CHALMERS

Efficient neuroevolution through accumulation of experience

Growing networks using function preserving mutations

Master's thesis in Complex Adaptive Systems

AXEL LÖF

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

MASTER'S THESIS IN COMPLEX ADAPTIVE SYSTEMS

Efficient neuroevolution through accumulation of experience

Growing networks using function preserving mutations

AXEL LÖF

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2019

Efficient neuroevolution through accumulation of experience
Growing networks using function preserving mutations
AXEL LÖF

Master's thesis 2019:13
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2019

Efficient neuroevolution through accumulation of experience
Growing networks using function preserving mutations
Master's thesis in Complex Adaptive Systems
AXEL LÖF
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

Abstract

In deep supervised learning the structure of the artificial neural network determines how well and how fast it can be trained. This thesis uses evolutionary algorithms to optimize the structure of artificial neural networks. Specifically, the focus of this thesis is to develop strategies for efficient neuroevolution.

The neuroevolutionary method presented in this report builds structures through architechtural morphisms that, approximately, preserve the functionality of the networks. The intended outcome of basing the mutations on the idea of function preservation was that new architechtures would start out in a high performance parameter space region. By skipping regions of low performance, the training of previous generations can be accumulated.

The proposed method was evaluated relative to version in which the preservating property of the mutations was removed. In the ablated version the parameters associated with the new structural change were randomly initialized. The two versions were benchmarked on five different regression problems. On the three most difficult problems the ablated version demonstrated better performance than the preservering version, while similar performance was observed for the two other problems. The performance difference between the two versions was inferred to a more frequent tendency for the function preserving version to get entrapped in stationary regions, compared to the ablated version. The parameter initializations associated with the ablated version allow the backpropagation to more easily escape these stationary regions.

The main contribution of this work is the conclusion that in order to efficiently utilize function preserving transformations to build structures in neuroevolution there need to be some mechanism that allows the backpropagation to esacpe stationary regions. The method is expected to improve by perturbating the parameters of the networks in a way that increase the gradient.

Keywords: Neuroevolution, ANN, Artificial Neural Networks, Function Preserving Transformations, Mutations, Competing Conventions

Acknowledgements

I would like to thank my thesis advisors Anna Samuelsson and Gustaf Johansson for all their support. Thank you for patiently listening to my ideas, encouraging me and proof reading my report countless number of times. I would also like to thank my thesis examiner Peter Forsberg for his thoughtful insights and feedback on my work. Finally, I would like to thank CPAC Systems AB, at which this thesis was carried out, for allowing me to use your resources and facilities.

Thesis advisors: Anna Samuelsson, and Gustaf Johansson, CPAC Systems AB
Thesis examiner: Peter Forsberg, Applied Artificial Intelligence

Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CoDeepNEAT	Coevolution Deep NEAT
DNN	Deep Neural Network
EXACT	Evolutionary Exploration of Augmenting Convolutional Topologies
FPT	Function Preserving Transformation
MLPNN	Multi Layered Perceptron Neural Network
NAS	Neural Architechture Search
NEAT	Neuroevolution of Augmenting Topologies
ReLU	Rectified Linear Units

Contents

1 Introduction

This chapter defines the scope of this thesis and provides a brief introduction to the interdisciplinary field called neuroevolution that uses evolutionary algorithms to configure Artificial Neural Networks (ANN).

1.1 Background

In deep learning the structure of the ANN determines its susceptibility to training and how well it is able to approximate the sought function or behavior. Difficult problems require custom made network architectures that has traditionally been tailored by researchers. The network design process includes an iterative trial and error procedure where the experimenter, guided by heuristics and experience, makes successive changes to the structure. Today, the current state of the art within deep learning was achieved through years of focused research and experiments. Naturally, automated ways for finding topologies have emerged and been successfully demonstrated on image recognition and captioning tasks [Rea+17].

Although experience, heuristics and transfer learning methods can be effective tools for certain problem domains they rarely generalize well to new areas. Manually designed network structures may also be unnecessarily large which is undesirable when running systems on embedded platforms where there are strict performance constraints. Automatized structure search mitigates both of these problems as, ideally, no previous experience is required and new effective architectures can be discovered. Ultimately, new knowledge can be gained from the structures found by the automated search.

A wide variety of approaches for automated topology search have been tried, ranging from grid search to statistical inference based methods. A proven and more intuitive approach is to base the structural search on evolution. Traditionally, evolutionary approaches have operated on both the structure and the parameters of the network. The idea to evolve neural networks were originally developed to solve reinforcement learning problems. These problems are often complex control tasks where the performance is determined by behaviors or strategies [Sta04]. In contrast to conventional reinforcement learning methods, evolutionary approaches need little or no information of what behaviors are required to solve the task. In principle, only a measure of the quality of the behavior is required to apply neuroevolution [Hau+13]. Thus, the relation between these types of problems and neuroevolution is natural since the outcome of an experiment can often easily be used as fitness measure of the network performance.

The last decades increased access to computational power allowed neuroevolution to be used for deep Neural Architechture Search (NAS). That meaning, for the supervised learning problem domain it became possible hybridize traditional neuroevolution with backpropagation. Optimizing the parameters of neural networks in supervised learning using evolution is not appropriate since gradient information is available in contrast to reinforcement learning.

Different approaches where neuroevolution based topology optimization has been combined with backpropagation have successfully provided state of the art architectures [Des17][Rea+17][Mii+17]. Typically, a population of neural network structures are trained through backpropagation and evaluated on the task. The fitness is assigned to the structure template. The best ranked architecthures are selected as templates to form the next generation and offsprings are produced by genetic crossing and random mutations.

The research has, so far, focused the attention on demonstrating neuroevolution as a competitive alternative to manually designed architectures. However, less attention has been given to developing methods for efficient neuroevolution which is the focus of this report.

In previous work, the structure and parameters are often treated separately. Although some approaches use parameter inheritance with the intentions that it could increase efficiency or accuracy, the results have not been unanimous [Des17][Rea+17][Mii+17]. Parameter inheritance refers to the concept that the parameters of the parents is transferred to the offspring instead of retraining the networks from a randomly initialized state. Inheritance can be viewed as a form of pretraining or transfer learning. Still, the evolutionary operators - how the structure is built - employed in these approaches have not treated structure and parameters in unison. Efficiency gains are expected if the evolutionary operators would consider the learnt experience of the networks.

1.2 Project description, scope and delimitations

The goal of the project is to develop efficient neuroevolutionary strategies. Specifically, this thesis examines if building structure through function preserving transformations is an opportune strategy for efficient neuroevolution.

The neuroevolutionary method put forward in this report does not include recombination or speciation. Crossing and speciation of ANN in a sensible way can only be achieved if certain structural growth restrictions are enforced. The neuroevolutionary algorithm put forward in this report does not constrain to these rules, crossover and speciations are therefore excluded. Sections 3.2 and 3.3 elaborates on these issues.

While most modern neuroevolutionary research has focused attention on building Convolutional Neural Networks (CNN) for image processing applications, the networks evolved in this thesis consist of fully connected perceptron layers. This delimitation was made since the computational resources required to evolve CNN were not available. Using evolution for neural architecture search of CNN requires enormous amount of computational power.

The application of the developed method is only intended for supervised learning problems without time dependence. Mainly because the method uses backpropagation to train each of the network structures.

1.3 Thesis outline

The thesis is divided into five chapters and follows a convential report outline consisting of: (1) Introduction, (2) Theory, (3) Method, (4) Evaluation and (5) Discussion. In the theory chapter the key concepts and previous work within the field of neuroevolution is described. The theory especially intends to emphasize the main problems in neuroevolution along with results and conclusions of previous research. The theory chapter lays a foundation that is required to understand the evolutionary algorithm and the design decisions presented in the method chapter. In the evaluation chapter, the neuroevolutionary algorithm is assessed through ablation in which the inherent function preserving mechanism of the mutations are disabled. The discussion interprets the results and puts forward ideas that could improve the algorithm.

2 Theory and related work

This chapter is devoted to explain the theoretical cornerstones that the algorithm in this report is based on. First, a description of the issue regarding crossover in neuroevolution is provided. Generally, crossing two ANNs is troublesome as the functionality of networks depend on an intracte relation between the parameters of the networks in a non-linear way. Next, a description of the neuroevolutionary method NEAT is introduced, as its structural growth principle provides a way to cross ANN. The ideas developed in NEAT also serve as a theoretical foundation in which neuroevolution can be discussed. The following section describe different approaches that combine evolutionary algorithms with backpropagation to serach for CNN architechtures. The methods utilize mutations and crossover that operate on the structure of the CNN while the parameters are update using backpropagation. This chapter ends with an introduction to a set of network morphisms that allows a network to be transformed to a larger network structure while preserving the functionality.

2.1 Crossover and the problem of competing conventions

Genetic, or evolutionary, algorithms are optimization schemes guided by the observed principles governing evolution in nature. In a sense, natural evolution is a form of optimization but without predefined goals [Wah08]. The mechanisms causing evolution are natural selection and random genetic variations. The most fit individuals of a population are more likely to survive and spread their genes. Through genetic recombination and random mutations new phenotypes appear. Genetic variations that are beneficial will improve an individual's chances to survive and reproduce. Mutations to the DNA allow new traits to emerge while recombination allows different traits in individuals to be combined in new ways [WT98].

Evolutionary algorithms conceptualize these mechanisms - selection, recombination and mutation - to solve optimization problems. In an evolutionary algorithm, a population of genetic encodings are held, where each individual correspond to an approximate solution to the optimization problem. Each individual, or solution, is evaluated on the problem and assigned a fitness determined by how well it solves the optimization problem. A selection is made, where fit individuals are selected for reproduction and through recombination and mutation the next generation is formed [Wah08].

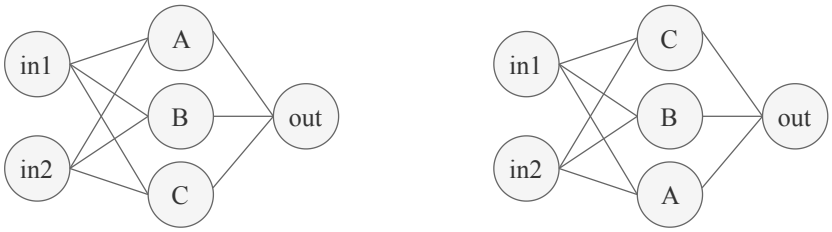

Figure 2.1: *An illustratation of the problem with competing conventions in neuroevolution. Both networks are identical in their functionality but are expressed in different ways. Recombining the networks can result in a network with a hidden layer with neurons (A, B, A) or (C, B, C). The possible offsprings are not likely to express a solution to the parameter optimization problem as both of them miss a third of the information of their parents.*

A problem in neuroevolution is how to recombine parameters of two networks due to the problem of competing convetions. The same solution, in terms of network functionality, to a parameter optimization problem can be expressed in a multitude of ways. Recombining the parameters of two solutions is not likely to produce an offspring with equal or better performance. [MD89][DR92]. To concretize, consider the two neural networks illustrated in Figure 2.1. The networks are identical in their functionality and are only permutations of each other. If both networks are solutions to a parameter optimization problem, recombining their parameters will most likely not produce a functional offspring. Crossing the networks visualized in Figure 2.1 will produce the damaged offsprings with hidden units (A, B, A) or (C, B, C). Both possible offsprings miss a third of the

information of their parents. In fact, for a single hidden layer with n neurons, there are $n!$ different encodings representing the same solution. In addition to the problem of competing conventions there may even be several characteristically different solutions to the same problem. Crossing such solutions will not produce a functional offspring [Sta04].

An even more difficult problem with crossover arise when the structure of the networks is subject to evolution. Two structurally different networks cannot generally be crossed in a way that represent a functional offspring. As an example consider the two networks illustrated in Figure 2.2. The two networks have different number of hidden layers and fusing the structures in a way that will capture the functionality of both parents is not plausible.

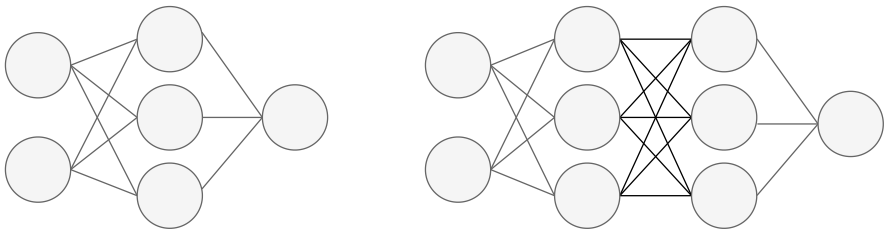

Figure 2.2: *The figure illustrates two different network topologies. Structural recombination of illustrated architechtures cannot be made in a principled way that is going to express a functional solution.*

2.2 NeuroEvolution of Augmenting Topologies - NEAT

NeuroEvolution of Augmenting Topologies (NEAT) is a neuroevolutionary algorithm for reinforcement learning. NEAT evolves both the topology and the parameters of ANN [Sta04]. In supervised learning, neuroevolution has been combined with backpropagation to search for network architechtures. The principles put forward in NEAT has been adopted and used in modern applications for neural architechture search, including the metod proposed in this work [Mii+17][Des17]. This section provide a brief overview of the principles in NEAT.

2.2.1 Genetic encoding and structural growth

The genetic encoding and the way structure is built in NEAT allow different network structures to be crossed in a principled way. The idea that allow NEAT to cross disparate topologies are historical markings encoded in the network representations. Whenever new structure is added it is also assigned a unique number. The numbers serve as indicators of historical origin of the different genes and is used to align the genome and in turn allow principled crossover [Sta04].

NEAT use a direct encoding scheme designed for genes to be easily lined up during crossover. The genome in NEAT contain two lists, one wich specifies what neurons that are present and the other is a list of connections. A connection gene specify an input-output node pair, a disable bit that determines whether the connection should be expressed, the weight of the connection and also an innovation number [Sta04].

As NEAT evolves both the structure of the network and the parameters the mutations include both structural changes and parameter perturbations. A parameter is perturbed probabilistically in each generation. The structural mutations come in two forms, *add connection* and *add node*. The mutations are illustrated in Figure 2.3. Any structural mutation is expressed by adding genes to the genome. When a new connection is created two previously unconnected nodes are chosen and a new gene is added to the connection genome along with a random weight. A new node is added by splitting a connection in two. A connection is selected at random, the connection is split by connecting the new node to the succeding and preceding nodes of the selected connection. The previous connection is then disabled. The weight associated with the incoming connection is given a value of 1 and the outgoing is given the same value as the previous connection. In this way, the new split edge will approximate the functionality of the previous connection [Sta04].

Whenever a structural mutation occur in NEAT new genes are created. Each new structural change is assigned an unique innovation number. The innovation number marks a chronology of when a gene was first

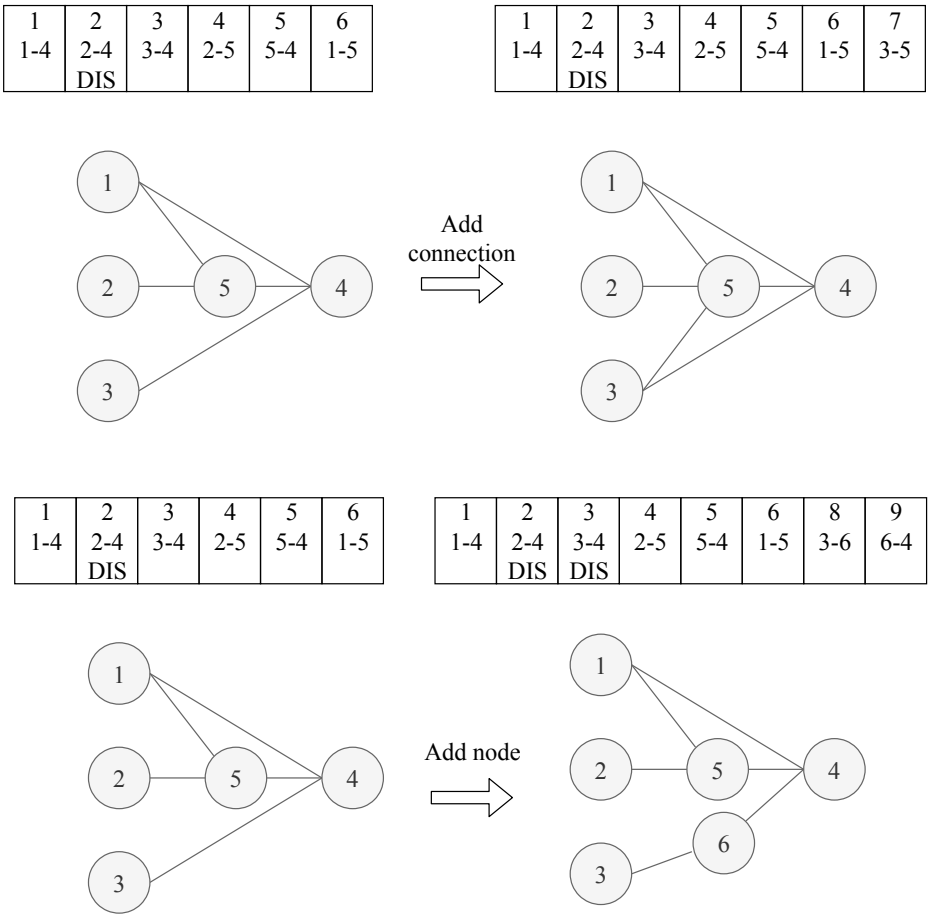

Figure 2.3: *An illustration of the two structural mutations used in NEAT. The lists above each network is the connection genome. The top number in each gene describe the innovation number. In this illustration the weights of the connections are not present. The top figure shows adding a connection. A new connection is created between node 3 and 5. It is created by adding a single gene to the connection genome. The bottom panel illustrates the add node mutation. A new node is added to the connection between node 3 and 4. The original connection is disabled and node 3 is connected to the new node with index 6. Node 6 in turn is connected to node 4. The two new connections are added to the connection genome along with corresponding innovation numbers.*

expressed and is used to align genomes during crossover. Genes in different networks but with the same innovation number express the same network structure. The innovation numbers in NEAT makes it easy to align and cross completely different structures. In NEAT during crossover, the genomes of two structures are lined up using the innovation numbers, see Figure 2.4. Genes that are present in both networks are called matching genes and are selected at random from either parent. Genes that are only present in either of the parents are always selected from the more fit parent. If both parents are equally fit, the non matching genes are selected at random[Sta04].

2.2.2 Speciation

In NEAT new topologies are built through mutation and recombination. Initially, new structures perform poorly before the parameters have had a chance to optimize. The initial decrease in performance will most likely cause the new individuals to die. To protect new individuals speciation is used in NEAT. A subdivision of the population allows each individual to only compete with similar individuals instead of the entire population. To organize the population in different species NEAT utilize a distance metric that determines how similar two

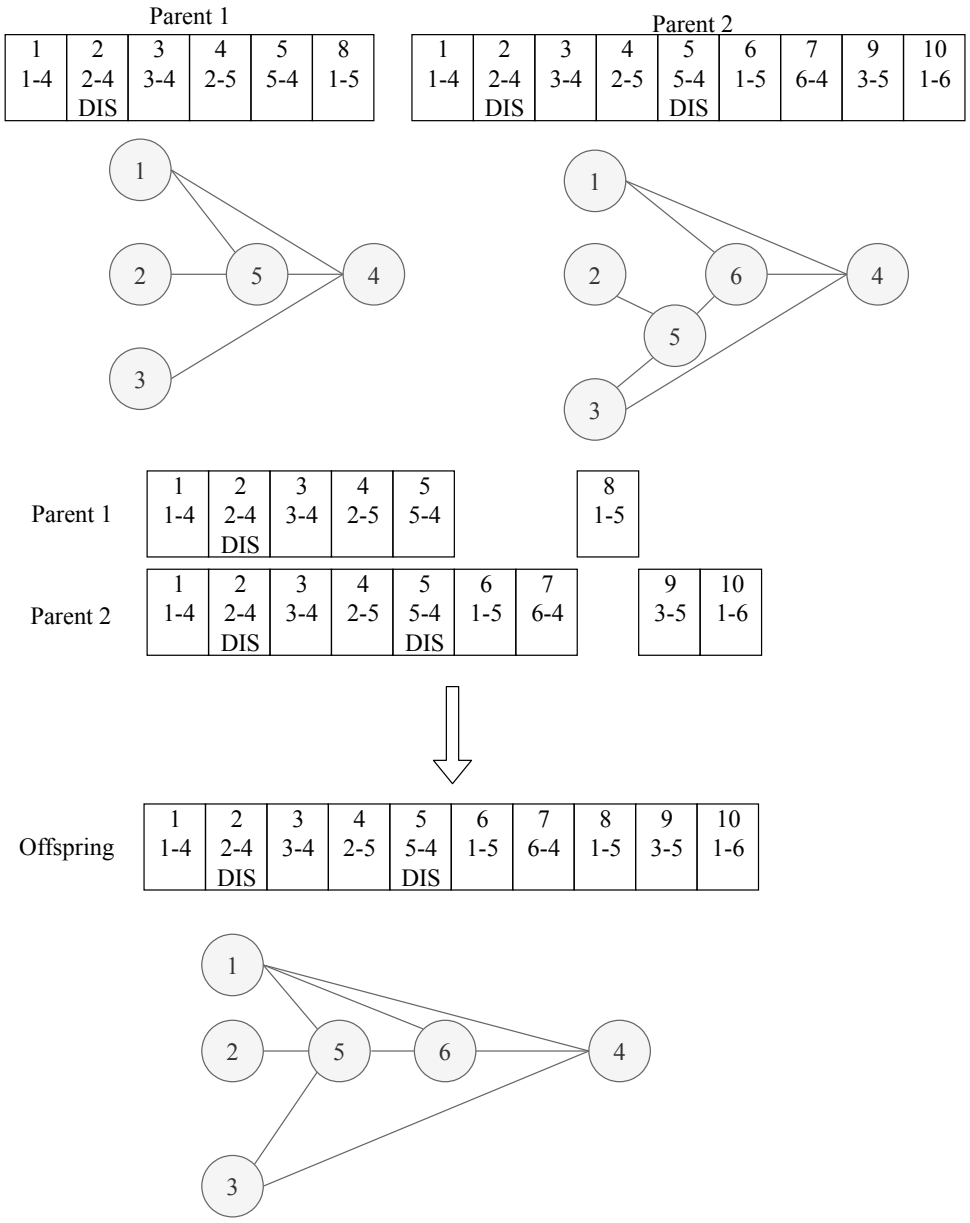

Figure 2.4: *An illustration of how crossover is performed in NEAT. First the connection genomes are lined up. The innovation numbers are used to match structure present in both networks. In this illustration both networks were equally fit and therefore the non-matching genes were selected at random from either parent.*

individuals are. The distance, or the compatibility metric, utilize the innovation numbers and is defined as

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \bar{W} \tag{2.1}$$

where E is the number of excess genes, D the number of disjoint genes and $\bar{W}$ the average difference in weights of matching connections, and N is the total number of genes. The hyperparameters c_1, c_2 and c_3 are used to weight the significance of E, D and $\bar{W}$ on the similarity [Sta04].

The distance metric determine the similarity between two networks, and is used in NEAT to speciate the population. Conceptually, if the distance between two networks are smaller then a userdefined threshold the two

networks are separated in different species. In the first generation all networks are placed in the same species. In succeding generations, the networks are placed sequentially into different species by comparing the distance between networks to the threshold. From each species in the previous generation, a random network is selected as a representative for the species in the current generation. The distance is calculated to the represantatives and the networks are assigned to species. If a network is not compatible with any of the existing species, a new one is created with the network as representative of that species [Sta04].

To prevent one species from taking over the entire population, fitness sharing within a species is used. The average fitness, F_k, of each species is calculated. A species is allowed to reproduce proportionally to its net contribution to the fitness of the entire population according to

$$n_k = \frac{F_k}{\sum F_k} P, \tag{2.2}$$

where n_k is the number of individuals assigned to the $k : th$ species and P is the population size [Sta04]. During reproduction, the fraction of the lowest performing individuals of each species is removed. Two individuals are selected at random for reproduction. The selected parents are crossed and the offsprings are mutated. This is repeated until the species has filled its quota. The best performing individuals of each species are also carried over unaltered to the next generation [Sta04].

2.3 Neural Architecture Search using Evolutionary Algorithms

This section describe different evolutionary approaches for automated Neural Architecture Search (NAS).

2.3.1 Extending NEAT for deep learning NAS - CoDeepNEAT

CoDeepNEAT is a coevolutionary version of a method called DeepNEAT which in turn is an extension of NEAT for NAS [Mii+17]. Coevolution means that the constituents of the solution are evolved independently but the performance of the individuals are measured in relation to how it performs in the full solution when assembled with other individuals [DS01][PD94].

DeepNEAT follows the same principles as NEAT but with the difference that each node in the graph now defines a layer in a Deep Neural Network (DNN) instead of single neuron. The node contains a set of hyperparameters that define the type of layer, for example convolutional or a fully connected layer along with mutable hyperparameters such as the number of filters, kernel size or number of neurons. The edges of a DeepNEAT genome do not encode weights contrary to NEAT but only the connectivity of the neural network. The algorithm starts out with a structurally uniform population that fulfills the input and output constraints. Historical markings are used to perform crossover by aligning similar structures. In the same way as in NEAT, the historical markings are used to speciate the population to protect structural innovations [Mii+17].

During evaluation, the graph is traversed and a CNN is assembled according to the node specification. The algorithm then proceeds to train the network and evaluates how well the encoded phenotype performs. The performance measure is then converted to a fitness and assigned to the corresponding genotype [Mii+17].

The structures evolved by DeepNEAT are unprincipled and complex and do not resemble those of human design. Heuristically, good architectures are often composed by repeating modules. To force the evolved structures to be more similar to human design, a coevolutionary extension of DeepNEAT were made, called CoDeepNEAT [Mii+17].

In **CoDeepNEAT**, two populations are evoloved using the same methodology as in DeepNEAT. One population contains high level architectures - blueprints - similar to DeepNEAT. The second population contains modules, small DNNs. The nodes in the blueprint graph contain references to species within the module population. During fitness evaluation an architecture is assembled by traversing the blueprint graph and replacing the nodes with randomly chosen modules from the referenced species in module population. The assembled network is trained as before and evaluated but the fitness of each component, blueprint or module, is the averaged fitness of all assembled networks it occurs in[Mii+17].

CoDeepNEAT was evaluated on the CIFAR-10 challange, with 25 blueprints and 45 submodules. In each generation, 100 full CNN were assembled and evaluated. Each assembled network was trained for eight epochs. After 72 generations the best networks were returned. These in turn, were trained for another 300 epochs and the accuracy was measured. CoDeepNEAT was able to find structures with an accuracy of 92.7% [Mii+17].

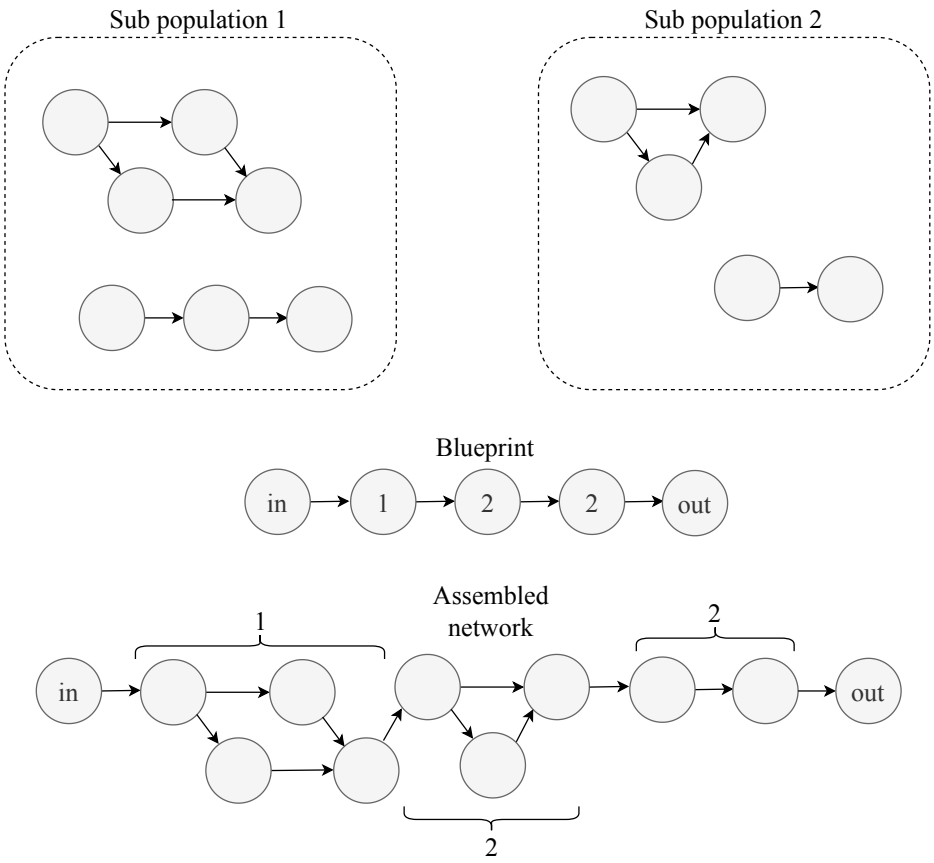

Figure 2.5: *The figure illustrates how modules and blueprints are assembled in CoDeepNEAT. The blueprint is a specification of what sub population to select the modules from.*

2.3.2 Large Scale Evolution of Image Classifiers - LSEIC

Another automated neuroevolutionary method for NAS was developed by Real *et al.* [Rea+17]. The method was not given a formal name but will in the context of this report be referred to as LSEIC, an abbreviation for Large Scale Evolution of Image Classifiers. While LSEIC combines evolution with backpropagation as in CoDeepNEAT, the way structure is evolved is subject to different mechanisms. CoDeepNEAT directly employs NEAT at large scales to develop deep networks [Mii+17]. LSEIC, on the other hand, builds structure solely through mutations, some reminiscent of the ones used in NEAT. In LSEIC, the learning rate is an individual specific mutable parameter[Rea+17].

The neural networks are represented as computational graphs. The edges of the graph represent transformations of how the tensors are mapped and propagated through the network. At the vertices, activation functions are applied. Two different activation functions are used, either batch normalization with Rectified Linear Unit (ReLU) or identity activation. The edges of the graph can either be a convolution or an identity mapping. By allowing identity edges, several activation functions can be sequentially applied without any intermediary convolutions. Similarly, several convolutions can process the data without activation. Results from experiments demonstrate that this allows novel and simple structures to emerge [Rea+17].

LSEIC starts out with a population of uniform structures with the input layer connected to a global pooling layer that maps the result to the output layer. Their implementation use workers that operates in parallel. In each step in the evolution a worker selects two random networks and removes the worst performing one from the population. The other one is selected to be the parent. A copy of the parent is made and a structural mutation operator is applied to generate an offspring. The mutation is randomly selected from a fix predetermined set. In total 11 different operators were used: (1) Mutating the learning rate. (2) No mutation, i.e the network is an identical copy of the parent. (3) The parameters of the network are reset to a random state. (4-5) A 2D-convolution is randomly inserted or removed. When a convolution is inserted, a vertex with an identity or

ReLU activation is also inserted with equal probability. (6) The stride of a convolutional kernel is altered. (7) The number of channels of a filter is mutated. (8) The kernel size of a convolution is randomly perturbated. (9) An identity connection is inserted between two vertices. (10-11) Add or remove a skip connection [Rea+17].

In a reproduction event the offspring is first created through any of the mutations specified above and then trained using backpropagation for a fixed number of iterations. In the work by Real *et al.* the number of training iterations was set to 25600 which are too few to fully train the architectures. Therefore, LSEIC employs parameter inheritance whenever possible. Some mutations can preserve all, for example a new identity mapping. Other transformation can preserve some parameters but not all, such as insertion of a new convolutional layer or changing the number of filters. Some mutations cannot preserve any parameters such as the weight resetting mutation [Rea+17].

Real *et al.* evaluated LSEIC on the CIFAR-10 and CIFAR-100 data sets. The best network, across five experiments, had an average accuracy of 94.1% on the CIFAR-10 data set. With the parameter inheritance mechanism disabled, the algorithm found networks with an accuracy of 92.2%. On the CIFAR-100 challange, the best network had an average accuracy of 77% [Rea+17].

The effect of the population size and number of training iterations in LESIC was examined by Real *et al.* The results show that increasing the population size increases the final accuracy of the best network in LSEIC. It was concluded that a larger population makes the population less likely to get temporary trapped in sub optimal solutions. Increasing the number of training iterations also increases the final accuracy of the best network. It was explained that fewer identity mutations were required to reach a higher level of training [Rea+17].

The motivation for employing the idea of parameter inheritance in LSEIC was to increase the efficiency of the evolutionary algorithm. In order for the algorithm to work it was required that the networks achieved a high level of training. To train a model from a randomly initialized parameter state in each reproduction event was believed to be too computationally expensive. Instead, parameter inheritance was used [Rea+17]. But, there is a drawback of parameter inheritance. Some individuals become more fit only because they had undergone more identity mutations and reached a higher level of training. In the long run, this can cause entrapment [Rea+17].

To escape entrapment a scheme was evaluated by Real *et al.* in which the mutation rate was dynamically changed whenever the population stagnated in order to encourage exploration. Instead of performing a single mutation during a reproduction event, five mutations were performed. The population was evolved with five mutations per reproduction event for a while and then switched back. Another scheme evaluated by Real *et al.* for escaping local entrapment was to randomize the parameters of all individuals. In that way, individuals that have become super fit beacuse they have undergone more identity mutation will loose its unfair advantage. The evaluations showed that both strategies were too inefficient to be practically used [Rea+17].

Further experiments by Real *et al.* evaluating crossover were carried out. In one version of LSEIC two network architechtures were fused in a reproduction event with the hope that the networks had learnt different features. In another experiment, a child was created by placing the parent structures parallell side by side. None of the recombination schemes improved the algorithm [Rea+17].

2.3.3 Evolutionary Exploration of Augmenting Convolutional Topologies – EXACT

Another neuroevolutionary method similar to both CoDeepNEAT and LSEIC is Evolutionary Exploration of Augmenting Convolutional Topologies (EXACT) developed by Desell. The method is based on the ideas of NEAT. As a result, EXACT is very similar to DeepNEAT with the difference that EXACT does not use speciation. The genome of the networks, similarly to LSEIC and CoDeepNEAT is represented with graphs that determine how different convolutional filters are connected [Des17].

EXACT uses a distributed setting where asynchronous workers train and evaluate network structures [Des17]. In a reproduction event, an offspring is either created through mutation or crossover. The genome is sent to a worker that train and evaluate the structure. If the new individual perform better then the least fit individual of the population, the master process replaces it with the offspring [Des17].

When a child is generated through mutation, it applies a user defined number of mutation operators to the structure. The operators are selected at random from a predetermined set with a user specified frequency. In total seven different mutations were used, four that operate on the high level connectivity, inspired by NEAT and three that operates on a low level and directly modify the convolutional filters. These are (1-4) Disable edge, enable edge, split edge and add edge. (5-7) Change both spatial dimensions of a convolutional filter,

change the filter size in the spatial x direction, and change spatial dimension of the y-direction [Des17].

Crossover is performed in a similar manner as in NEAT. Edges and nodes that exist in both parents are represented in the offspring. Two hyperparameters, *more fit crossover rate* and *less fit crossover rate*, determine the fate of disjoint structure only represented in one of the networks. The more fit parent's excess structure is represented in the offspring probabilistically determined by the *more fit crossover rate*. Similarly, the less fit parent excess structure is represented probabilistically by *less fit parent crossover rate*. Edges that are not copied from a parent to a child are still represented but set as disabled [Des17].

In the basic implementation of EXACT, parameter inheritance was not used. However, experiments with parameter inheritance were performed by Desell. The version using parameter inheritance had similar progression as the baseline, both in regards to finding CNN with high accuracy and the number of training epochs for a CNN to reach its minimal training error [Des17].

Desell evaluated EXACT on the MNIST data set and was able to attain structures with an accuracy of 97.89% with random parameter inheritance and 98.32% when parameter inheritance was used. The evolved networks diverged from typical stacked convolutional topologies [Des17].

2.3.4 The effects of parameter inheritance - Lamarckian evolution

The effects of parameter inheritance in NAS using neuroevolution were examined by Prellberg and Kramer. In these experiments the networks were restricted to stacked modules of convolutional filters with batch normalization and ReLU activation. For the output, a global average pool and a fully connected layer with softmax activation were used[PK18].

The evolutionary algorithm operates on the number of stacked modules along with the following defining parameters: kernel size, number of filters and stride. The algorithm furthermore use a population size of 1. In each generation an offspring is created by applying a structural mutation. The mutation operator are chosen with weighted probability from a predefined set consisting of the following mutation operators: add or remove a convolutional block at random, increase or decrease the number of filters in a convolution, change the kernel size or the stride of a filter. The offspring is trained for a fixed number epochs and then evaluated. If the offspring has a higher fitness, it will replace the parent network. To avoid premature convergence the offspring was probablistically explored even if they performed worse than the parent [PK18].

The algorithm tries to preserve as many parameters from the parent to the child network as possible. As an example, inserting a new convolutional block introduce new parameters that cannot be inherited. These parameters are randomly initialzied while the rest of the parameters of the ANN is reused [PK18].

The results from Prellberg and Kramer shows that parameter inheritance increase the data efficiency for challenging image data sets while no observed difference was found for the easier data sets. It is argued that the observed difference is because more challenging data sets require larger networks that in turn requires more training iterations to optimize [PK18].

2.4 Accelerated learning using function preserving transformations

The design process of a neural network often includes an iterative trial and error procedure where successively larger network topologies are tried, often based on how well previous models were able to perform. In an effort to reduce the training time required for each model a new set of transformation techniques was developed by Goodfellow *et al.* The strategies allow the knowledge learnt by a smaller network to be transferred to a larger one[PK18].

Conceptually, the method is based on structural transformations that preserve the functionality of the network. The method allows for a trained CNN to become both wider and deeper through so called Function Preserving Transformations (FPT). Two different function preserving transformations were presented: increasing the size of a CNN layer and increasing the depth of the network by inserting a specially configured convolutional filter [PK18].

A layer in a neural network can be expanded by replication of a neuron. By setting the incoming weights and bias of the new neuron to the same values as the template neuron the output of the two neurons will always be the same. If the weights associated with outoging connections are divided by two, the functionality of the network will not have changed. The other network transformation inserts a new layer between two already fully connected layers. If the incoming weight matrix is set to identity and the bias to zero, the output of the new layer will be similar as the previous layer [PK18].

3 Method

The approach for developing an efficient neuroevolutionary strategy is based on the idea that backpropagation in combination with NEAT is an efficient strategy. That meaning, instead of using an evolutionary scheme for both structure and parameters, the method presented in this report utilizes mutations to configure the structure while weights and biases are updated using backpropagation.In this chapter, this approach is presented, with focus on motivation of the different decisions made.

With increasing problem complexity, the minimal size of the network required to solve the problem increase. Therefore, high efficiency can be expected if the network structures are allowed to grow fast. In NEAT this can be achieved by increasing the mutation rate in each reproduction event. However, NEAT builds structure in an unorganized way not suited for fast backpropagation. To allow fast structural growth while enabling efficient backpropagation the method presented in this report builds structure by adding or altering fully connected perceptron layers. This way of building structure is analogue to CoDeepNEAT, LSEIC and EXACT but for Multi Layered Perceptron Neural Networks (MLPNN).

An intriguing idea for efficient neuroevolution would be to base the structural mutations on function preserving transformations, similar to the ones described in Section 2.4. If the functionality of a network can be preserved as the network undergoes a structural modification, the population would be able to accumulate experience across generations. If an offspring approximates its parent's functionality, a trained network, it is reasonable to believe that the offspring can reach a higher level of training fast as it starts out in region in parameter space of high performance. Function preserving mutations can be viewed as an extension to the idea of parameter inheritance.

To evaluate the ideas put forward a modular network representation is required. This is accomplished by representing the networks with acyclic directed computational graphs and is described in the next section.

3.1 Network encoding using acyclic directed graphs

A neural network is in this thesis represented with a graph that defines the computational flow. The edges of the graph represent affine transformations between two vertices. The affine mappings between the vertices include both a weight matrix multiplication and subtraction of bias. The vertices apply a non-linear activation to the incoming affine transformations. In a forward pass, the input tensor is loaded into the input vertex, which is without activation, and then propagated through the graph. The input to each node is the sum of the mappings from all directly connected preceding vertices. In Figure 3.1 a computational graph representing an ANN is visualized. In a forward pass the output of each layer is

$$
\begin{aligned}
&\mathbf{x}_{in} && \text{Network input}\\
&\mathbf{x}_{(1)} = \sigma\left(T\mathbf{x}_{in}\right) && \text{Output node 1}\\
&\mathbf{x}_{(2)} = \sigma\left(Z\mathbf{x}_{(1)}\right) && \text{Output node 2}\\
&\mathbf{x}_{(3)} = \sigma\left(R\mathbf{x}_{(1)} + Q\mathbf{x}_{(2)}\right) && \text{Output node 3}\\
&\mathbf{x}_{out} = s\left(W\mathbf{x}_{(3)}\right) && \text{Network output}
\end{aligned}
$$

where T, Z, R, Q and W are affine transformations that represent the edges of the graph. The non-linear activation function is denoted by σ. Based on the character of the problem a different activation function can be used for the output layer. Thus the activation for the output layer is denoted s.

The graph representation described above is a high level abstraction of the common low level synaptic layered neuron to neuron view. A vertex in the graph representation is equal to a layer of neurons. The affine transformations corresponds to the weights and biases used in the neuron to neuron view.Figure 3.2 illustrates the corresponding low-level view of the computational graph example presented in Figure 3.1. The computational graph representation was used as it enables dynamic modification of the network structure without advanced topological analysis.

3.2 Crossover

In evolutionary algorithms crossover is used to combine favourable features in different individuals in new ways. But, due to the problem of competing conventions it is difficult to recombine solutions in neuroevolution. In

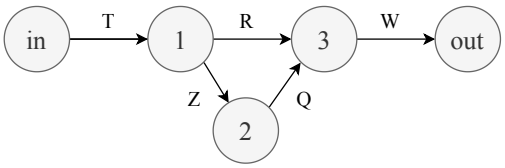

Figure 3.1: *The figure illustrates a computational graph used to represent a neural network. The nodes marked with in and out are the input respectively the output layer of the neural network. The edges represent affine transformations that map the values stored in one node to another. In a forward pass, the input is mapped from the input layer to node 1 using the affine transformation T followed by non-linear activation. Next, node 2 is activated by mapping the values at node 1 using transformation Z followed by non-linear activation. Node 3 is activated by adding the values obtained by mapping the values at vertex 1 and 2 with transformations R and Q and then followed by activation. The output is obtained by mapping the values of node 3 to the output vertex.*

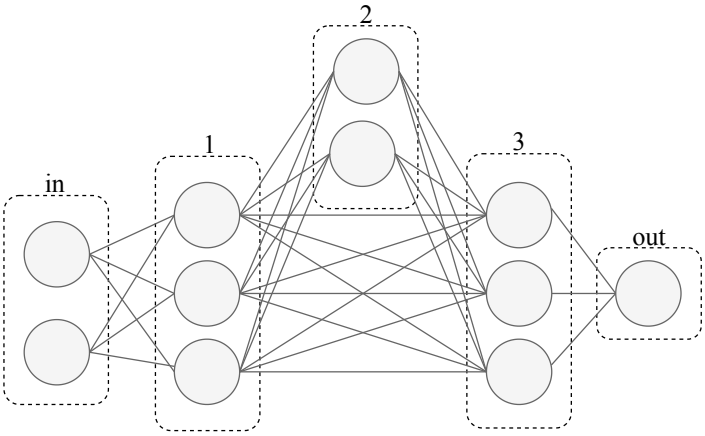

Figure 3.2: *The figure illustrates the neural network representation of the computational graph in Figure 3.1. In this figure the individual connections between the neurons in the layers are shown.*

NEAT, the problem of competing conventions is avoided by enforcing structural growth constraints.

The network encoding and how structure is built in this thesis is not compatible with crossover in a way that allows the offsprings to combine the functionality of their parents due to the problems with crossover in neuroevolution, see Section 2.1. If the offsprings cannot represent the functionality of their parents, it would defeat the purpose of using parameter inheritance and function preserving mutations. This perception is supported by the fact that epigenetic parameter initialization in EXACT, that used crossover, did not improve the performance of the algorithm. On the contrary, LSEIC that only built structure through mutations did benefit from parameter inheritance. In addition, versions of LSEIC using recombination were evaluated and the results showed that it did not improve the performance.

Another argument to not use crossover is that crossover appear to increase the complexity of the ANN. In both DeepNEAT and EXACT the evolved networks were unprincipled and complex. On the contrary, LSEIC that only built structure through mutations found simple solutions that were similar to human design.

In summary, the genetic encoding and the way structure is built is not compatible with organized crossover. In previous work crossover appear to increase the complexity of networks and reduce the efficiency of the evolutionary algorithm. Beacuse of the reasons stated, crossover is not included as a part of the algorithm presented in this report.

3.3 Speciation

In NEAT, speciation serves the function to protect structural innovation as it is unlikely that new structure will immedeatiley express useful function. Speciation is therefore necessary in NEAT to allow the structural innovations to become self-sustainable. In this work where the mutations preserve the functionality of the parent networks there is no need for speciation as the structural configurations of the offsprings already express

useful functionality. Through backpropagation, new structure is phased in.

More generally, speciation is used to prevent premature convergence by maintaining a genetically diverse population that explores different peaks of the solution space. Ideally, a speciation scheme could improve the efficiency of the evolutionary algorithm but the encoding and structural growth scheme does not allow an appropriate structure based similarity metric to be defined. To illustrate this issue, consider the three networks illustrated in Figure 3.3. The dark gray marked neurons are structure that is present in all three networks. Although they have common structure it is difficult to distinguish which of the three networks that are more structurally similar.

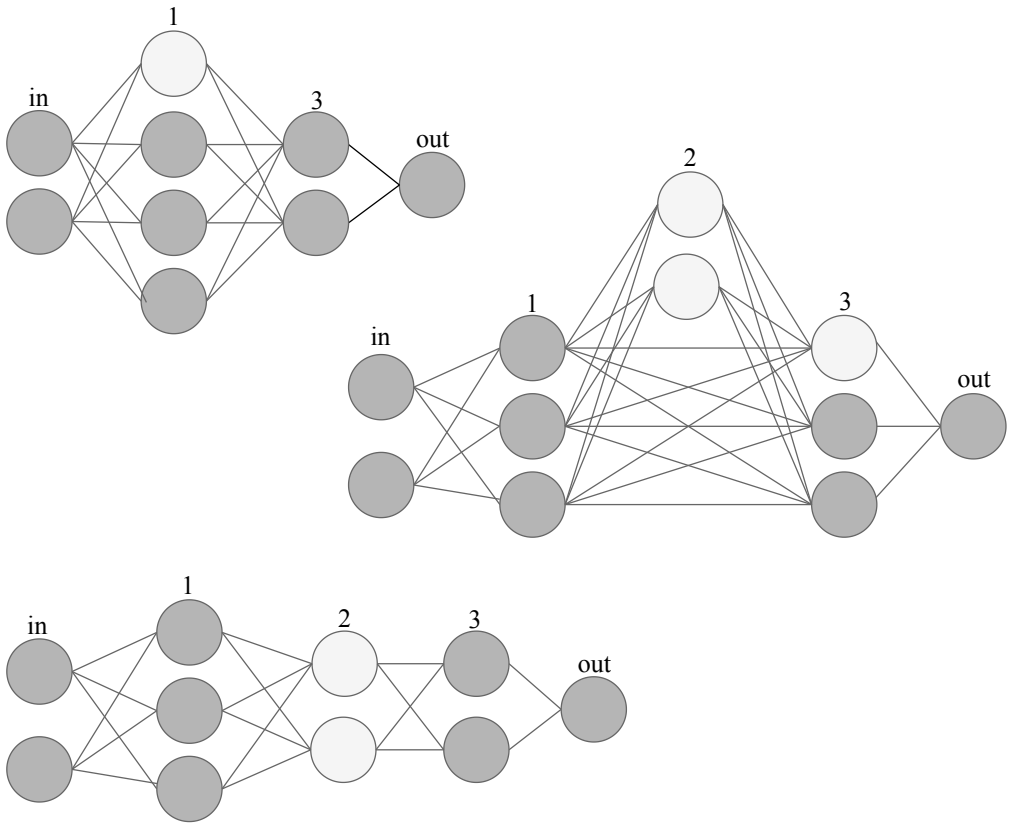

Figure 3.3: *The figure illustrates three different network topologies with some common structure. The gray marked neurons indicate structure that can be viewed as common in all three networks. Despite their common structure they are completely different.*

3.4 Selection and reproduction

This section aims to provide an overview of the evolutionary algorithm with special emphasis on the selection mechanism. In Figure 3.4 a flow chart of the evolutionary procedure is illustrated. A structurally uniform population is initialized. The structures are trained using back propagation and their performance evaluated. If the best ranked network performs better than the desired threshold the algorithm is terminated. If none of the networks have an accuracy that is good enough, the next generation is formed by selection and mutation of high ranking individuals.

The parent networks are selected through both elitism and tournament selection. First, the n best individuals are selected for reproduction and removed from the population, where n is determined by a hyperparameter.The remaining parents are selected using tournament selection. The tournament selection is performed by selecting two candidates from the population. The better candidate are chosen with probability determined by a user

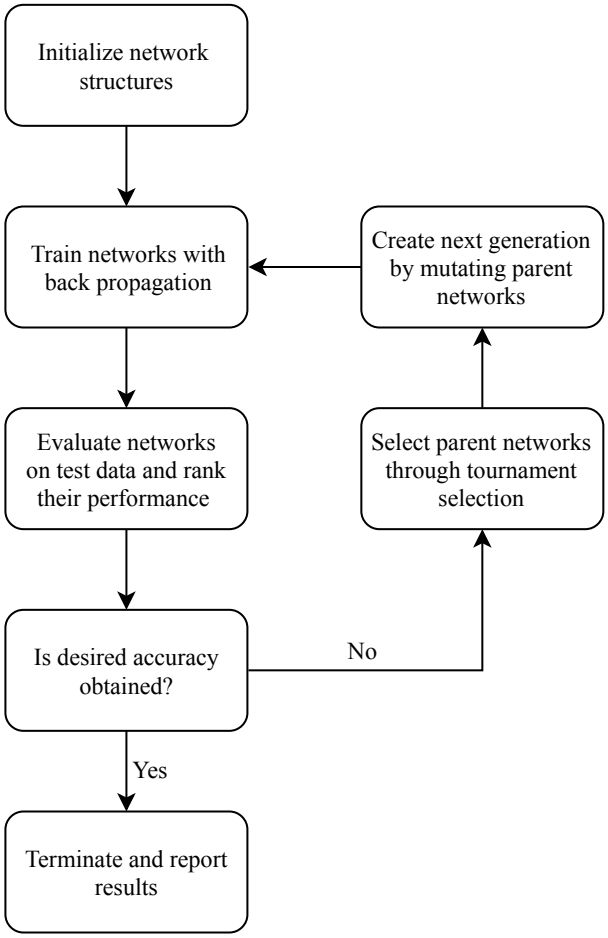

Figure 3.4: *The figure illustrate a flow chart of the evolutionary algorithm. The algorithm starts with a population initialization of neural networks. In each generation the network structures are trained with back propagation, evaluated and ranked. If the best individual performance exceeds a user defined threshold the algorithm is terminated. If not, the algorithm proceeds to create the next generation by first selecting the parent networks through tournament selection. The offsprings are created by altering the structure of the parent networks through a random selection of a mutation operator.*

defined value referred to as *tournament_parameter*. The winner is removed from the population to prevent it from being selected multiple times. The tournament selection is repeated until the total number of parents equal a user defined threshold.The remaining individuals of the population are discarded.

The offsprings are generated by applying function preserving structural mutations to copies of the parents. In a single reproduction event, a parent and a mutation operator is chosen at random. The operator is applied to the network to create the offspring. In Section 3.5 the mutation operators are presented in detail. The parents are transferred to the next generation unaltered and the reproduction procedure is repeated until the number of parents and offsprings equals the population size.

The selection and reproduction scheme was chosen to attain high exploration. In other selection schemes, each individual usually produce offsprings proportionally to its fitness relative to all the other individuals. In fitness proportional selection, the risk for premature convergence is palpable if one individual becomes considerably more fit. Tournament selection explores super-fit individuals to the same extent as any other parent. Furthermore, by using a small tournament size of only two individuals, weak candidates are more likely to survive selection. By using elite survivors the population will never decrease in performance with only a small cost to the level of exploration.

3.5 Structural mutation operators

The mutation operators were designed to allow any network structure to be morphed to any other network structure through a series of a repeated mutations. The mutations are inspired by how an experimenter would change the structure iteratively during a manual search: expand or shrink a neuron layer, increase or decrease the depth of the network by insertion or removal of a layer and add or remove skip connections. Skip connections were introduced to allow arbitrary structures to emerge.

The method used in this report includes mutations to remove structure. Structural reduction can have several benefits. It reduces the number of parameters to update during back propagation. It allows the evolutionary process to search among smaller topologies. Smaller topologies train faster than a larger ones and may even be less prone to get stuck in stationary regions. Small structures are also desirable for integration in embedded systems as they can execute a forward pass faster.

Another consideration during the design of the mutation operators was that they should allow rapid structural change. By increasing or decrasing the size of a layer proportional to its original size it allows the layer to reach an optimal size more rapidly compared to constant rate of change. When a layer is inserted it is given the same size as the preceding layer to obtain a faster growth rate compared to inserting layers of fixed size.

A mutation operator is selected at random, in some cases the chosen operator may not be compatible with the structure. For example, if the parent is a network without any hidden layers, then adding a skip connection is not possible nor meaningful. In such scenarios, a new mutation is selected at random until a compatible mutation operator is found. In the succeeding sections the principles of the mutation operators are described.

3.5.1 Network layer expansion through neuron functionality multiplication

The layer expanding mutation is based on the work by Goodfellow *et al.* described in Section 2.4. In the context of this report the layer expanding mutation will be referred to as neuron multiplication.

Expanding a layer can be accomplished by changing the transformations that map to and from the computational vertex. Increasing the size of a layer layer can be accomplished by adding rows respectively columns to the weight matrices associated with the incoming and outgoing edges. The bias vector of the transformation that precedes the vertex must also be extended to the same extent as the number of neurons that are added. If there are several preceding and succeeding vertices, the explained procedure needs to be repeated for each edge.

The functionality of a single neuron is recreated in new neurons added to the layer. To concretize, consider the procedure of increasing the size of a layer with one new neuron. Figure 3.5 illustrates the functionality of neuron beeing duplicated. The weights of the incoming connections are directly copied from the template neuron to the new neuron. The bias is also copied to the new neuron as it is. For any input that is passed through the network, the values in these two neurons will always be the same. To compensate for this duplication, the weights of the outgoing connection are divided by two to compensate for both the template and the new neuron.

If the incoming and outgoing weights of two neurons are identical, they will change exactly in the same way during back propagation. To allow the parameters associated with the new neurons to diverge during backpropagation the weights associated with the new neurons are perturbated according to

$$w_{new} = w_{old}(1 - \Sigma) \text{ where } \Sigma \sim N(\epsilon, \epsilon^2) \tag{3.1}$$

where ϵ is a hyperparameter that determines the size of the perturbations. The size of the perturbations scale proportionally to the size of the weights. In that way large variations between the different weigths are accounted for. For $\epsilon = 0$ the functionality of the new neurons are identical to the originals and will never diverge. For $\epsilon = 1.0$, the new weights are randomly sampled from a normal distribution and scaled in proportion to the original weights.

When the layer expanding mutation operator is chosen, a hidden layer is selected at random. Half of the neurons in the layer is chosen at random for duplication.

3.5.2 Network layer insertion through identity matrix weight initialization

In the split edge mutation in NEAT the weight of the outgoing connection is set equal to the weight of the original connection. The incoming weight of the new connection is set to 1. In that way the new structure

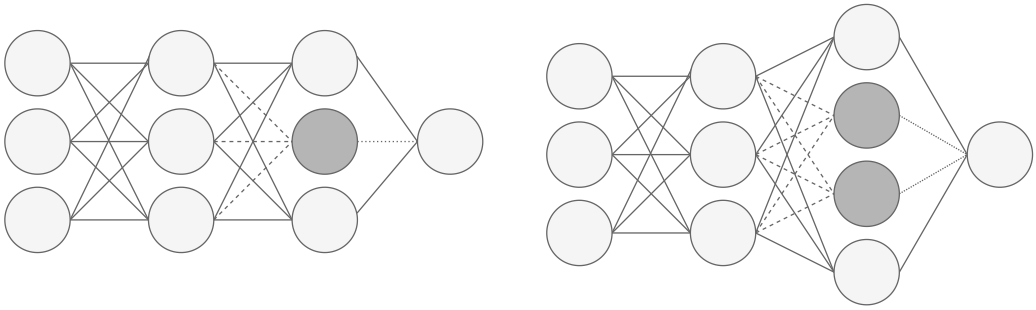

Figure 3.5: *An illustratation of the layer expansion transformation. The left and right panels show the network before and respectively after the transformation. The dark grey colored neuron is selected and duplicated. The incoming weights and bias, associated with the template neuron are directly copied to the new neuron, marked with dashed lines. The weights associated with the outgoing connections are set to half of the original values and are illustrated with the dotted lines in the right panel.*

approximates the function of the original connection, disregarding the non-linear activation. Conceptually, the idea can be adopted to insertion of an entire perceptron layer. This procedure is illustrated in Figure 3.6. The weight matrix of the incoming connection is set to identity. In that way the new layer will output values close to the original layer. By setting the weight matrix of the outgoing edge identical to the original connection, the effect of the new layer is small with regards to the functionality of the network.

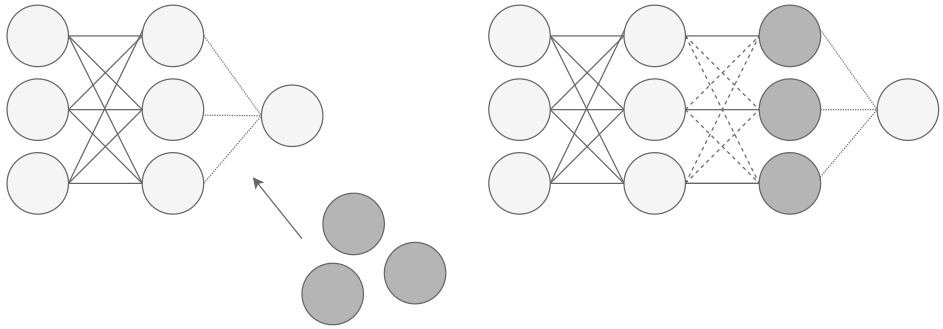

Figure 3.6: *The figure illustrates network made deeper by layer insertion. The weights and bias associated with the outoging transformation is directly copied and marked with dotted lines. The incoming connection to the new layer is the identity matrix with a zero valued bias vector.*

In reality, inserting a layer between two connected nodes in the manner described above will most likely cause a significant decrease in performance due to the non-linear activation function. To mitigate the non-linearity of the newly inserted vertex the sub network consisting of the preceding, succeeding and the new inserted vertex is trained to behave as the original transformation. During a single training iteration a mini batch is propagated through the unaltered network. The input and output of the layers preceding and succeeding the edge that was selected for layer insertion are stored and used as trainig data for the sub network. The details regarding this conceptual procedure is described further in Section 3.5.6, but where the only difference is that a layer is removed instead of inserted.

3.5.3 Add skip connection

The structural transformations described so far allow networks to become both wider and deeper, but confines them to a stacked layers topology. Each layer of neurons has one preceding and one succeding layer, except the input and output layer. To allow more general solutions to be found, new connections between layers that

originally were not connected must have the option to emerge. However, new connections will not preserve the functionality of the network unless the combination of weigths and biases cancel out and create a net contribution of zero as output – input to the next layer. For arbitrary inputs, the only parameter combination that meet such requirements is when all parameters are zero. However, experimentation showed that new edges can be adapted faster and configured as useful structure if the weights were not set to zero. Therefore, when a new edge is created the initialization of weights are sampled according to

$$w \sim U(-\zeta\sqrt{k}, \zeta\sqrt{k}) \quad \text{with} \quad k = 1/n, \tag{3.2}$$

where n is the preceding layer size and ζ a hyperparameter that is used to control how much the functionality of the networks are allowed to change.

3.5.4 Network layer contraction through pruning

It is generally impossible to preserve the functionality while shrinking a layer, but the impact can be reduced by a smart selection of neurons to remove. When a network is evolved and trained, some neurons become less important or even obsolete to the overall functionality. This concept is utilized to decrease the size of a layer. This procedure can be viewed as form of pruning, and will for simplicity be referred to as layer pruning.

A hidden layer is selected at random for reduction. The pruning is performed by passing through a subsample of the training data set while storing the activation tensor associated with the selected layer. Each row in the matrix contains the activation values for i:th neuron in the layer. The matrix is used to rank the neurons by their relative activation level. Below, a numerical example is presented. The layer consists of three neurons and a subsample of 4 was used. As 3 is not divisible by 2, only the neuron with the lowest activation will be removed.

$$\begin{bmatrix} 0.92 & 0.87 & -0.21 & 0.34 \\ 0.07 & -0.03 & 0.04 & 0.02 \\ -0.65 & -0.95 & 0.04 & 0.63 \end{bmatrix} \xrightarrow{\text{abs}} \begin{bmatrix} 0.92 & 0.87 & 0.21 & 0.34 \\ 0.07 & 0.03 & 0.04 & 0.02 \\ 0.65 & 0.95 & 0.54 & 0.63 \end{bmatrix}$$

$$\xrightarrow{\text{mean}} \begin{bmatrix} 0.585 \\ 0.040 \\ 0.693 \end{bmatrix} \xrightarrow{\text{ranking}} \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

The absolute value of each element in the matrix is calculated and the mean of the rows is computed. The matrix from these computations contain information on the average activation level of each neuron. The neurons are ranked and the 50% lowest scoring ones are deleted, in the numerical example above the second neuron. In practice this means that incoming and outgoing edges to the chosen vertex are modified. Incoming transformations are modified by removing rows from the weight matrix and elements from the bias vector. For the outgoing edges columns of the weight matrices are removed.

3.5.5 Remove edge

All connections in the network cannot be removed. Removing the wrong edge can cause some layers to not be connected to a preceding nor a succeding layer. Such structures are not meaningful and therefore special care was made to prevent this. An edge is allowed to be removed if both the succeding and preceding vertices of the edge are connected to other vertices. To reduce the distortion of the network functionality the edge associated with the smallest loss is removed. First, all edges that can be removed are listed. One edge at the time is disabled and a forward pass of a subsample of the training data is made and the loss is stored. All losses are compared, and the connection corresponding to the smallest loss is removed in the mutatation.

3.5.6 Network layer deletion

To delete a layer without reducing the performance is not possible but the damage can be mitigated by replacing two layers in sequence with a wider layer that approximates the function of the two layers. The scheme is illustrated in Figure 3.7. A training batch is propagated through the network. The input and output tensors

from the sub network consisting of the two layers, C and D in the figure, are stored and used to train the wider layer. The inputs are fed to the smaller network with and the output is used to backpropagate.

The size of the wider layer is set so that the number of parameters of the new structure equals the number of parameters of the orignal two layers. The layer size can be calculated with

$$N_{new} = \frac{(N_B + 1)N_C + (N_C + 1)N_D + N_D N_E}{N_B + N_E + 1}, \tag{3.3}$$

where N_B, N_C, N_D and N_E is the size of the layers, see Figure 3.7. The weights of the incoming and outgoing connections are uniformly sampled according to

$$w_{in} \sim U(-\sqrt{1/N_B}, \sqrt{1/N_B}), \tag{3.4}$$

$$w_{out} \sim U(-\sqrt{1/N_{new}}, \sqrt{1/N_{new}}). \tag{3.5}$$

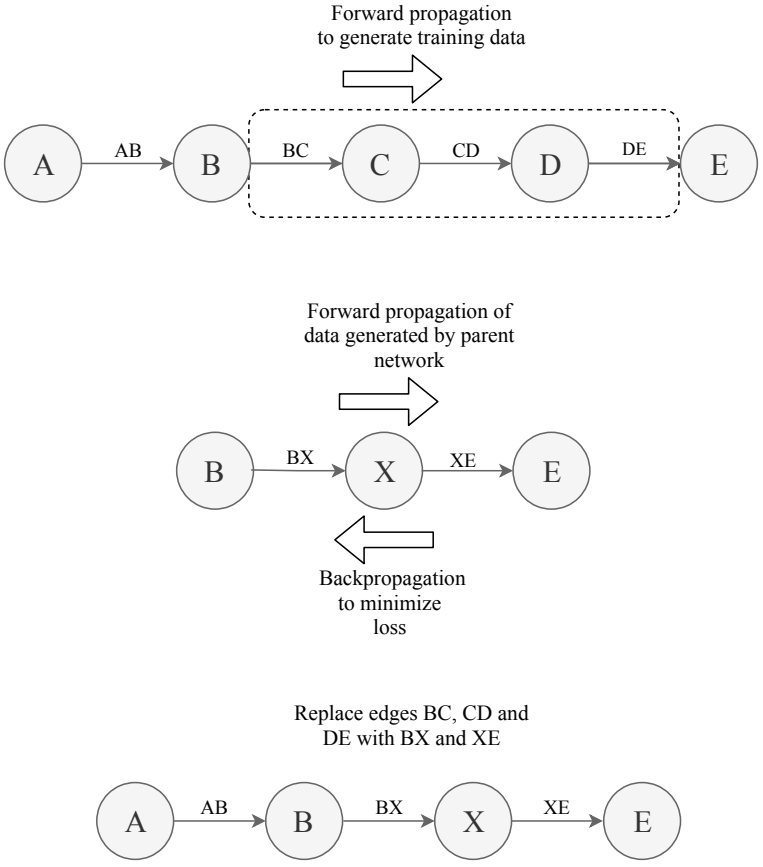

Figure 3.7: *The figure illustrates the procedure used to decrease the depth of a network with minimal distortion to its functionality. The procedure replaces two consecutive hidden layers with a single, wider, layer. The boxed part of the network on the top is the part that will be replaced by a smaller network. The procedure is carried out by training the smaller network, seen in the middle, to approximate the function of the boxed sub network. In one training iteration, a mini batch is propagated through the top network. The output tensor of layer B is used as input to the smaller network and the input tensor to layer E is used as target tensor to define the loss. Once the middle network has trained it replaces the boxed sub network of the orignal network.*

4 Evaluation

In this chapter the strategy to build structure through function preserving transformations is evaluated with regards to efficiency. The chapter is divided in two parts. The first part describe experimental setup, that is how this strategy was evaluated along with the details governing the experiments. The second part present the results of the experiments.

4.1 Experimental setup

To evaluate the hypothesis that function preserving mutations constitutes an efficent strategy for neuroevolution, the function preserving algorithm is compared to an ablated version in which the preservating properties of the mutations are disabled. The ablated version is created by adjusting the hyperparameters so that the parameters associated with the mutation become random. For some mutations, an additional level of noise is added to randomize new structure. Below the difference between the baseline, i.e the function preserving algorithm, and the ablated version is explained for each of the mutation operators.

- The function preserving property of the layer expanding mutation can be removed by increasing the size of perturbations determined by ϵ, see Equation (3.1). By increasing ϵ, the weights associated with the new neurons diverge from their duplicates. For small values of $\epsilon \approx 0$, new neurons are simply copies of other neurons present in the layer before the expansion. For $\epsilon = 1.0$, the neuron multiplication is reduced to random padding. For all the experiments, the baseline use $\epsilon = 0.1$ while the ablated version is represented with $\epsilon = 1.0$.

- In the baseline a new layer is inserted so that weights of the incoming connection are set to identity, and outgoing weights are equal to those of the original connection. To account for non-linear activation, the sub network consisting of the in- and outgoing edges of the new layer is trained through backpropagation to represent the functionality of the orignal connection. The function preserving property can be disabled by adding a level of noise to weights associated with the incoming connections instead of retraining the sub structure. In the experiments, the noise is uniformly sampled on the interval $(-1, 1)$. In the baseline, no noise is added and 500 retraining iterations are used.

- The function preserving property of the new edge mutations is based on the idea that a new edge can be added without distorting the functionality too much if the parameters associated with the edge are small. Different values of ζ, see Equation (3.2), are used represent the baseline respectively the ablated version. In the baseline $\zeta = 0.1$ and in the ablated version $\zeta = 1.0$.

- The layer reducing mutation is based on pruning. The neurons with the average lowest level of activation are removed in the baseline. In the ablated version the neurons are selected at random instead.

- In the baseline the depth of the neural network is reduced by replacing two consecutive layers with a single wider one. The sub network consisting of the wider layer is retrained using backpropagation to represent the function of the two original layers. In the ablated version the number of retraining iterations are set to zero, in that way the new structure will be randomly initialized. In the baseline representation 500 retraining iterations are used.

- The remove edge mutation removes the edge that is associated with the smallest loss in the baseline version. In the ablated version, an edge is selected at random instead.

4.1.1 Benchmark problems and experiment details

The relative performance of the two versions is tested on a set of 5 regression problems, defined by

$$p_1(x_1, ..., x_3) = \sin(2\pi(x_1 + x_2 + x_3)) \tag{4.1}$$
$$p_2(x_1, ..., x_6) = \sin(2\pi(x_1 + x_2 + x_3 + x_4 + x_5 + x_6)) \tag{4.2}$$
$$p_3(x_1, ..., x_9) = \sin(2\pi(x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9)) \tag{4.3}$$
$$p_4(x_1, ..., x_9) = \frac{\sin(2\pi(x_1 + x_2 + x_3)) + \sin(2\pi(x_4 + x_5 + x_6)) + \sin(2\pi(x_7 + x_8 + x_9))}{3} \tag{4.4}$$
$$p_5(x_1, ..., x_9) = \frac{\sin(3\pi(x_1 + x_2 + x_3)) + \sin(3\pi(x_4 + x_5 + x_6)) + \sin(3\pi(x_7 + x_8 + x_9))}{3}. \tag{4.5}$$

For each of the five functions p_1 to p_5 the goal is for the algorithms to approximate them as good as possible. The benchmark functions were chosen to represent different levels of complexity with highly alternating function surfaces. The data used constitutes of 11000 data points uniformly sampled from functions (4.1) - (4.5) where $x_i \in (0, 1) \ \ \forall i$. The data is divided in a training and a validation set of 10000 respectively 1000 data points. The loss on the validation set is used for selection. Before training, the data is scaled to unit variance and zero mean.

For the hidden layers tanh was used as activation while identity activation was used for the output layer. In the following experiments, a solution is found if a network has a Mean Squared Error loss (MSE-loss) lower then 0.01 on the validation data. In all experiments a population size of 30 individuals was used, with a total of five parents whereof one elite survivor. For the tournament selection, the *tournament_parameter* was set to 0.8. The individuals of the population is intialized with a hidden layer of size two. During reproduction all mutations are selected with equal probability.

For the training the gradient descent method ADAM was used with a learning rate of 6×10^{-3}, and $\beta = (0.9, 0.999)$, $\epsilon = 1 \times 10^{-8}$ and with weight decay of 0. A mini-batch size of 100 was used.

4.2 Results

The purpose of this experiment was to determine whether function preserving mutations is an efficient strategy for neuroevolution. In Table 4.1 the average number of generations required to find a solution is presented for the baseline and ablated versions. Different variations of the number of training iterations per individual is shown. For problems 2, 3, and 5 the performance is significantly higher for the ablated version. For problem 1 and 4, the two algorithms show similar performance, possibly in favour of the baseline.

It can be concluded that the efficiency is related to the parameter initializaion associated with the structural mutation. For problem 2 and 3 the average number of generations required to find a solution is 2.4 times fewer for the ablated version than the baseline.

The average number of generations required to find a solution does not reveal qualitative information about the shape of the distribution. Potential skewness in the distribution or thick tails may cause the mean to be misguiding. The distribution of solutions for problem 2 and 3, was chosen for further examination as the results differ between them. Problem 5 was not examined as the function preserving version did not find any solutions. In Figures 4.1 and 4.2 the solution distributions for problem 2 respectively 3 is presented. A curve represent the number of solutions found as a function of the generations passed. Each curve can be interpreted as an estimate of the cumulative distribution function (CDF) that a solution has been found after n generations. The black colored curves correspond to the baseline and the gray colored curves with the ablated method. The different line types mark different number of training iterations per individual and generation.

The distributions show both that the minimum number of generations to find a solution is higher for the baseline and also that on average the rate at which solutions are found is lower. The effect of increasing training iterations seem to be a more rapidly increasing CDF. This is expected as fewer generations are required to attain the same level of training compared to if the number of training iterations per individual and generation would have been lower.

The results provide evidence against the hypothesis that function preserving mutations is an efficient strategy for scalable neuroevolution as the baseline require more generations to find a solution in comparison to the ablated version. To conclude why the baseline demonstrates poor performance, the progression of the

Table 4.1: The table show the average number of generations required to find a solution on the five problems for different number of backpropagation iterations.

Nr of backpropagation iterations	**Avg nr of generations: Baseline**	**Avg nr of generations: Ablation**
Problem 1		
50	20.0	18.4
150	9.95	9.55
300	10.0	6.6
Problem 2		
250	27.9	13.2
500	23.6 (90% successrate)	10.0
1000	17.6	6.8
Problem 3		
2000	28.5	11.2
5000	16.3	6.85
Problem 4		
300	15.0	15.7
600	9.9	10.7
1200	7.3	7.5
Problem 5		
5000	N/A, no solution found	17.6 (90 % successerate)

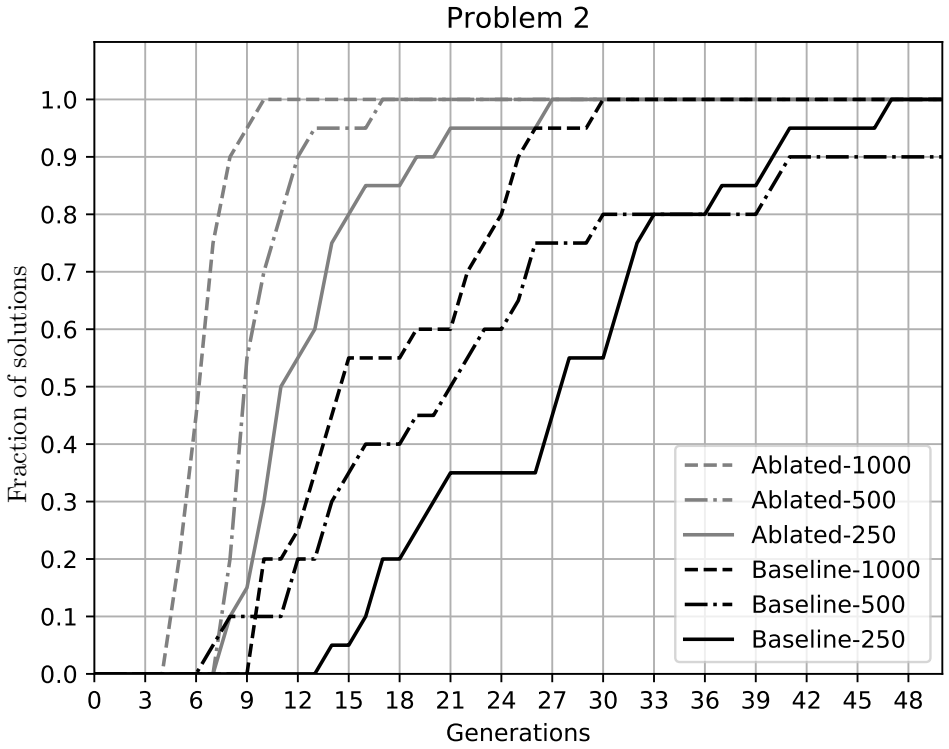

Figure 4.1: *Each curve represent an estimate of the cumulative distribution function that a solution has been found at a given time for problem 2. The gray lines are associtaed with the ablated version and the black ones with the function preserving version. The ablated versions grow more rapidly than the corresponding function preserving versions.*

evolutionary algorithm for problem 3 and 5 was chosen for further inspection. The loss of each individual in

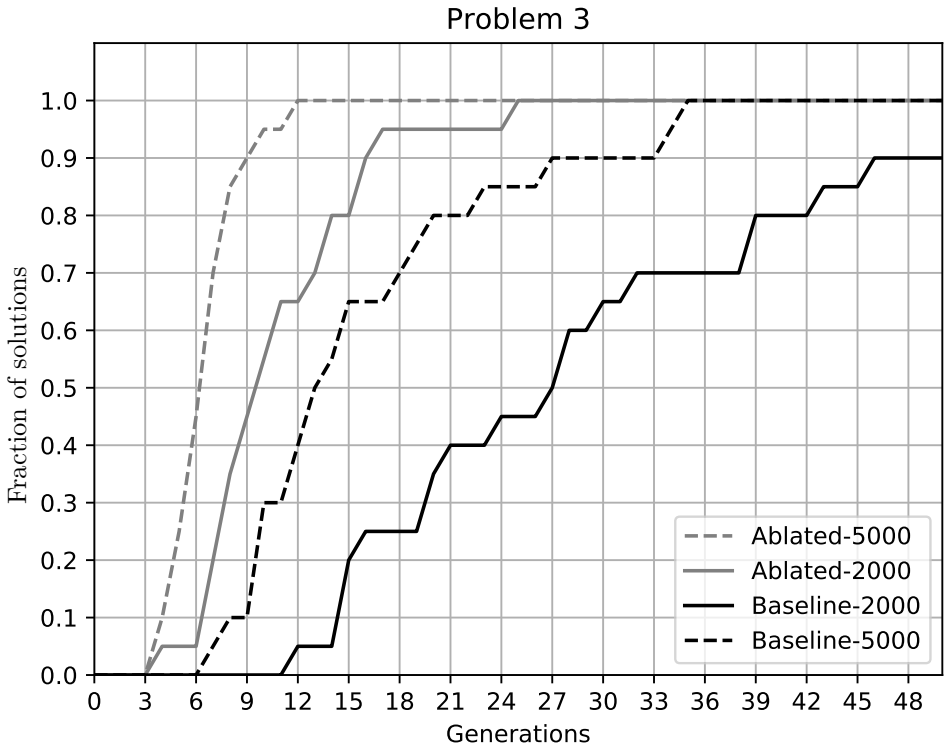

Figure 4.2: *The figure shows the CDFs associated with problem 3. The gray lines represent the distribution of the ablated versions and the black ones with the function preserving versions. The curves show that the ablated version converge faster than the function preserving version as they grow more rapidly.*

the population throughout history is plotted, see Figures 4.3 and 4.4.

Figure 4.3 shows the progression of the baseline and the ablated version for problem 3 with 2000 training iterations, this corresponds to the solid lines in Figure 4.2. The black dots correspond to individuals associated with the baseline and the gray with the ablated version. In the figure, there is a line of dots around a loss of 0.5. It means that the networks have learnt to output values close to zero, the mean of the data. The stagnation in non-optimal function representations of the data occur for both algorithms. However, the ablated version appear to be able to more rapidly diverge from such non-optimal representations.

Similar stagnations can be observed for problem 5. In Figure 4.4 the distribution of individuals for problem 5 is visualized. The individuals seem to be distributed in three groups around 0.17, 0.12 and 0.07.s The first group corresponds to that the networks have learnt the average of the function and only output zeros. The second optima means that the network has learnt to approximate one of the three sinus terms. The third stagnation point means that the network has learnt to approximate two out of three sinus terms of problem 5. As concluded before, the ablated version can more easily diverge from sub-optimal representations of the sought function.

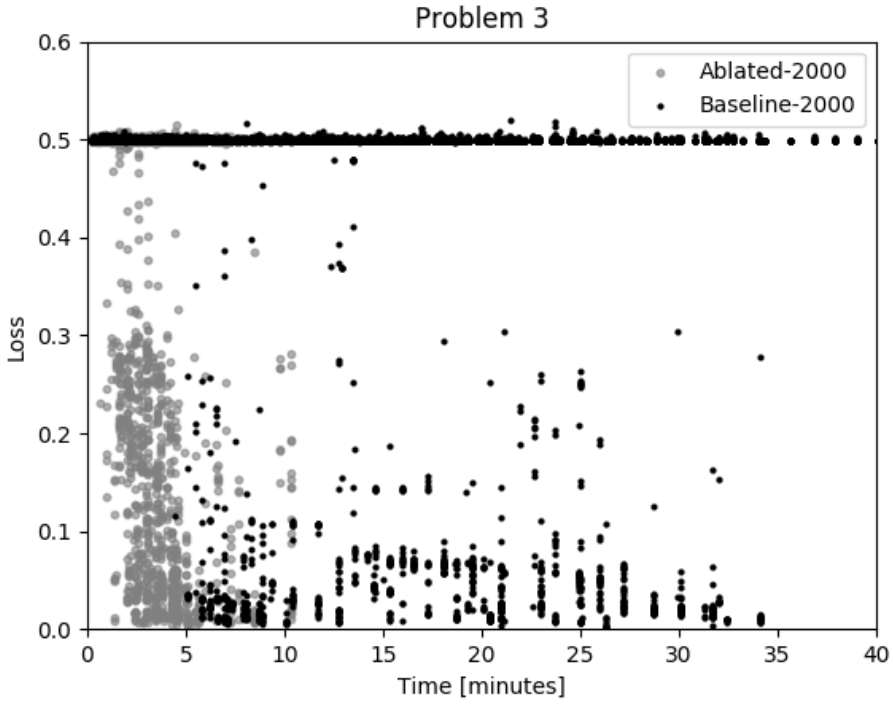

Figure 4.3: *The figure shows the distribution of individuals through time for the two evaluated methods on problem 3 with 2000 training iterations per individual and generation. The gray dots corresponds to the individuals of the ablated version and the black dots with the baseline. Notice the line of dots distributed around a loss of 0.5.*

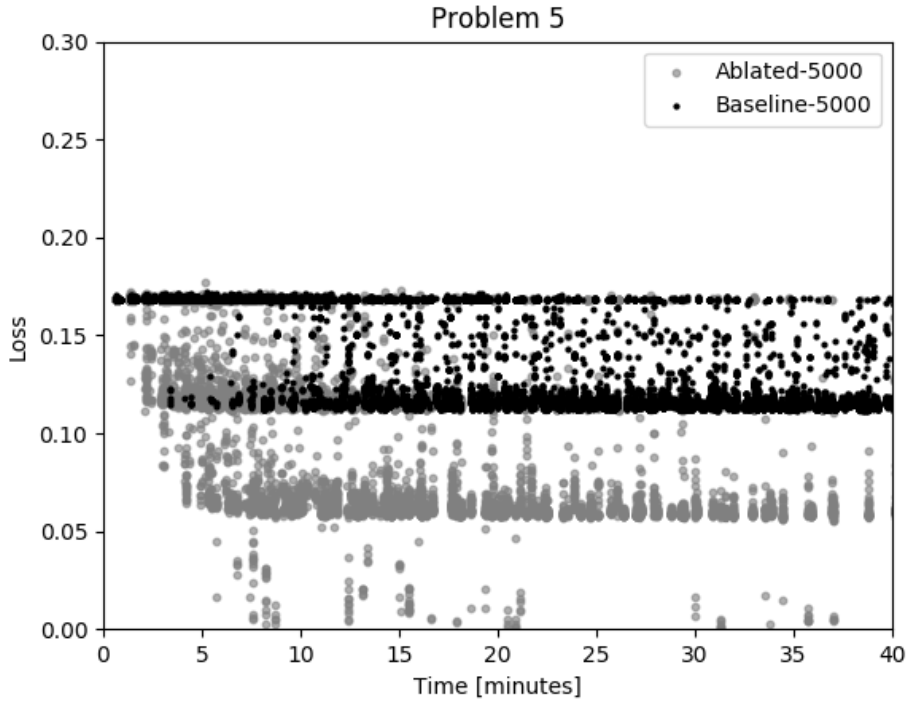

Figure 4.4: *The figure show the distribution of individuals for problem 5 with 5000 backpropagation iterations. The black dots represent the baseline and the gray dots mark the individuals of the ablated version. Notice the three groupings of individuals at loss 0.17, 0.12 and 0.07.*

5 Discussion

This section analyze the results from the evaluation and put forward ideas that can increase the efficiency in neuroevolution.

5.1 Analysis of result

The performance of the proposed method was evaluated relatively to an ablated version in which the function preservating property of the mutations was disabled. The two methods were evaluated on a set of five different regression problems. On the three most difficult problems, 3-5, the ablated version outperformed the baseline. The performance difference appear to be related to a tendency for the baseline to halt at sub-optimal function approximations and must be related to how new structural configurations are initialized.

During training the loss is minimized by updating the parameters in the gradient descent direction. The networks in each generation will eventually reach a minima or stationary point - region of low gradient. Once the training is completed it is likely that the networks are in a stationary region. While the mutations change the networks structurally the functionality is preserved. This gives reason to believe that the loss landscape is preserved as well. In other words, the networks of each generation starts out in the vicinity of a stationary point in a region of low gradients. In Figure 5.1 the described procedure is illustrated. The left panel illustrates the loss surface of a network. The network starts out with parameter configuration A and through backpropagation it ends up in B, a stationary region. The network with parameter configuration B is structurally transformed to point C in the right panel. The mutated network starts out with a parameter configuration that is in the vicinity of a stationary point.

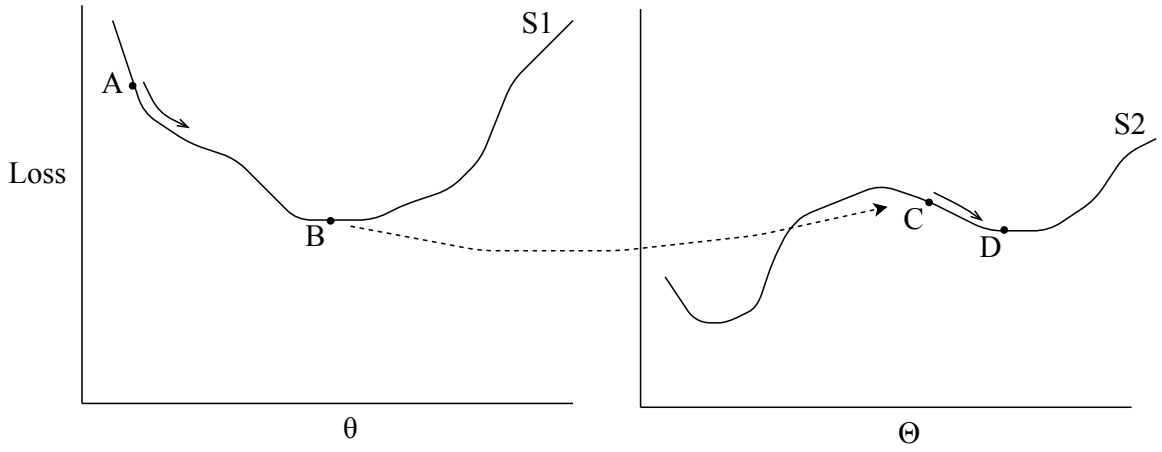

Figure 5.1: *The figure illustrates why the baseline stagnate in sub-optimal function approximations. The left panel shows a parameterized loss surface, $S1(\theta)$, of the parent network structure. The right panel shows the parameterized loss surface, $S2(\Theta)$, of the offspring network. The parent starts with parameter configuration A and through gradient descent ends up with configuration B. The trained network, B, is mutated with a function preserving mutation, indicated with the dashed arrow, to a new network structure with parameter configuration C. The new network is close to the vicinity of a local optima and through backpropagation it will be end up in D, that represent the same sub-optimal representation of the data as the parent network with configuration B.*

The ablated version converge faster because the parameter initialization allows the networks to escape the stationary region. The structural mutations are the same as in the baseline but with the difference that the parameters are randomly initialized. This gives reason to believe that the idea of function preserving transformations can still constitute as a feasible strategy for efficient neuroevolution. If the magnitude of the perturbations can be adjusted in such a way that it allows the networks to escape the stationary region

but simultaneously keeps the network in the vicinity of high performance. In other words, for the function preserving transformations to work some control mechanism is needed that allows networks to escape stationary regions. In the paper *Escaping Flat Areas Via Function-Preserving Structural Network Modifications* [1], recently submitted for blind review, a method is presented that allows networks to escape saddle points. The method add hidden units with parameter initializations that maximizes the gradient. A fusion of neuroevolution based on function preserving mutations with gradient maximization to escape flat regions in parameter space requires further examination.

5.2 Crossover revisited

Crossover was not attempted as fusion of different networks was not expected to improve the efficiency. However, there are other ways to combine networks than through structural fusion. Large efficiency gains can be achieved if the learnt knowledge of two networks can be combined. For example, consider problem 5 in which the networks seem to learn the sinus terms one at a time. If two different networks have learnt to approximate two different terms their offspring can immedeatiley approximate both of the terms. Or in a classification problem two different networks might each have learnt to classify different data. By combining the learnt knowledge of two networks the offspring can immedeatiley improve even without further training.

A possible way for learnt knowledge to be combined without fusing the actual structures would be to create a hierarchical and modular network. A single network can be viewed as a module or a constituent that is constrained by some high level controller. The purpose of the controller is to associate the input data, $\mathbf{x}_{in}$, with the network that is most suited to process the data. Alternatively combine the outputs, o_1 and o_2, in a way that minimizes the loss of the total system.

The problem can be formulated by introducing the function $f(o_1, o_2, \mathbf{x}_{in})$ that represents an optimal combination scheme of o_1 and o_2 that minimizes the loss of the total system. The recombination is now stated as a regression problem where the objective is to find the function f. If f is approximated with a neural network, the controller becomes a natural part of the offspring and its parameters can be updated during training. In Figure 5.2 the recombination scheme is conceptually illustrated where o_1, o_2 and $\mathbf{x}_{in}$ are fed as inputs to the controller network. During crossover the parameters of the parent networks are kept fix while the parameters of the controller network are configured with backpropagation.

5.3 Dynamic adaptation of the number of backpropagation iterations

The performance of the algorithm is reliant on the number of training iterations used per individual and generation. It is difficult to select the optimal number of training iterations prior to starting the evolutionary algorithm. Increasing the number of training iterations will improve the efficiency with regards to the number of generations required to find a solution as a network needs to be chosen for reproduction fewer times to undergo the same number of backpropagation iterations.

Efficiency measured by the number of generations required to find a solution is not, alone, a justified efficiency metric. Instead, the execution time until a solution is found should also be considered. Increasing the number of training iterations is not certain to decrease the exeeution time. If the number of training iterations are high, much time is spent on training bad network structures. If fewer iterations are used, suitable structures that optimize fast could be attained more quickly. On the contrary, if the number of training iterations are too few the simulated evolution is not able to distinguish between the good and the bad networks. Training the networks for only a few iterat¡ions can be sufficient for the selection to guide the algorithm in the right direction.

Effciency gains can be obtained if the number of training iterations are dynamically adjusted. Baker *et al.* demonstrated that standard frequentist regression models can predict the performance of not fully trained CNN by utilizing certain architechtural features along with time-series of the performance on the validation data[Bak+17]. Such methodologies can possible be incorporated in the algorithm. By doing so, the potential performance of each structure could be predicted and the number of training iterations can be adjusted so that promising networks are trained more and less computational resources are spent on unfit structures.

[1] The paper can be found at: `https://openreview.net/pdf?id=H1eadi0cFQ`

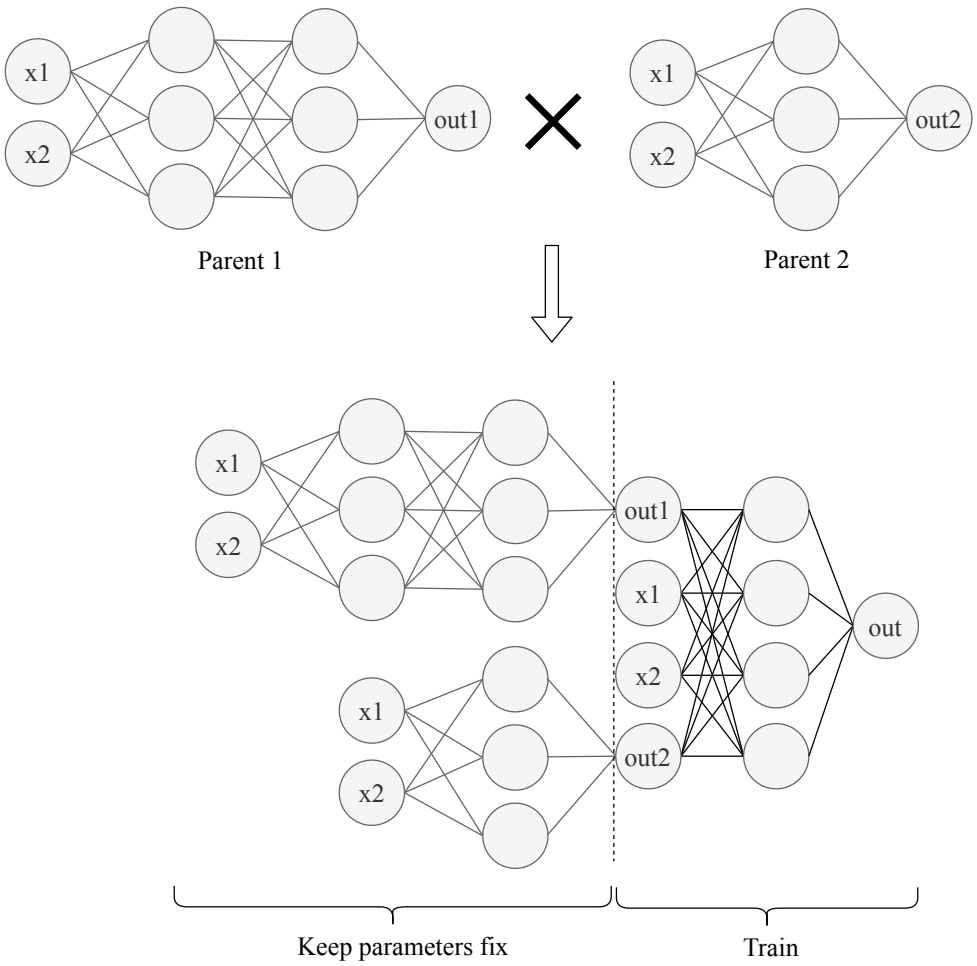

Figure 5.2: *The figure illustrates conceptually how two different networks can be recombined by introducing a neural network that is trained to approximate the function $f(o_1, o_2, \boldsymbol{x}_{in})$. The function f defines the optimal way to combine the output of the parent networks. Once the regulator network has been configured all parameters are released to be updated during regular training.*

5.4 Increasing efficiency by tracking novel topologies

The mutation operators both add and remove structure. Therefore, it is likely that new topologies have been represented before. By keeping a record of all structural configurations through time, new performance increasing opportunities arise. This information can be used to guide the evolution towards novel and previously unrepresented topologies. By guiding the search in the direction of previously unrepresented structures, the optimal structure could be found quicker. Another application can be to estimate the potential of individuals that have been structurally represented before. Instead of re-evaluating such structures the computational resources can be directed towards novel and unexplored structures. A third application can be to replace the networks in the population with the corresponding historic version. This can increase the efficicency as the historic version has been trained for more iterations and is more likely to be closer to the solution in parameter space.

The intention with uniqueness search is to prevent the algorithm from redoing calculations already made and instead divide the computational resources in a way that is more likely to yield a solution. But, as the networks are not trained to completion, continuing training historic structures may be harmful as new versions with different parameter configurations could yield better results more quickly if the network structure is trained again. In other words, uniqueness search could be a useful tool for efficient neuroevolution but it needs to be balanced in order to not miss simple solutions.

5.5 Function preserving mutations for minimal solutions

A possible application for function preserving mutations is for structural reduction. Minimal structure is desirable in most real applications as they generally reduce the execution time. An example of this is if a network structure that can be trained to a satisfactory degree has been found but it does not fulfill the time constraints of the application. Reducing the structure of the network can possibly solve this issue. To manually decide how the structure can be altered while maintaining its performance is difficult.

The function preserving mutations, with low noise level, prevent networks from escaping stationary regions. This behavior may be utilized as a way to reduce the size of networks that already solves the problem. To guide the evolution to search for smaller topologies the selection frequency of the mutation operators that reduce structure can be increased. In that way the algorithm more frequently selects mutations that reduce structure. Instead of basing the selection solely on fitness it can also be based on the number of parameters. This reductional scheme was briefly evaluated and was in some instances able to reduce the number of parameters up to a magnitude of 10.

6 Conclusion

The objective of this thesis was to develop efficient strategies for neuroevolution. An algorithm that builds structure through function preserving mutations were implemented and evaluated. By building structure through these transformations the intention was that the offspring of the next generation would start out in a region in parameter space corresponding to high performance. However, the results of the evaluation show that with the function preserving property of the mutations disabled the efficiency increased. The results was presumed to be caused by the gradient beeing invariant under a function preserving mutation thereby preventing the networks from escaping stationary regions. This insight is important as it provide a direction of future research. In order to efficiently employ the strategy of function preserving mutations there need to be a way to allow mutated offsprings to escape regions of low gradient.

Neuroevolution appear to be a promising strategy for NAS, but it need to become more efficient to be practically employed. A major challange is to find a way to efficiently recombine the functionality of different networks. Previous work has focused attention on crossing networks with the intention that their functionality will be combined as a secondary consequence. Instead the research should focus attention on how functionality of different networks could be combined. A proposed example is that two two networks can be combined by creeating a controller ANN that determine how their outputs should be combined in a way that minimize the loss.

The strategies developed in this work for efficient neuroevolution did not work as intended, but instead the research of this thesis contribute to a better understanding of neuroevolution for NAS.

References

[Bak+17] B. Baker et al. Practical Neural Network Performance Prediction for Early Stopping. *CoRR* **abs/1705.10823** (2017). arXiv: 1705.10823. URL: http://arxiv.org/abs/1705.10823.

[CGS16] T. Chen, I. Goodfellow, and J. Shlens. "Net2Net: Accelerating Learning via Knowledge Transfer". *International Conference on Learning Representations*. 2016. URL: http://arxiv.org/abs/1511.05641.

[Des17] T. Desell. Large Scale Evolution of Convolutional Neural Networks Using Volunteer Computing. *CoRR* **abs/1703.05422** (2017). arXiv: 1703.05422. URL: http://arxiv.org/abs/1703.05422.

[DR92] D. Dasgupta and D. R. McGregor. "Designing application-specific neural networks using the structured genetic algorithm". July 1992, pp. 87–96. ISBN: 0-8186-2787-5. DOI: 10.1109/COGANN.1992.273946.

[DS01] P. Darwen and W. Spencer Churchill. Co-Evolutionary Learning by Automatic Modularisation with Speciation (Feb. 2001).

[Hau+13] M. Hausknecht et al. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games* (2013). URL: http://nn.cs.utexas.edu/?hausknecht:tciaig14.

[Li+17] H. Li et al. Visualizing the Loss Landscape of Neural Nets. *ArXiv e-prints* (Dec. 2017). arXiv: 1712.09913.

[MD89] D. J. Montana and L. Davis. "Training Feedforward Neural Networks Using Genetic Algorithms". *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'89. Detroit, Michigan: Morgan Kaufmann Publishers Inc., 1989, pp. 762–767. URL: http://dl.acm.org/citation.cfm?id=1623755.1623876.

[Mii+17] R. Miikkulainen et al. Evolving Deep Neural Networks. *CoRR* **abs/1703.00548** (2017). arXiv: 1703.00548. URL: http://arxiv.org/abs/1703.00548.

[PD94] M. A. Potter and K. A. De Jong. "A cooperative coevolutionary approach to function optimization". *Parallel Problem Solving from Nature — PPSN III*. Ed. by Y. Davidor, H.-P. Schwefel, and R. Manner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 249–257. ISBN: 978-3-540-49001-2.

[PK18] J. Prellberg and O. Kramer. Lamarckian Evolution of Convolutional Neural Networks. *CoRR* **abs/1806.08099** (2018). arXiv: 1806.08099. URL: http://arxiv.org/abs/1806.08099.

[Rea+17] E. Real et al. "Large-Scale Evolution of Image Classifiers". 2017. URL: https://arxiv.org/abs/1703.01041.

[SM02] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* **10**.2 (2002), 99–127. URL: http://nn.cs.utexas.edu/?stanley:ec02.

[Sta04] K. O. Stanley. "Efficient Evolution of Neural Networks Through Complexification". PhD thesis. Department of Computer Sciences, The University of Texas at Austin, 2004. URL: http://nn.cs.utexas.edu/?stanley:phd2004.

[Wah08] M. Wahde. *Biologically inspired optimization methods : an introduction*. WIT Press, 2008. ISBN: 9781845641481. URL: http://proxy.lib.chalmers.se/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat06296a&AN=clc.b1378449&site=eds-live&scope=site.

[WT98] R. C. Woodruff and J. N. Thompson. *Mutation and Evolution. [electronic resource]*. Contemporary Issues in Genetics and Evolution: 7. Springer Netherlands, 1998. ISBN: 9789401152105. URL: http://proxy.lib.chalmers.se/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat06296a&AN=clc.b1969685&site=eds-live&scope=site.