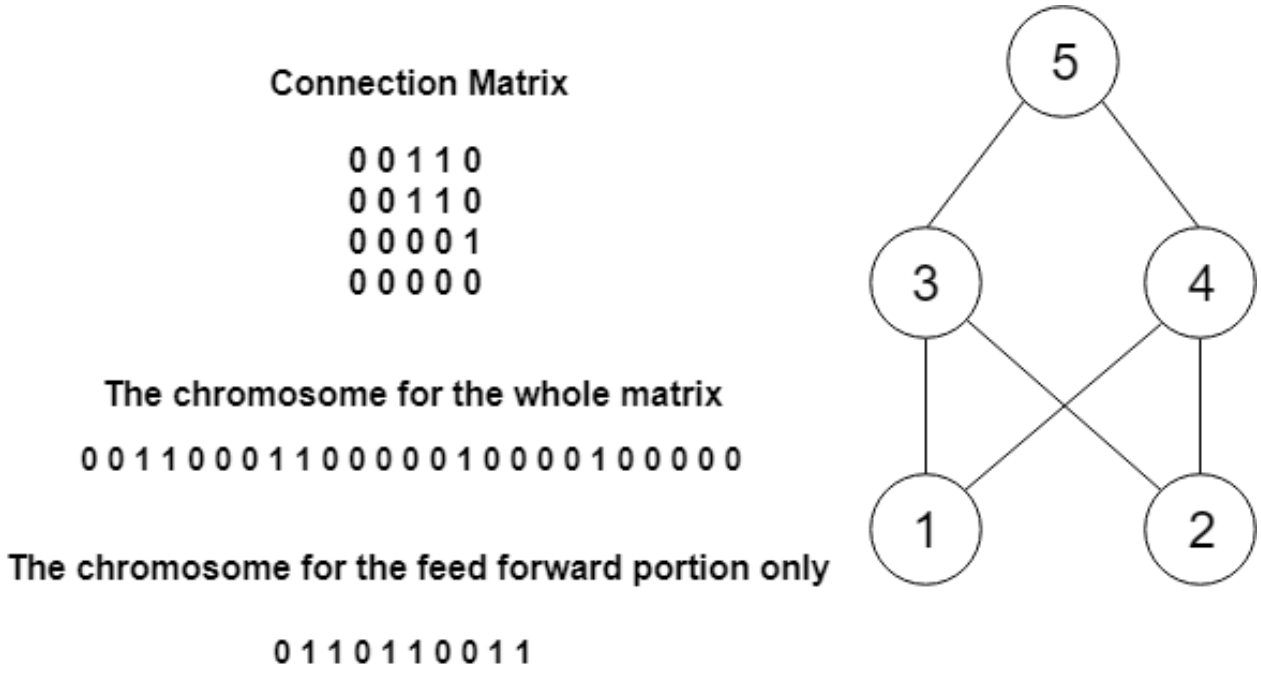


Figure 2.1: Direct encoding of genome given as a Connection matrix with values restricted to 0 and 1, 0 being no connection and 1 being a full connection. New nodes are inserted by extending the matrix with both a row and column. The figure is recreated from [23]

two nodes, the nodes are assumed to be not connected. Thus the algorithm does not need to store the relationship between all nodes, only the nodes which are connected. As a solution to competing conventions, NEAT gives all genes an innovation number, (see Section 3.3.2), which can be compared between genomes to lower the probability of losing information in crossovers. NEAT encounters difficulties when trying to solve problems requiring large networks because of the number of possible permutations large networks can have which makes the solution space slow to search through, as stated in [15].

The ideas behind NEAT were further developed by [25], in which Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) were proposed. HyperNEAT tackles the problems of scaling from NEAT by using an indirect genome representation instead of the genome representation in NEAT. HyperNEAT works on the assumption that weight values of an Artificial Neural Network (ANN) can be described as a function of the topology of the network. An assumption which has proven to be quite successful in many problems. HyperNEAT utilizes two different genomes, the first being what is called a substrate. The substrate is a neural network with a fixed geometrical topology where only the weights need to be defined. The weights are

defined with the use of the other genome, a Compositional pattern-producing network (CPPN). The CPPN takes as inputs spatial information of nodes in the substrate and outputs connection weights between the nodes. The fitness of the CPPN is calculated from the results from the substrate, and used to evolve the CPPN using NEAT. CPPNs tend to produce non-random patterns when provided with spatial inputs which tend to create networks with less chaotic outputs. In HyperNEAT the geometric ordering of the nodes in the substrate plays a large role for the final result, however, there are variations where the substrate is evolved as well [21]. Despite promising results, [10] showed that the efficiency of HyperNEAT decreases as problems become more complex by testing on multiple different problems.

Neural Architecture Search with Reinforcement Learning (NAS), proposed by [31], is an approach where a controller is given responsibility for learning to create the best neural network architectures through reinforcement learning. The paper works on the observation that Convolutional Neural Network (CNN)s can be described by their parameters such as number of filters, filter height, filter width, stride height, etc. Picking these parameters can be seen as picking actions in a reinforcement setting. The controller, being a Recurrent Neural Network (RNN), is trained through reinforcement learning by picking actions, looking at the fitness of the resulting CNN (after being trained) and using that fitness to update its own weights. Achieving impressive results by hitting about 4% error rate on CIFAR-10, the algorithm is still very resource intensive, using 800 GPUs to train 800 networks at any given time.

A similar approach is given in [15] where the authors propose an approach to adapt NEAT to effectively evolve deep neural networks. The authors firstly propose DeepNEAT where the nodes in the original NEAT genome no longer represents a neuron but instead represents a layer in the deep neural network. The networks are trained with gradient descent for a fixed number of epochs and fitness is evaluated based on the performance of the network. According to the paper, this approach often produces complex and irregular structures and so Coevolution DeepNEAT (CoDeepNEAT) is proposed next. CoDeepNEAT, mainly inspired by the work of [16], define the concepts of modules and blueprints. Modules represent small deep neural networks while blueprints are graphs where nodes contain pointers to specific modules. In this way CoDeepNEAT can produce repetitive modular networks. The results in the paper are promising but as stated in the [15], training time for each deep neural network is measured in days on

modern GPUs and during evolution up to thousands of networks must be trained. As such the algorithm is only viable if no time constraint is given.

Also related to NAS, [5] suggests a simpler approach to architecture search. The proposed algorithm is called Neural Architecture Search by Hillclimbing (NASH), noticeably replacing reinforcement learning with hillclimbing. Network architecture search by hillclimbing is a relatively straightforward method, but presents undeniable results, in this case, hitting 4.7% on CIFAR-100 with two days of training on one GPU representing a clear speedup from many state of the art algorithms. NASH involves an iterative process of selecting the best current solution and creating k neighbors which are modified n times by applying one of three mutations each time. The process is illustrated in Figure 2.2. The networks are trained for a set amount of epochs with stochastic gradient descent on each evolutionary iteration, and the trained weights are inherited by the children. This means that all networks except for the first iteration start training from the same weights as the parent architecture. A problem often encountered with hillclimbing is getting stuck in a local optimum, however, their results do not show this to be the case in this instance.

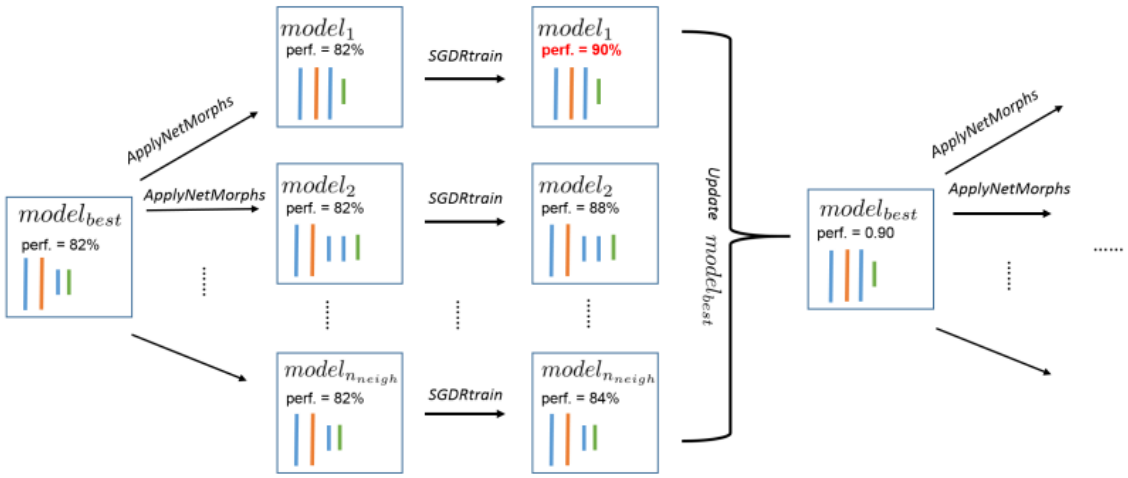


Figure 2.2: Showing how the current best method is morphed into multiple children repeatedly. The figure is retrieved from [5]

It is also worth noting that the authors propose a baseline of randomly constructing and training networks with stochastic gradient descent. This very simple algorithm is able to achieve 6% - 7% test error on CIFAR-10.

[12] introduces a novel way of representing architectures. Architectures are represented in three hierarchies, going from more to less detailed as shown in Figure 2.3. Since the genome is given as a directed graph, manipulations to the network can be done through mutation of the graph. Adding, removing and altering of edges are approaches given in the paper. As all three levels are given as graphs, this enables the same mutations to be done on regardless of the level. This enables the algorithm to efficiently construct larger networks consisting of sub-parts in various different configurations to solve problems.

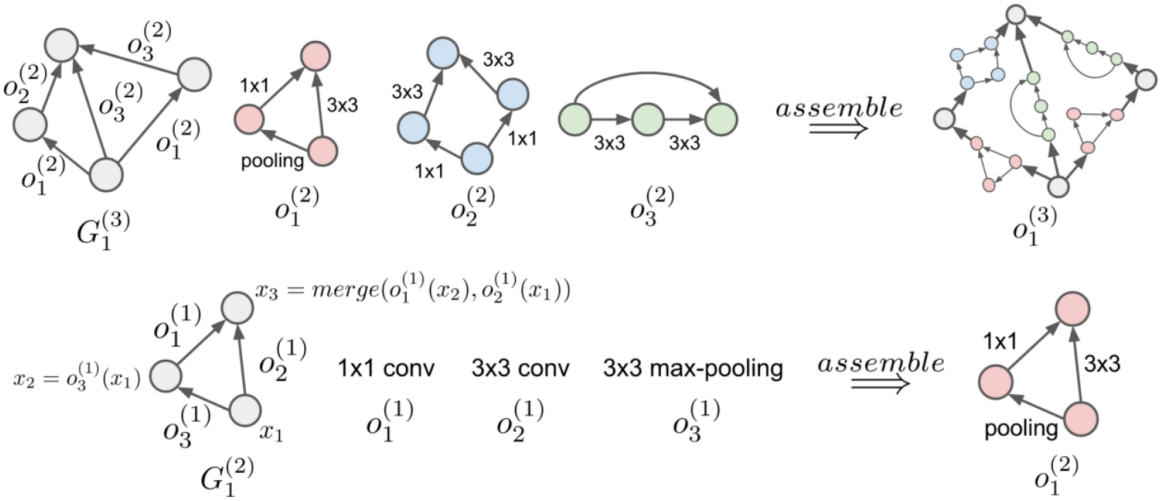


Figure 2.3: Figure showing assemblies from level 1 ($o_1^{(1)} o_2^{(1)} o_3^{(1)}$) to a level 2 structure ($o_1^{(2)}$). Level 2 structures ($o_1^{(2)} o_2^{(2)} o_3^{(2)}$) can then be assembled into a level 3 structure ($o_1^{(3)}$). Genomes are given as a directed graph. The figure is retrieved from [12]

PathNet, proposed in [8], seeks to create one large neural network which can be trained to perform multiple different tasks. To achieve this, PathNet initiates populations of competing pathways through the network. The pathways are trained for a set number of iterations before the best pathways are mutated and trained again in the next population. When a satisfactory result is achieved, the winning pathway is fixed, so that further training does not delete already learned solutions. Interestingly, the evolution, in this case, is used to guide where learning (gradient descent) should be applied in the network, instead of evolving the topology itself.

2.2 Monte Carlo Tree Search

MCTS has been successfully utilized in various programs requiring some sort of decision process, most notably perhaps in various computer games. However, the method can be used in various problems which can be represented as tree traversal-problems. The term was first coined in [4] where the algorithm was used for a 9x9 GO-game. Recently it has often been used in games with large search spaces ([7], [13], [11]), and maybe most notably in alphaGo [24] where it was used in conjunction with value- and policy networks to explore the search space of GO.

2.3 Architecture Search using Monte-Carlo Method

[29] achieves notably results by treating architecture search as a tree problem and utilizing Monte-Carlo planning as well as two derivations of Upper Confidence Bound applied to trees (UCT), explained in Section 3.4, to quickly search through architectures. One of the derivations being a reward predictor which can estimate the reward from doing changes to existing networks. Notably, since almost every new network has to be trained before fitness can be evaluated they only have time to run a relatively small number of roll-outs, which means that the algorithm must converge to a solution quickly. Knowledge transfer using Net2Net [3] were used to speed up the algorithm as well.

Recently [28] presented AlphaX, an algorithm which combines MCTS and a meta Deep Neural Network (DNN) to predict the value of sampled architectures. The meta DNN uses previous experience to predict the performance of architectures. This is used to 1) heuristically guide the search towards promising regions, and 2) preemptively backpropagate the accuracy with only the score given by the meta DNN as a placeholder until a real score can be given. This is done since each new network must be trained which will give a time between selection of an architecture and the fitness of that architecture being returned. The overall networks in this algorithm are structured in chunks, called cells, which are fixed. There are two types of cells, normal and reduction, where Normal cells maintain the input and output sizes, while reduction cells reduce the size of inputs and outputs, by half. Inside each cell, there are blocks containing neural network structures, which are evolved as the algorithm progresses. New architectures are selected using MCTS, according to the fitness evaluation done by the meta DNN. The algorithm

runs using multiple GPUs over hours and days, meaning this algorithm is not viable for various researchers or groups.

Chapter 3

Background Theory

This section will give brief introductions to the main concepts and algorithms used in this thesis.

3.1 Artificial Neural Networks

ANNs are a set of systems based originally on biological processes in brains [27]. ANNs contain neurons (also called nodes) joined by directed connections controlling the flow of information through the network. Often, the neurons are divided into layers, with the directed connections connecting the layers, as can be seen in Figure 3.1. This is called a fully connected feedforward network since data propagates from left to right, and all neurons in a layer are connected to the layer before.

Neurons in neural networks contain a function called an activation function, which takes as input a weighted sum of the inputs to that neuron and calculates an output. This process is shown in Figure 3.2 for one neuron. The output of the neuron is either passed to the next neuron in the network, or it may be considered the output of the network if the neuron is the last of the network.

The connections between neurons represent weights which decide the importance of the input to the neurons by multiplying the input to the corresponding weights. As can be seen in Figure 3.2, if an input is multiplied with a low number, say 0.0, the original value of that weight will have no say in the final output of the neuron. It is these weights that are changed when the network is trained so that the network can learn how each input in the network should be

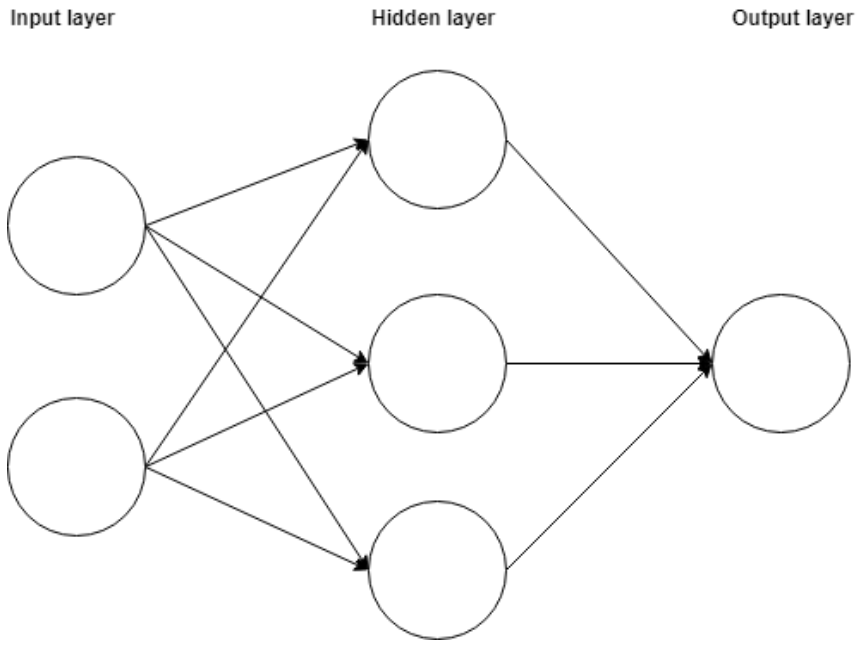


Figure 3.1: An illustration of a simple feed-forward neural network consisting of two input-neurons, three hidden-neurons and one output-neuron. This network is fully connected, meaning all neurons are connected to all neurons in the layer before.

modified.

Multiple activation functions exist and are used for different purposes in different problems. Popular ones are identity, tanh, softmax and sigmoid. Notably, activation functions map the original output value of a neuron between two values, such as (-1,1) for sigmoid, or (0,1) for the logistic activation function.

3.1.1 Neural Architecture Search

Neural architecture search is the process of searching through architectures to find topologies according to some search strategy [6]. Figure 3.3 shows an abstract illustration of a general search strategy. The search space defines the set of architectures that can be represented, in theory. The search strategy is the method with which the search space is traversed. A problem with search strategies is often to find the optimal balance between exploration and exploitation where too much exploration will cause the algorithm to run for too long while too much exploitation will lead to premature convergence on a sub-optimal result.

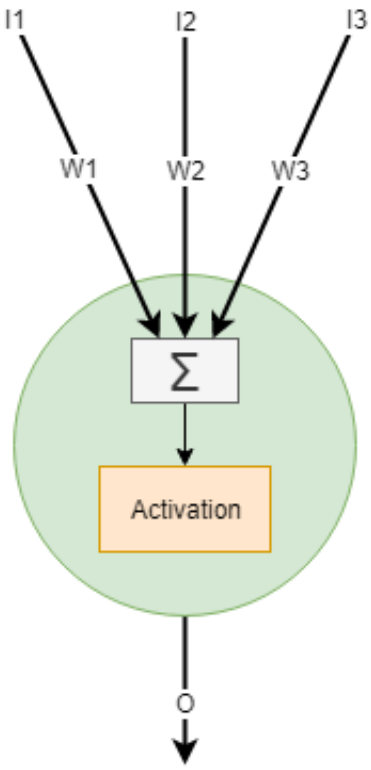


Figure 3.2: A general activation function showing how the product of the inputs $I1, I2, I3$ and weights $w1, w2, w3$ are summed and passed to an activation function, before being sent as output O

3.2 Evolutionary Algorithms

Genetic algorithms are a set of algorithms focusing on solving optimization problems. Inspired by nature, these algorithms mimic the process of biological evolution by utilizing the principle of evolving better and better solutions through an incremental process of population creation and fitness evaluation. By creating new populations based on good traits from earlier populations, as well as some rate of random mutations, these algorithms generally are able to move towards more and more optimal candidate solutions. Genetic operators are specific operators used to change populations over time. Specific operators are selected for specific problems, however, the most used are $crossover, selection$ & $mutation$.

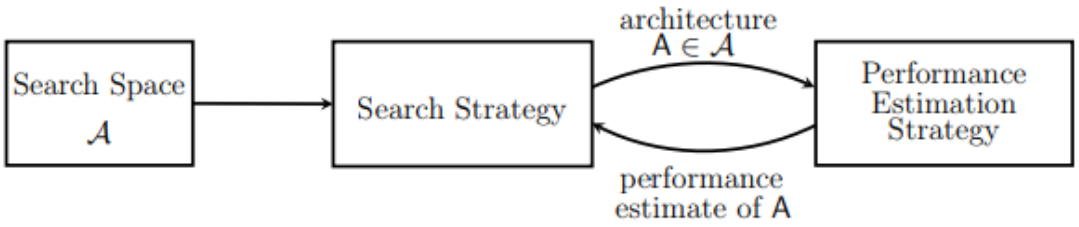


Figure 3.3: Simplified illustration of a neural architecture search scheme. A predefined search strategy selects an architecture **A** from the search space. The architecture is then evaluated by a performance estimation strategy which passes the estimation back to the search strategy. Figure copied from [6]

Base Algorithm

Implementation variation will naturally differ from problem to problem, however most will contain the basic instructions shown in Algorithm 1.

Algorithm 1 Base Evolutionary Algorithm copied from [20]

```
1: procedure BASE EA(a, b)
2:     initialize population
3:     evaluate population
4:     while !stopCondition do          ▷ End when good enough solution is found
5:         select the best-fit individuals for reproduction
6:         breed new individuals through crossover and mutation operations
7:         evaluate the individual fitness of new individuals
8:         replace least-fit population with new individuals
```

The initial population is generated and evaluated in lines 2 and 3. The best k individuals of the population are selected in line 5, and these are used as the basis for the next group of individuals through crossover and mutation. In lines 7 and 8 the least fit individuals of the population are replaced with new and hopefully better individuals.

3.2.1 Genotype to Phenotype

Another concept inspired by biological evolution is the distinction between genotype and phenotype where genotype is the hereditary information while the phenotype is the observed behavior of the individual. This distinction is used to simplify the process of making changes to the solution via the genetic operators. To illustrate the difference the well-known traveling salesman problem will be used. Given the genome $[city_1, city_2, city_3, city_4, city_5, city_6]$, each element

represents a unique id for a city, and the order of the elements represents the order in which the salesman will have to visit the cities. A possible phenotype for this genome can be seen in Figure 3.4.

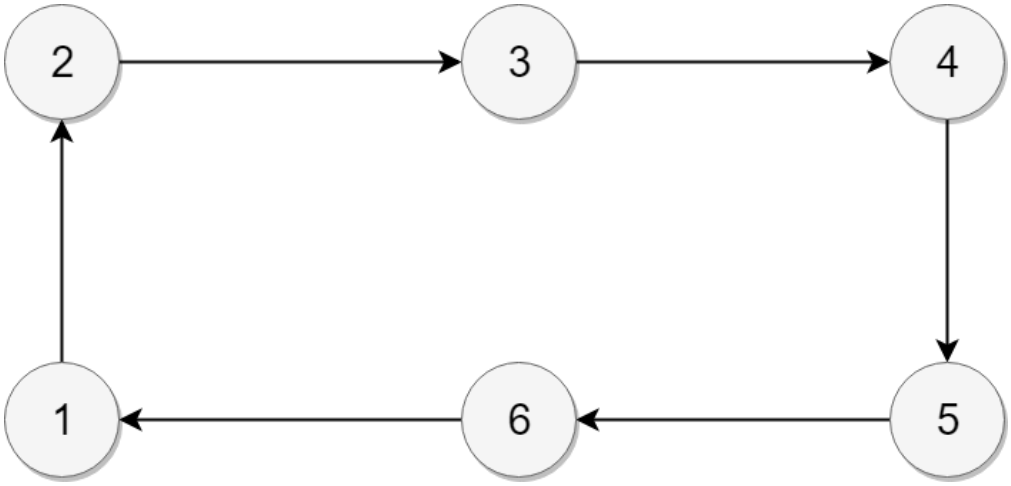


Figure 3.4: Figure showing a possible phenotype from the genome $[city_1, city_2, city_3, city_4, city_5, city_6]$

3.2.2 Genetic Operators

Crossover is the process of combining the hereditary data from two parent individuals to create a child individual. The operation is inspired by natural sexual reproduction which combines traits from parents in nature. By combining the information from different individuals with different traits multiple times it is hoped that at least one of the offspring will contain the best traits from both parents.

Selection is used to decide which parents to use in reproduction. Multiple selection strategies exist, however, most generally have a higher chance of selecting the more fit parents rather than the less fit. This ensures that good traits continue to the next population.

Mutation is used to diversify the population by introducing new traits that cannot be produced through crossover. Mutations are often random changes to genomes which often creates solutions that are worse than the original genome. However, by doing many mutations on many individuals, some good traits may be discovered which did not exist in the population before the mutation.

3.2.3 Fitness Evaluation

To determine how good individuals are, they must be evaluated in some manner. The evaluation is problem specific, but the fitness is always used to guide the evolutionary process towards better and better solutions.

3.3 Neuroevolution of Augmenting Topologies

NEAT is a specific algorithm developed by Stanley et al. in [26] as a method to generate artificial neural networks through an evolutionary process. NEAT is part of a group of algorithms called TWEANNs, where the algorithm will simultaneously try to optimize both the weights and the topology of a neural network.

Incremental Complexification

The initial structures in NEAT are very simple starting with input-nodes, output-nodes, and connections between some, or all, of them. The genomes are made incrementally more complex over time to ensure solutions with small networks gets a chance to evolve weights. Small networks a preferable since they take less time to further evolve, as well as using less time to compute a result. In addition, evolving small structures first keeps the search space of possible structures as small as possible which heightens the performance of the overall algorithm.

3.3.1 Representation

In NEAT the genotype of a neural network is divided into two different gene types: Nodes, containing an $node\_id$ and whether it is an input, output or a hidden node, and connections which contain the connection's input node, output node, weight, whether it is enabled or not and the innovation number of the connection. The $node\_id$ of the node genes are used in the connection genes to keep track of which nodes each connection gene connects, and is simply globally incremented when new nodes are added in the evolutionary process. $node\_id$ and $innovation\_number$ also serve a purpose in the crossover process, explained in Section 3.3.2. An illustration of the genome representation, and how it can be decoded into a phenotype can

be seen in Figure 3.5.

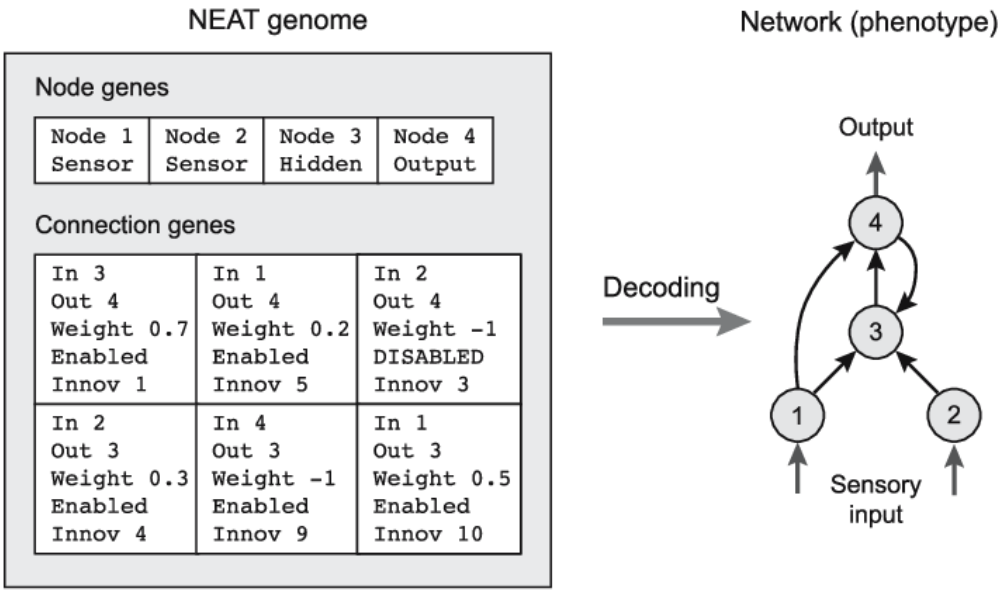


Figure 3.5: On the left: Illustration of how the genome in NEAT is divided between Node genes and Connection genes. On the right: The phenotype of the genome, showing the nodes and connections. Image copied from [9]

3.3.2 Crossover

Doing crossover on neural network architectures is not trivial since the computer will have to distinguish between similar and different nodes and connections in different genomes to be able to discern which exist in one parent and which does not. This is needed so that the crossover process does not create children who are missing important traits from either parent. To solve this problem, NEAT introduces $node_id$s for nodes and $innovation_number$s for genes, collectively named *history markers*. The *history marker* is a global counter which is incremented each time a new connection is created in the evolutionary process.

When doing crossover the genomes can be lined up using the *history markers* to ensure that genes that exist in both parents are always continued in the child, and genes that exist in one parent but not the other is continued in the child if they come from the best parent.

Figure 3.6 illustrates how two parents with similar, but not identical genomes can be combined to form a child. For gene one through five the genes are taken from a random parent, while the *disjoint* and *disjoint* genes are passed on from the best fit parent, or both if they are

equally fit.

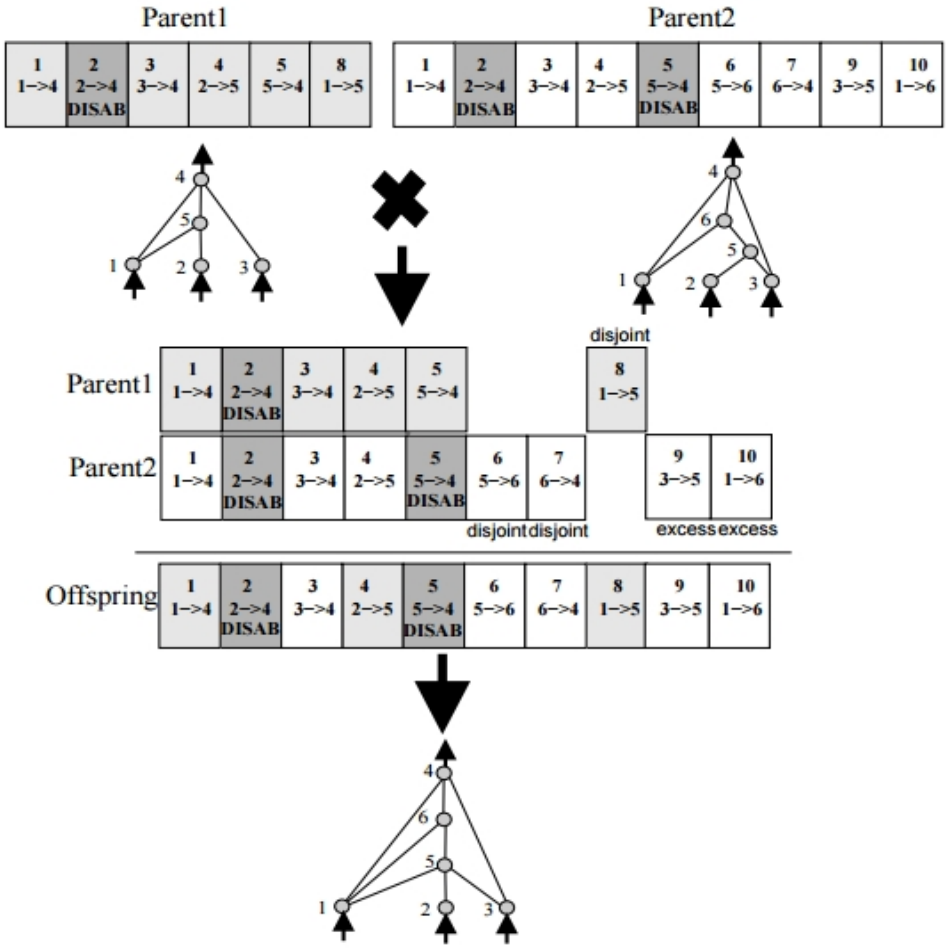


Figure 3.6: Crossover operation in NEAT showing how two parents can be combined to create a child. Image copied from [26]

3.3.3 Mutation

The structure of a network in NEAT can be changed by two different mutations, both of which are shown in Figure 3.7. In short, the two possibilities are adding a node or adding a connection. When adding a node it is always done by selecting a connection which is divided into two connections, with the newly made node as the node in between them. When adding a new connection, two nodes with no connection between them are selected and a new connection between them is created. New nodes and connections are given a *historical marker* to distinguish them from previous nodes and connections.

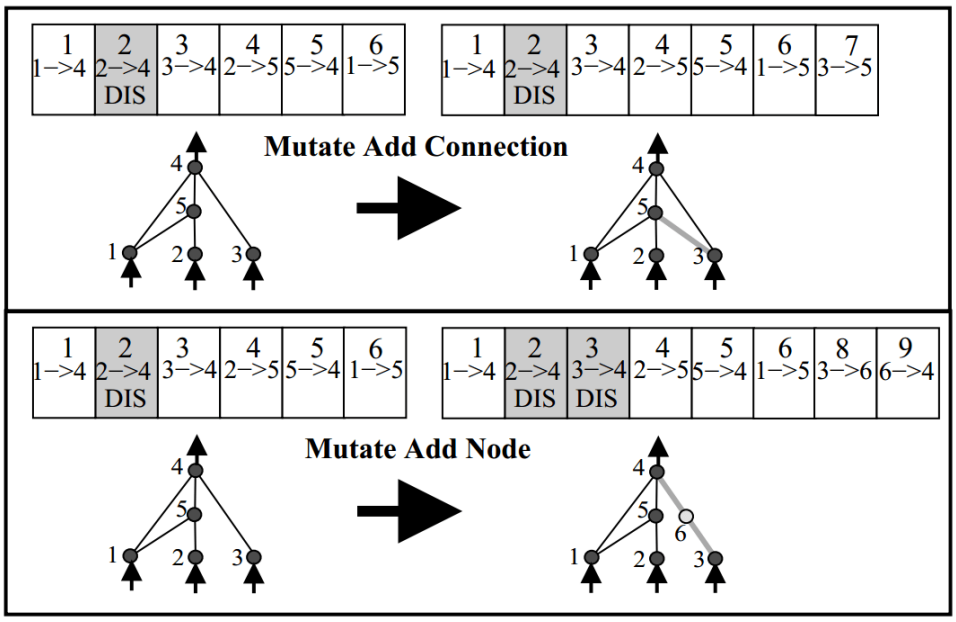


Figure 3.7: Top: Showing how a connection is added between node 3 and 5 and given the innovation number 7. Bottom: Showing how connection 3 is divided into 2 connections by adding a node in the middle. Image copied from [26]

Speciation

To prevent the extinction of solutions that are not yet viable or satisfying in comparison to other solutions, a speciation scheme is utilized. Speciation allows individuals containing specific traits to improve those traits by making the individuals only compete with other individuals containing the same overall traits. This prevents good, but not optimal individuals, from taking over the global population which will often lead to the entire population stagnating at a local optimum.

The history markings used in NEAT is utilized to calculate the distance between individuals (genomes) by looking at the number of excess and disjoint genes between individuals. The distance δ between the structures is calculated by adding the disjoint D and excess E genes, and the average weight difference of matching genes $\overline{W}$. The Equation 3.1 explained in [26] explains the δ - value.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{3.1}$$

Where c_1, c_2 and c_3 are constants used to prioritize the importance of the three factors and N is the number of genes in the largest genome used to normalize the values. According to [26], N can be set to 1 for genomes smaller than 20 genes.

3.4 Monte Carlo Tree Search

MCTS is a statistical search method which utilizes a Monte Carlo method of random sampling to create a heuristic for tree traversal [1]. The base algorithm consists of four steps: *Selection, expansion, simulation* and *backpropagation.* An illustration of these points can be seen in Figure 3.8.

3.4.1 Selection

Selection is done from the root node in the tree until a leaf node is encountered. The goal of the selection process is to guide the search towards the most promising section of the tree, while also ensuring that unsearched, or little searched sections of the tree aren't ignored. This balance is categorized as exploration vs exploitation with the balance between them given from the formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

where w_i is the best-found fitness of the subtree from the current node, n_i is the current simulations ran from the current, N_i is the number of simulations ran from the parent of the current node and c is a number weighting exploration and exploitation.

3.4.2 Expansion

When a leaf node is encountered after the selection process, one or more children is created such that the tree is expanded. Available children represent legal actions that can be done from the current node.

3.4.3 Simulation

Simulation involves doing random actions from a child node of an expanded node, either until a solution is found, a designated time frame is reached or a depth of the tree is reached.

3.4.4 Backpropagation

After a simulation is run the result is backpropagated from the node the simulation was run from, up to the root node in the tree. This value is used to calculate whether or not the branch is worth exploring more, or if other branches should be prioritized.

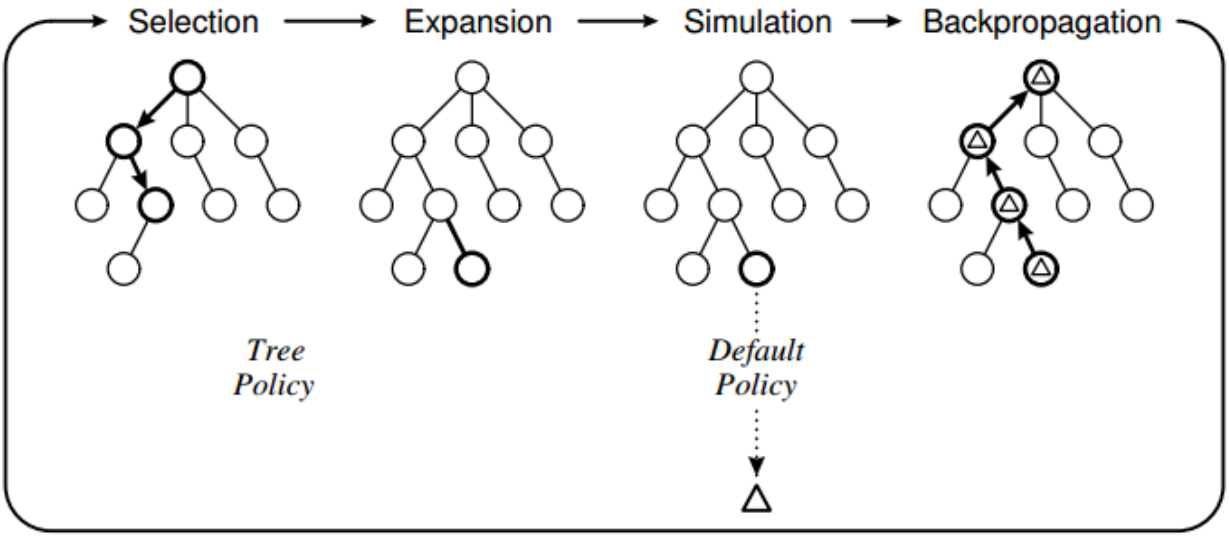


Figure 3.8: One iteration in MCTS. Image copied from [1]

Chapter 4

Methods

In this chapter the reimplementation of the reproduction scheme in NEAT will be explained, as well as the reasoning behind the implementation decisions. The implementation is written entirely in the coding language Python and is adapted from existing code written by [14], more specifically the newest fork by Github user "drallensmith"[1]. The reasons for using this existing code are mainly in regards to time constraints and result consistency. The project by [14] has been developed since 2015 and been tested extensively, which will result in more reliable results than would be possible with a system created during the time period of this thesis.

4.1 Initial Experimentation

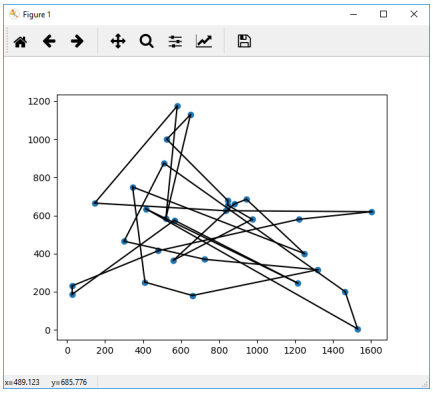
To test the initial efficiency of applying Monte Carlo Tree Search to an evolutionary problem the algorithm was first applied to the much simpler Traveling Salesman Problem (TSP). TSP is an NP-hard problem with a smaller, but still very large, solution space than searching through network topologies. This initial work was done to test the hypothesis of being able to efficiently treat an evolutionary problem as a tree search, as well as to measure the performance of such a solution versus a more standard genetic algorithm.

The algorithm was implemented by iteratively selecting and creating sub solutions by building a order of cities as the algorithm traverses down the tree. Each step down the tree will add one city to the existing sub solution. More formally, if we have 5 cities $[city_0, city_1, city_2, city_3, city_4]$

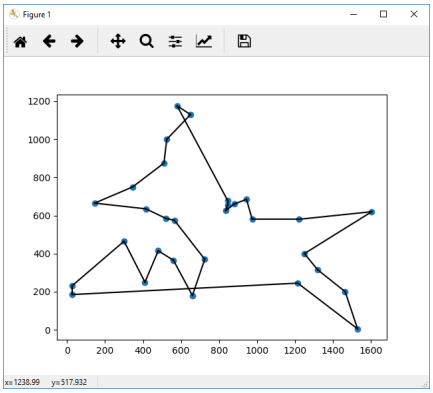
[1] https://github.com/drallensmith/neat-python

the root node in the tree will start with one of those cities. If the root node starts with $city_1$, the possible children of the root node will be $[city_1, city_0]$, $[city_1, city_2]$, $[city_1, city_3]$ and $[city_1, city_4]$. The cities of these children are fixed and cannot be changed, such that simulations on these nodes are only looking at the order of the rest of the cities. Selection is done based on the best-encountered score for each node when a set of simulations has been run.

As can be seen from figures 4.1(a) and 4.1(b) there is a noticeable difference between running a pure MCTS algorithm and using MCTS with Beam Search. When using only MCTS the algorithm was only able to generate very sub-optimal solutions because the search tree became too big. When running MCTS with Beam Search to reduce the number of nodes being searched the result is markedly better, however noticeably still not optimal.



(a) Using only Monte-Carlo Tree Search



(b) Reducing the search space by applying Beam Search

Figure 4.1: Comparisons between using only Monte-Carlo Tree Search and Monte-Carlo Tree Search with Beam Search on 30 cities.

Adding more cities resulted in increasingly worse solutions, however as can be seen in Figure 4.2 with 52 cities the algorithm is still fairly consistent inside the sub-solutions, that is, there are only one large jump between two far away cities.

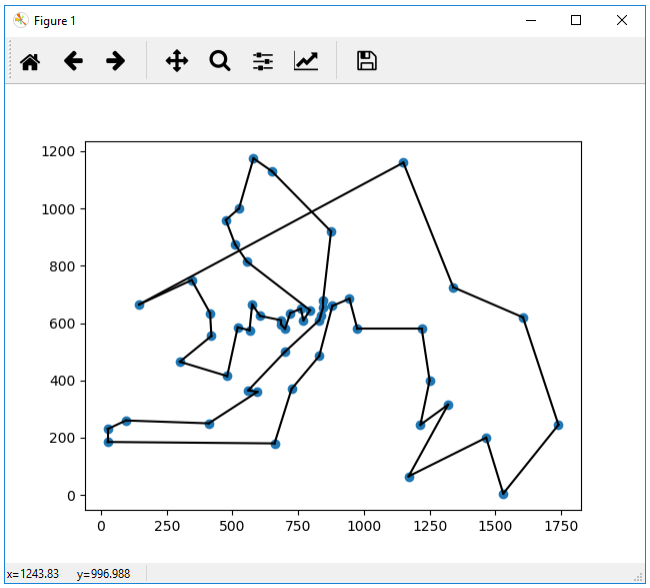


Figure 4.2: MCTS with Beam Search on 52 cities.

4.2 Main Implementation

Two different methods for updating the weights of the neural networks are explained below. They were implemented to see the difference in approaches as each theoretically has their own advantages and disadvantages: Hill climb tends to be faster and able to quickly climb towards a solution, however, they often stagnate at a local optimum. Evolution is nearer to the standard NEAT implementation. Evolution tends to climb slower towards solutions, but are also better able to reach global optima. Both of the learning schemes are done on one node at a time as the selection process progresses down the search tree, with a certain percentage.

4.2.1 Tree Implementation

The main element of the proposed system is the MCTS and the process of expanding the tree. As can be seen in Figure 4.3 the tree expands by mutating the topology of an initial root network A. The root network, in this case, has two input nodes and one output node, with both input nodes being connected to the output. There are two possibilities for generating child nodes from a network: Splitting an existing connection and adding a neuron in the middle or connecting two previously unconnected neurons. The two only possible children of node A are therefore

nodes B and C. As can be seen in node D, a connection has been added between I_2 and the hidden node. The tree in the figure is not complete but it should be apparent that the width of the tree will grow exponentially as the number of nodes and connections becomes larger. The number of children of a network is the sum of connections and unconnected nodes, not including connections between input nodes and between output nodes.

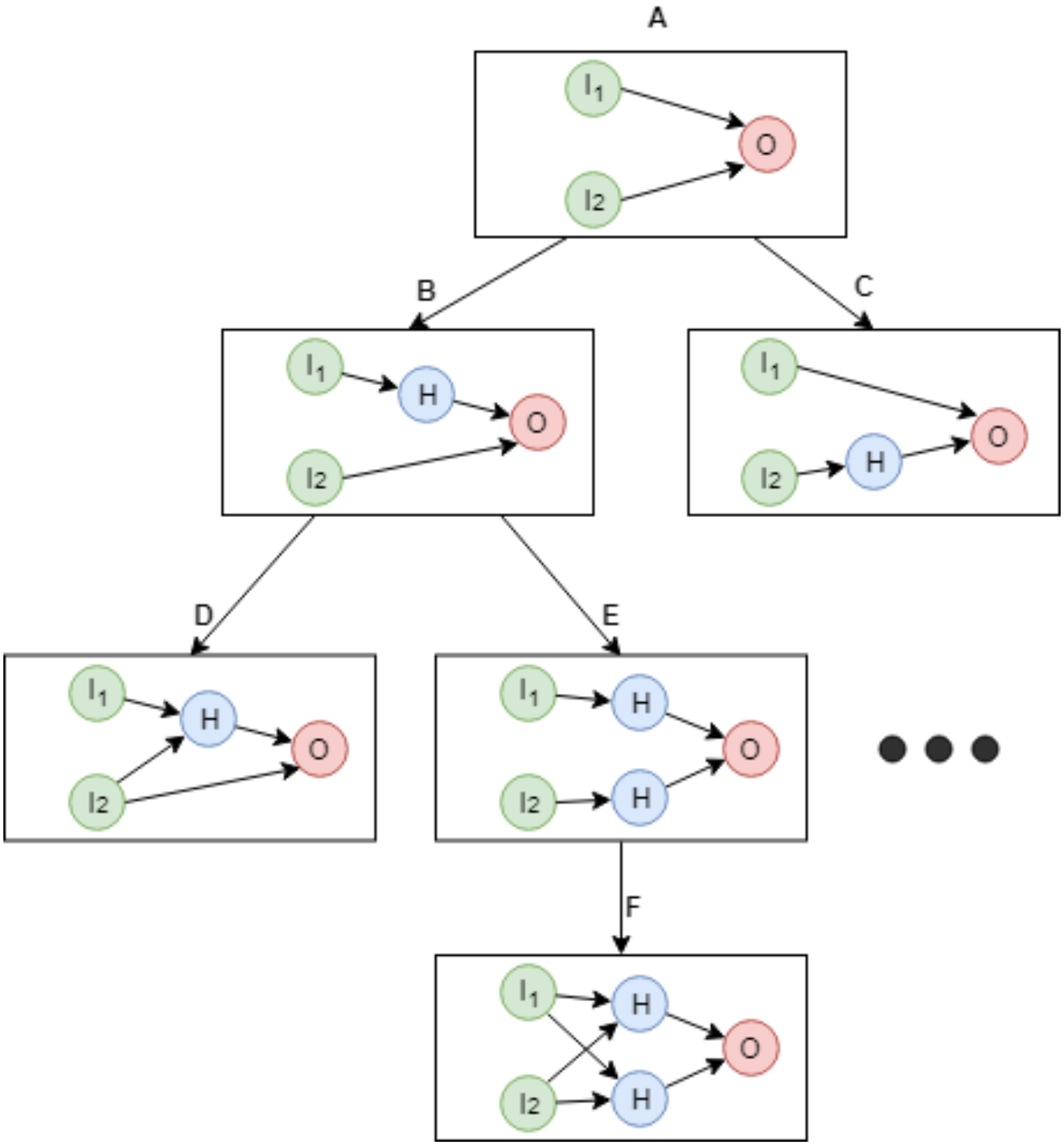


Figure 4.3: Incomplete progression of structures as search tree expands.

Pruning

Since the search tree will grow at a fast rate, an aggressive pruning method is vital so that the algorithm can manage to find solutions within a reasonable timeframe. As the selection strategy progresses down the tree there is a chance for each node selected to prune a set percentage of its children based on the best-found fitness in that subtree. Figure 4.4 illustrates how node C prunes one of its children, thereby removing the entire subtree from the search tree.

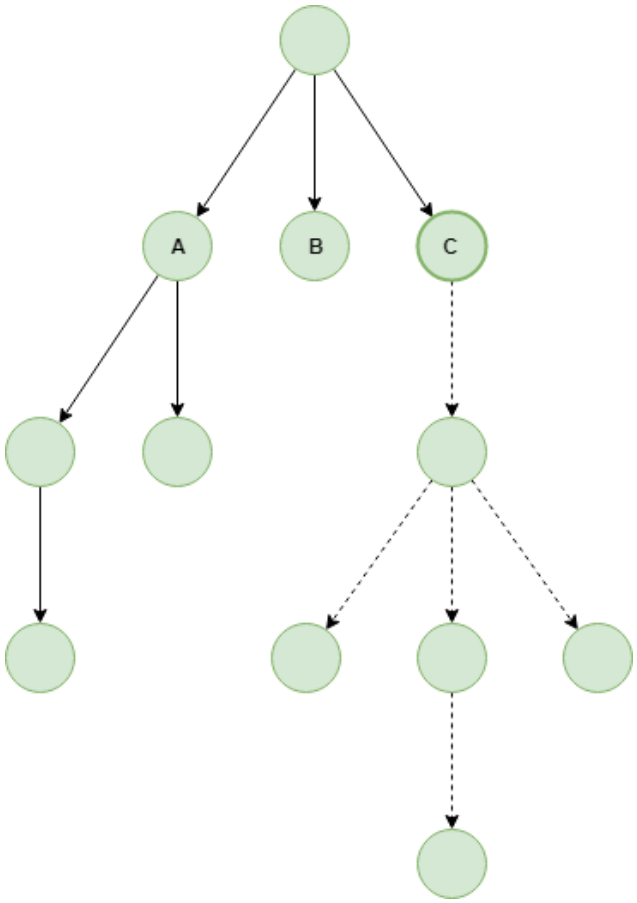


Figure 4.4: A sub-tree is pruned by removing node C from the tree.

4.2.2 MCTS with Hill Climb Local Search

Hill climb is an iterative process of creating or selecting the best current node, generating different neighbors of that node and setting the next current if there is a new solution which is better than the currently found best node. A more formal explanation can be found in Algorithm 2. This algorithm will be identified as **mcHillNEAT** for the remaining of this thesis.

Algorithm 2 Hill climbing algorithm

```
1: procedure HILL CLIMB(initial_state)
2:     priorityQueue = [initial_state]
3:     best_found_solution = initial_state
4:     while !stopCondition do                ▷ End after a certain number of iterations
5:         current = priorityQueue.pop()
6:         best_found_solution = current if current is more fit than best_found_solution
7:         create n-neighbours of current
8:         add all neighbours to priorityQueue
```

The hill climbing process is done for only a single node at a time with a certain percentage as the selection process proceeds down the tree. Multiple copies of the topology of that node are created with small mutations to the weights. Each solution is then added to a priorityQueue, and the process is repeated. See Figure 4.5 for an overview.

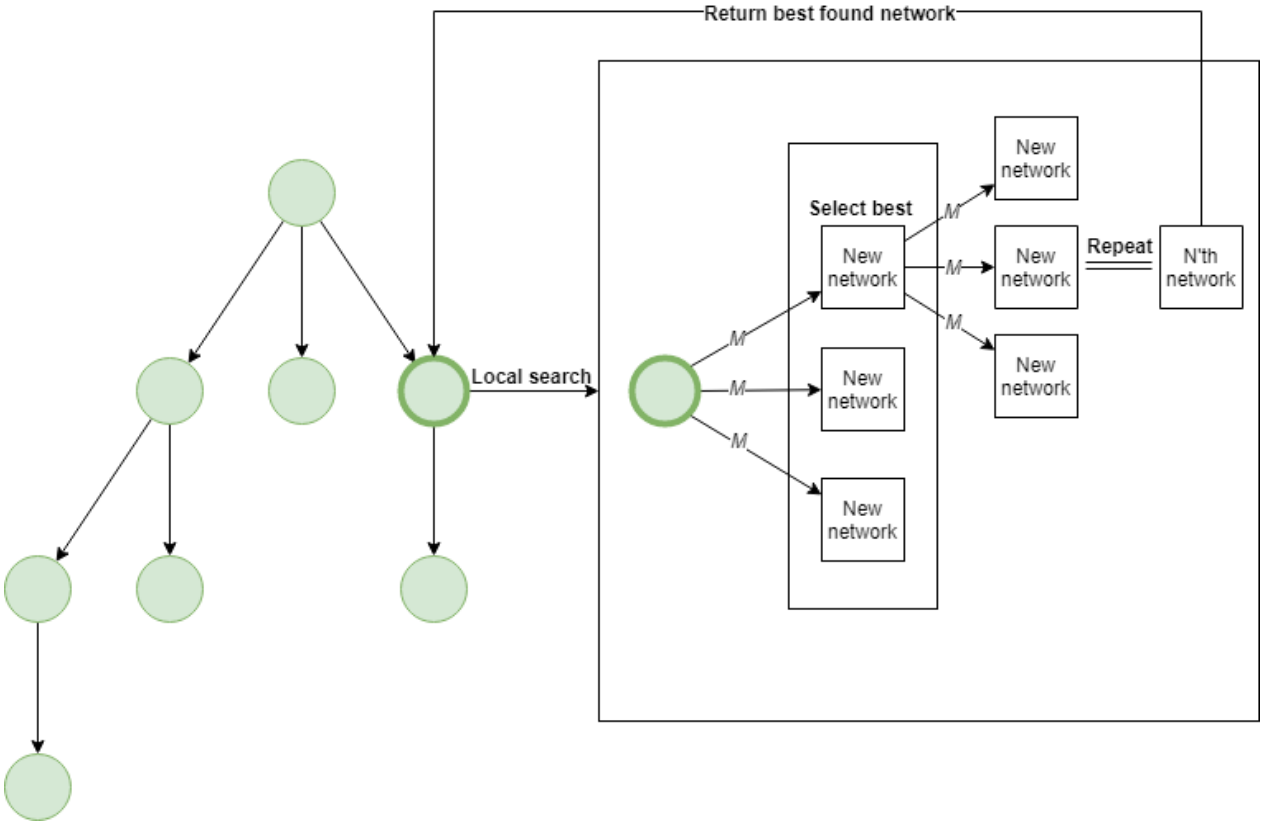


Figure 4.5: A node in the tree is selected for local search. The weights of the network are mutated to create several neighbors, the best of which are selected for further mutation. This process repeats N-times. The last network replaces the original node in the tree.

4.2.3 MCTS with Genetic Algorithm Local Search

This algorithm is more true to the standard NEAT algorithm by using the standard crossover-operation as well as a genetic algorithm which is close to that used in standard NEAT. This algorithm is done for only a single node at a time with a certain percentage as the selection process proceeds down the tree. Multiple copies of the topology of that node are created with small mutations to the weights. The copies are then selected as the initial population and placed into a cycle of selection, crossover and mutation (see Figure 4.6. In each cycle, a set of individuals are selected from the population for crossover. More fit individuals have a higher chance of being selected, but it is possible for all individuals to get selected. When new parents have been selected, children are created using the standard crossover method explained in Section 3.3.2. When the new child has been created it is mutated according to some chance and placed into a new population. This is repeated until the new population contains as many individuals as the last. The new population then replaces the old population and the cycle starts again. At each iteration, the best individual is compared to see if it is better than already found solutions and replaces the old solution if it is. This algorithm will be identified as **mcWeightEvolNEAT** for the remaining of this thesis.

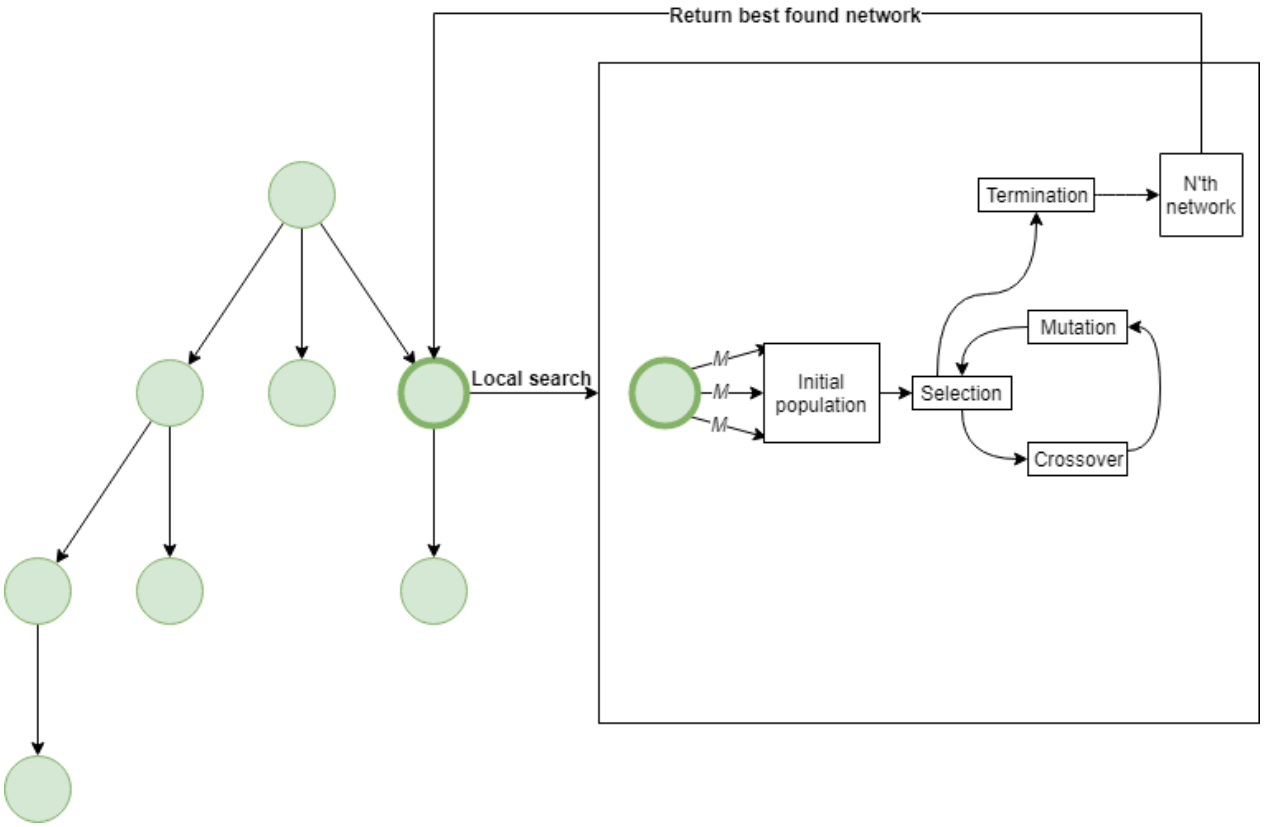


Figure 4.6: A node in the tree is selected for local search. The weights of the network are mutated to create the initial population. A genetic algorithm is run to find better weights. The last network replaces the original node in the tree.

4.2.4 MCTS with Partial Genetic Algorithm Local Search

This algorithm were created after the testing of *mcHillNEAT* and *mcWeightEvolNEAT*, based on the observations from the behavior of *mcWeightEvolNEAT*. As can be seen in figures 6.1, 6.2 and 6.3, the fitness graph for *mcWeightEvolNEAT* has a tendency to initially climb very fast and then stagnate in a local optima. This behavior is consistent standard hill climbing explained in [22, Chapter 4.1]. Figure 4.7 illustrates how hill climbing algorithms may climb towards sub-optimal solutions, and the theory is that a similar situation happens with *mcWeightEvolNEAT* since for a node in the network is replaced after each local search. This forces the node to search in the direction found by the very initial local search for that node. To combat this, the new algorithm, which still has the same method of running weight evolution, only runs the genetic algorithm one epoch. That is, only one new population is created when the local search is run. The entire genetic algorithm is then stored until a local search is run on the node again, at which point the

genetic algorithm runs one more population. Figure 4.8 illustrates this process.

The theory is that this will enable a larger part of the search tree to move in parallel towards solutions, thereby searching a greater part of the search space and ensuring that one node in the network does not get a disproportionate amount of training compared to other nodes. This algorithm will be identified as **mcWeightEvolPartialNEAT** for the remaining of this thesis.

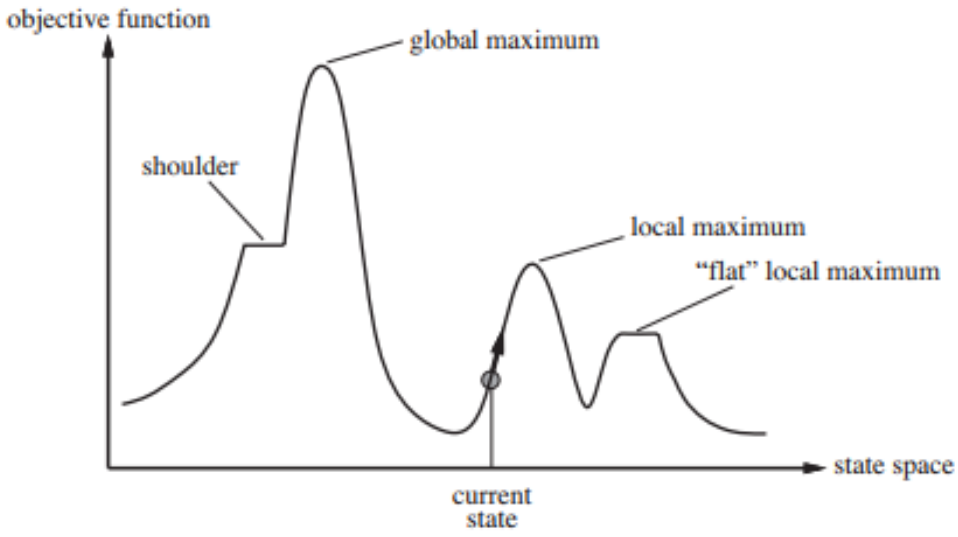


Figure 4.7: Figure illustrating the search space of a hill climb algorithm, with the agent currently traveling towards a local maximum. The figure is copied from [22, Chapter 4.1].

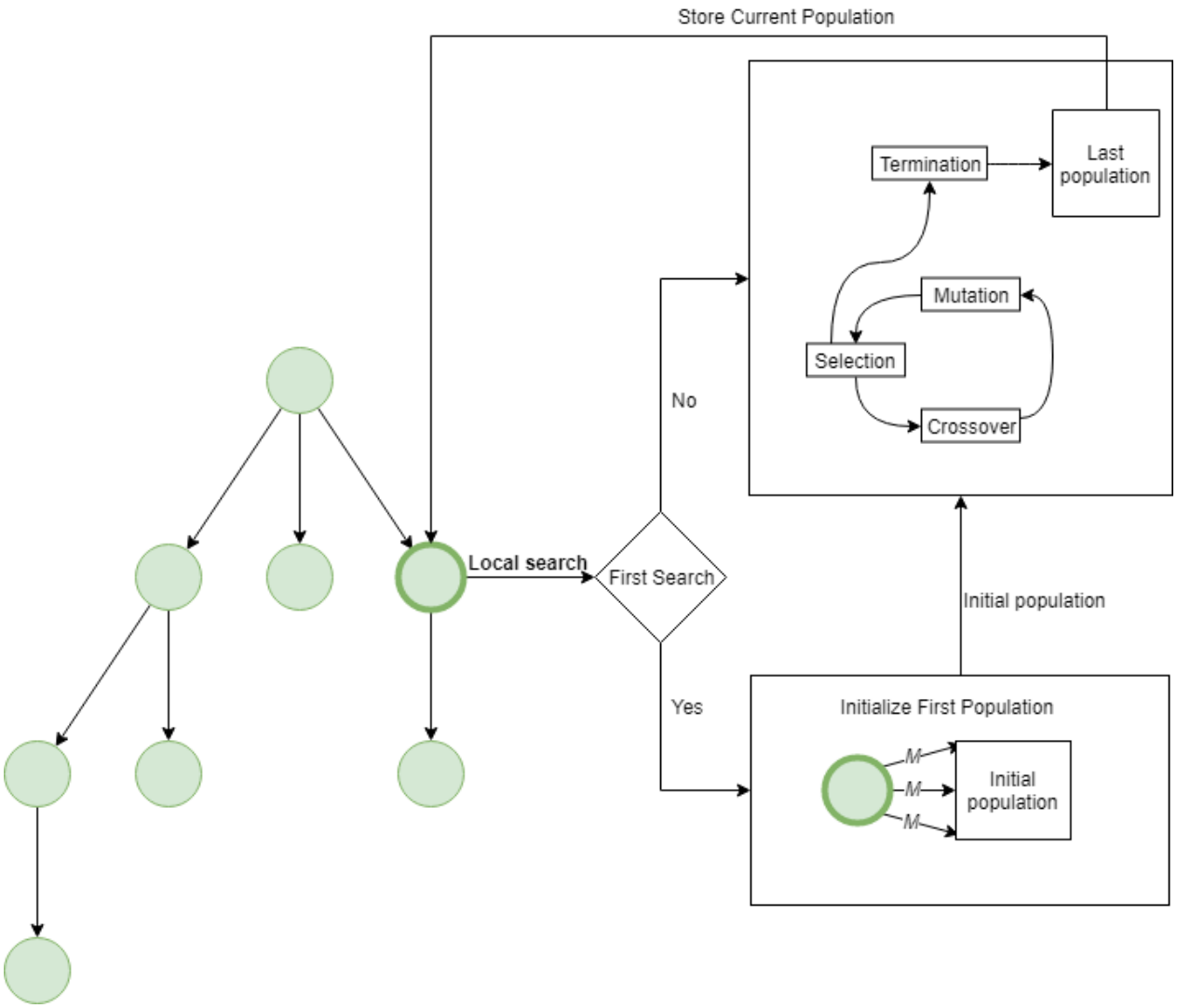
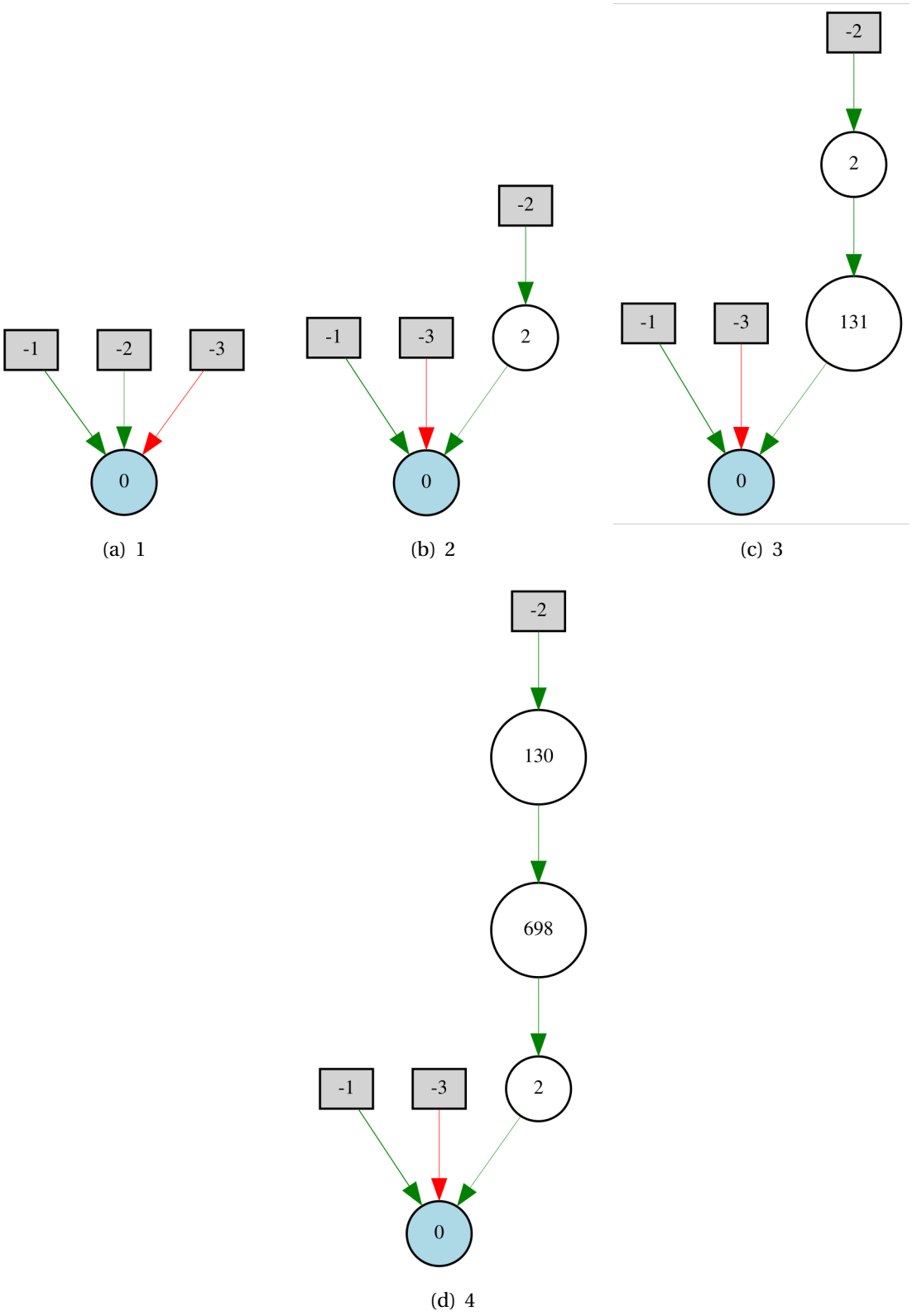


Figure 4.8: A node in the tree is selected for local search. The weights of the network are mutated to create the initial population. A genetic algorithm is run to find better weights. When an iteration of the genetic algorithm is finished the new population is stored until the next iteration is run.

4.3 Tree Progression Example

This section will illustrate a real case scenario of tree progression. Figure 4.9 show how networks are expanded. In this case it is clear that depth of the tree have been prioritized by the algorithm, and only input *-2* are connected to hidden nodes in the initial generations.



(a) 1 (b) 2 (c) 3

(d) 4

Figure 4.9: Example of how a tree search can progress down the search tree. Gray rectangles are the inputs to the network, blue circles are outputs and white circles are hidden nodes. Green and red arrows are positive and negative connection weights respectively.

4.10 is an illustration of how the best found neural network can look. In this case it can be seen that node 2 from 4.9 is still in the final solution, however the other nodes have been replaced.

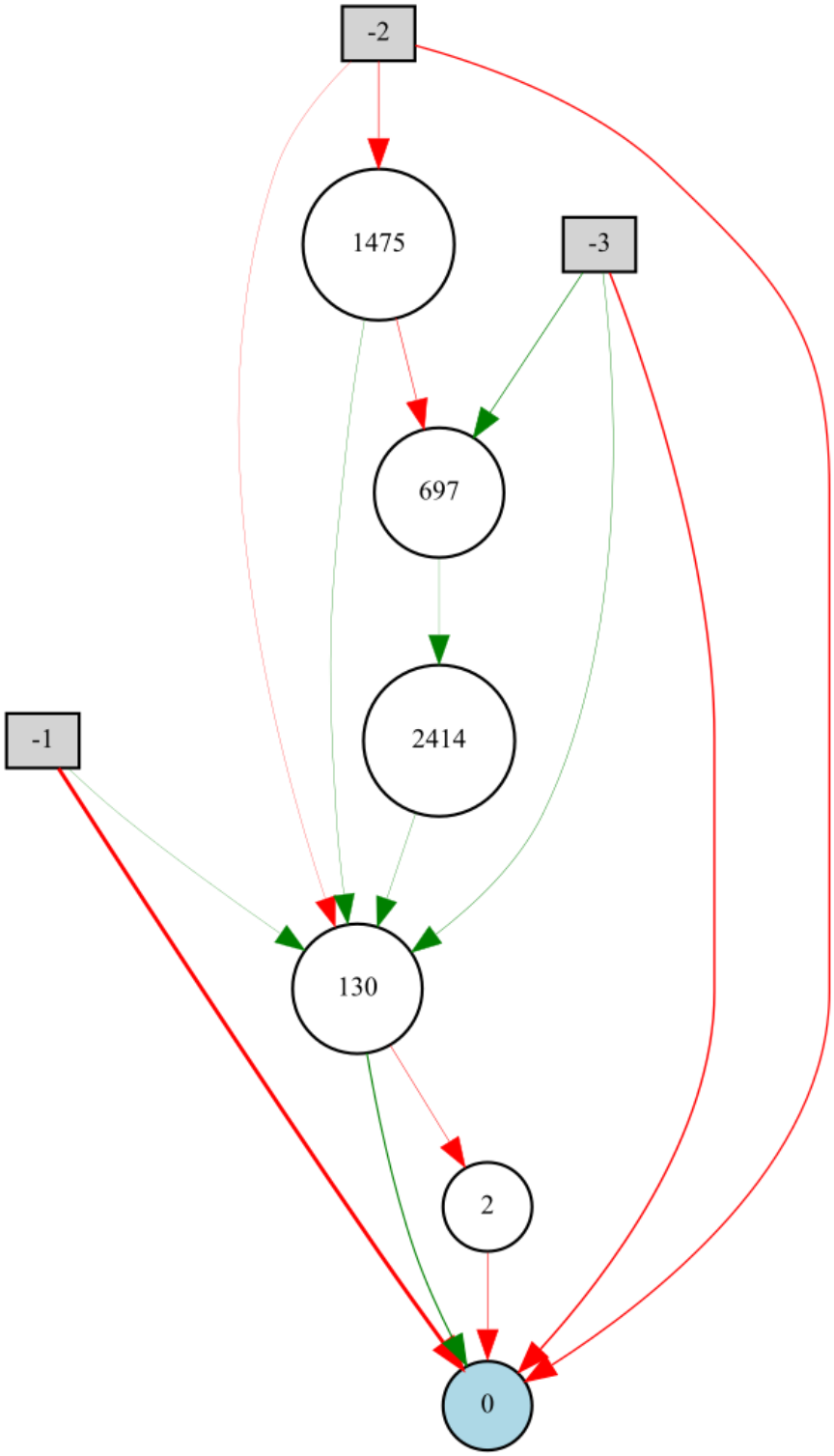


Figure 4.10: Example of how the best found neural network from running the algorithm to its finish can look. Gray rectangles are the inputs to the network, blue circles are outputs and white circles are hidden nodes. Green and red arrows are positive and negative connection weights respectively.

Chapter 5

Experimental Setting

This chapter will explain the environments used in testing the algorithms proposed in this thesis, as well as the original NEAT algorithm. The environments explained below are specifically created for comparing various reinforcement learning algorithms, and as such are well suited in this case. The environments have been selected in increasing difficulty, from mostly trivial to relatively hard. The goal of using this methodology of testing is to reveal the difference in solving capability for the three algorithms tested, as well as highlight the strengths and weaknesses in the three approaches. Another important aspect is how well the algorithms are able to scale, that is: How will the complexity of the problem influence the run-time of the algorithm.

5.1 Test Environment

Open AI [17] has developed a toolkit for testing and evaluating learning algorithms. The toolkit provides a set of virtual environments, called gyms [18], which can be observed and influenced by an agent. Each gym operates in time steps where the agent performs an action in, or on, the environment and receives an updated observation of the world. It is important to note that each action taken increments the time step by one. Figure 5.1 illustrates this relationship on one of the problems. Different environments have different complexity regarding the number of possible actions and the number of observations that are returned. The possible actions and observations are listed below.

Also of note is that all figures of the environments in this chapter, excluding Figure 5.1, are

copied from the Open AI website.

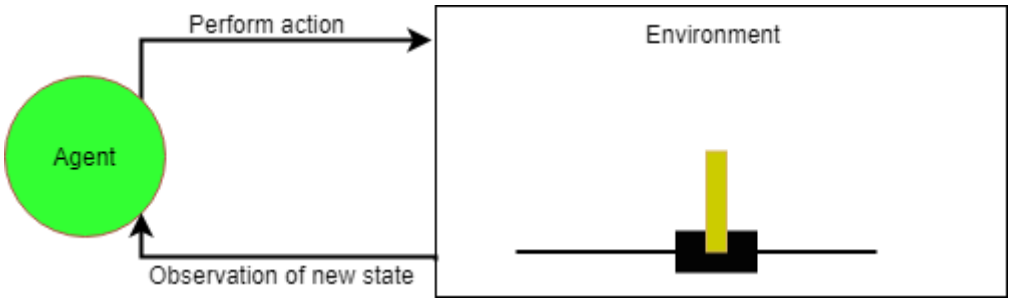


Figure 5.1: Showing the relationship between an agent and an environment.

5.1.1 Discrete & Continuous Actions and Observations

Actions and observations in the test environments can either be continuous or discrete, that is a range from minimum to maximum, or either minimum or maximum. For actions where the action are discrete the number of the selected action is passed to the environment e.g. in Table 5.2, the action selected are either 0 or 1. In Table 5.6 there is only one action, however, the action is continuous, which means that the value which can be passed to the environment is a value between -2.0 and 2.0 depending on the force which should be applied.

5.1.2 Environments

Cart Pole

The goal in this environment is to keep a pole in an upright position by moving the cart side to side. An agent will fail when the pole reaches a certain angle, the cart moves offscreen or when the number of time steps reaches 200. Rewards are given for each time completed time step. Figure 5.2 represents how this environment looks while Table 5.1 and 5.2 shows the possible observations and actions respectively. Specific evaluation code can be found in Appendix A.1.

Number	Observation	Minimum	Maximum
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-41.8°	41.8°
3	Pole Velocity At Tip	-Inf	Inf

Table 5.1: Cart Pole Observations

Number	Action
0	Push cart to the left
1	Push cart to the right

Table 5.2: Cart Pole Actions

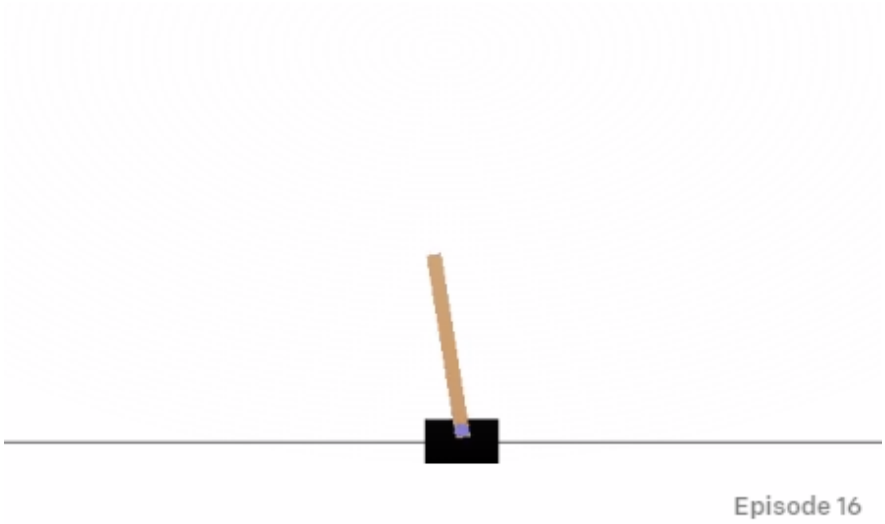


Figure 5.2: Cart pole environment from OpenAI gym

Mountain Car

A cart is placed in a valley with the goal of reaching the flag to the right, as can be seen in Figure 5.3. The motor in the cart is not strong enough to simply drive up the slope. The agent must, therefore, rock the cart from side to side until it has enough momentum to reach the top, using the actions in Table 5.2. The maximum number of timesteps is 200 and the episode will end

either at 200 timesteps or when the flag is reached, with a reward being subtracted for each timestep such that fast solution will get a better score. Observations from the environment can be seen in Table 5.3. Specific evaluation code can be found in Appendix A.2.

Number	Observation	Minimum	Maximum
0	Car position	-1.2	0.6
1	Car velocity	-0.07	0.07

Table 5.3: Mountain Car Observations

Number	Action
0	Push car to the left
1	Do nothing
2	Push car to the right

Table 5.4: Mountain Car Actions

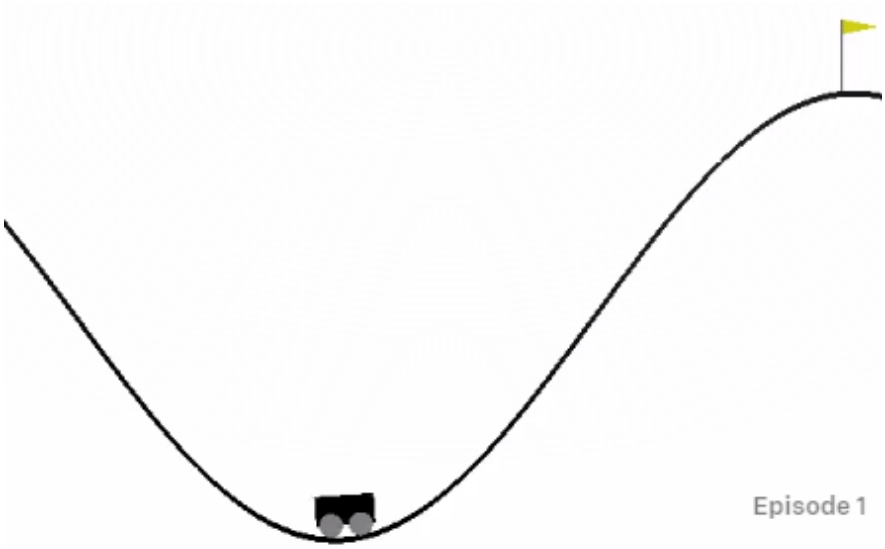


Figure 5.3: Mountain car environment from OpenAI gym

Lunar Lander

Two flags are placed on uneven ground and it is the goal of the agent to steer a lander to land between the flags. A representation can be seen in Figure 5.4. The agent can steer the lander

by firing its engines as can be seen in Table 5.6. The episode ends when the lander crashes or lands, with the rewards being -100 and 100 respectively. Additionally, the agent receives 10 reward when a leg is contacting the ground and -0.3 reward when firing the main engine. Observations from the environment can be seen in Table 5.5. Specific evaluation code can be found in Appendix A.3.

Number	**Observation**	**Minimum**	**Maximum**
0	Lander x-position	-1.0	1.0 (normalized width of screen with the middle being 0.0)
1	Lander y-position	Minimum and maximum dependent on the height of helipad	
2	Lander x-velocity	-Inf	Inf
3	Lander y-velocity	-Inf	Inf
4	Lander angle	-pi	pi
5	Lander angular velocity	-inf	inf
6	Lander left leg contacts ground	0.0	1.0
6	Lander right leg contacts ground	0.0	1.0

Table 5.5: Lunar Lander Observations

Number	**Action**
0	Do nothing
1	Fire left engine
2	Fire middle engine
3	Fire right engine

Table 5.6: Lunar Lander Actions

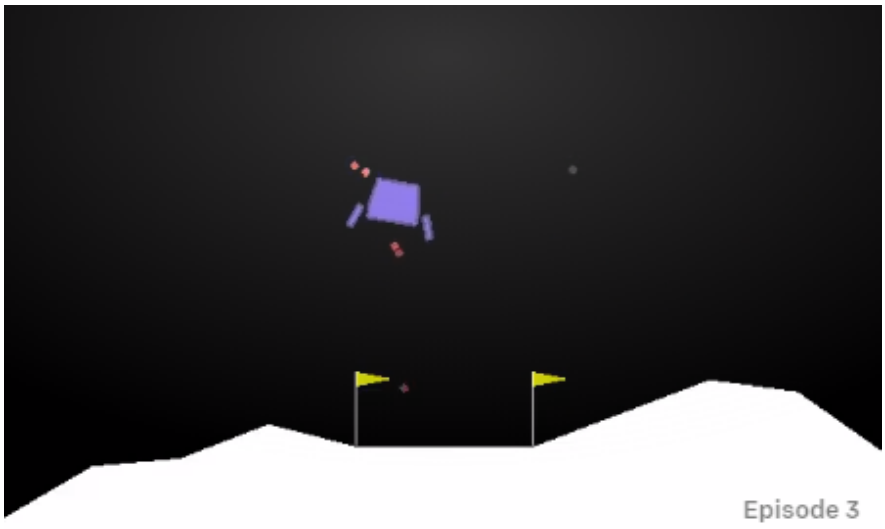


Figure 5.4: Lunar lander environment from OpenAI gym

Pendulum

The goal in this problem is to swing a pendulum such that it stays upright as long as possible. Figure 5.5 shows how problem is illustrated. The possible action, as can be seen in Table 5.8 are continuous, meaning that the size and whether it is positive or negative corresponds to the direction and force of a swing. Rewards are given by equation 5.1. Episodes terminate after a user-defined number of timesteps. Observations from the environment can be seen in Table 5.7. Specific evaluation code can be found in Appendix A.4.

$$-(theta^2 + 0.1 * theta\_dt^2 + 0.001 * action^2) \tag{5.1}$$

Equation 5.2: Theta is the angle of the pendulum and theta_dt is the rotational velocity.

Number	Observation	Minimum	Maximum
0	cos(pole angle)	-1.0	1.0
1	sin(pole angle)	-1.0	1.0
2	Angular velocity of pole	-8.0	8.0
3	Lander y-velocity	-Inf	Inf
4	Lander angle	-pi	pi
5	Lander angular velocity	-inf	inf
6	Lander left leg contacts ground	0	1
7	Lander right leg contacts ground	0	1

Table 5.7: Pendulum Observations

Number	Action	Minimum	Maximum
0	Add force	-2.0	2.0

Table 5.8: Pendulum Actions

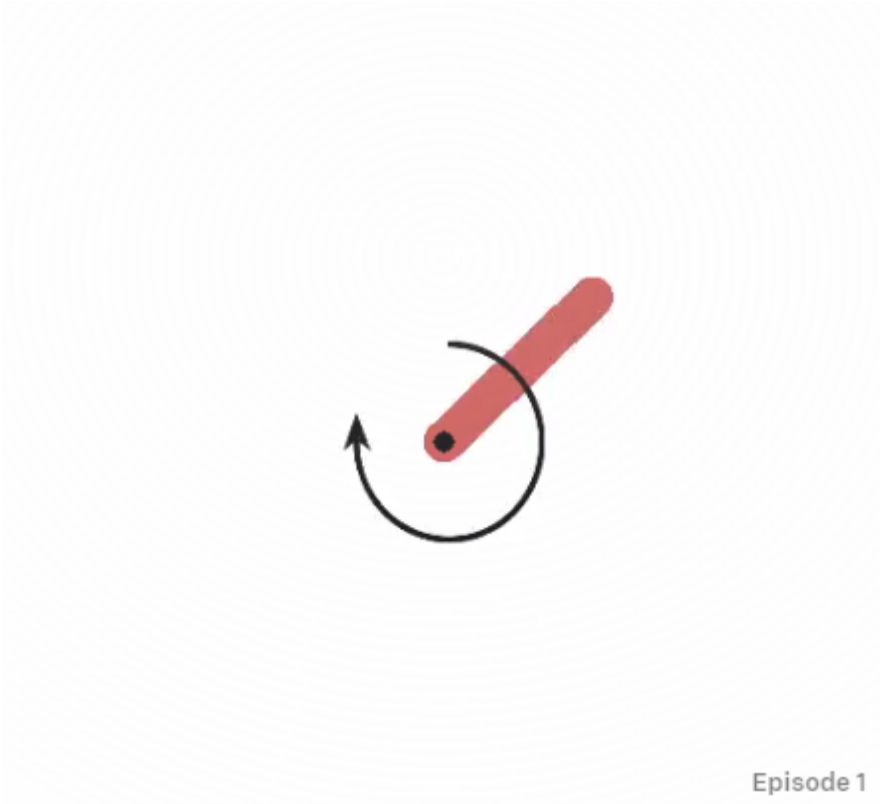


Figure 5.5: Pendulum environment from OpenAI gym

Bipedal Walker

The agent must learn to move the legs of a walker in such a way as to make it move from left to right, as can be seen in Figure 5.6. Multiple readings of the state can be observed such as the speed of the different parts of the walker. Table 5.9 shows all the possible readings. Rewards are given for moving from left to right, with a penalty of -100 being given when the walker falls. Actions are shown in 5.10. Specific evaluation code can be found in Appendix A.5.

Number	Observation	Minimum	Maximum
0	Hull angle	0.0	2*pi
1	Angular velocity of hull	-Inf	Inf
2	Hull x-velocity	-1.0	1.0
3	Hull y-velocity	-1.0	1.0
4	Hip joint 1 angle	-Inf	Inf
5	Hip joint 1 speed	-Inf	Inf
6	Knee joint 1 angle	-Inf	Inf
7	Knee joint 1 speed	-Inf	Inf
8	Leg 1 contacts ground	0.0	1.0
9	Hip joint 2 angle	-Inf	Inf
10	Hip joint 2 speed	-Inf	Inf
11	Knee joint 2 angle	-Inf	Inf
12	Knee joint 2 speed	-Inf	Inf
13	Leg 2 contacts ground	0.0	1.0
14-23	10 lidar (distance) readings	-Inf	Inf

Table 5.9: Bipedal Walker Observations

Number	Action	Minimum	Maximum
0	Add force to hip 1	-1.0	1.0
1	Add force to knee 1	-1.0	1.0
2	Add force to hip 2	-1.0	1.0
3	Add force to knee 2	-1.0	1.0

Table 5.10: Bipedal Walker Actions

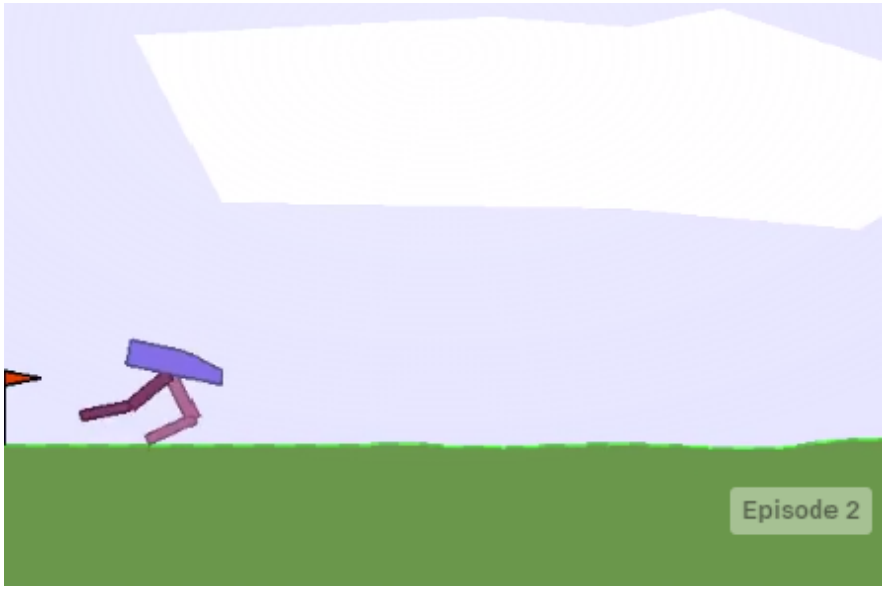


Figure 5.6: Bipedal Walker environment from OpenAI gym

Chapter 6

Results and Discussion

6.1 Experimental Setup

The algorithms were run on each of the environments for a comparable amount of time, and with the same evaluation methods, that is with the same environment observations and associated rewards. These algorithms are *nondeterministic*, meaning that for the same input, the algorithms may generate distinct solutions for each time they are run. The results can, therefore, vary from run to run, meaning that while an algorithm may encounter a good solution fast in one run, it may take considerably longer in the next. The results in this chapter are based on the average for each algorithm over multiple runs to give a more representative picture on how they compare, instead of using single cases where one algorithm may be lucky or unlucky. See appendix A.2 for an example of 20 consecutive runs with the same algorithm on the same problem.

6.2 Quantitative Results and Discussion

In this section, I will go through the quantitative results achieved for each algorithm in detail. Later, I will aim to explain the differences in results as well as describe the behavior of the algorithms in context through the use of fitness graphs for single runs and then average fitness graphs for multiple runs. Firstly it is important to note that, as mentioned in 4.2.4, *mcWeightEvolPartialNEAT* is created after the testing of *mcHillNeat* and *mcWeightEvolNEAT*,

and based on the observations from *mcWeightEvolNEAT*.

Cart Pole			
Algorithm	**Run-time**	**Achieved Fitness**	**Solution Complexity**
NEAT	< 1	200/200	1/4
mcHillNEAT	< 1	200/200	3/13
mcWeightEvolNEAT	< 1	200/200	2/7
mcWeightEvolPartialNEAT	< 1	200/200	1/4

Table 6.1: Cart Pole Results: All results are average over 20 runs. The run-time is given in seconds. The fitness is given as (achieved fitness / maximum fitness). The complexity is given as (number of nodes / number of connections). The number of nodes does not include input and output nodes.

As a proof of concept, the cart-pole problem was tested initially. Being a relatively small problem with only four inputs and two possible binary outputs combined with the fact that the pole initializes as being upright means that the problem is quite forgiving even if the steering agent were to input some sub-optimal inputs. As can be seen from Table 6.1, all four algorithms achieved 200 out of 200 fitness, that is, they managed to keep the pole upright and the cart on-screen for 200 time-steps, in on average less than a second. As can be seen, however, is that the complexity of the solution-networks for *mcHillNeat* and *mcWeightEvolNEAT* are considerably larger on average than for NEAT and *mcWeightEvolPartialNEAT*. This implies that they have to proceed further down the search tree to find solutions, indicating that the local-search scheme does not adequately find good weights on early architectures before moving on. Since this problem is so small, however, it is hard to draw any definitive conclusions other than that *mcWeightEvolPartialNEAT*, seemingly, is closer to NEAT in the ability to utilize small architectures.

Mountain Car		
Algorithm	**Run-time**	**Solution Complexity**
NEAT	50	1/4
mcHillNEAT	60	3/13
mcWeightEvolNEAT	70	2/7
mcWeightEvolPartialNEAT	50	6/13

Table 6.2: Mountain Car Results: All results are average over 20 runs. The run-time is given in seconds. Fitness is not included here since finding a concrete solution is more important. All three algorithms found a solution. The complexity is given as (number of nodes / number of connections). The number of nodes does not include input and output nodes.

The mountain car problem, seen in Table 6.2, can initially seem simpler than the cart-pole problem since it only has two inputs and three possible actions. What makes the problem more complex is the combination of greater forces impacting the mountain car than the cart as well as the car not starting in the winning position as the pole does. This heightened complexity naturally produces longer run-times before solutions are found. Table 6.2 shows that, while not a large difference overall, *mcHillNeat* and *mcWeightEvolNEAT* takes longer to produce solutions, with *mcWeightEvolNEAT* being the slowest. Interestingly, *mcWeightEvolPartialNEAT* is on average as fast as NEAT, however, the network-solutions are often much larger than the solution-network created by NEAT. NEAT prioritizes starting with small architectures which are slowly enlarged as the algorithm progresses, while the algorithms created in this thesis are able to quickly create larger networks. Generally, in regards to both resource usage and training, small solution-networks are preferable over larger networks if the produced fitness and run-time is comparable.

Lunar Lander			
Algorithm	**Run-time**	**Achieved Fitness**	**Solution Complexity**
NEAT	1835	209	4/17
mcHillNEAT	2000	-150	6/35
mcWeightEvolNEAT	2000	-100	6/35
mcWeightEvolPartialNEAT	2000	-50	5/34

Table 6.3: Lunar Lander Results: All results are average over 20 runs. The run-time is given in seconds. The fitness is given as (achieved fitness / maximum fitness). The complexity is given as (number of nodes / number of connections). The number of nodes does not include input and output nodes.

The fact that NEAT is better able to exploit smaller networks becomes apparent when observing Table 6.3 where the three proposed algorithms fall well behind NEAT in both fitness and solution complexity. None of the three algorithms are able to find adequate solutions, struggling more than NEAT in exploiting small networks, similar as in the mountain car problem, but also struggling to search deep in the network because of the number of possible children of each node. This is in spite of a rather strict pruning scheme, explained in Section 4.2.1.

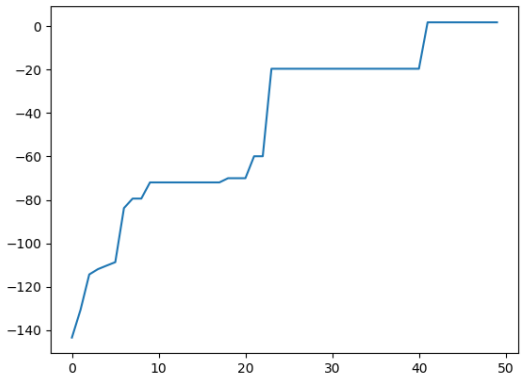
Pendulum			
Algorithm	**Run-time**	**Achieved Fitness**	**Solution Complexity**
NEAT	500	-270	9/5
mcHillNEAT	500	-375	14/19
mcWeightEvolNEAT	500	-377	13/21
mcWeightEvolPartialNEAT	500	-250	6/35

Table 6.4: Pendulum Results: All results are average over 20 runs. The run-time is given in seconds. The fitness is given as (achieved fitness / maximum fitness). The complexity is given as (number of nodes / number of connections). The number of nodes does not include input and output nodes.

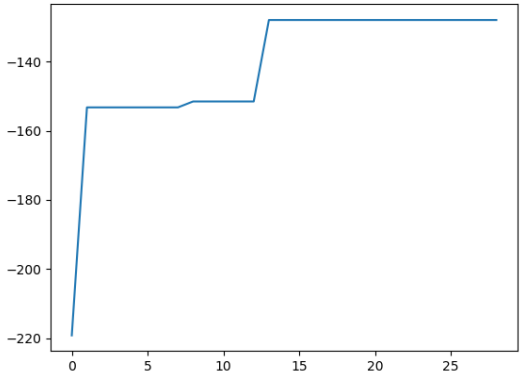
Bipedal Walker			
Algorithm	**Run-time**	**Achieved Fitness**	**Solution Complexity**
NEAT	2000	12	5/75
mcHillNEAT	2000	9	5/92
mcWeightEvolNEAT	2000	9	6/93
mcWeightEvolPartialNEAT	2000	11	6/121

Table 6.5: Bipedal Walker Results: All results are average over 20 runs. The run-time is given in seconds. The fitness is given as (achieved fitness / maximum fitness). The complexity is given as (number of nodes / number of connections). The number of nodes does not include input and output nodes.

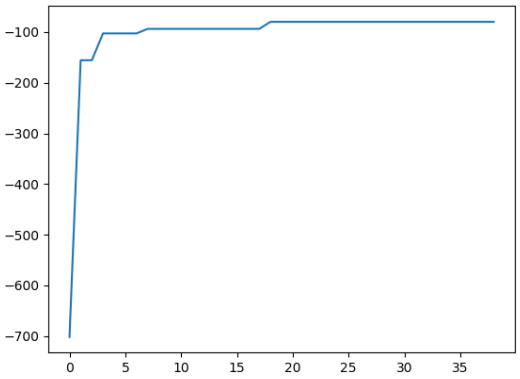
This changes somewhat, however, when looking at table 6.4 and 6.5. In the pendulum problem NEAT has converged on a rather small network, while *mcWeightEvolPartialNEAT*s network is substantially larger and performs slightly better. While this do not indicate that *mcWeightEvolPartialNEAT* is overall better than NEAT, it does provide some nuance and indicates that there are cases where a faster complexification of the solution networks may be beneficial. Likewise, in the bipedal walker problem it can be observed that while *mcHillNeat* and *mcWeightEvolNEAT* lags behind, *mcWeightEvolPartialNEAT* and NEAT have very comparable average fitness values and, in some very few test runs, *mcWeightEvolPartialNEAT* even performed better.



(a) Neat fitness graph

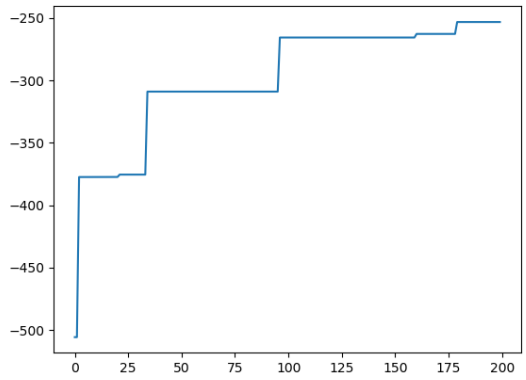


(b) mcHillNEAT fitness graph

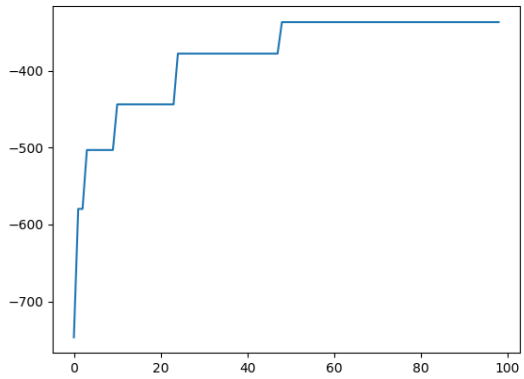


(c) mcWeightEvolNEAT fitness graph

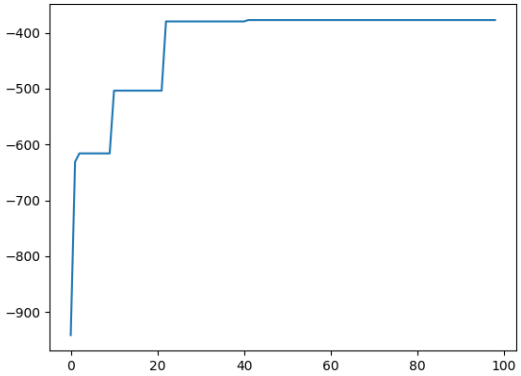
Figure 6.1: Fitness graph comparisons between NEAT, mcHillNEAT and mcWeightEvolNEAT for the lunar lander environment. x-axis is number of generations and y-axis is the fitness. These graphs are for a single run.



(a) Neat fitness graph

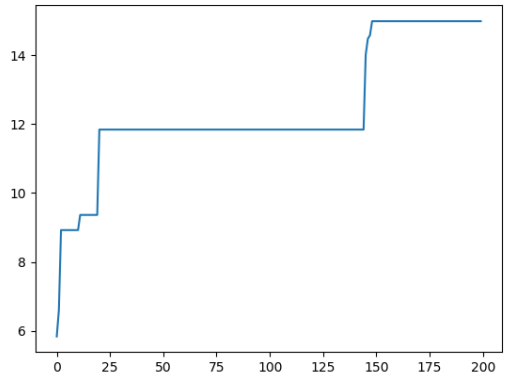


(b) mcHillNEAT fitness graph

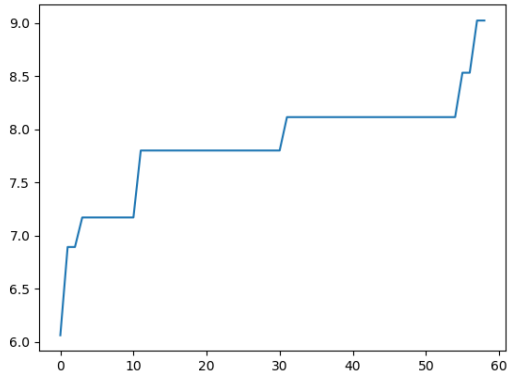


(c) mcWeightEvolNEAT fitness graph

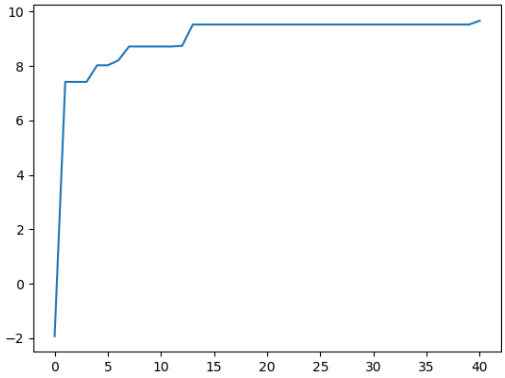
Figure 6.2: Fitness graph comparisons between NEAT, mcHillNEAT and mcWeightEvolNEAT for the pendulum environment. x-axis is number of generations and y-axis is the fitness. These graphs are for a single run.



(a) Neat fitness graph



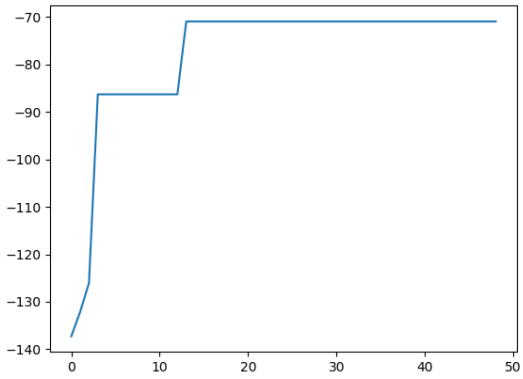
(b) mcHillNEAT fitness graph



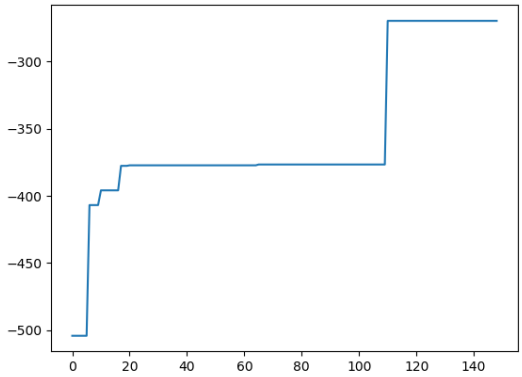
(c) mcWeightEvolNEAT fitness graph

Figure 6.3: Fitness graph comparisons between NEAT, mcHillNEAT and mcWeightEvolNEAT for the bipedal walker environment. x-axis is number of generations and y-axis is the fitness. These graphs are for a single run.

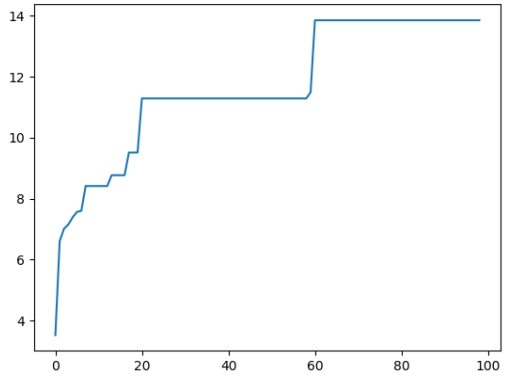
Figures 6.1, 6.2 and 6.3 shows the difference in search strategies for NEAT, *mcHillNeat* and *mcWeightEvolNEAT*. The images, showing fitness progression for the three algorithms, illustrates how NEAT searches a larger search space before committing to a path while the two other algorithms will very narrowly proceed towards the nearest local optimum. This is illustrated by *mcHillNeat* and *mcWeightEvolNEAT* by an initial sudden jump in fitness, and then stagnation at a low fitness. In contrast NEATs fitness rises more slowly and stagnates at a higher fitness.



(a) Lunar lander mcWeightEvolPartialNEAT



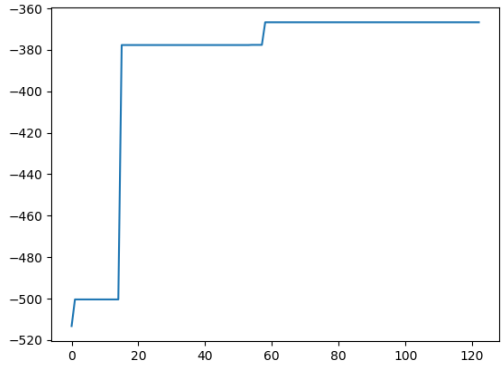
(b) Pendulum mcWeightEvolPartialNEAT



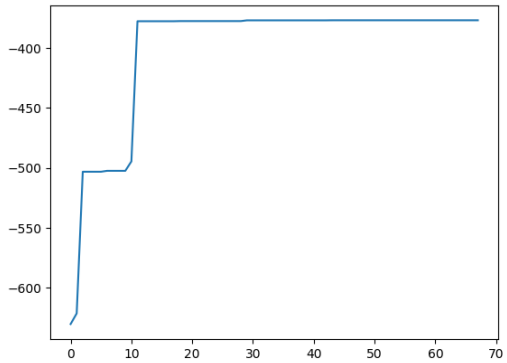
(c) Bipedal walker mcWeightEvolPartialNEAT

Figure 6.4: Fitness graphs for mcWeightEvolPartialNEAT showing for the problems lunar lander, pendulum and bipedal walker. x-axis is number of generations and y-axis is the fitness. These graphs are for a single run.

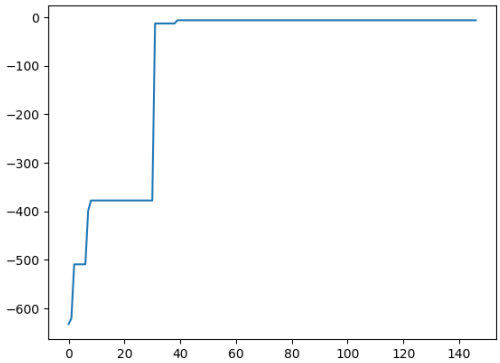
As mentioned in 4.2.4, it was observed that a vital flaw with the proposed algorithms was the rapid convergence on local optima and the adaption *mcWeightEvolPartialNEAT* were created as an attempt to correct it. In Figure 6.4 the effect this change can be seen. While still not ideal, the fitness graphs are now a step closer to those that are produced by NEAT, showing that the change was a step in the right direction.



(a) mcHillNEAT average fitness graph



(b) mcWeightEvolNEAT average fitness graph



(c) mcWeightEvolPartialNEAT average fitness graph

Figure 6.5: Average Fitness graphs for *mcHillNEAT*, *mcWeightEvolNEAT* and *mcWeightEvolPartialNEAT* for the pendulum problem over 20 runs. x-axis is number of generations and y-axis is the fitness. Note: For each generation only the best found fitness is graphed, and as such the average fitness can never decrease.

Though illustrations of single runs are helpful when discussing algorithms, it is often more advantageous to look at the overall behaviour of the algorithm by considering the average results over multiple runs. Figure 6.5 show the average fitness for 20 consecutive runs. From the three figures it can be observed that all three show signs of having sudden and great leaps in fitness. This i caused by the algorithms getting lucky in finding a path in one run which adds to the average. It is important to note that although 6.5(c) in this case shows that *mcWeightEvolPartialNEAT* achieves a good score, since this is largely because of a lucky leap in fitness, it does not mean that the algorithms always will find as good solutions. Often, the results will vary greatly.

The quantitative results and associated discussion in this chapter demonstrate that the proposed solutions do not currently outperform the original NEAT algorithm. Consistently they achieve worse results and/or provide more complex solution-networks. This seems to be a problem with the size of the search space the algorithms are able to traverse in the same amount of time as NEAT. One reason for this may be that a genetic algorithm is able to make small 'jumps' in the search space and thus avoids having to explore every possibility in the vicinity of already found solutions. In contrast, the proposed tree search expands a node by creating all possible children of that node, causing the tree to grow exponentially. This likely causes the algorithms to create many different topologies, making the resources required to train the found architectures larger than that needed in NEAT.

Chapter 7

Conclusion

This thesis has explored the possibility of using MCTS in combination with NEAT to search for both neural network topologies as well as weights. For this reason, three algorithms were created, tested, and compared to the original algorithm. Experiments were run in the same environments so that valid comparisons could be made. Results showed that while all three algorithms were able to learn simple tasks, none were equal, or better, than NEAT. Notably, on the Bipedal walker-problem, *mcWeightEvolPartialNEAT* achieved results which are very nearly as good as NEAT, and on some few runs were even able to get a better score. Some of the results were promising, however, and further research in the field could still reveal variances of the proposed algorithms which can compete with NEAT.

In Chapter 1 two research questions were introduced, and in consecutive chapters I have aimed to answer them.

RQ1 is the focal point of Chapter 4, where an initial study was done to explore the possibilities in applying MCTS to problems traditionally solved with genetic algorithms. In this study, it became apparent that a strict pruning method would need to be applied for the algorithm to have a chance of progress down the search tree.

RQ2 is explored in chapters 4 and 6 where firstly the algorithms are proposed and explained in detail and later evaluated. In Chapter 6 the concrete results are discussed and contextualized, and the behavior of each algorithm is analyzed.

7.1 Further Work

While the experiments in this thesis provide a basis for the evaluations of the proposed algorithms, several improvements and adaptions should be tested in later experiments. A strength of MCTS is parallelization; the nature of tree searches makes them ideal for parallelizing the search of different subtrees [2]. This would enable the algorithms to simultaneously explore multiple separate parts of the search space.

A clear advantage, and important aspect, of NEAT is the speciation aspect. In the proposed algorithms, speciation has not been implemented in the local search, possibly giving up an important advantage.

Another interesting experiment would be to test a non-random simulation scheme. That is, when running simulations it may be possible to generate some heuristics to guide the simulation. In the same vein, the leaf node of a simulation currently contains random weights. It may be preferable to train the last network slightly to more accurately identify good and bad branches.

Bibliography

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[2] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel monte-carlo tree search. In H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, editors, *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] T. Chen, I. J. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer. *CoRR*, abs/1511.05641, 2015.

[4] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[5] T. Elsken, J.-H. Metzen, and F. Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[6] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *CoRR*, abs/1808.05377, 2018.

[7] T. Ewals. *Playing and solving Havannah*. PhD thesis, University of Alberta, 2012.

[8] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017.

[9] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: From architectures to learning. *Evol Intell*, 1, 03 2008.

[10] T. G. van den Berg and S. Whiteson. Critical factors in the performance of hyperneat. pages 759–766, 07 2013.

[11] T. Kozelek. Methods of mcts and the game arimaa. 2009.

[12] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436, 2017.

[13] R. J. Lorentz. Amazons discover monte-carlo. In H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, editors, *Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] A. McIntyre, M. Kallada, C. G. Miguel, and C. F. da Silva. neat-python. https://github.com/CodeReclaimers/neat-python.

[15] R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[16] D. E. Moriarty and R. Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5:373–399, 1997.

[17] OpenAI. Open ai website, 2019.

[18] OpenAIGym. Open ai gym website, 2019.

[19] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge. The design science research process: A model for producing and presenting information systems research. *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST*, 02 2006.

[20] M. Rashid, M. A. H. Newton, M. Hoque, and A. Sattar. Mixing energy models in genetic algorithms for on-lattice protein structure prediction. *BioMed research international*, 2013:924137, 09 2013.

[21] S. Risi, J. Lehman, and K. Stanley. Evolving the placement and density of neurons in the hyperneat substrate. pages 563–570, 01 2010.

[22] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[23] J. Schaffer, D. Whitley, and L. J. Eshelman. Cogann-92 combinations of genetic algorithms and neural networks. pages 1 – 37, 07 1992.

[24] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[25] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life*, 15(2):185–212, Apr. 2009.

[26] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, June 2002.

[27] D. Svozil, V. Kvasnicka, and J. Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43 – 62, 1997.

[28] L. Wang, Y. Zhao, and Y. Jinnai. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *CoRR*, abs/1805.07440, 2018.

[29] M. Wistuba. Finding competitive network architectures within a day using UCT. *CoRR*, abs/1712.07420, 2017.

[30] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 38:1–38:10, New York, NY, USA, 2014. ACM.

[31] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[32] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

Appendix A

Additional Information

A.1 Evaluation Code

A.1.1 Cart Pole

```
def eval_genomes(genomes, config):
    observation = env.reset()
    for genome_id, genome in genomes:
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        genome.fitness = 0

        for _ in range(500):
            action = net.activate(observation)[0]
            action = 1 if action > 0.5 else 0
            observation, reward, done, info = env.step(action)
            genome.fitness += reward

            if done:
                observation = env.reset()
                break
```

Listing A.1: Cart pole evaluation

A.1.2 Mountain Car

```
def eval_genomes(genomes, config):
    observation = env.reset()
    for genome_id, genome in genomes:
      observation = env.reset()
      net = neat.nn.FeedForwardNetwork.create(genome, config)
      genome.fitness = 0
      max_height_reached = -inf
      max_vel_reached = -inf

      for _ in range(200):
        actions = net.activate(observation)
        action = actions.index(max(actions))
        observation, reward, done, info = env.step(action)
        if done:
          observation = env.reset()
          break
        genome.fitness += reward
        max_height_reached = observation[0] if observation[0] > max_height_reached else
    max_height_reached
        max_vel_reached = observation[1] if observation[1] > max_vel_reached else
    max_vel_reached
      genome.fitness +=  (1 + abs(max_height_reached)) ** 2
      genome.fitness +=  (1 + abs(max_vel_reached)) ** 2
```

Listing A.2: Mountain car evaluation

A.1.3 Lunar Lander

```
def eval_genomes(genomes, config):
    observation = env.reset()
    for genome_id, genome in genomes:
      worst_found_fitness = inf
      net = neat.nn.FeedForwardNetwork.create(genome, config)
      for __ in range(10):
        observation = env.reset()
        curr_fitness = 0

        for _ in range(1000):
            action = net.activate(observation)
            high = action.index(max(action))
            observation, reward, done, info = env.step(high)
            curr_fitness += reward

            if done:
              observation = env.reset()
              break
        worst_found_fitness = curr_fitness if curr_fitness < worst_found_fitness else
    worst_found_fitness
      genome.fitness = worst_found_fitness
```

Listing A.3: Lunar lander evaluation

A.1.4 Pendulum

```
def eval_genomes(genomes, config):
    observation = env.reset()
    for genome_id, genome in genomes:
      observation = env.reset()
      net = neat.nn.FeedForwardNetwork.create(genome, config)
      genome.fitness = 0
      for __ in range(2):
        for _ in range(100):
            action = net.activate(observation)[0]
            action *= 2
            observation, reward, done, info = env.step([action])
            genome.fitness += reward
```

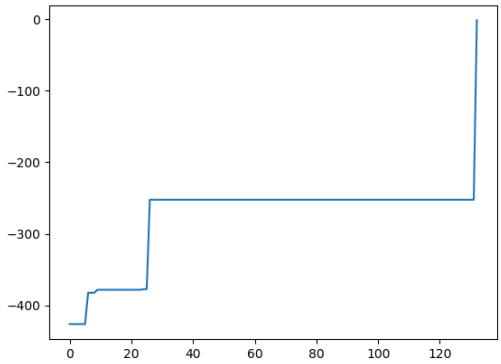
Listing A.4: Pendulum evaluation

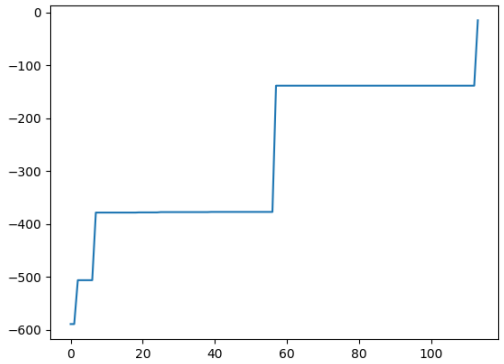
A.1.5 Bipedal Walker

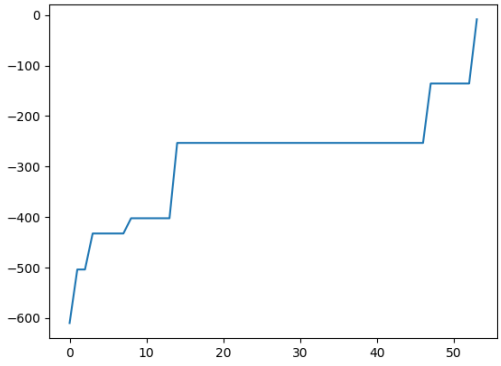
```
def eval_genomes(genomes, config):
    observation = env.reset()
    for genome_id, genome in genomes:
      observation = env.reset()
      net = neat.nn.FeedForwardNetwork.create(genome, config)
      genome.fitness = 0

      for _ in range(100):
          action = net.activate(observation)
          observation, reward, done, info = env.step(action)
          genome.fitness += reward

          if done:
            observation = env.reset()
            break
```

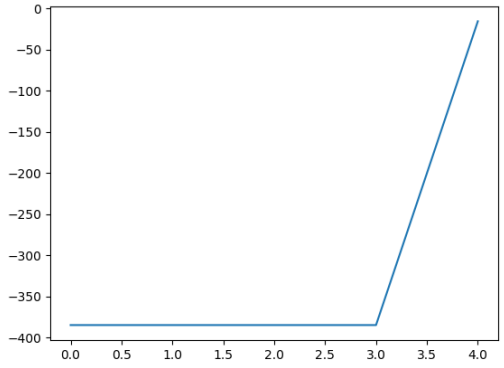
Listing A.5: Bipedal walker evaluation

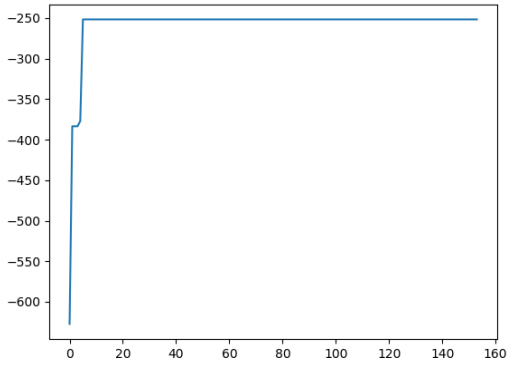
A.2 Example Runs

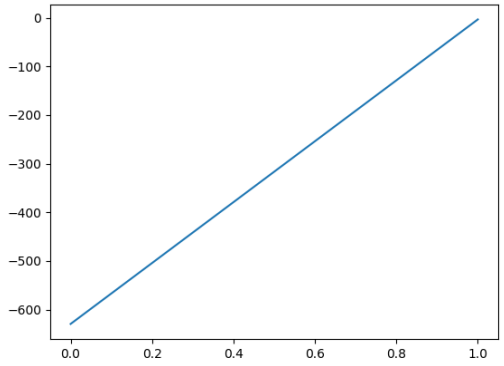


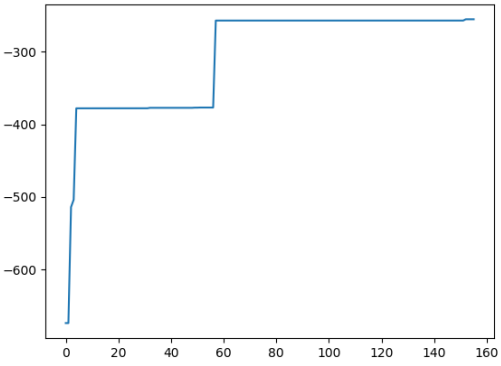


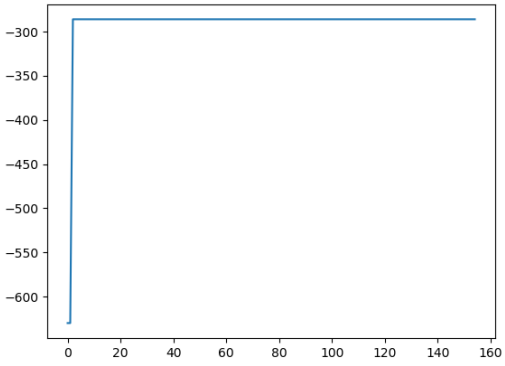


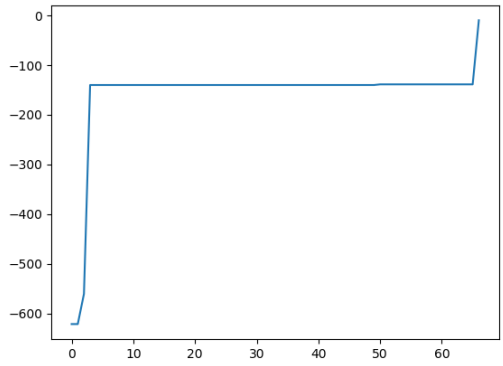


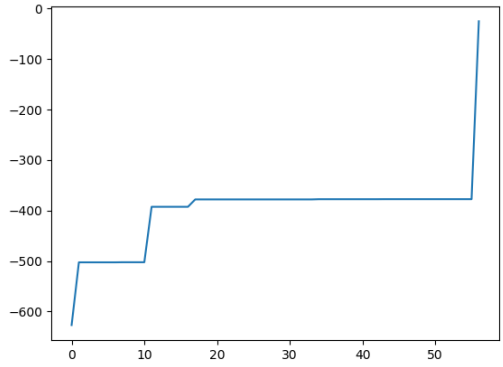


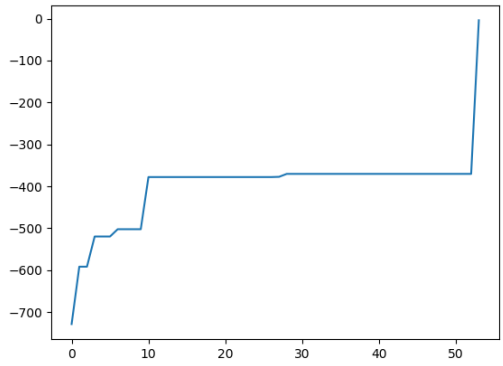


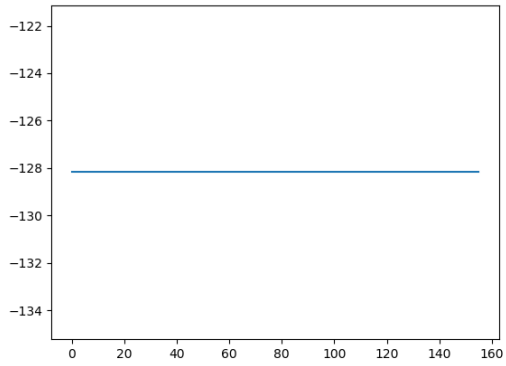


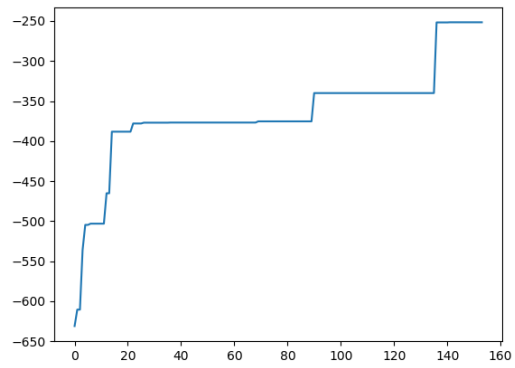


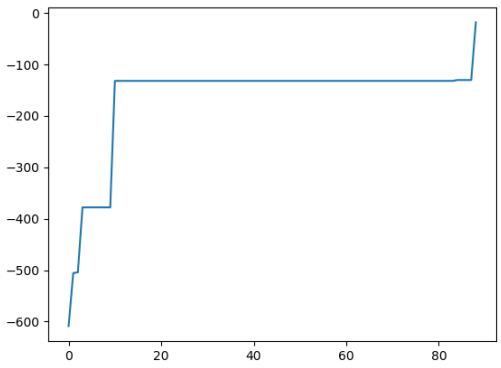

0
−100
−200
−300
−400
−500
−600
0
10
20
30
40
50
60

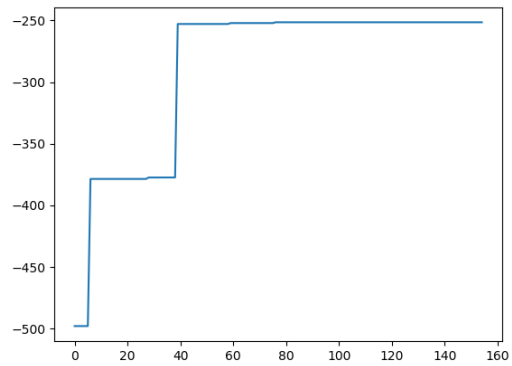

0
−100
−200
−300
−400
−500
−600
0
10
20
30
40
50

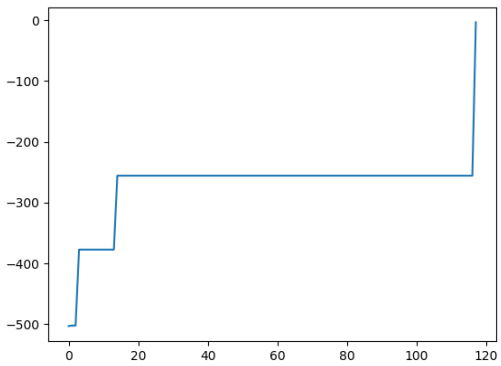

0
−100
−200
−300
−400
−500
−600
−700
0
10
20
30
40
50


−122
−124
−126
−128
−130
−132
−134
0
20
40
60
80
100
120
140
160


−250
−300
−350
−400
−450
−500
−550
−600
−650
0
20
40
60
80
100
120
140
160


0
−100
−200
−300
−400
−500
−600
0
20
40
60
80


−250
−300
−350
−400
−450
−500
0
20
40
60
80
100
120
140
160


0
−100
−200
−300
−400
−500
0
20
40
60
80
100
120

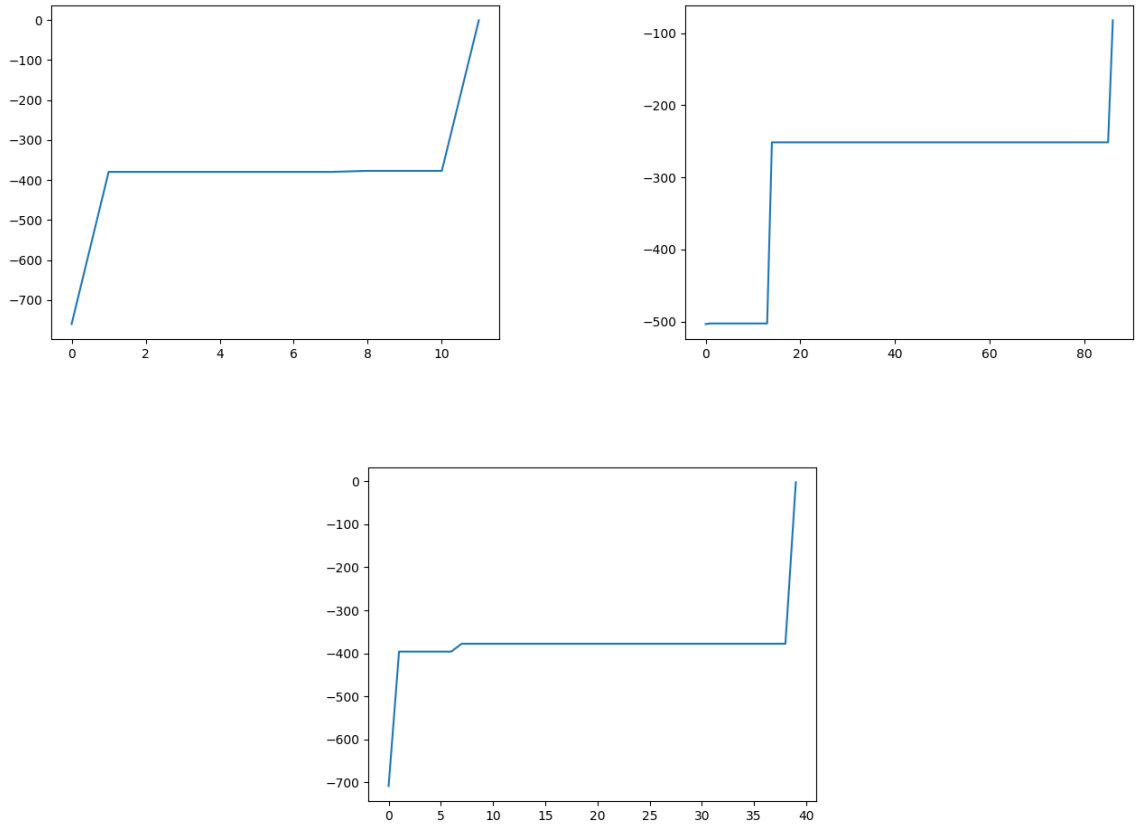
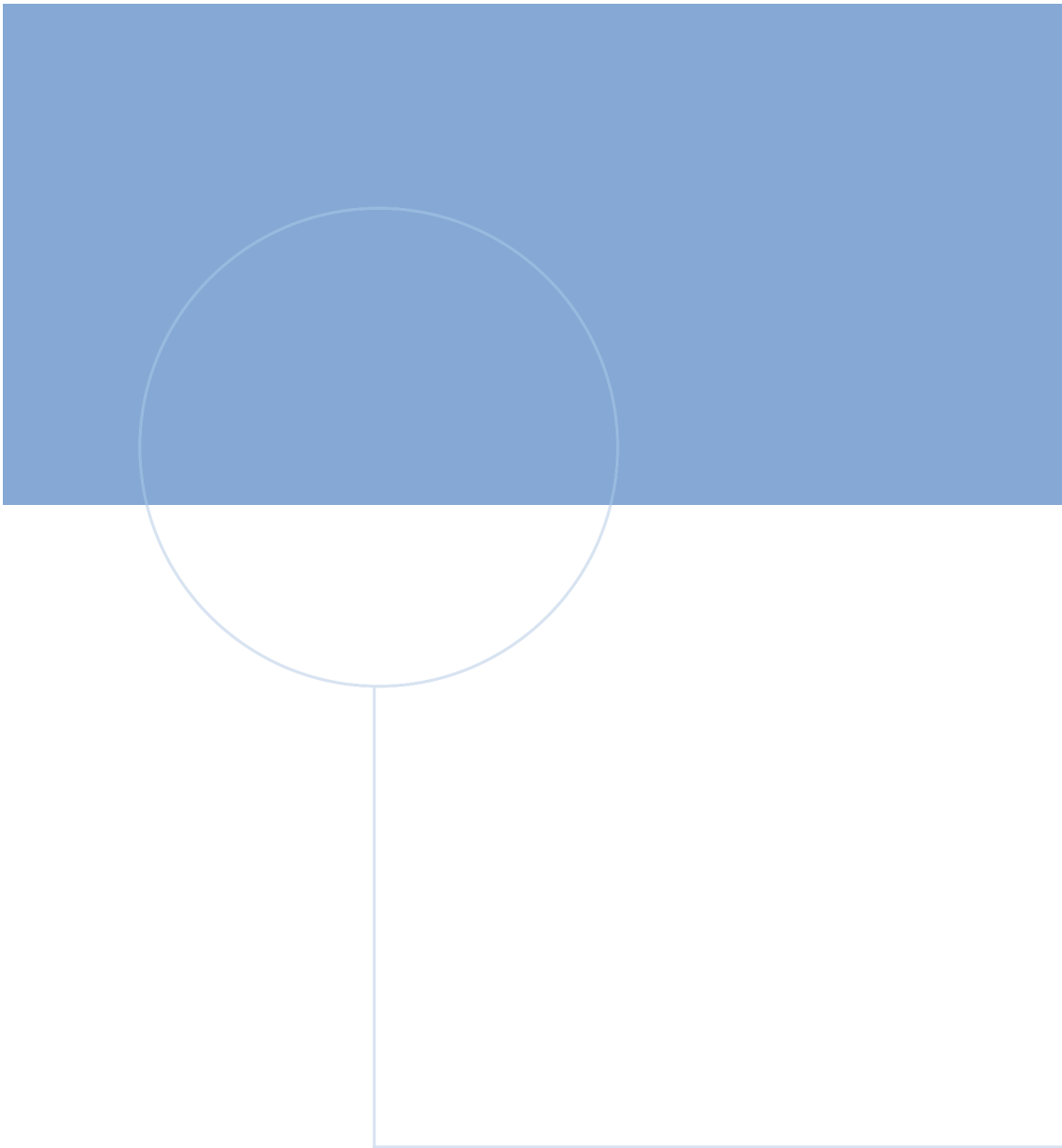


Figure A.1: Example of mcWeightEvolPartialNEAT being run on the inverted pendulum problem 20 times



NTNU
Norwegian University of
Science and Technology