Childhood Habituation in Evolution of Augmenting Topologies (CHEAT)

Anton Moberg
Department of Astronomy and Theoretical Physics, Lund University

Date: September 26, 2020

LUND UNIVERSITY
Faculty of Science

A master thesis supervised by Patrik Edén, Ph.D FYTM04 (60 ECTS)

"Imagine a puddle waking up one morning and thinking, "This is an interesting world I find myself in — an interesting hole I find myself in — fits me rather neatly, doesn't it? In fact it fits me staggeringly well, must have been made to have me in it!" This is such a powerful idea that as the sun rises in the sky and the air heats up and as, gradually, the puddle gets smaller and smaller, it's still frantically hanging on to the notion that everything's going to be alright, because this world was meant to have him in it, was built to have him in it; so the moment he disappears catches him rather by surprise. I think this may be something we need to be on the watch out for."

- Douglas Adams

Abstract

Neuroevolution is a field within machine learning that applies genetic algorithms to train artificial neural networks. Neuroevolution of Augmenting Topologies (NEAT) is a method that evolves both the topology of the network and trains the weights of the network at the same time, and has been found to successfully solve reinforcement learning problems efficiently and the XOR problem with a minimal topology. However, NEAT has not been shown to solve more complex labelling problems and has a vaguely motivated heuristic concept of speciation needed to keep a diverse population and protect new structural innovations from instant elimination. In this thesis a new algorithm was developed, the Childhood Habituation in Evolution of Augmenting Topologies (CHEAT) algorithm, which removes the need for the heuristic speciation concept and its associated hyper-parameters by splitting topology evolution and weight training into two distinct phases. CHEAT also allows for structured topology evolution by having the option of forcing fully connected layers. The algorithm was tested on the XOR problem and the spiral problem with two turns, with a result showing performances on par with NEAT for the XOR problem, and better results on the spiral problem where CHEAT is able to solve the problem while NEAT is not. It was found that without an early stopping criterion for gradient descent training, new structural innovations were quickly eliminated from the population before being optimally tuned, and thus the stopping criterion is vital to be able to remove the NEAT speciation heuristics. It was also found that restricting the algorithm to evolve in a structured manner by forcing fully connected layers was vital to solving any problems more complex than the XOR problem, likely due to the feature selection behaviour fully connected layers exhibit. The work done in this thesis opens further areas of research into evolving Artificial Neural Networks, where the most interesting leads lie in other weight training methods, different stopping criterion for gradient descent training, and finally letting the algorithm take control of evolution of its own hyper-parameters for automatic model construction.

Popular Abstract

Artificial Neural Networks are the computer equivalent to the human nervous system. Just like the nervous system is a network of neurons connected to other neurons by axons, the artificial neural network is a network of nodes connected to other nodes by links. The artificial neural network takes an input signal, transmits the signal through the network during which the nodes perform mathematical operations to alter the received signal in some way. Once transmitted all the way through the network a result can be extracted. These networks are used in basically all modern technology and has during the past decade completely changed the way technology is working in society.

The theory behind artificial neural networks is rather simple, but the problem lies in how these networks are created efficiently. It involves a lot of fiddling with the size of the network, the so called topology, since a network that is too big will be computationally inefficient while a network that is too small will not be able to solve the problem it has been tasked with. Furthermore the strength at which the links transmit the signal needs to be individually tuned to a very precise unknown value. This tuning is called training of the network and is one of the foundations to machine learning, and there are many methods used for this process.

One such training method is called Neuroevolution which uses Darwin's theory of evolution to essentially artificially simulate natural selection. It works by having a bunch of networks (a population), all with different link strengths and then mutating these strengths at random. The mutated networks are evaluated on how well the perform a given task, and based on this evaluation they are ranked from best to worst, where the worst ones are eliminated from the population while the best ones are allowed to reproduce and produce offspring to replace the eliminated ones. The reproduction works by combining information on the strength of the links from two parent networks, and thus producing an offspring that, ideally, will perform better. This process is the repeated until a solution is found. This method solves the tuning problem, but there is the second piece to the puzzle of finding a suitable network size. This is where Neuroevolution of Augmenting Topologies, or simply NEAT, comes into the picture.

NEAT evolves both the strength of the links and the actual network structure at the same time. The evolution of the network structure works in the same way as for the link strength, but instead of the link strength at random the actual networks structure is mutated. This happens by either adding a new node on an existing link, or adding a new link between two previously not linked nodes. NEAT is a very powerful algorithm as it removes the need for a human to specify the network structure, but it also has some trouble being able to evolve a network that is able to solve some more complex problem. NEAT also has a heuristic principle called speciation which introduce new parameters that a human needs to tune by hand, which of course is not ideal. Speciation splits the population into different groups in which each network resembles the other networks in that species. This is done to ensure that there is some diversity in the population so that evolution does not stagnate.

Enter the new algorithm Childhood habituation in Evolution of Augmenting Topologies, or CHEAT. CHEAT works on the same base principle as NEAT but it splits the training of the link strengths and the evolution of the structure into two separate phases. The phase for structural innovation can be seen as an analogy to two parents mixing their genes to produce an offspring, however when the offspring is born it cannot care for itself and needs its parents help to learn to survive. The training phase can be seen as the time where the parents protect their offspring and teach it necessary skills to survive on its own. Once trained it will go out and face the world on its own and if it has inherited good genes and been trained well, it will be able to find a mate and keep the cycle going.

The results of studying this algorithm has been bigger networks must be allowed to train for longer than smaller networks, which can be seen as an analogy to how bigger animals usually stay with their parents for longer while smaller animals stay shorter. Second, to solve more complex problems there is a need to structure how the mutations of the network occur. This is done by forcing the network to have a layered structure in which all nodes of that a layer have links to all the nodes of the previous layer and the following layer. With these modifications CHEAT solves the problems that NEAT has, as well as removes the speciation heuristic and the associated difficult to tune parameters. This opens up for exciting further research into the matter, where the ultimate goal would be to let the algorithm take control of all of the remaining parameters currently needing human input and tuning.

Populärvetenskapligt abstract

Artificiella Neuronnät är datorvärldens ekvivalens till det mänskliga nervsystemet. Precis som nervsystemet är ett nätverk av neuroner som är kopplade till andra neuroner med nerver så är det artificiella neuronnätet ett näverk av noder som är kopplade till andra noder med länkar. Det artificiella neuronnätet tar en inputsignal som skickas igenom nätverket och där noderna signalen stöter på på vägen modifierar den matematiskt innan den skickas vidare. När signalen nått hela vägen till slutet av nätverket så kan ett resultat extraheras. Dessa nätverk används i princip i all modern teknologi och har under det senaste decenniet totalt förändrat teknikens plats i samhället.

Teorin bakom artificiella neuronnät är relativt simpel, men problematiken ligger i hur dessa nätverk skapas på ett effektivt vis. Det involverar mycket pysslande med storleken av nätverket i sig, den så kallade topologin, då ett nätverk som är för stort är beräkningsmässigt ineffektivt, medan ett nätverk som är för litet inte kommer lyckas lösa det givna problemet. Dessutom, som om det inte vore nog så behöver styrkan på länkarna som skickar vidare signalen i nätverket individuellt ställas in till ett okänt optimalt värde. Denna process kallas för träning av nätverket och är ett av fundamenten för maskininlärning. Det finns många olika metoder för träning av ett nätverk.

En metod är Neuroevolution, som bygger på Darwins evolutionsteorier för att simulera naturligt urval. Det fungerar genom att ha en mängd olika nätverk (en population), alla med olika styrka på länkarna, och sen mutera denna styrka slumpvis. De muterade nätverken testas sen på ett givet problem och baserat på detta test rankas nätverken i populationen från bäst till sämst. De sämsta nätverken elimineras medan de bästa får lov att föröka sig och producera avkommor som ersätter de eliminerade nätverken. Förökningen sker genom att kombinera information om styrkan på länkarna mellan två föräldranätverk, och på så sätt skapa en avkomma som förhoppningsvis kommer prestera bättre. Denna process återupprepas till dess att en lösning på det givna problemet hittats. Detta löser problemet med att finjustera styrkan på länkarna, men det andra problemet om att hitta en optimal nätverksstruktur återstår. Det är här Neuroevolution of Augmenting Topologies, förkortat NEAT, kommer in i spelet.

NEAT använder neuroevolution för att både utveckla styrkan på länkarna och den faktiska strukturen av nätverket. Utvecklingen av strukturen på nätverket fungerar på samma sätt som för styrkan av länkarna, men istället för att styrkan på länkarna muteras så muteras nätverket i sig. Detta sker genom att antingen lägga till en ny nod på en existerande länk, eller en ny länk mellan två tidigare icke sammankopplade noder. NEAT är en väldigt kraftfull algoritm då den tar bort behovet för en människa att bestämma nätverksstrukturen, men har vissa problem när det kommer till att utveckla ett nätverk som kan lösa lite mer komplexa problem. NEAT introducerar också ett heuristiskt koncept vid namn artuppdelning (*eng. speciation*) vilket introducerar nya svårbestämda parametrar som behöver justeras av en människa. Artuppdelning delar, precis som namnet antyder, upp populationen i olika arter där varje nätverk inom en art liknar alla andra nätverk i samma art. Detta görs för att forcera en mångfald av nätverk i populationen så att utvecklingen inte stagnerar.

Det är här den nya algoritmen Childhood Habituation in Evolution of Augmenting Topologies, eller förkortat CHEAT, gör entré. CHEAT använder samma grundpricniper som NEAT men delar träningen av länkstyrkan och utvecklingen av nätverksstrukturen till två separata faser. Fasen för utvecklingen av nätverket kan ses som en analogi till hur två föräldrar mixar genetiskt material för att skapa en avkomma, men denna avkomma klarar sig inte själv utan behöver sina föräldrars hjälp. Träningsfasen kan ses som en analogi till hur föräldrarna skyddar och uppfostrar avkomman så att den lär sig att klara sig själv. När den är uppfostrad blir den utslängd i världen och om den har fått en bra mix av gener och har tränats väl så kommer den hitta en partner och cykeln återupprepas.

Resultaten av att studera denna algoritm har varit att det är mycket viktigt för större nätverk att tränas längre än små, vilket kan ses som en analogi till hur stora djur oftast stannar med sina föräldrar under än längre period än små. För det andra så är det viktigt att styra nätverkens utveckling för att lösa komplexa problem. Detta görs genom att forcera nätverket till en lagerstruktur där noder i ett visst lager har länkar till alla noder i föregående samt efterföljande lager. Med dessa modifikationer lyckas CHEAT lösa de problem som uppstod för NEAT, samtidigt som CHEAT helt tar bort det heuristiska artuppdelningskonceptet, och de associerade svårbestämda parametrarna. Detta öppnar upp för intressant uppföljande forskning där det ultimata målet hade varit att låta algoritmen ta över kontrollen över alla parametrar som annars behöver hjälpande mänsklig hand.

Contents

1 Introduction

Genetic Algorithms (GAs) are a set of optimization algorithms that use Darwin's theory of natural selection to computationally evolve a solution to an optimization problem. When applied to train Artificial Neural Networks (ANNs), these methods collectively go under the name of Neuroevolution (NE). Neuroevolution of Augmenting Topologies (NEAT) is one of these methods developed by K.O Stanley and R. Miikkulainen [1] which evolves both the topology and the weights of the ANNs. NEAT freely evolves from a minimal structure to find the smallest possible network needed to solve a problem. It has been shown to be able to find that network for the XOR problem and also managed to solve the double pole balancing problem with (markovian) and without (non-markovian) information on the velocity of the cart better than other popular NE and Reinforcement Learning (RL) algorithms at the time [1]. However, NEAT has not been shown to be able to solve more complex labelling problems. It also introduces a speciation heuristics with the argument of diversifying the population, which in turn introduces a few hyper-parameters that are difficult to optimize and are heavily problem specific.

With NEAT as the basis, this thesis describes and tests a new algorithm called Childhood Habituation in Evolution of Augmenting Topologies (CHEAT), which removes the need for the speciation heuristics by splitting the evolution of the topology and the training of the weights into two separated phases of the algorithm. This additionally allows for implementation of different training methods tailored to the problem at hand. CHEAT also has an option to enable structured evolution of the topology by forcing fully connected layers, which allows it to evolve bigger networks for solving more complex labelling problems.

The thesis is structured by first going through the basic background regarding genetic algorithms, how they can use neuroevolution to train artificial neural networks, as well as the theory behind the NEAT algorithm. The following section describes the fundamentals of CHEAT algorithm and outlines the important differences compared to the NEAT algorithm. In the results section the important features of the algorithm are tested and presented, and then discussed in the section after. Finally the thesis is rounded off with a summary of the major findings and conclusions as well as the outlook for potential future research.

2 Background

2.1 Genetic Algorithms

The general idea of a genetic algorithm is to let a *population* of *individuals* breed, mutate, and compete to evolve the population and solve a problem. The individuals of the population are different potential solutions to said problem. Each individual is encoded to a string of bits, a so called *genome* or *genotype*. Each bit defines the expression of different features, where a 1 means expressed while 0 means absent [2, 3].

A GA consists of 2 main phases: the evaluation phase which measures the performance and assigns a fitness score to the individual, and the reproduction phase which creates offspring that replaces the parents by means of crossover (mixing of genes), and/or sometimes mutation. The fitness score can be calculated in many different ways, since it is technically only a measure of the performance, but an example could be the cross-entropy error function based on the target/output pairs for labelled problems. Each cycle of these phases is called a *generation* [2, 3]. The mutation operator is used in GAs to model mutation of genetic material by flipping random bits in the genome of an individual. This operation causes a smaller change to the genome than crossover, and is used to make fine-tuning adjustments to the genome. Mutation occurs during the reproduction phase [3].

2.1.1 Tournament Selection

Tournament selection is a popular scheme used in some GAs to determine which individuals are selected for further genetic processing out of the population pool. The scheme is simple, n individuals are chosen out of the pool at random and the best is allowed to proceed and the worse is eliminated, i.e. the one with the best fitness score survives. Most of the times the tournament is done as a duel between a pair of individuals ($n = 2$), so called *binary tournament* [4]. There is also a method called *Restrictive*

Tournament Selection which restricts tournaments to only occur between individuals of the same niche[5]. Niching is a way to divide the population into groups of similar individuals (*niches*) to allow a diverse population that covers a larger global search space [6, 7].

2.1.2 Neuroevolution

Neuroevolution (NE) is a general term for methods that evolve Artificial neural networks (ANN) using GAs. In NE, the genome of an individual decodes for an ANN, and mutation and crossover cause modification to said network. NE has been found to be both faster and more efficient in solving some traditional Reinforcement Learning (RL) tasks than the popular Q-learning algorithms [8]. In most NE algorithms the population is filled by individuals with a fixed topology and randomized weights. Genetic Algorithms are then applied to train the weights of the networks in the population and evolve a solution to the problem. These NE algorithms are very reasonable to use for problems where the user have a good understanding of the problem and the required topology to solve the problem. However, in many cases this is a trial-and-error process, thus an argument for automating the process of selecting a topology to save time can be made, as shown by Grau et al.[9].

2.2 NeuroEvolution of Augmenting Topologies (NEAT)

NeuroEvolution of Augmenting Topologies, known as NEAT, is an algorithm developed by Kenneth O. Stanley and Risto Miikkulainen [1] at the University of Texas in 2002, and belongs in a subclass of NE algorithms known as Topology and Weight Evolving Artificial Neural Networks (TWEANN). NEAT is an attempt to show that the evolution of topology alongside the weights can be a powerful and efficient NE method. In this section I will give a description of NEAT and its components, as well as some flaws and vaguely argued heuristics which serves as the basis for the development of CHEAT.

2.2.1 Genome encoding

There are a few different methods of encoding a genome within the TWEANN methods. They are split into two main groups, *Indirect* or *Direct* encoding. *Indirect encoding* defines some rules on how to construct the phenotype (network) and is useful as it allows for a more condensed representation than direct encoding. *Direct encoding* is the more commonly used method and it specifies each node and connection in the phenotype and their unique properties. NEAT uses a variation of direct encoding which solves a common direct encoding problem called *Competing Conventions.*

The competing conventions problem occurs for any algorithm applying crossover between two different genomes and means that a specific network can be represented by more than one unique genome, as illustrated in **Figure 2.1**. If crossover between any two identical networks with different genomes would occur, the result would be lost information of an offspring's parents.

This problem becomes more severe for TWEANN methods because crossover between networks with differing topologies are possible. NEAT solves this by introducing a global innovation number that tracks new topological structure throughout the population by incrementing and assigning the number to each newly added structure as a historical marker. By using this method NEAT knows which genes match up with which during crossover, and thus reducing the competing conventions problem and ensures that less information is lost in the offspring.

During the crossover process each gene with the same innovation number from the parents are lined up and are considered so called *matching* genes. These genes represent the same exact structure in both of the parent networks. Any genes that do not have a matching gene in the other genome are called either *disjoint* or *excess* and they represent structures that do not exist in one of the two parents. A *disjoint* gene exists within the range of the other parent's innovation number range, i.e. a gene with innovation number 4 while the other parent have 7 as the maximum. An *excess* gene exists outside of the other parent's innovation number range, i.e. a gene with innovation number 8 while the other parent have 5 as the maximum[1]. See **Figure 2.2** for a schematic representation of this process.

The genome consists of two types of genes, *connections* and *nodes.* A connection gene links two nodes together and have the following properties: Innovation number, the 'in' node, the 'out' node, a boolean parameter specifying if the connection is enabled, and the weight. The node genes provide a list of all the

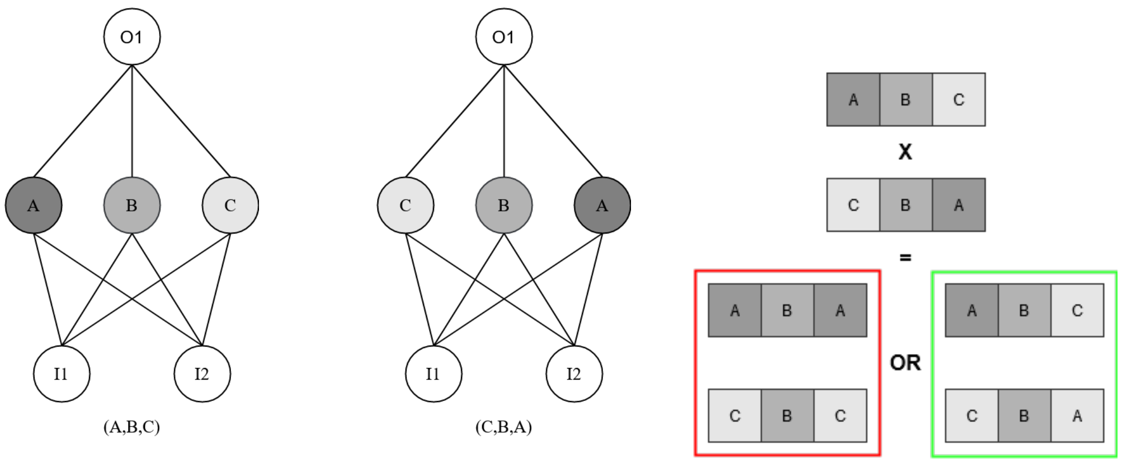


Figure 2.1: Simple schematic of how the same network can be represented in two different ways, the so called competing conventions problem. When crossover is applied on these genomes, 2 of the possible resulting networks are (A,B,A) or (C,B,C) which have lost information about the parent genomes.

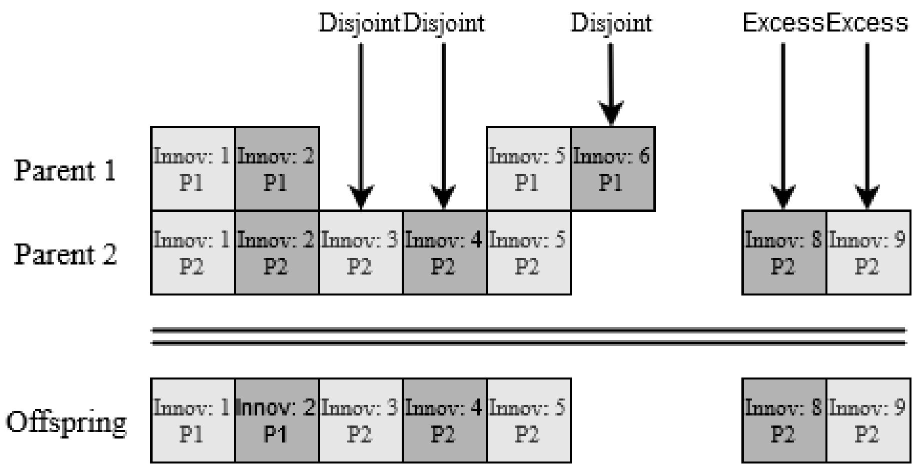


Figure 2.2: Schematic representation of two parent genomes creating and offspring genome by crossover. The parent genomes are aligned by matching genes with the same innovation number. Non-matching genes are either disjoint, if they exists within the innovation number range of the other parent or excess if not. Matching genes are inherited randomly while disjoint/excess genes are inherited from the more fit parent, in this case *Parent 2*.

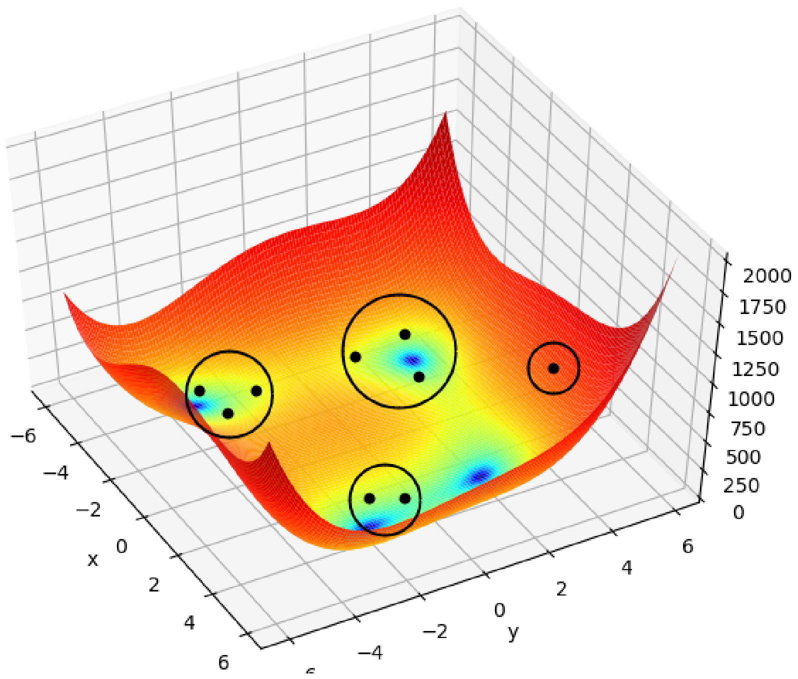


Figure 2.3: Individuals (black dots) of a population subdivided into niches (black circles) spread out over the search space landscape. Right most niche/individual is protected from elimination, even though it is performing badly, due to adjusted fitness and limited competition.

available nodes to be linked by the connection genes to form a network. The corresponding phenotype (the neural network) is constructed by reading the genome and linking the available nodes as specified. For a schematic representation of the encoding and decoding of the NEAT genome, see **figure2.4**.

2.2.2 Speciation

NEAT uses a niching method called *explicit fitness sharing*, which forces individuals of the same niche to share the fitness. Niching is a way to subdivide the population into smaller groups (niches) based on some criterion, and competition is limited to within the niches. In the case of NEAT that criterion is the genomic distance or in other words the genomic similarity. Fitness sharing adjusts the fitness of each individual based on the number of individuals in its niche. This makes it unfavourable to have many similar individuals and thus forces the population to diversify and protects topological innovation. See **figure 2.3** for an example of how niching can subdivide a population and protect unoptimized topological innovation. The individual fitness is adjusted accordingly:

$$f_i^{'} = \frac{f_i}{\Sigma_{j=1}^{n}\mathrm{sh}(\delta(i,j))} \tag{1}$$

where f_i is the fitness of the genome, $\delta(i,j)$ is the topological distance between genome i and j, and $\mathrm{sh}(\delta(i,j)$ is the sharing function which evaluates to 0 if $\delta(i,j)$ is larger than the threshold δ_{th}, and evaluates to 1 if $\delta(i,j)$ is below[1].

The topological distance between two genomes, i and j, is determined by

$$\delta(i,j) = \frac{c_1 E(i,j)}{N} + \frac{c_2 D(i,j)}{N} + c_3 \cdot \bar{W}(i,j) \tag{2}$$

where $E(i,j)$ is the number of excess genes between the genomes, $D(i,j)$ is the number of disjoint genes between the genomes, $\bar{W}(i,j)$ is the average difference in weights of the matching genes, and N is the number of genes in the larger of the two genomes. The coefficients c_1, c_2, and c_3 are hyper-parameters that can be adjusted to determine the importance of E, D, and $\bar{W}$ [1].

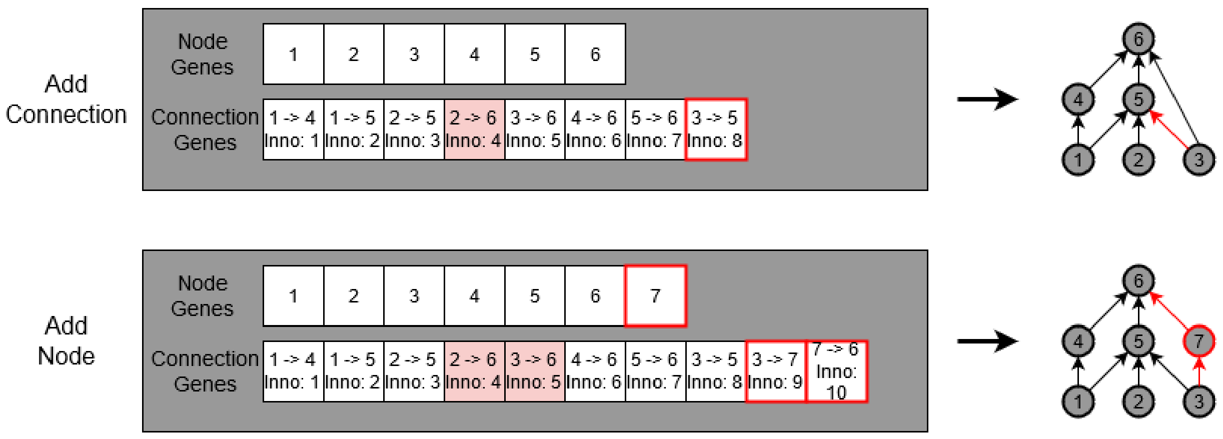


Figure 2.4: Shows a schematic representation of how the genome of an individual is encoded and decoded in NEAT. It also shows how structural mutations affect the genome. Genes with the red border represent a newly added gene, while genes with a red background are disabled genes, i.e. not expressed in the network. Adding a connection adds 1 new connection gene, and adding a new node adds 1 new node gene and 2 new connection genes as well as disabling the connection gene that the node was added on top of.

2.2.3 Mutation and Crossover

The n best individuals of each species with more than N individuals, are copied unchanged into the next generation (elitism). The rest of the offspring are created from either mutation only or from mutation with crossover. The parents for each species are the $r\%$ best performing individuals of that species. If a species does not increase its maximum fitness after g generations it is eliminated.

Mutation of the connection weights are determined by hyper-parameters defining if and how the weights are mutated. There is an $m\%$ chance that the weights of a genome are mutated. In that case there is an $m_{uniform}\%$ chance of a uniform perturbation of the weight, and otherwise the genome is assigned random new weight values. Structural mutations are determined by hyper-parameters defining the probability to add a new link and/or node. There is an $a_{node}\%$ chance to add a new node and an $a_{connection}\%$ chance to add a new connection. A new node is always added on an existing link by disabling the link and adding two new links to and from the node, i.e. splitting the link.

There is an $o_{wc}\%$ chance of an offspring being produced from mutation without any crossover, while the rest are produced from mutation with crossover. There is a low $c_{interspecies}\%$ chance for crossover between two genomes from different species, so called inter-species reproduction. Crossover happens by randomly selecting two genomes from the parent pools of each species. The genes of the genome are then lined up using the innovation number. The offspring is created by randomly inheriting one of the parents matching genes, and inheriting the disjoint/excess genes from the more fit parent. If both parents happen to have the same fitness the excess/disjoint genes are also inherited randomly as seen in the example given by **Figure 2.2**. For a schematic representation of how mutations modify the genome of an individual, see **figure 2.4**. For a schematic representation of how crossover works in NEAT, see **figure 2.2**. The hyper-parameters listed above are manually selected by a human, and optimal parameters are determined by trial-and-error.

2.2.4 neat-python package

For convenience, the neat-python package [10] was used for any comparisons to the NEAT algorithm. The neat-python package is a python adaptation of Stanley's original C++ code-base, based on the NEAT papers.

3 Childhood Habituation in Evolution of Augmenting Topologies (CHEAT)

CHEAT is a new NE algorithm based on NEAT that removes the need for the speciation concept to protect topological innovation by separating the topology evolution and weight training into two distinct phases. CHEAT also expands on the domain of solvable optimization problems by allowing for the forced evolution of fully connected layers instead of completely freely connecting links.

3.1 Protected Childhood Training

3.1.1 An analogy to biology

K.O. Stanley & R. Mikkulainen claims that speciation in NEAT is analogous to how speciation works in biology. However, arguments could also be made that such is not the case, since speciation in NEAT limits competition to within each species which is clearly not the case in biology. In nature species are also competing against one another, while there is protection of the offspring within the species/family itself. In fact, one could argue that a protected childhood training is more analogous to biology, where the offspring is trained to survive on its own in the wild by its parents in a protected environment for the first part of its life. This concept of a protected training phase serves as the basis of CHEAT and allows for removal of the speciation heursitics otherwise required in NEAT to ensure that new topological innovation has a chance to survive in the genome. This analogy to biology and nature is interesting, but what is more important is how it works in practice.

3.1.2 Training black-box

The protected childhood training is in practice a black-box for training since it is independent from the rest of the CHEAT algorithm and whichever training algorithm best suitable for problem can be used. For example RL algorithms and GAs for unlabelled problems, or gradient descent for labelled problems. The only requirement being that the black-box takes an untrained population as an input and returns a trained population. For the purpose of this thesis a mini-batch stochastic gradient descent method with a momentum term was implemented as a training method.

3.2 Genome Encoding

The genome in CHEAT differs slightly from NEAT in that links and nodes are not represented by two different genes. Instead the links in CHEAT are encoded as a property of the node gene. Each gene of the genome represents a node in the network and contains a list of its properties: a number which represents the layer index, the node bias weight, its activation function, and finally a list of its input links with their respective weight and the previous weight update, which is used in calculation of the momentum term in gradient descent. The bias and link weights are randomly assigned during the initialization phase by a normal distribution with mean 0 and a user-defined variance. The genome also encodes the fitness score and all the existing layer indices of the network.

The genome of the initial population is created during the initialization phase, where the topology of all individuals are constructed according to the user-defined parameters N_{in}, N_{hidden}, and N_{out}, which defines the number of input, hidden, and output nodes respectively. The initial topology can be layer-wise either fully connected or partially connected defined by c_p, between 0 and 1, which determines the probability for each initial connection to exist.

3.3 The CHEAT algorithm

CHEAT is like NEAT a TWEANN, but with the fundamental difference that in CHEAT, the evolution of the topology and the training of the weights are separate from one another. This means that there are three main phases during one generation of the algorithm: the topology evolution phase, the protected weight training phase, and the population recombination phase. Before the first generation there is also an initialization phase.

The initialization phase starts by constructing a population of n individuals (ANNs), where n is the population size. Each individual is initialized with normally distributed weights around 0 for a given topology (can be minimal, i.e. inputs directly connected to outputs, or not). Each individual's performance is then evaluated on the given problem and assigned a fitness score. The assigned fitness score can be based on an error function, like cross-entropy (as is used for the scope of this thesis), if the training data have known labels, or some other predefined metric able to measure the performance. The purpose of the CHEAT algorithm is to either minimize or maximize said fitness score. After the initialization phase the algorithm proceeds to the topology evolution phase of the first generation.

The topology evolution phase starts by selecting the top $r_{co}\%$ performing individuals of the population to enter a reproduction pool. The individuals in the reproduction pool produce n new offspring, so that the total combined size of the population is $2n$ individuals. The reproduction occur by randomly selecting two individuals from the reproduction pool and applying crossover, in the same way as NEAT, to produce one new offspring and then also having an $m\%$ chance to mutate each individual weight in the genome. Structural mutation occurs after the crossover has been done by adding a node to the offspring genome if $R_1 < a_{node}$, and a connection if $R_2 < a_{connection}$. R_1 and R_2 are two random numbers uniformly generated between 0 and 1, and a_{node} and $a_{connection}$ are two hyper-parameters between 0 and 1, which determines the probability of adding a node and/or a connection respectively.

In the protected weight training phase the produced offspring population is passed to the training black-box. The trained offspring coming out of the training black-box is then evaluated and assigned a fitness score, and then recombined with the rest of the population in the final phase of the algorithm.

In the recombination phase, the $2n$ individuals of the surviving population are reduced by a simple binary tournament selection scheme as described in **Section 2.1.1**. At this point the first generation of the CHEAT algorithm is done, and this process is repeated until an ending criterion, such as target fitness or maximum generations, has been met. For a step-by-step representation of thee algorithm see **figure 3.1**.

3.4 Layer indexing

To make decoding of the genome easier, CHEAT uses layer indices to keep track of the position of the nodes in the network. The initial nodes of the network are assigned a fixed layer index, and any subsequent added node is assigned a layer index calculated by the following

$$l_n = \frac{l_{n_{in}} + l_{n_{out}}}{2}. \tag{3}$$

Where l_n is the index assigned to the added node, $l_{n_{in}}$ is the index of the node that is the input to the added node, and $L_{n_{out}}$ is the index of the node that is the output to the added node. If the input nodes are assigned the layer index 0 and the output nodes are assigned the layer index 2, then the first hidden node added would be assigned the layer index $l_n = \frac{0+2}{2} = 1$. Layer indices can be floating point numbers, and thus the depth of a network is limited by the floating point precision and the choice of index for the output layer.

3.5 Stopping criterion

Intuitively the bigger a network is, the longer it will take to train. The reason being an increased number of tunable parameters for larger networks. It can therefore be of interest to define a stopping criterion that automatically stops the training process of a network once converged. This can save computation time and require less human interference and/or input.

In the case of the Gradient Descent training method implemented into CHEAT, the training of an offspring is stopped when the absolute difference between the average error of the last w_1 epochs and the average error of the w_2 last epochs before that, is smaller than a given factor of the average error of the two windows, i.e. the past $w_1 + w_2$ epochs. The stopping criterion is triggered when

$$\left| \frac{1}{w_1} \sum_{k=0}^{w_1} E_{i-k} - \frac{1}{w_2} \sum_{m=0}^{w_2} E_{i-w_1-m} \right| < c \cdot \frac{\left(\frac{1}{w_1} \sum_{k=0}^{w_1} E_{i-k} + \frac{1}{w_2} \sum_{m=0}^{w_2} E_{i-w_1-m} \right)}{2} \tag{4}$$

where w_1 is the size of first window, $w_2 = w_1$ is the size of the second window, E_i is the error at the i:th epoch of training, and $c < 1$ is the stopping factor.

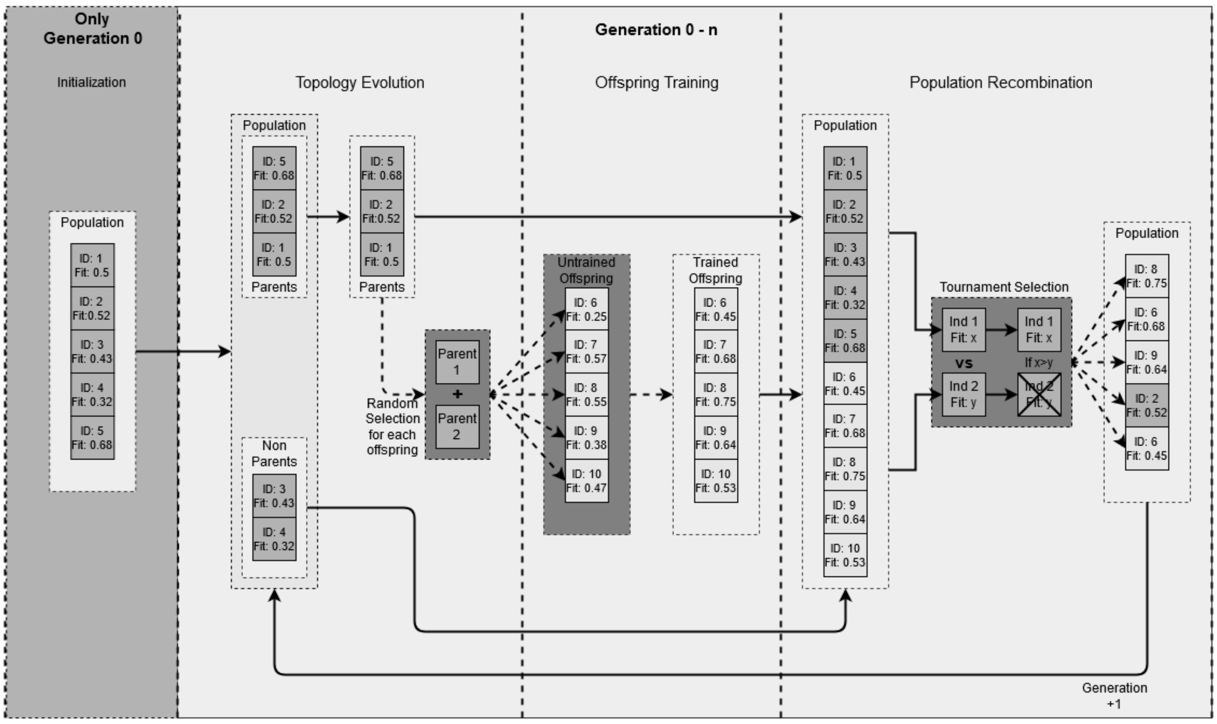
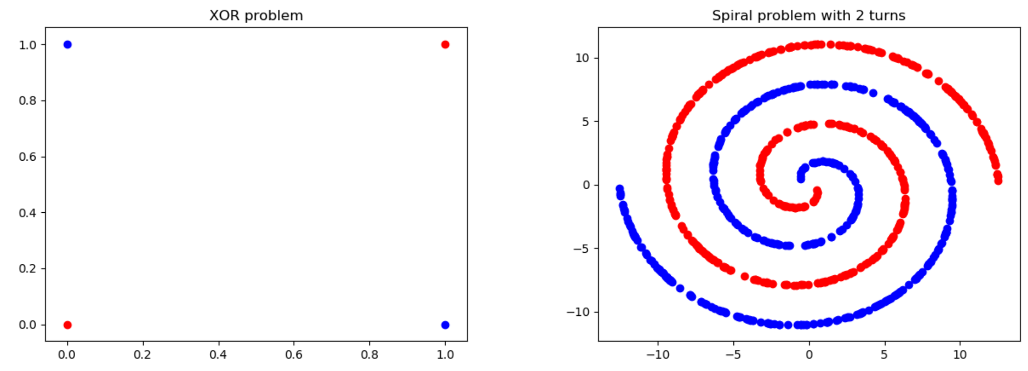


Figure 3.1: A schematic representation of the CHEAT algorithm. Initialization of a random population with minimal topology only occurs in Generation 0. After initialization the three phases Topology Evolution, Offspring Training, and Population Recombination is repeated once per generation. Topology Evolution phase splits the population into parents and nonparents. Two parents are picked at random to produce one offspring, this is repeated until the offspring population is the same size as the main population. The offspring population is in the offspring training phase trained with an applicable neural network training algorithms. Once trained the offspring is recombined with the parents and nonparents to form one population. This population is then halved by selecting two individuals from the population at random and comparing their fitness score. The higher scoring individual is allowed to survive while the other one is eliminated. Once the population is halved the generation counter is incremented and the procedure starts over from the topology evolution phase.

Alongside this convergence-based stopping criterion, there exists a parameter limiting the maximum number of epochs allowed during the training phase. If training has not been stopped before the epoch limit is reached, the training will be stopped anyways to avoid long inefficient training in case of a badly tuned convergence-based stopping criterion.

3.6 Network connectivity

In NEAT the network is dynamically evolved without any condition for fully connected layers, and also allows for connections between nodes to skip layers. This evolution condition is required to solve some problems that require small topologies. However, fully connected layers can serve as a feature selection filter which can be a crucial part to solve more complex problems. Therefore there is an option to enforce fully connected layers without skipping connections in CHEAT and if enabled, another parameter, r_{db}, that controls the depth versus the breadth of the topology evolution is introduced. If $r_{db} = 0$ then all new nodes will be placed in the existing hidden layers unless no hidden layer exists, in which case the node will create a new hidden layer. If $r_{db} = 1$ then all new nodes will always create a new hidden layer. For values of r_{db} between these extremes, a uniform random number, R, between 0 and 1 is generated. If $R < r_{db}$, then the new node will create a new hidden layer, and vice versa.



(a) (b)

Figure 3.2: The two colours (blue & red) represents two different classes to be identified by the ANN. (a) shows the layout of the XOR problem in euclidean space. (b) shows the layout of the spiral with 2 turns in euclidean space.

3.7 Adaptive growth

Adaptive growth in CHEAT means that the rate of topology evolution varies with network size. The idea is that if a problem requires a large network to solve, you wish to evolve the network faster in order to reach such a network size quickly. The adaptive growth in CHEAT is determined by a growth parameter g, and the sum of the number of nodes and connections in the network. For each generation of the topology evolution phase, a number is calculated for each individual in the population which indicates the number of structural changes that are to be made. This number is determined by

$$n_{ag} = \lfloor g \cdot (N_n + N_c) \rceil \tag{5}$$

where $\lfloor x \rceil$ denotes the nearest integer to x that is not 0, N_n is the total number of nodes in the network, and N_c is the total number of connections in the network. The algorithm then runs the loop of attempting to add a node and/or connection to the topology if the network is not fully connected, or add only nodes if network is fully connected. The steps of that loop is:

```
while $n < n_{ag}$ do:
    if $R_1 < a_{node}$: add node to topology & n++
    else: do nothing
    if $R_2 < a_{connection}$: add connection to topology & n++
    else: do nothing
```

where n is the total number of structural changes to that individual.

3.8 Problem Definition

The simple XOR problem and the slightly more complex spiral problem with two turns was used measure the performance of the CHEAT algorithm compared to the NEAT algorithms, as well as understanding the effects that different aspects within the CHEAT algorithm have. These are two labelling problems of differing complexity. The aim of the XOR problem is to create an ANN that is able to act as an XOR logical gate. The XOR gate consists of two binary inputs that yield an output of 0 if both the inputs are either 0 or 1, and 1 if either of inputs are 1 and the other 0. The spiral problem is slightly more complex and the aim is to correctly identify two lines which spiral outwards from origo for a set number of turns, which in this thesis is always 2 full turns, while only giving the euclidean coordinates as the two inputs to the ANN. See **figure 3.2** for a representation of the layout for the two problems in euclidean space.

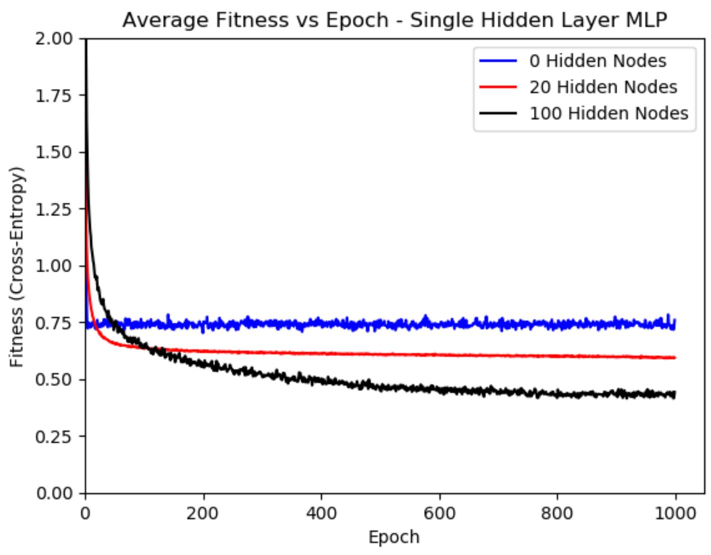


Figure 4.1: The figure shows how three different plots which each represent the average error at each epoch for 64 networks of a fixed topology when training with gradient descent. The blue line shows the results for networks with 0 hidden nodes, the red line for networks with 20 hidden nodes in one layer, and the black line for networks with 100 hidden nodes in one layer. Fitness score assigned as the Cross-Entropy error of the ANN.

4 Results

A quick investigation of different values for the available hyper-parameters lay as the foundation to the values chosen to produce the results in this section. All hyper-parameters constant to all problems and tests can be found in **Appendix A**. The problem/task specific hyper-parameters can be found in **Appendix B,C,D,E**.

4.1 Stopping criterion

Figure 4.1 shows a comparison between the training error for three different network sizes. Each line shows the average error per epoch for 64 networks of a given network topology trained with gradient descent on the spiral problem with 2 turns. The different topologies are: 0 hidden nodes, 20 hidden nodes in one layer, and 100 hidden nodes in one layer. The figure shows that bigger networks take longer to converge, and that smaller networks at some points perform better than the bigger networks after the same number of epochs. However, eventually the bigger networks start performing better and converge to a lower error.

Figure 4.2 shows when the stopping criterion is triggered using different values of c, on the same data-sets as shown in **figure 4.1**. The result show that, the smaller the value of c is, the later in the training process the stopping criterion will trigger. When $c = 0.1$ the stopping criterion is triggered on each data-set such that networks with 20 hidden nodes on average perform better than networks with 100 hidden nodes, while the networks with 0 hidden nodes on average perform the worst. However, when c is smaller the stopping criterion is triggered on each data-set such that networks with 0 hidden nodes perform the worst, networks with 20 hidden nodes perform the second worst, while networks with 100 hidden nodes perform the best.

4.2 Network connectivity

Figure 4.3 shows the resulting network topologies produced when using CHEAT on the XOR problem. **Figure 4.3a** shows the resulting network when CHEAT is allowed to evolve the topology freely. The resulting topology is the minimal topology needed to solve the XOR problem successfully. **Figure 4.3b** shows the resulting network when CHEAT is forced to construct topologies with fully connected layers.

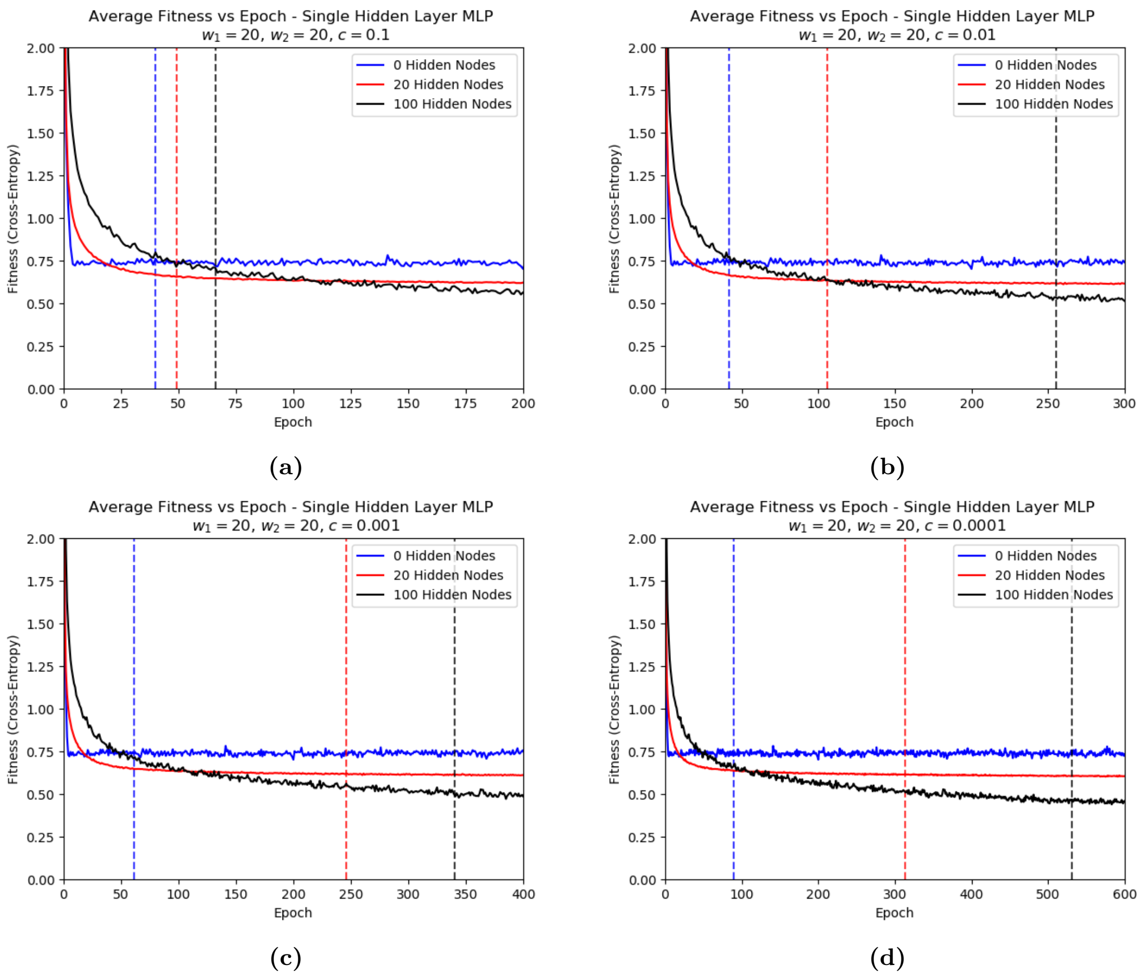
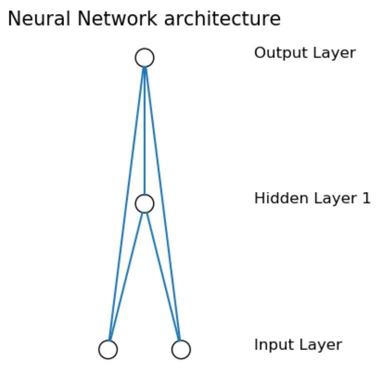


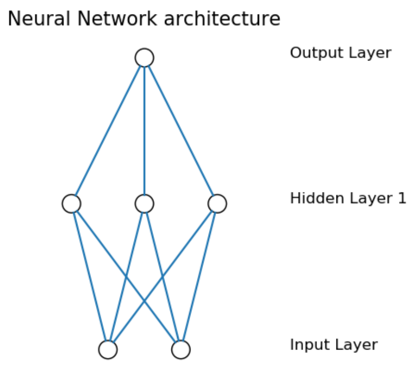
Figure 4.2: Results of the stopping criterion using different values of c. The stopping criterion is applied on three different data-sets. Each data-set is the average fitness of each epoch of 64 different networks trained on the spiral problem with 3 turns using gradient descent. The blue data-set is the average fitness of 64 networks with random initial weights and 0 hidden nodes, the red data-set with 20 hidden nodes, and the black data-set with 100 hidden nodes. Fitness score assigned as the Cross-Entropy error of the ANN.

The topology consists of 2 input nodes, 3 hidden nodes, 1 output node. The input nodes fully connects to all 3 hidden nodes, the 3 hidden nodes then fully connect to the single output node. This is the second smallest fully connected topology possible, where the smallest possible one consists of 2 hidden nodes instead of 3.

Figure 4.4 compares the outputs of two networks produced by the CHEAT algorithm on the spiral problem with 2 turns. **Figure 4.4a** shows the resulting boundary plot for the produced network where the topology is allowed to evolve freely. This network is unable to solve the problem satisfactorily and produces a network that give outputs close to 0.5 in most cases. **Figure 4.4b** shows the same plot, but for the CHEAT algorithm where the topology is forced to have fully connected layers and no skipping connections. This network is able to solve the problem well, and is giving outputs that are close to either 1 or 0 in most cases leaving a distinct boundary between the two classes and a clear outwards spiralling band for each class. In both the XOR problem and the spiral problem with 2 turns, gradient descent was used as the training algorithm.

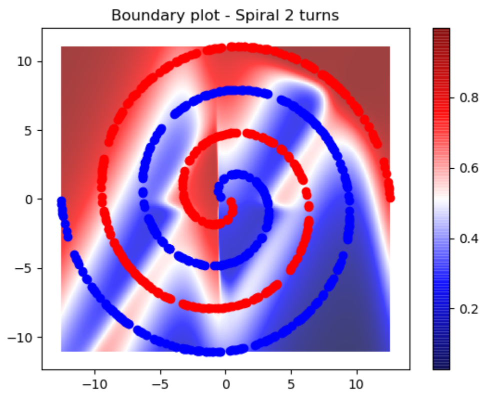


(a)

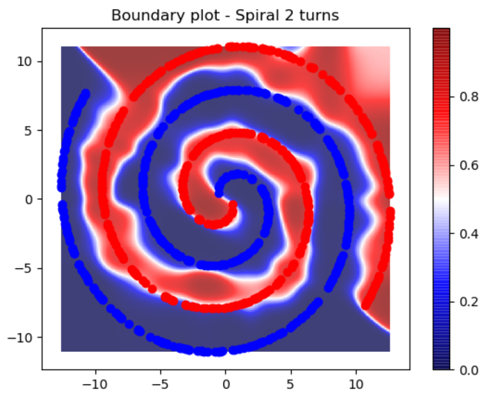


(b)

Figure 4.3: Shows the networks produced with the CHEAT algorithm on the XOR problem. (a) shows the produced network when CHEAT was allowed to evolve the topology freely, while (b) shows the network produced when CHEAT was forced to produce topologies with fully connected layers. (a) produced the minimal possible topology for solving the XOR problem, 2 input nodes, 1 hidden node, and 1 output node. The hidden node have connections from both input nodes, and to the output node. The output node have connections from the input nodes (skipping the hidden node), and one from the hidden node. (b) produced a network with 2 input nodes, 3 hidden nodes, and 1 output node, where all node are fully connected to the previous layer.

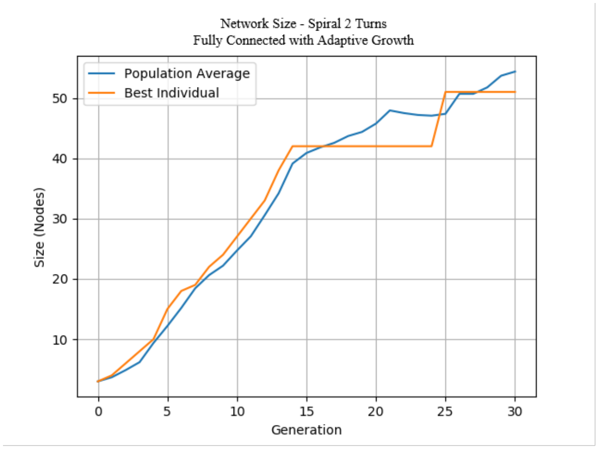


(a)

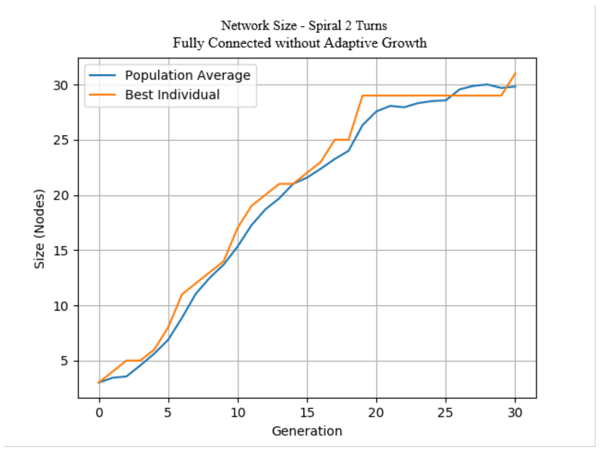


(b)

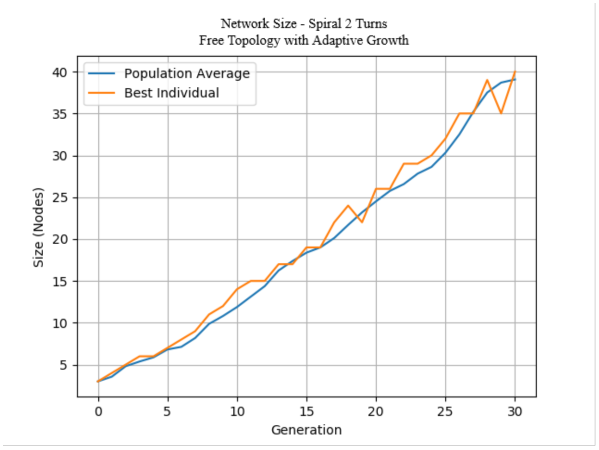
Figure 4.4: Shows the boundary plot for running the CHEAT algorithm on the spiral problem with 2 turns; (a) without forcing fully connected layers, and (b) with forcing fully connected layers. All other hyperparameters were identical. (a) shows the network being unable to solve the problem, while (b) shows the network being able to solve it. The background of the figure shows the output of the network given the two coordinate points as input. Outputs > 0.5 evaluate as identified to belong to the red spiral, while outputs ≤ 0.5 are evaluated to belong to the blue spiral, while 0.5 is the boundary between the two classes.



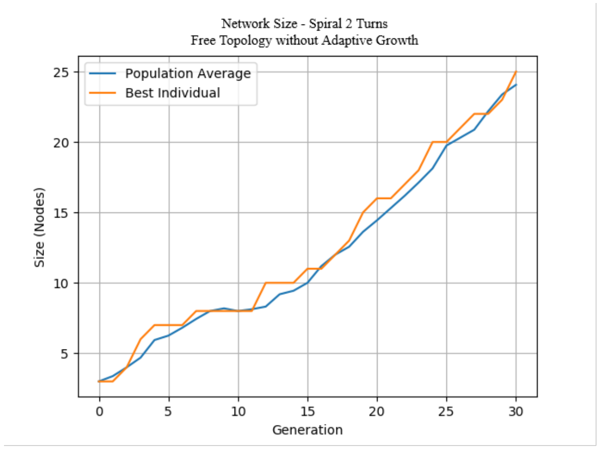
(a)



(b)



(c)

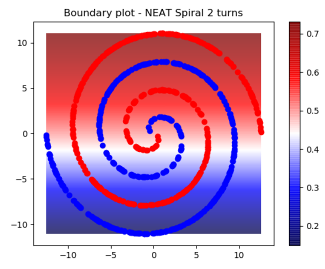


(d)

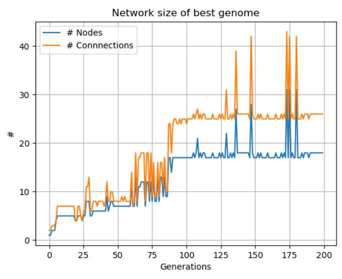
Figure 4.5: Shows the evolution of the size (number of nodes) of the network. The blue line shows the population average for each generation, and the orange line shows the size of the best performing individual in the population for each generation. Each figure is run with CHEAT on the spiral problem with 2 turns. The results shown are for: (a) fully connected layers with Adaptive Growth turned on ($g = 0.01$), (b) fully connected layers with Adaptive Growth turned off, (c) free topology with Adaptive growth turned on ($g = 0.01$), (d) free topology with Adaptive Growth turned off.

4.3 Adaptive Growth

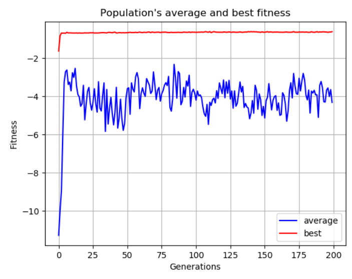
Figure 4.5 shows the evolution of the network size, i.e. the number of nodes in the network itself, when running the CHEAT algorithm with/without fully connected layers on the spiral problem with 2 turns with/without using adaptive growth. **Figure 4.5a** shows the result with using adaptive growth ($g = 0.01$), and **figure 4.5b** shows the result without using adaptive growth. The best network produced at the end of 30 generations with adaptive growth turned on had a cross-entropy error of $E = 0.123$ and a size of $N = 51$, while the best network with adaptive growth turned off had a cross-entropy error of $E = 0.126$ and a size of $N = 31$. Both of the runs produced networks that managed to solve the problem with 100% accuracy, and both runs had converged by 15 generations and only very minor and insignificant improvements happened after that. **Figure 4.5c** shows the result with using adaptive growth ($g = 0.01$), and **figure 4.5d** shows the result without using adaptive growth. The best network produced at the end of 30 generations with adaptive growth turned on had a cross-entropy error of $E = 0.521$ and a size of $N = 40$, while the best network with adaptive growth turned off had a cross-entropy error of $E = 0.493$ and a size of $N = 25$. Neither of the runs produced networks that managed to solve the problem.



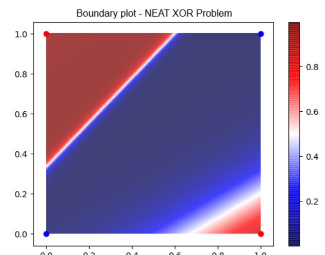
(a) Boundary plot of the produced network using NEAT on the spiral with 2 turns.



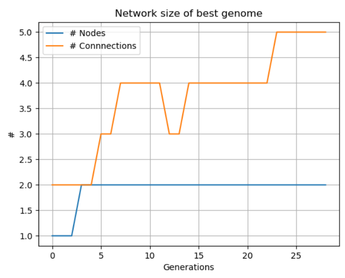
(b) Size evolution of the best network per generation using NEAT on the spiral with 2 turns.



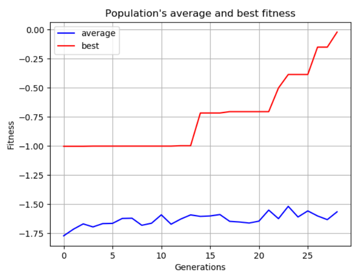
(c) The fitness (negative cross-entropy) per generation using NEAT on the spiral with 2 turns.



(d) Boundary plot of the produced network using NEAT on the XOR problem.



(e) Size evolution of the best network per generation using NEAT on the XOR problem.



(f) The fitness (negative squared absolute error) per generation using NEAT on the XOR problem.

Figure 4.6: Shows the boundary, network size, and fitness plots for NEAT applied on the spiral problem with 2 turns and the XOR problem.

4.4 Comparison to NEAT

Figure 4.6 shows the results of running the NEAT algorithm on the spiral problem with 2 turns and the XOR problem using the neat-python package. NEAT does not manage to solve the spiral problem after 300 generations, and stagnated in terms of fitness after only a few generations. The network size kept increasing until around generation 80 where it too stagnated at just under 20 nodes and just under 30 connections. The unsuccessful solution found with NEAT is linear. NEAT did, however, manage to solve the XOR problem in under 30 generations with the minimal structure needed, 1 hidden node, 1 output node, and 5 connections, as seen in **figure 4.3a**.

5 Discussion

5.1 Computational timescales

The software used to run CHEAT for testing purposes during this thesis is highly unoptimized and perform relatively poorly in terms of computational efficiency. To give a general idea of the performance: The algorithm was parallellized by multi-threading and run on an AMD Ryzen 7 3700X CPU with 8 cores (16 threads) @ 4.2 GHz. On the spiral problem with 2 turns with a stopping criterion active, it took approximately 6-24 hrs (depending on parameter tuning etc.) to finish training a population of 16 individuals for 30 generations at a final network size of approximately 20 nodes. As currently implemented, the training aspect using gradient descent is the limiting factor computationally speaking, and the time to finish 1 epoch for a single individual (could run 16 at the same time due to parallelism) scales linearly with the number of connections in the network, which in the worst case scenario scales exponentially with the number of nodes in the network. To run the CHEAT algorithm on even more complex problems which require larger networks, optimization of the code-base is vital to make it produce any usable results within a reasonable time. Two examples of such optimization could be to vectorize the gradient descent method, and parallellize the gradient descent method on to the GPU instead of the CPU.

5.2 Stopping criterion

If stopping occurs too soon, as can be seen in **figure 4.2a**, the result is that the bigger networks have not had enough time to converge and thus ends up performing worse, even though the bigger networks should if fully trained perform better on the training data. On the other hand, if stopping occurs too late, as can be seen in **figure 4.2d**, the algorithm is wasting time by continuing to train these networks. **Figure 4.2b** and **figure 4.2c** shows two alternatives where the stopping criterion is tuned correctly in such a way that the results are kept representative while not wasting computational power.

If the stopping criterion where stopping occurs too early were be used in CHEAT, it would mean that the topology evolution would stall. This is because individuals with a smaller network would perform better than individuals with large networks and thus win the tournament selection during the recombination phase. Stalling of topology evolution means that the best individual in the population would never be able to become better than what that specific topology size would allow for, and potentially never be able to solve the given problem.

Using a correctly tuned dynamic stopping criterion is thus vital and also allows the maximum number of training epochs to be set fairly high without the risk of wasting computational power since the training of an individual will be stopped once converged. This in turn will allow bigger networks to train for longer while smaller networks are stopped earlier, which is necessary to get a representative result as seen in **figure 4.2**.

5.3 Network connectivity

Fully connected layers can be seen as a way to extract features from a complex structure which for some problems can be vital to successfully solve the given problem within reasonable time and with a reasonably sized network. However, by forcing fully connected networks some freedom is taken away from the CHEAT algorithm to dynamically evolve the network. For some problems this can lead to the construction of networks with redundancy or in other words a network which is not the minimal possible topology required to solve the given problem. Having redundancy in the network means longer training times without any other benefits, due to more tunable parameters.

An example of redundancy can be seen in **figure 4.3**, where the CHEAT algorithm was used construct a network able to solve the XOR problem. It compares the networks constructed when CHEAT was allowed to evolve freely versus when fully connected layers were forced. During free topology evolution CHEAT manages to find the smallest possible solution to the XOR problem, which is an impossible network to construct if fully connected layers are forced. In turn, forcing fully connected layers resulted in CHEAT evolving a network with redundant nodes and connection. The algorithm didn't even find the smallest possible topology for a fully connected network on the XOR problem, which has 2 hidden nodes instead of 3, using the otherwise same hyper-parameters. This result shows that forcing fully connected layers can lead to construction of less efficient networks, which for this particular problem is not necessarily devastating due to the very simple nature of the XOR problem. However for a problem that is more complex, redundancy should be avoided since it may increase the time it takes to train the network.

In contrast, **figure 4.4** shows a result where forcing fully connected layers is needed to solve the given problem at all. The spiral problem is a more complex problem than the XOR, and so it makes sense that it would require fully connected layers to find features in the data-set. Even though the spiral problem with 2 turns is slightly more complex than the xor problem, it is still nowhere to close as complex as some deep-learning tasks are. Thus it seems as if forcing fully connected layers is required for any problem that cannot be solved by a very simple shallow network with only a few nodes. This goes against the base principles of NEAT, which is to let the topology evolve freely from a minimal structure. Simply put, when the problems get more complex it seems as if a guided evolution of the topology is the most efficient way of reaching a solution.

5.4 Adaptive growth

Based on the results seen in **figure 4.5**, for the spiral problem with 2 turns there does not seem to be any benefit to having adaptive growth or not. However, this is most likely due to the fact that in order to solve said problem, not very many nodes are needed to begin with. The trend observed though is that when using adaptive growth the network size evolves quicker than without it, but still in a seemingly linear fashion. The linearity is an interesting point since in theory we should see an exponential growth behaviour as the generations increase. However, this linear growth can be attributed to the use of a small growth factor $g = 0.01$, since the network sizes evolved during the first 30 generations could result in $g \cdot (N_n + N_c)$ being close to 1, and thus not resulting in a rapid increase in the number of structural changes per generation. When using large values of g though, the network sizes grew very rapidly to sizes which made the algorithm too slow to practically finish more than only a few generations within reasonable time given the hardware available (total estimated time for 30 generations where on the order of days to week). That said, there is a slight increase in the network sizes between the run with adaptive growth on versus off, but without any seeming benefit. Therefore in this specific case, the adaptive growth only slows the CHEAT algorithm down due to the fact that larger networks need to be trained. The same applies for when the CHEAT algorithm is allowed to evolve the network freely, with the only exception that neither run managed to solve the problem at all.

Adaptive growth could potentially avoid local optima due to small networks since the algorithm is forced to rapidly evolve larger networks. However, with the dynamic stopping criterion introduce in the CHEAT algorithm, stagnation to such local optima were not observed. Adaptive growth therefore mainly serves as a potential tool to speed up the network evolution in very complex problems.

5.5 Comparison to NEAT

Another very interesting aspect to look at is how the CHEAT algorithm stands in comparison to the NEAT algorithm, to see if any improvements were made and if so, in what way. As seen in **figure 4.6** managed to solve the XOR problem using the minimal possible structure, however failed to solve the spiral problem with 2 turns. This is also what the CHEAT algorithm without forcing fully connected layers managed. However, when forcing fully connected layers the CHEAT algorithm solved the spiral problem with ease. Why the NEAT algorithm gives a linear solution is unknown and somewhat odd since the network produced technically should be able to create a more complex boundary. A more expected result would be a solution similar to that of freely connecting CHEAT algorithm. A deeper analysis of this behaviour was not conducted due to time-constraints, but potential candidates for the cause is either something in the NEAT algorithm itself or potentially in the neat-python packaged used.

Based on these results, CHEAT seems to be able to evolve networks from minimal topology and solve the same problems as NEAT but with the addition that CHEAT is able to also solve more complex problems requiring a bigger more well-structured networks. CHEAT manages this while completely removing the speciation heuristics which in turn means the removal of several heuristic hyper-parameters connected to determining species. CHEAT also completely decouples the evolution of the topology and the training of the weights. This allows for freedom in selection of the training method, and thus can be selected and/or implemented according to the specifications of the problem. It also means that the evolution phase only have to find a good network topology to the problem, rather than both a good topology and optimal weights. This slims down the search space for the evolution algorithm significantly, and thus can more efficiently evolve the topology in a structured way, while the training phase only deals with finding the optimal weights for that particular topology. In short, two separate specialized phases instead of one generalist phase that does everything.

6 Conclusions

To summarize and conclude the findings of this thesis, CHEAT is a completely new Topology and Weight Evolving Artificial Neural Network algorithm based on the general idea of NEAT method created by K.O Stanley and R. Miikkulainen. In contrast to NEAT it separates the topology evolution and the weight updates into two phases. The weight training protects the offspring produced in the topology evolution phase, and is a black-box which takes an untrained population as input and gives a trained population as an output. Any valid training method for the given problem can be used. This removes the need to speciate the population as done in NEAT with a heuristic speciation method, and thus removes the task of optimizing the hyper-parameters associated to it.

It was found that a dynamic stopping criterion during gradient descent training was very important to ensure representative results. Without it, the algorithm risk either over-training and potentially wasting computational power on training an already converged network, or not training bigger networks long enough. This would cause them to perform worse than smaller networks and thus stagnating the topology evolution before the optimal structure had been found.

It is also very important to force fully connected layers in CHEAT in order to solve more complex problems that benefit from feature selection in some kind. This was very clear when running the CHEAT algorithm on the spiral problem with 2 turns, where without fully connected layers the problem was unsolvable. However, if the goal is to find the minimal possible structure to solve a simpler problem with, having fully connected layers means that those networks are not obtainable by the algorithm as showcased with the XOR problem. It is therefore important to have the option to chose whether or not to force fully connected layers.

When it comes to adaptive growth of the topology evolution, no clear immediate benefit was found. That said, it was only ever tested on a rather simple case, the spiral problem with 2 turns, which does not require very many nodes to solve to begin with. There is therefore a possibility that adaptive growth could be beneficial for more complex problems since it is clear from the results presented that it does increase the speed at which the networks grows at.

By comparing CHEAT to NEAT, it seems as if CHEAT is able to do what NEAT can do, and more. CHEAT is able to freely evolve from a minimal structure to find the smallest possible solutions to simple problems, which follows the same principle as NEAT. However, CHEAT is also able to evolve the topology in a structured way which makes the algorithm able to solve more complex problems which are outside the domain of possible problems for NEAT.

7 Outlook

The principles and results of CHEAT opens up further interesting points of research, which were not covered under the scope of this thesis. For starters, only gradient descent was used as a training method in this thesis which limits the algorithm to classic labelling problems, and thus excludes a lot of use-cases such as the domain of RL. The CHEAT algorithm as used in this thesis project was also limited to strictly feed-forward networks which excludes application on problems that require recurrent connections. The algorithm was also not very well optimized which meant analysis of problems requiring deeper networks was infeasible due to long training times. Optimizing the algorithm and including recurrent networks as well as more choices of training algorithms is critical to test the limits of the CHEAT algorithm and define its niche within the world of machine learning algorithms.

Secondly, since the stopping criterion have been found to be of high importance to the functionality of the algorithm it is worth spending more time on finding a good stopping criterion. An example of an interesting stopping criterion to potentially explore is the *Evidence-Based Criterion (EB-criterion)* [11] which is a stopping criterion based on the local statistics of the computed gradients which removes the need for a withheld validation set.

Finally, the algorithm touches on the subject of automatic model selection which is a hot topic. Allowing the algorithm to automatically evolve and set its own parameters while running means that the user does not have to spend time on model selection. It can therefore be of interest to see how CHEAT behaves when the control of the hyper-parameters is handled by the algorithm, i.e. mutating and/or crossover of hyper-parameters during the evolution phase.

Acknowledgements

I would like to thank Patrik Edén for his invaluable supervision and support throughout this project. You have a supervision style that encourages a lot of exploration and trying out your own ideas, while at the same time making sure that you don't stray to far off the track. I really value my time working with you, and I take new knowledge with me into the future. Thank you!

I would also like to thank Robin Emanuelsson for the discussions and company you provided. You helped me take a step back and see any problems I encountered in a new light, leading to ingenious solutions to problems that would otherwise still be unsolved.

Finally, I'd like to thank the members of my family, both humans and animals, for patiently listening to my incoherent and incomprehensible ramblings about problems beyond their understanding. Without your encouragement and belief in me throughout my studies, I would not have reached this point as painlessly as I have.

Bibliography

[1] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. doi: 10.1162/106365602320169811.

[2] John H. Holland. *Adaption in Natural and Artificial systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[3] David E. Goldberg. *Genetic Algorithms in search, Optimization, and Machine Learning.* Reading, MA:Addison-Wesley.

[4] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann Publishers, Inc, 1991.

[5] Gorges R. Harik. Finding multimodal solutions using restricted tournament selection. In Larry J. Eshelman, editor, *Proceedings of the 6th International Confereance on Genetic Algorithms*, pages 24–31. Morgan Kaufmann Publishers, Inc, 1995.

[6] Samir W. Mahfoud. *Niching Methods for Genetic Algorithms.* PhD thesis, University of Illinois, May 1995.

[7] Bruno Sareni and Laurent Krähenbühl. Fitness sharing and niching methods revisited. *IEEE Transactions On Evolutionary Computation*, 2(3):97–106, 1998. doi: 10.1109/4235.735432.

[8] David E. Moriarty and Risto Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996. doi: 10.1023/A:1018004120707.

[9] Frédéric Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 81–89, Cambridge, MA, USA, 1996. MIT Press. ISBN 0262611279.

[10] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. neat-python. URL `https://github.com/CodeReclaimers/neat-python`. latest version as of September 26, 2020.

[11] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. Early stopping without a validation set, 2017.

Appendix A: Constant Hyper-Parameters

Parameter	Value	Description
N_{in}	2	*Number of input nodes*
N_{hidden}	0	*Number of initial hidden nodes*
N_{out}	1	*Number of output nodes*
σ_w	5	*σ-value for weight initialization normal dist.*
μ_w	0	*μ-value for weight initialization normal dist.*
v_{i_m}	30	*Weight max/min value*
f_{activ}	Sigmoid	*Activation function*
m_p	1	*Weight mutation probability*
σ_m	3	*σ-value for weight mutation normal dist.*
μ_m	0	*μ-value for weight mutation normal dist.*
$a_{connection}$	0.5	*Probability to add new connection*
N_{pop}	16	*Number of individuals in population*
f_{goal}	'min'	*Fitness goal - Maximize or Minimize ('max'/'min)*
G_{max}	30	*Maximum number of generations*
r_{co}	0.2	*Crossover rate, i.e. fraction of population selected as parents*
μ_{gd}	0.1	*Gradient descent momentum parameter*
L_E	Cross-Entropy	*Gradient descent loss function*
e_{max}	10000	*Gradient descent maximum number of training epochs*

Table 1: Lists the constant hyper-parameters common to all problems and tests in this thesis.

Appendix B: Gradient Descent Hyper-parameters

Spiral problem with 2 turns

Parameter	Value	Description
ζ	0.005	*Gradient descent learning rate*
N_{bs}	25	*Gradient descent batch size*
a_{node}	0.5	*Probability to add new node*

Table 2: Lists the hyper-parameters for gradient descent unique to the spiral problem.

XOR problem

Parameter	Value	Description
ζ	0.3	*Gradient descent learning rate*
N_{bs}	1	*Gradient descent batch size*
a_{node}	0.1	*Probability to add new node*

Table 3: Lists the hyper-parameters for gradient descent unique to the XOR problem.

Appendix C: Dynamic Stopping Hyper-Parameters

Parameter	Value	Description
c	0.000005	*Stopping factor*
w_1	20	*Size of stopping criterion window 1*
w_2	$w_1 = 20$	*Size of stopping criterion windows 2*

Table 4: Lists the additional hyper-parameters used when dynamic stopping is enabled.

Appendix D: Fully connected layers hyper-parameters

Parameter	Value	Description
r_{db}	0.01	*Depth vs breadth ratio, i.e. ratio of adding a node to new layer or existing layer*

Table 5: Lists the additional hyper-parameter used when fully connected layers are forced.

Appendix E: Adaptive growth hyper-parameters

Parameter	Value	Description
g	0.01	*Growth parameter*

Table 6: Lists the additional hyper-parameter used when adaptive growth is enabled.

Appendix D: Software Availability

The CHEAT software written and used in this thesis is made available under the BSD-3-clause license at `https://github.com/AntonMoberg/CHEAT/releases` as release v0.1.0. Any potential future releases will also be found here with an incremented release tag.