Technical University of Cologne

Technology
Arts Sciences
TH Köln

Revisiting NEAT
A detailed analysis and modern implementation with a strong focus on formal and programmatic correctness

Silvan Tristan Büdenbender
Student ID 11117250

Supervisors:
Prof. Dr. Hubert Randerath
Prof. Dr. Chunrong Yuan

February 26, 2021

Abstract

After 19 years the procedure of ”Neuroevolution of Augmenting Topologies” (NEAT) is still applied and relied upon in modern research. One of the most recent publications of the original author Kenneth O. Stanley still bootstraps using the novelty search enhanced NEAT algorithm [2]. This works aims to articulate each aspect of its methodology very explicitly, shed light on them from potentially novel perspectives and formalize possible weak spots. A modern implementation is constructed from insights developed in this work.

Contents

List of Figures

List of Tables

1 Introduction

Evolution has intrigued mankind for decades and more. Entire fields of science work in order to figure out its processes and to understand the marvelous results of these processes, namely all organic lifeforms. Further it bears huge potential should it be possible to recreate those processes which result in unprecedented complexity in programmatic nature. This is exactly what the field of Evolutionary Algorithms is set out to explore. It shines when tasks are inherently complex and traditional methods unfeasible [23].

The basic model of programmatic evolution involves a population of solution candidates that are evaluated on a task which determines their so called fitness. This task is the problem to be solved. After evaluation on the task only the top-performers are selected to persist, the other candidates are discarded. Then the population is repopulated with new candidates created via random changes to the top-performers, which is mutation, and interpolation between the top-performers, which is crossover.

An example could be a group of programs all tasked to play a video game and archive the highest score, like in a tournament. Following the tournament, where only a few of the best are permitted to the next round as winners, the group is refilled with various slightly altered clones of the winners, hoping that some of the clones play better than the originals.

The basis of this work is the paper "Evolving Neural Networks Through Augmenting Topologies" often referred to as NEAT [28], which applies an evolutionary algorithm to evolve the topology and parameters of artificial neural networks. This is called neuroevolution [30].

1.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are widespread among a huge number of services and research fields [1]. They are capable of approximating how to perform various tasks via several learning strategies, which alleviates the operator of the ANN from specifying how the solution to a problem works, necessary is only to be able to assess the quality of a solution. This work uses so called reinforcement learning, where agents, i.e. ANNs to be trained, interact with an environment, i.e. the task. For every action taken in the environment agents receive a reward which indicates some form of value the action had in the given situation; the goal is to archive the highest cumulative reward [14].

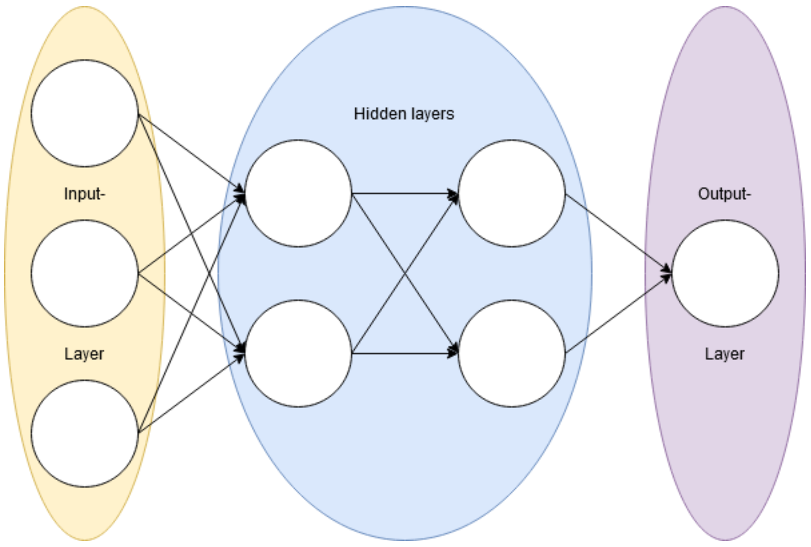

Figure 1.1: Exemplaric ANN topology

ANNs are intended to roughly capture a very simple model of how the brain works, they are made up of neurons also called nodes and connections between them. The exact composition of nodes and connections of an ANN is called its topology which has to be defined before it can learn the task it is hoped to perform. An ANNs topology is usually defined in layers. There exist one input layer and one output layer with so called hidden layers in between. These layers are composed of nodes. Each node in each layer is usually connected to all nodes in the next layer, called "fully-connected". The number of hidden layers and their respective amount of nodes is theoretically unconstrained, as well as their connectivity could be anything less then fully-connected, i.e. connections could be removed as long as each node still has one feeding into it and one leaving it. Being able to use many fully-connected layers due to computational resources becoming more readily available gave rise to the name ”deep neural network” and ”deep learning”. A simple ANN topology is shown in figure 1.1 with two hidden layers each composed of two nodes, all layers fully-connected.

The only known constraints on the topology are the width of the input and output layers as those are most often implied by the task. Designing the hidden layers is mostly backed by experiences gained from past attempts, i.e. knowledge generated by trial end error. More complex components to build ANNs from have been developed [12, 20], sparse neural networks are being researched [17] and several works propose automatic procedures to develop topologies [7] in order to improve on this situation.

1.2 NEAT

The NEAT algorithm is one of those works, providing an alternative to hand-crafting network topologies by evolving them alongside the learning process. More recent work has shown that it is also possible to embed domain knowledge into the topology itself [9], which NEAT could take advantage of as well. There also has been work to compose the mentioned components using the NEAT method [18] and several recent works build on top of the NEAT genome and its operators [2, 22, 10]. This highlights the ongoing interest in the method and reveals that befits could arise by investigating it further.

This work aims to articulate a strong formalism covering the NEAT method, which partially is missing in the original work, to add explainability and predictability to it as well as to simplify some of the original mechanics. It does so by conducting a detailed analysis of each part of the method to end up with it being well-reasoned about and understood or discarded.

2 NEAT in detail

As mentioned before NEAT is a method to evolve ANNs until they are capable of solving a given task. In order to evolve anything every genetic algorithm has to choose a representation for the genes and the genome. A NEAT genome and its encoded ANN can be see in figure 2.1.

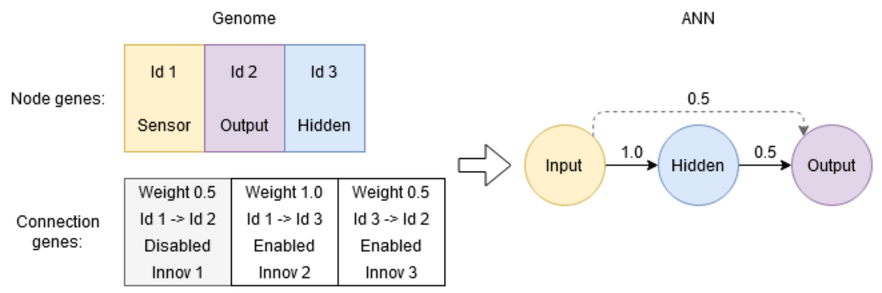

Figure 2.1: NEAT genome with corresponding ANN

In NEAT there exist two types on genes: Node genes and connection genes. The genome is made up of two lists, one for each type of genes. Node genes consist of an identifier and a specifier which is one of sensor, hidden or output, depending on the role of the node. All nodes use a steepened sigmoid activation function. Connection genes consist of a weight and identifiers describing its origin and destination nodes. Further they carry an "disabled" bit and an innovation number.

2.1 SET-NEAT

The genome in this works version of NEAT is represented as a collection of sets. It is called "Set-Encoded Topology NEAT" or SET-NEAT for short.

The decision to represent the components of a genome as sets was made with the intuition that it is supposed to encode a directed, acyclic graph (DAG). The acyclicity is important to have an unambiguous interpretation of how the encoded ANN has to be computed. Further it allows for a simple comparison of genomes via set operations.

The genome is made up of the following structural components:

- I, set of input node genes
- H, set of hidden node genes
- O, set of output node genes
- $F \subseteq \{(x, y) \mid x \in I \cup H \wedge y \in H \cup O \wedge x \neq y\}$, set of feed-forward connection genes

Further a function $f_\alpha \colon I \cup H \cup O \longrightarrow f \colon \mathbb{R} \longrightarrow \mathbb{R}$ is defined that maps every node gene to an associated activation function and a function $f_\omega \colon F \longrightarrow \mathbb{R}$ that maps every connection gene to an associated weight. A full genome thus is represented via the tuple $(I, H, O, F, f_\alpha, f_\omega)$.

In order to actually encode a DAG every $f \in F$ is interpreted as a directed edge and no sequence of edges, where the next edge starts where the previous edge ended, is allowed to exist that traverses any vertex more than once. The union $I \cup H \cup O$ are the vertices of said DAG.

The DAG invariant has to be preserved under the mutation and crossover operations and is the most important difference to the original NEAT genome, which is indeed lacking a definite interpretation on how to be computed as cycles can arise easily, elaborated further in the following sections. That severely limits the reproducibility and comparability of experiments as implementation details can have a major impact on their result.

2.2 Innovation is key

The previously mentioned innovation number is key to the ideas and mechanics employed by the NEAT algorithm. It is a globally incremented counter, assigned to connection genes when they first appear. Topologically identical connection genes appearing in the same generation receive the same innovation number. This allows to uniquely identify every connection over the course of evolution, independent of the overall ANN structure encoded in the genome.

The NEAT paper itself highlights three key mechanisms enabled by the innovation numbers, namely they allow for a proper comparison and crossover of genomes, the proper comparison in turn enables grouping genomes into species separated by genetic differences which results in the capability to start with minimal structure as new but not yet useful structure can be protected in its own species.

What constitutes those capabilities in detail will be thoroughly discussed in the following sections.

2.3 Identity is innovation

The NEAT algorithm works with said innovation numbers. It uses a cache to assign the same innovation number to connection genes encoding identical structure, even when arising in different genomes. The cache keeps track of connection genes and their corresponding innovation numbers. It is only kept for one generation, i.e. connection genes encoding topologically identical structure from different generations do not share the same innovation number.

This work slightly alters and enhances the cache to get rid of innovation numbers while still preserving the same functionality inherently encoded in the identity of the set elements. The initial population is created with identical I and O sets, the H set is empty in the beginning. From the definition of F it is clear that connection genes are identical, should they originate from and are destined to the same node gene. Because the initial node gene identities are identical across all genomes every connection gene created from those will express topologically identical structure. So when the node gene identities introduced over the course of evolution are kept in sync across all genomes this property will continue to hold.

Algorithm 1: Get identity for new node gene

```
Input: $f \in F$ is connection gene to be split, $N$ is $I \cup H \cup O$ of
       mutating genome, $C$ is cache from $F$ to [id], $id\_gen$ is
       function to generate identities
Output: identity of the new node
if $f \in C$ then
    forall $id \in C(f)$ do
        if $id \notin N$ then
            return $id$
        end
    end
    assign $new\_id$ to result of $id\_gen()$;
    append $new\_id$ to $C(f)$;
    return $new\_id$
else
    assign $new\_id$ to result of $id\_gen()$;
    assign $C(f)$ to $[new\_id]$;
    return $new\_id$
end
```

In order to keep identities in sync the the *AddNode* mutation (further explained in subsection 2.4.1) is examined, as it is the only way new identities emerge in the genome. Important for now is that it works by splitting a connection gene, which introduces two new connection genes and a new node gene. This procedure needs to be reproducible, i.e. should the identical connection gene be split in another genomes it has to result in a node gene with the same identity. To archive this the cache C is a permanent mapping from F to an ordered list of identities introduced by splitting connection genes. Notice the mapping goes to an ordered list, as the same connection gene could potentially be split several times due to future mutations targeting it again. When a connection gene $f \in F$ is split the cache is consulted regarding f. Should a list of identities exist, the first in order that is unknown to the mutating genome is selected. In the case that all identities are known a new identity is generated, appended to the list in the cache and selected. Otherwise, when f is not yet present in the cache, a new list with a new identity is added to the cache under f and that same new identity is selected. This method implies that structure is considered identical independent of the generation it appeared in.

By following algorithm 1 the node gene identities are in sync and should two connection genes have the same identity they are guaranteed to encode the same structure without the need for innovation numbers.

2.4 Mutation and Crossover

Mutation and crossover are the two drivers of change in individuals. Mutating a genome changes parts of it randomly while crossing two genomes over recombines parts of the two into one. The possible mutations and the crossover operator are explained and analyzed in the following sections.

2.4.1 *AddNode* Mutation

Originally an existing connection gene is split to add a new node gene, i.e. it is disabled and two new enabled connection genes with their according innovation numbers and the new node gene are added to the genome. The first new connection gene runs from the origin of the split connection gene to the newly introduced node gene with a connection weight of one. The second new connection gene originates from the new node gene and runs to the destination of the split connection gene with its original weight. This is to limit the initial impact of the new non-linearity.

The ”enable/disable” mechanism causes trouble as it does not maintain the DAG invariant by producing disconnected structure, elaborated in subsection 2.4.5. Due to this issue this work incorporates a suggestion from [27] of resetting a connection weight to zero when it is split during an *AddNode* mutation and removing the ”disable” bit entirely. Following this procedure the split connection gene might not be relevant for the computed output for a while but will still express a computable structure by propagating a zero value.

2.4.2 *AddConnection* Mutation

In the original algorithm this mutation adds an enabled connection gene with a random weight and a new innovation number between two nodes that where not previously connected .

In this work the additional constraint applies that the new connection gene is not allowed to introduce a circle into the encoded ANN. Further the innovation number and ”disable” bit are superfluous. That implies, as stated in the definition of F, that only feed-forward connections can mutate, while the original imposes no such restrictions. That restriction will be lifted in chapter 3.

2.4.3 *ChangeWeight* Mutation

Changing the weights of a genome originally happens to all weights at once by a user-defined chance. Should they be mutated each weight is then either uniformly perturbed or set to a random value.

This work always perturbates a configured percent of all weights of each genome with perturbations sampled from a normal distribution with a user-defined σ and zero mean. Further it constrains the possible weights to the interval $[-cap, cap]$ where *cap* is user-defined in order to define a comparable weight difference, elaborated in subsection 2.5.1.

2.4.4 *ChangeActivation* Mutation

As presented in several other works [26, 9] this work as well allows the mutation to change activation functions from a configured pool. The pool is the same as used in [9] without the "square" function, as it potentially explodes the propagated values. This shall allow for the potential benefits gained by encoding domain knowledge into the topology as well as allowing to build HyperNEAT [29] on top of this work.

2.4.5 *Crossover* Operator

Originally crossing over two genomes happens by some chance inside the same species or even between different species. The crossover operator would line up the connection genes of the two involved genomes by their innovation number. It then operates on the genes that did line up to their same innovation number counterpart. From those matching genes either would be inherited by a fair 50/50 chance, effectively resulting in picking one of the two possible weights for the identical gene. All of the differing structure is included from the fitter genome. NEAT also shows that by enabling crossover solutions are found faster.

It is explicitly illustrated that crossover of two equal fitness genomes results in inheriting all their matching and differing structure into the offspring. Despite that procedure being given, NEAT does not provide a mechanism to give meaning to that operation, likely resulting in structure violating the DAG invariant as can be seen in figure 2.2. It displays the minimal structural setup that does so as the crossover of these two genomes by the original rules creates a cycle in the ANN (highlighted in red). It then is missing a definite interpretation on how to be computed.

This work suggests to avoid that problem by always determining a fitter genome, either the fitter by fitness value, in case of equal fitness the shorter

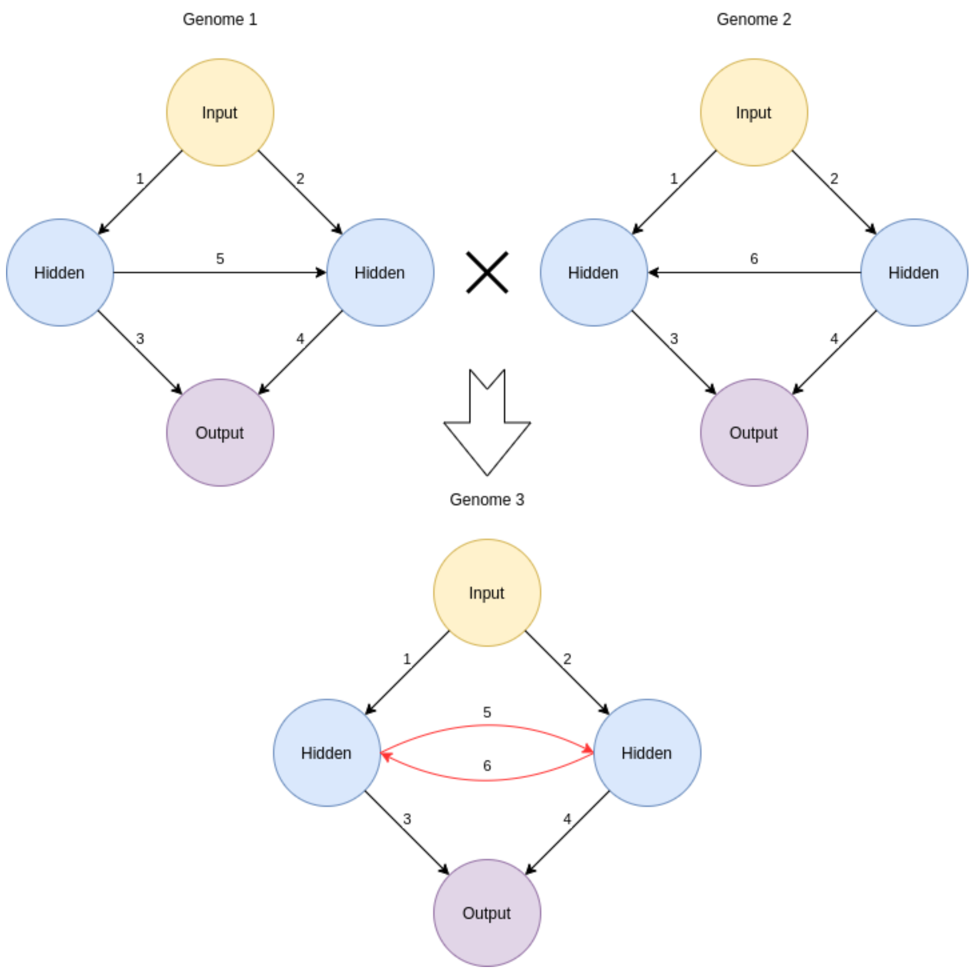

Figure 2.2: Crossover resulting in circle

genome as suggested in [25] or else the partner genome. Let's call that last case "the selfless genome".

Another problem during crossover is caused by the "disable" bit as it is inherited by chance in shared connection genes. Thereby the NEAT algorithm can introduce dangling structure in the sense that hidden nodes might not have incoming or outgoing connections. That also violates the DAG invariant and has to be prohibited. A minimum example displaying this behavior can be seen in figure 2.3 where genome 1 is assumed to be fitter and red indicates a connection is disabled.

It can be concluded that the crossover operator is not operating on structure directly, it only might alternate details of the shared structure, namely the connection weights and, in this works version, activation functions. So applying the crossover operator to two genomes selects the fitter one and might replace connection weights of shared connections with the correspond-

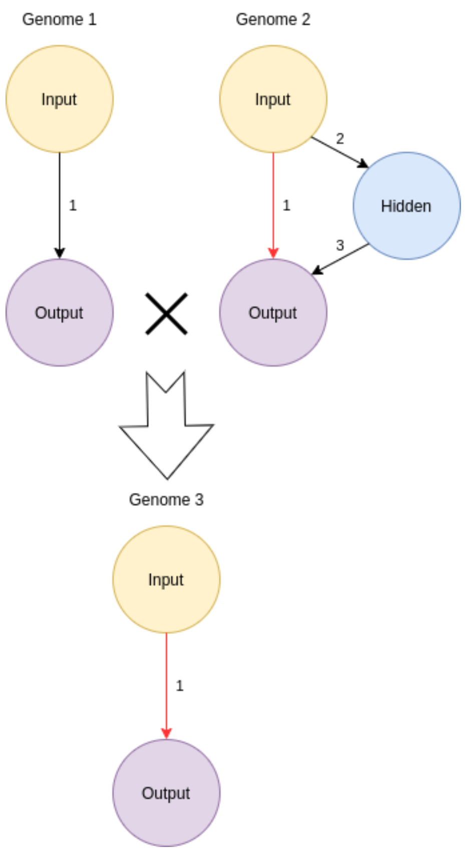

Figure 2.3: Crossover resulting in disconnected structure

ing weights of the partner genome, analogously it might replace activation functions of shared nodes with the corresponding activation functions of the partner genome. This work always reproduces via crossover but only inside the same species as being in another species by definition indicates very little shared structure that could benefit from such a crossover.

2.5 Speciation

Speciation is responsible to divide genomes into species based on a genetic difference metric. A new species emerges when an individual is not compatible to any existing species. This gives new structure time to develop value by being protected inside its own niche as survival and reproduction are determined inside each species. A species s is defined via its first member, which simultaneously becomes its representative r and a list of individuals M belonging to that species, as can be seen in equation 2.1.

$$s = \{\, (r, M) \mid r \in P \land M \subseteq P \,\} \tag{2.1}$$

Inside a species the fittest x percent of the individuals get to reproduce, where x is a user-defined survival rate. The average fitness of those top-performers is compared to the other species average fitness of top-performers and the amount of offspring each species produces is in proportion to the ratio of those averages.

2.5.1 Measuring genetic difference

The original NEAT paper computes genetic difference (δ) via equation 2.2.

$$\delta = c_1 * \frac{E}{N} + c_2 * \frac{D}{N} + c_3 * \overline{W} \tag{2.2}$$

It distinguishes between disjoint and excess genes for the computation, which are determined by lining up the connection genes by their innovation numbers, like for a crossover. Every gene, iterated backwards from the genome with the highest innovation number, until the first matching innovation number is encountered, is called "excess". Every following gene that does not have a matching innovation number is called "disjoint". Their counts are given correspondingly by E and D, while N is defined as the number of genes in the larger genome. $\overline{W}$ is the average weight difference of matching genes. These terms are parameterized by user-defined factors c_1, c_2, c_3. The difference between disjoint and excess genes only comes into play when the corresponding terms are parameterized differently. Further the distinction seems arbitrary, when considering how excess genes can turn into disjoint genes by just one mutation: In the case of a genome with just one connection gene $G_1 : [0]$ and genome $G_2 : [0, 1, 2, 3, 4, 5]$ with several connection genes all of the genes $[1, 2, 3, 4, 5]$ are considered excess genes. If now $G_1 : [0, 6]$ happens by mutating a new connection gene, all genes $[1, 2, 3, 4, 5]$ are considered disjoint and $[6]$ is considered excess which might significantly influence the computed genetic difference between G_1 and G_2 despite very little actual change in the structure of the genome.

This work dropped the distinction between excess and disjoint genes as the distinction is not well reasoned about and they are already considered the same regarding their parametrization in the original experiments and when it comes to how they are inherited (either both or none).

The ratios expressed in the original paper are also adapted. Instead of normalizing by the bigger genome, it is normalized by the total amount of unique connection genes, expressing more precisely the percentage of shared structure between the two genomes. Also, as in works before, a term to express the difference in activation functions of shared structure is incorporated. Between all shared node genes, the differing activation functions are counted and expressed as a ratio to the count of shared node genes, giving a precise percentage of differing activation functions in the shared structure. Regarding weights the formula is adjusted to express a ratio of the present difference to the potential maximal difference with regards to the interval defined by *cap* as mentioned in 2.4.3. The final difference is expressed by normalizing the factored terms by their summed factors.

So for two genomes $(I_1, H_1, O_1, F_1, f_{\alpha_1}, f_{\omega_1})$ and $(I_2, H_2, O_2, F_2, f_{\alpha_2}, f_{\omega_2})$ their genetic difference δ is defined in equation 2.3.

$$\begin{aligned}
\delta &= \frac{c_1 * \delta_F + c_2 * \delta_\omega + c_3 * \delta_\alpha}{c_1 + c_2 + c_3} \\
\delta_F &= \frac{|F_1 \triangle F_2|}{|F_1 \cup F_2|} \\
\delta_\omega &= \frac{\sum \{\, diff_\omega(\mathrm{f}) \mid \mathrm{f} \in F_1 \cap F_2 \,\}}{|F_1 \cap F_2| * 2 * cap} \\
\delta_\alpha &= \frac{\sum \{\, diff_\alpha(h) \mid h \in H_1 \cap H_2 \,\}}{|H_1 \cap H_2|} \\
diff_\omega(\mathrm{f}) &= |f_{\omega_1}(\mathrm{f}) - f_{\omega_2}(\mathrm{f})| \\
diff_\alpha(h) &= \begin{cases} 0, & \text{if } f_{\alpha_1}(h) = f_{\alpha_2}(h) \\ 1, & \text{otherwise} \end{cases}
\end{aligned} \tag{2.3}$$

When considering recurrent ANNs the terms for the feed-forward connections are duplicated accordingly for the recurrent connections, shown in appendix B.

2.5.2 Initializing and adapting the threshold

The previously defined genetic difference metric is compared to a threshold in order to determine species membership of the individuals in the population. This threshold directly influences the amount of existing species.

It can be hard to predict how many species a given threshold will produce and certainly there is somewhat of a useful amount of species with regard to the population size (neither all in one nor everyone in their own). Therefore the compatibility threshold is automatically initialized and adapted, a form of adaptation can already be seen in [24]. The user shall configure a desired amount of species and the algorithm tries to stay as close as possible to the desired value.

Initialization of the threshold (see equation 2.4) is performed by determining the average species size $\overline{|s|}$. Then for each initial individual the difference to every other individual in the population is computed as defined in equation 2.3. Those are listed in $differences_{|P|\times|P|}$ below where $\delta_{x,1}$ denotes the least difference and $\delta_{x,|P|}$ the largest difference with regard to individual x. The $\overline{|s|}$th difference from each comparison is then summed and finally averaged over the population size $|P|$.

$$\overline{|s|} = \frac{|P|}{target\_species\_count}$$

$$differences_{|P|\times|P|} = \begin{pmatrix} \delta_{1,1} & \dots & \delta_{1,\overline{|s|}} & \dots & \delta_{1,|P|} \\ \vdots & \dots & \vdots & \dots & \vdots \\ \delta_{|P|,1} & \dots & \delta_{|P|,\overline{|s|}} & \dots & \delta_{|P|,|P|} \end{pmatrix} \tag{2.4}$$

$$\mathcal{T}_{initial} = \frac{\sum_{p=1}^{|P|} \delta_{p,\overline{|s|}}}{|P|}$$

The intuition behind this initialization is as follows: It looks at every individual and figures out what the threshold $\mathcal{T}$ would have to be to produce the average species size should that individual represent a species. Then those estimates are averaged. This can only be a rough approximation of the threshold desired to produce the configured amount of species but has served well enough in the experiments. Further the adaptation mechanism has been observed to be robust against imprecise initialization.

The adaptation formula is given in equation 2.5 where $\mathcal{T}_{t+1}$ denotes the next i.e. adapted threshold value. $\mathcal{T}_t$ refers to the current threshold value and $\#existing_t$ is the number of existing species, both at timestep t. $\#desired$ is the configured number of desired species.

$$\mathcal{T}_{t+1} = \mathcal{T}_t * \sqrt{\frac{\#existing_t}{\#desired}} \tag{2.5}$$

It is evaluated at every generation to approach the desired species count.

2.5.3 Starting minimal

The capability to add new structure with the potential to persist by being protected in a new species allows the topological setup to be minimal, i.e. the population is initialized with ANNs that only have the input layer fully-connected to the output layer with random weights. Starting minimally leads to the smallest possible solution and to a reduced search space which is beneficial to the speed of the overall process.

To adapt better to high input domains this work incorporates a suggestion from [27] and therefore allows to start with only a configured percentage of inputs connected to all outputs as the initial topology. Thereby evolution may decide which inputs are actually relevant for the task and incorporate more inputs when appropriate.

3 Evaluating ANNs of arbitrary topology

Given some description of a genome introduced in chapter 2 there is the requirement to somehow compute the function encoded in the genome. It is a particularly interesting task as the evolutionary process can produce about any topology imaginable, given sufficient time to do so.

This chapter will elaborate on two distinct approaches, one from the original NEAT paper and one devised as a part of this work.

3.1 Original Procedure

The evaluation strategy is not discussed in the NEAT paper itself though [27] points to the source code [25] where the procedure can be examined.

Given the description of nodes and connections inside the ANN the original approach to evaluating the encoded ANN is to loop all nodes continuously, checking if any incoming connection provides a value, then activate the node with that potentially partial input. The loop stops once all outputs nodes have been activated. A simplified version of the procedure is shown in algorithm 2, the full code can be seen in the original source [25] or in the Rust port [3].

Algorithm 2: Original evaluation strategy

```
Input: $g$, genome to evaluate, $i$ inputs to encoded ANN
Output: $o$, values of activated $g.O$
activate $g.I$ with $i$;
while $g.O$ not active do
    forall node of $g.H \cup g.O$ do
        collect all active inputs;
    end
    forall node of $g.H \cup g.O$ do
        if node has collected inputs then
            activate node with inputs;
        end
    end
end
```

It is hypothesised that the output of this process is hard to predict as changes in network topology can alter the path over which the outputs receive a signal, potentially short-circuiting the ANN. Further it is hypothesised that the networks response is sensitive to the order in which the nodes are looped, which would make the output of an ANN depend on implementation details due to lacking specificity of the procedure.

This sensitivity to order can be seen in figure 3.1 on the left side, where two possibilities are displayed how the procedure could turn out. Nodes with green borders indicate that they are activated, orange connections indicate an active input. It can be seen clearly that in possibility one the hidden node does not influence the final output as it is not activated. It is also easily imaginable that a connection gene connecting the input and output like in the figure or bridging similar wide distances in the ANN could be added over the course of evolution, resulting in the hypothesised short-circuiting of the ANN.

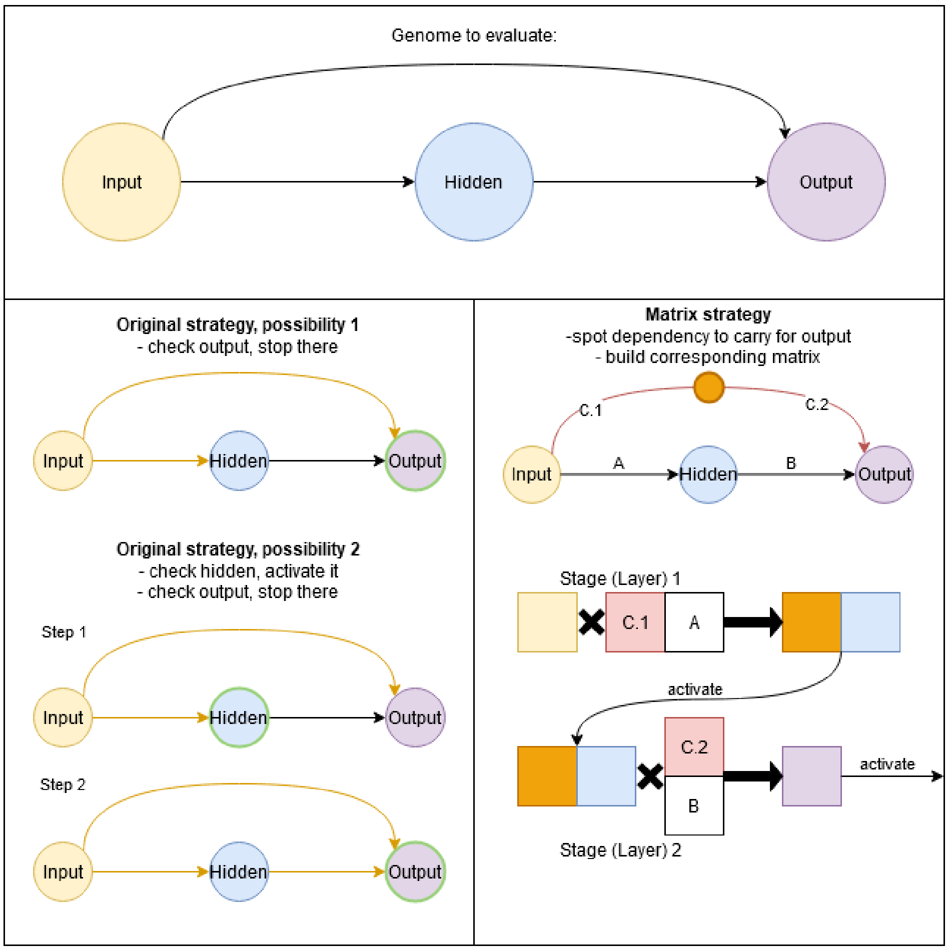
Genome to evaluate:
Input
Hidden
Output
Original strategy, possibility 1
- check output, stop there
Input
Hidden
Output
Original strategy, possibility 2
- check hidden, activate it
- check output, stop there
Step 1
Input
Hidden
Output
Step 2
Input
Hidden
Output
Matrix strategy
-spot dependency to carry for output
- build corresponding matrix
C.1
C.2
Input
A
Hidden
B
Output
Stage (Layer) 1
C.1
A
activate
C.2
B
activate
Stage (Layer) 2

Figure 3.1: Comparison of evaluation strategies

3.2 Alternative Procedure

This work constructs a predictable and potentially more efficient representation before starting evaluation. The idea is to decompose the ANN into several matrices in order to compute as much as possible in each step and potentially benefit from SIMD operations or GPUs.

The ANN is decomposed into so called stages where every stage is made up of a matrix and a vector of activation functions. A stage resembles a layer in a common ANN. The evaluation of the ANN can then be computed in the same way ANNs with fully connected layers are. Matrix-multiply the input with the matrix, apply activation functions and repeat with the newly computed vector as the input for the next stage/layer.

Algorithm 3: Construct matrix representation from genome

```
Input: $g$, genome to be evaluated
Output: $S$, list of stages and $A$, list of activation vectors
construct dependency graph $D$ for $g.H$ and $g.O$;
initialize $Av$, list of available nodes with $g.I$;
while $D$ not empty do
    initialize list of column vectors (stage) $s$;
    initialize list of activations $a$;
    initialize list of next available nodes $Av\_next$;
    forall $n \in D$ do
        if all $n.dependencies \in Av$ then
            add column vector of corresponding weights to $s$;
            add $f_\alpha(n)$ to $a$;
            move $n$ from $D$ to $Av\_next$;
        end
        if some $n.dependencies \in Av$ then
            add carry vectors for available $n.dependencies$ to $s$;
            add linear activation for each carry vector to $a$;
            add carried $n.dependencies$ to $Av\_next$;
        end
    end
    add $s$ to $S$;
    add $a$ to $A$;
    set $Av$ to $Av\_next$;
end
```

In order to accomplish the decomposition a simple basic schema is applied, see algorithm 3. Initially a dependency graph for every node but the inputs is computed and all input nodes are marked as 'available'. The available values are represented as d-dimensional vector $Av = \big(a_1, \ldots, a_d\big)$.

The dependency graph is iterated as long as it is not empty and n is the dependent node being checked at the current iteration. The dependencies of n are compared to the available nodes and depending on the outcome one of three actions happens:

- Should Av contain every node n depends on a column vector containing the weights of the relevant inputs is constructed and added to the current partial matrix s that represents everything computable from the currently available nodes when complete. That vector is defined as c in equation 3.1.

  $$\begin{gathered} c = \big(\mathcal{W}(a_1), \ldots, \mathcal{W}(a_d)\big) \\ \mathcal{W}(a_d) = \begin{cases} f_\omega((a_d, n)), & \text{if } f_\omega \text{ is defined at } (a_d, n) \\ 0, & \text{otherwise} \end{cases} \end{gathered} \tag{3.1}$$

  The associated activation function $f_\alpha(n)$ is added to the current activation vector a that will be applied after the matrix multiplication. n is then removed from the dependency graph.

- Should Av only contain a subset of nodes n depends on, then for every node a_d of this subset a carry vector in form of the standard unit vector e_d is added as a column to the partial matrix s. The identity function is added to a as an activation function. Carry vectors and their activation functions are only added once per stage for every a_d.

- Should neither be the case i.e. no value that n depends on is present nothing is changed.

All computable dependencies and all carried values are marked as available for the next iteration of the dependency resolution process. Each iteration will produce a matrix s and an activation function vector a and after all dependencies are resolved the list of matrices and list of activation function vectors encodes the entire function of the ANN.

An illustration of this procedure can be seen in figure 3.1 on the right side. The smaller orange node indicates a carry whose necessity has been discovered by the dependency resolution process. Carries can be imagined as phantom nodes that do not actually exist in the ANN but are required to compute it as a matrix. Connections leading into carry nodes (C.1) always have a weight of one and the nodes themselves always have a linear activation function. The outgoing connection (C.2) has the original weight of the connection that was extended by the carry node. With the carry node the decomposition into two matrix multiplications becomes possible, indicated as Stage 1 and Stage 2 in the figure.

Following this procedure, as many values as possible can be computed per step, in as few steps as possible. A potential problem are unnecessarily bloated matrices when lots of carries are added. This could be mitigated by using sparse matrix representations.

3.3 Handling recurrent connections

Recurrent connections allow the ANN to express a dependency on past internal state. That means every recurrent connection resembles a memory cell storing a value from the current evaluation to be available in the next. Some tasks benefit greatly from being able to observe data over time as it allows to interpret single inputs in context.

3.3.1 Original

The original procedure handles those by buffering the last activation value of every node and should an incoming connection of a node be marked as recurrent this buffered value is collected as an input. The buffer is updated when its corresponding node is activated. The genome only reflects recurrent connections by that additional marker.

3.3.2 Alternative

To encode recurrent connections in the genome of this work a new set and a new function are introduced:

- $R \subseteq \{\, (x, y) \mid x \in I \cup H \land y \in H \cup O \,\}$, set of recurrent connections

R is similar to F and they can hold elements with matching identity but their semantics differ. Further elements in R are allowed to express connection genes that originate from and destine to the same node, i.e. an encoded node can depend on its own last value. Due to the overlapping identities the weights for elements of R have to be captured in a new function f_{ω_R} and f_ω is now called f_{ω_F}. A genome with recurrent connection genes is thereby defined as $(I, H, O, F, R, f_\alpha, f_{\omega_F}, f_{\omega_R})$.

To compute ANNs with recurrent connections it would be useful to replace the recurrent structure with the equivalent feed-forward structure and then use the evaluation procedure defined in section 3.2. To obtain a feed-forward only representation of a recurrent genome the operation $\mu(R, f_{\omega_R})$ is introduced which maps the recurrent structure present in the any genome to the tuple $(I_\mu, O_\mu, F_\mu, f_{\omega_\mu})$, which denotes the necessary structure to express the recurrent connections in a feed-forward manner. μ is not specific to any genome.

$$\begin{aligned} I_{all} &= I \cup I_{\mu} \\ O_{all} &= O \cup O_{\mu} \\ F_{all} &= F \cup F_{\mu} \\ f_{\omega_{all}} &= f_{\omega_F} \cup f_{\omega_{\mu}} \end{aligned} \tag{3.2}$$

With equation 3.2 a feed-forward representation of a genome then is:

$$(I_{all}, H, O_{all}, F_{all}, f_{\alpha}, f_{\omega_{all}})$$

The explicit procedure of μ is also called "unrolling" and is possible due to the DAG invariant preserved throughout evolution. Unrolling happens by mapping every recurrent connection gene to a new input node gene, a new output node gene and two new feed-forward connection genes.

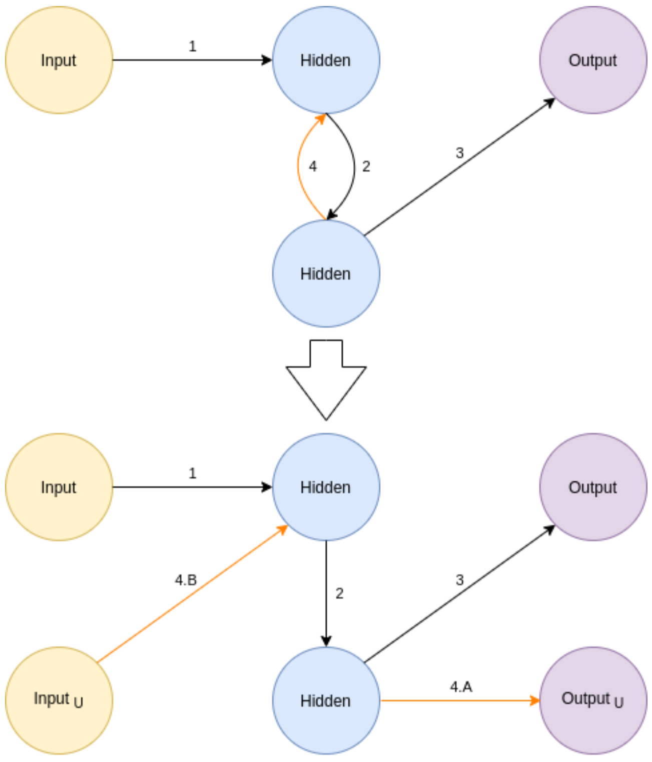

Figure 3.2: Unrolling a recurrent connection

One connection gene runs from the origin of the recurrent connection gene to the added output node gene with a weight of one. The second connection gene runs from the added input node gene to the destination of the recurrent connection gene with its original weight. This can be seen in figure 3.2 where the orange structure represents the recurrent connection and its unrolled counterpart. Between evaluations the value of the added output node gene $Output_U$ is transferred to the value of the added input node gene $Input_U$. Every outward leading connection gene (4.A) i.e. to $Output_U$ is only created once, should multiple recurrent connection genes originate from the same node. Only new inward pointing connection genes (4.B, 4.C, ...) with their according weights are added to express those further recurrent connection genes originating from the same $Input_U$.

To allow recurrent connection genes to emerge the *AddConnection* mutation has a configurable chance to mutate the gene into R instead of F.

4 Demonstrating capabilities with experiments

In order to observe and assess the behavior and performance of the proposed changes to the NEAT method a series of experiments are conducted. All experiments are performed with the default configuration stated in appendix A and the software version of the git-tag ”thesis” unless stated otherwise. The relevant software is listed in [4] and [3] and written Rust. It is a modern, compiled language with the focus on compile-time memory safety and performance and an overall good deal in terms of speed, memory consumption and energy consumption [21].

This chapter will present the data and conclusions from experiments conducted on three tasks. First the classic XOR task will be solved to validate all basic capabilities of the method were preserved, just as in the original NEAT paper. Secondly two tasks from the OpenAI-Gym will be tackled in order to show the capability to solve more complex problems as well as to gather data that the two evaluation strategies elaborated in chapter 3 can be compared on.

Every tasks has a bias input with a constant signal of one added to the default input as done previously in NEAT to allow the evolution of a bias term without incorporating it into the nodes themselves.

The following tables present the amount of hidden nodes $|H|$, feed-forward connections $|F|$ and recurrent connections $|R|$, number of generations as well as the score and the set goal, each averaged over 100 successful evolutions and their corresponding standard deviation denoted as $(\pm\ x)$.

4.1 Verification with XOR

The simplest task to verify basic functionality on is to evolve the XOR gate as a neural net, the fitness function used is the same as in [28]. The configuration deviates from the default as the weight perturbation sigma is increased to 3.0 with a cap of 9.0, the output node has a sigmoid activation function as in the original NEAT tasks and recurrent connections are turned off. The difference between the two setups is the evaluation procedure as explained in chapter 3.

evaluation	$\|H\|$	$\|F\|$	$\|R\|$
original	1.61 (± 0.80)	7.79 (± 1.87)	0
matrix	1.52 (± 0.83)	7.79 (± 1.99)	0

Table 4.1: XOR structure, task configuration

evaluation	#generations	score	goal
original	10.38 (± 4.16)	15.98 (± 0.03)	15.90
matrix	15.32 (± 7.27)	15.99 (± 0.03)	15.90

Table 4.2: XOR results, task configuration

Table 4.1 and 4.2 shows the results from the experiments run with the configuration described above. The two experiments are repeated with the restriction that hidden nodes always use the sigmoid activation function instead of the full repertoire, the weight mutation rate is at 80%, new nodes appear with 3% chance, new connections with 5% chance. This configuration closely resembles the original, results are shown in table 4.3 and 4.4.

evaluation	$\|H\|$	$\|F\|$	$\|R\|$
original	2.92 (± 1.35)	10.72 (± 3.18)	0
matrix	4.90 (± 2.98)	20.05 (± 9.74)	0

Table 4.3: XOR structure, nearly original configuration

evaluation	#generations	score	goal
original	27.35 (± 14.18)	15.97 (± 0.03)	15.90
matrix	142.05 (± 96.99)	15.97 (± 0.03)	15.90

Table 4.4: XOR results, nearly original configuration

The results from the XOR task show that the adapted methodology and evaluation mechanism still exhibit the basic expected behavior. Both evaluation strategies find solutions with the task configuration in less generations than the original NEAT XOR verification result which took 32 generations. It provides both with good results with neglectable differences in structure (see table 4.1) establishing a solid baseline. This works strategy still took 1.47x generations more to reach the solution. If now the possibility to change activation functions, which is available in the task configuration, is taken away it falls further behind, resulting in a 1.68x increase in $|H|$ and 1.87x increase in $|F|$. Apart from bloat it takes 5.19x more generations to solve the task. This hints at some major differences between the two evaluation strategies, possibly a bias of the original towards the task or a major benefit this works strategy has from switching activation functions. With the nearly original configuration the original strategy in this work is still slightly better than in the original work indicating a potential overall increase in evolution performance.

4.2 OpenAI-Gym

Many papers in the field of GAs and neuroevolution bring their own problem with the solution or method they are proposing. Due to this circumstance it is not straight forward how those methods can be compared. This work tries to validate itself by testing its method against a collection of known tasks, namely the OpenAI-Gym. It comes with several types of tasks with varying difficulties. A task is considered solved when the average score of an individual over 100 simulations is equal or higher than the set goal.

When training not just one ANN but evaluating an entire population of them a crucial factor regarding time and score accuracy is how many simulation an ANN is evaluated on, as those are randomized to some degree. A bad or good score from just one simulation has difficulties representing the abilities of the ANN when confronted with the entire spectrum of the environment that comes from the 100 consecutive simulations used to validate a possible solution. Increasing the number of simulations better assesses the overall capability of the individual ANN to handle the task but comes at significant computational cost.

4.2.1 Classic control

Classic control tasks are "Control theory problems from the classic RL literature" [6]. The following task is evaluated in four scenarios: Twice with the original evaluation strategy, each time with one and two simulations respectively. The same happens with this works "matrix" evaluation strategy, the two simulations are denoted with the suffix "+1" in the following tables.

CartPole-v1

CartPole-v1 [5] is about balancing a pole on a cart which can slide along a fixed horizontal rail. It can move with fixed speed in either direction. The provided inputs are the carts position and velocity along its variable axis. Further the angle and angular velocity of the pole are available information. Table 4.5 shows the evolved structure of the solution, table 4.6 shows the generations passed and the archived score.

evaluation	$\|H\|$	$\|F\|$	$\|R\|$
original	1.55 (± 0.99)	12.71 (± 4.13)	1.70 (± 1.49)
original+1	2.17 (± 1.47)	11.98 (± 4.26)	1.26 (± 1.19)
matrix	1.38 (± 1.02)	9.59 (± 3.45)	0.5 (± 0.75)
matrix+1	1.73 (± 1.09)	10.87 (± 3.68)	0.74 (± 0.84)

Table 4.5: CartPole-v1 structure, default configuration

evaluation	#generations	score	goal
original	120.15 (± 101.40)	266.71 (± 59.22)	195.00
original+1	70.83 (± 81.97)	271.07 (± 64.96)	195.00
matrix	22.63 (± 45.17)	286.03 (± 63.67)	195.00
matrix+1	27.59 (± 38.11)	287.77 (± 69.65)	195.00

Table 4.6: CartPole-v1 results, default configuration

Both strategies in both setups archive and outperform the score required to solve this relatively simple task without major differences between the final scores or structure. A remarkable difference unveils itself when the amount of generation passed until a solution was discovered is examined. The original evaluation strategy on average takes more than 5.31x the generations to reach a solution while at the same time deviating more than 2.24x from that average compared to this works matrix evaluation strategy, portraying it as less reliable and more time consuming. Its reliability increased when taking two simulations into account decreasing the amount of generation passed by 0.59x and the corresponding standard deviation by 0.81x. Increasing the amount of simulations on this works strategy resulted in an 1.22x increase in average generations passed and an 0.84x decrease in standard deviation, displaying increased consistency at a higher base cost. It did not benefit greatly from raising the amount of simulations on this specific task.

Notably the amount of species over all computed evolutions with varying amounts of generations passed, stayed very close to the desired goal of 10 species with on average 9.08 (± 2.42) species per generation even though it was initialized in a way that produced 24.44 species on average in the first generation.

4.2.2 Box2D

The Box2D type of tasks involves controlling a robot in various 2D environments with simulated physics.

LunarLanderContinuous-v2

The LunarLanderContinuous-v2 [16] environment tasks the ANN to learn how to land a moon lander as central as possible without crashing it into the ground. The task is considered solved when 200 or more points are scored on average over 100 simulation runs. The state vector fed into the ANN is composed of eight sensor values plus one bias input that is always one. The outputs are two floats in the range [-1.0, 1.0] indicating the activation of a main engine (upwards thrust) and orientation engines (nudging left/right) respectively.

The experiments were conducted with the chance to add a new node decreased to 0.5%. The weight cap was decreased to 1.0 and the sigma for sampling the weight perturbations was decreased to 0.1. Due to the smaller changes in weight the factor for weight difference was raised to 3.0, the factor for connection difference was raised to 2.0 as new nodes are added less often. The score per ANN was determined over 3 simulations. This configuration produced successful consecutive runs in smaller experiments, less simulations had fairly unstable fitness trajectories.

Table 4.7 reports on the developed structure of 100 experiments on average with its standard deviation. Interestingly the solutions rarely incorporate hidden nodes, indicating that the initial topology is almost sufficient to solve the task once the weights are tuned correctly. Almost all solutions incorporate a recurrent connection which indicates that this task does gain the previously discussed benefit from being able to observe data over time.

evaluation	$\|H\|$	$\|F\|$	$\|R\|$
original	0.08 ($\pm$ 0.27)	18.61 ($\pm$ 2.14)	1.06 ($\pm$ 0.88)
matrix	0.09 ($\pm$ 0.32)	18.66 ($\pm$ 2.33)	1.15 ($\pm$ 1.04)

Table 4.7: LunarLanderContinuous-v2 structure, task configuration

evaluation	#generations	score	goal
original	31.4 ($\pm$ 9.38)	219.99 ($\pm$ 14.76)	200.00
matrix	31.62 ($\pm$ 10.37)	219.18 ($\pm$ 15.95)	200.00

Table 4.8: LunarLanderContinuous-v2 results, task configuration

Table 4.8 reports the generation and score average over the 100 experiments with its standard deviation. It demonstrates that both methods are capable of finding solutions consistently on a more complex task, seemingly without significant differences in their respective results. But when it is recalled that the solutions are mostly absent of hidden nodes, it becomes apparent that the instabilities of the original evaluation strategy do not appear in a simple fully-connected network without hidden layers. This can also be derived from the examples in figure 3.1 and examination of algorithm 2. Regarding the evolved topology, which in this case is the initial topology, both strategies should be expected to behave identical which is what the experimental results show.

4.3 Final Remarks

The experiments proved that the basic capabilities of the NEAT method were preserved and even showed success regarding the species count maintenance while being backed by exact formalization. Further the experimental results regarding the developed "matrix" evaluation strategy give evidence of it being more reliable. This is likely due to the underspecification in the original evaluation strategy which leads to the hypothesised inconsistencies discussed in chapter 3. This potentially slows down the overall evolutionary progress by mutations introducing destructive changes to already developed functionality.

Better individual results for some tasks could potentially be archived by experimenting more with the configuration of the algorithm as well as the training regime i.e. number of simulations per individual, normalization of inputs and interpretation of outputs. Tinkering further with the experiments to archive better scores and running other experiments is interesting and necessary but out of the scope of this work and should be picked up again in future work.

5 NEAT and beyond

Since the invention of the NEAT method roughly 19 years have passed. To acknowledge the progress made in those years and to put this work into perspective some ideas and research in the field will be discussed here.

5.1 Novelty Search

Novelty search [15] has been a new approach that works almost as a drop-in replacement for a fitness metric. Every individual is assigned a behavior characterization represented as some vector instead of a fitness value. Then the sparseness of those vectors is computed as the average distance to their k-nearest neighbors. This sparseness is interpreted as a novelty score. The more sparse a behavior is the less that exact or similar behaviors have been expressed by other individuals, i.e. it is novel. If the novelty score of any individual exceeds a certain threshold its behavior vector is added to an archive which is always taken into account when computing the sparseness. That drives a dynamic where individuals are pushed to explore the behavior space ever more until they enter the intersection of the behavior space and the solution space. The assumption behind this approach is that the novelty gradient enters areas that the fitness gradient would not have explored. Also much of the search space (weights and topological changes) potentially collapses into similar points in behavior space thereby greatly reducing the total space to be explored. An apparent problem is that the behavior space might still be very large and exploring it without any sense of direction but "somewhere new" deems unfeasible, effectively resulting in a random exploration. This challenge has been approached from various directions and a empirical comparison of those approaches can be found in [10].

5.2 Converging vs. Diverging Search

Novelty search advanced the field with respect to so called "deceptive" environments, where strictly following the fitness gradient inevitably leads to dead ends. It shed light on and offered an approach to overcome this fundamental limitation. It inspired further investigation leading up to the inception of Quality Diversity (QD) algorithms [22]. QD algorithms are bound to a behavior descriptor and aim for several, diverse, high quality solutions at the same time, i.e. they diverge. This is analogous to natural evolution where a vast variety of organisms exists with an equally vast amount of different approaches to continuing there existence. It is likely that multiple attractors exist in most search spaces and a traditional genetic algorithm guided by a fitness metric alone is likely to stick to the first one found i.e converge quickly. Potentially even a different attractor is converged to each time the algorithm is run. Conversely QD algorithms try to illuminate all attractors by uniformly covering the behavior space. Notably they also diverge from the assumption that evolution works as a black box optimization. They try to explore other crucial aspects of evolution such as diversity and creativity in the sense that various approaches yield different but working solutions. One such algorithms is MAP-Elites [19]. It articulates a simple mechanism to allow for quality and diversity without those conflicting each other, i.e. they act orthogonally. First the behavior space is segregated into even chunks. Then individuals are placed into those chunks according to their behavior characterization. Diversity is preserved by permanently keeping one individual per chunk (should it have been filled by evolution) and quality is improved by replacing individuals inside chunks should a better performing one be placed in the same chunk. The chunks form the population and the archive at the same time and individuals are randomly selected for reproduction. MAP-Elites makes no assumptions about the individuals.

The goal of continuing ones lineage is very implicit in natural evolution and it is questionable if it is the actual goal or rather just a precondition for playing the game of life. A similar thought is picked up by the Minimal Criterion Coevolution (MCC) [2] algorithm. It removes the explicit behavior characterization mechanism by designing the selective pressure and genetic drift in a way that drives organisms towards desired capabilities and fosters diversity by resource limitation. It does so by evolving two populations at the same time, solvers and problems alike. The solvers mutate to improve their problem solving capabilities, the problems mutate to become more difficult and complex. Those two populations are coupled via a minimal criterion, namely each solver has to solve at least one of any problems, conversely each problem has to be solved by at least one solver. Resource limitation means one problem can only be used a limited amount of times to satisfy a solvers minimal criterion, i.e. not all solvers can solve the easiest problem and continue to exist. This approach is one that incorporates so called open-endedness of the search into its process. As before no assumptions about the individuals are made.

5.3 And now this

Evolution is separable into two parts, namely the representation part spanning the genome and its associated operations (mutations and crossover) and the mechanism part which is responsible for the evaluation and selection processes. This work put its focus on strictly formalizing the genome, computations derived from it and its interpretation as an ANN. Since the genome is the representation aspect, the formalism developed in this work can be mixed and matched with the selection mechanisms of other evolutionary algorithms while improving reliability and explainability. Potential candidates for this are the mentioned MCC and MAP-Elites algorithms.

6 Concluding thoughts and future work

Being unsatisfied by the apparent arbitrariness of ANN topologies and then inspired by the idea of evolving the topology alongside the learning process this work came to be. During research it was realized that significant progress in the field of neuroevolution had been made since the publication of "Evolving Neural Networks through Augmenting Topologies" proposing the NEAT method in 2002. Catching up and not getting lost or distracted along the way became a major focus. Some of the research results inspired chapter 5 and the following conclusions and proposed ideas.

First and foremost evolution is a deceivingly simple mechanic. In terms of Manollis Kellis: "[...] biology is not intelligent, it's just ruthless selection, random mutation." [8], again highlighting the two parts of evolution. Each part, the genome with mutations and crossover and the selection mechanism, can be made arbitrarily complex to model observed and desired behaviors more closely. This task of capturing the evolutionary dynamics encountered in nature is a key part of designing evolutionary algorithms. Current research seems to shifts the focus towards Quality Diversity and open-endedness, as mentioned in chapter 5.

Another major insight gained is that evolving topologies does not alleviate the operator of the ANN from a lot of experimentation as it shifts the complexity of topological choices into the complexity of the hyper-parameters. A partial goal of this works NEAT implementation became to reduce the hyper-parameter surface which succeeded in some areas, notably the automated speciation threshold, but not all. A significant amount of time was spend experimenting then improving the code as well as the configuration triggering new experimentation, resulting in a feedback loop. Some of the discussed shortcomings of the original work were only discovered as a result of this process. The software finally used and presented on Github [4, 3] is thereby the result of several iterations and improvements.

Self-organized critically (SOC) was encountered during research and could potentially become a new building block in evolutionary algorithms and more so in neuroevolution as it seems inherently linked with neural systems [11]. It could serve as a selection or mutation mechanism of sorts as it has been employed to generate search patterns in optimization problems before [13]. This is particularly interesting due to the lack of parametrization when dealing with SOC as opposed to heavily parameterized custom mechanisms. Another perspective could be to investigate existing EAs and probe their results for SOC-like behavior or to design mutations and/or selective pressure to have the evolved neural structures exhibit SOC-like behavior.

The observed differences of the activation strategies is another direction that could be explored in more detail, as strategies potentially lend themselves better towards some type of task indicated by the XOR experiment.

Bibliography

[1] Oludare Isaac Abiodun et al. “State-of-the-art in artificial neural network applications: A survey”. In: *Heliyon* 4.11 (2018), e00938. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2018.e00938. URL: https://www.sciencedirect.com/science/article/pii/S2405844018332067.

[2] Jonathan Brant and Kenneth Stanley. “Diversity preservation in minimal criterion coevolution through resource limitation”. In: June 2020, pp. 58–66. DOI: 10.1145/3377930.3389809.

[3] Silvan Buedenbender. *favannat*. 2021. URL: https://github.com/SilvanCodes/favannat (visited on 02/11/2021).

[4] Silvan Buedenbender. *SET-NEAT*. 2021. URL: https://github.com/SilvanCodes/set-neat (visited on 02/11/2021).

[5] *CartPole-v1*. URL: https://gym.openai.com/envs/CartPole-v1/ (visited on 02/20/2021).

[6] *Classic Control*. URL: https://gym.openai.com/envs/#classic_control (visited on 02/17/2021).

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. *Neural Architecture Search: A Survey*. 2019. arXiv: 1808.05377 [stat.ML].

[8] Lex Fridman and Manollis Kellis. *Manollis Kellis: Human Genome and Evolutionary Dynamics*. July 2020. URL: https://open.spotify.com/episode/6SHos1Idu5VgM9dmlAPMUj?si=YLzBMriMQFm8tQZHoU3b4Q (visited on 12/14/2020).

[9] Adam Gaier and David Ha. “Weight Agnostic Neural Networks”. In: (2019). https://weightagnostic.github.io. eprint: arXiv:1906.04358. URL: https://weightagnostic.github.io.

[10] Jorge Gomes, Pedro Mariano, and Anders Christensen. “Devising Effective Novelty Search Algorithms: A Comprehensive Empirical Study”. In: Jan. 2015, pp. 943–950.

[11] Janina Hesse and Thilo Gross. “Self-organized criticality as a fundamental property of neural systems”. In: *Frontiers in Systems Neuroscience* 8 (2014), p. 166. ISSN: 1662-5137. DOI: 10.3389/fnsys.2014.00166. URL: https://www.frontiersin.org/article/10.3389/fnsys.2014.00166.

[12] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[13] Heiko Hoffmann and David W. Payton. “Optimization by Self-Organized Criticality”. In: *Scientific Reports* (Feb. 2018). DOI: 10.1038/s41598-018-20275-7.

[14] L. Kaelbling, M. Littman, and A. Moore. “Reinforcement Learning: A Survey”. In: *J. Artif. Intell. Res.* 4 (1996), pp. 237–285.

[15] Joel Lehman and Kenneth Stanley. “Exploiting open-endedness to solve problems through the search for novelty”. In: *Artificial Life - ALIFE* (Jan. 2008).

[16] *LunarLanderContinuous-v2*. URL: `https://gym.openai.com/envs/LunarLanderContinuous-v2/` (visited on 02/20/2021).

[17] R. Ma and L. Niu. “A Survey of Sparse-Learning Methods for Deep Neural Networks”. In: *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. 2018, pp. 647–650. DOI: `10.1109/WI.2018.00-20`.

[18] Risto Miikkulainen et al. *Evolving Deep Neural Networks*. 2017. arXiv: `1703.00548 [cs.NE]`.

[19] Jean-Baptiste Mouret and Jeff Clune. *Illuminating search spaces by mapping elites*. 2015. arXiv: `1504.04909 [cs.AI]`.

[20] Keiron O’Shea and Ryan Nash. “An Introduction to Convolutional Neural Networks”. In: *ArXiv e-prints* (Nov. 2015).

[21] Rui Pereira et al. “Energy efficiency across programming languages: how do energy, time, and memory relate?” In: Oct. 2017, pp. 256–267. DOI: `10.1145/3136014.3136031`.

[22] Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. “Quality Diversity: A New Frontier for Evolutionary Computation”. In: *Frontiers in Robotics and AI* 3 (2016), p. 40. ISSN: 2296-9144. DOI: `10.3389/frobt.2016.00040`. URL: `https://www.frontiersin.org/article/10.3389/frobt.2016.00040`.

[23] Andrew N. Sloss and Steven Gustafson. “2019 Evolutionary Algorithms Review”. In: *CoRR* abs/1906.08870 (2019). arXiv: `1906.08870`. URL: `http://arxiv.org/abs/1906.08870`.

[24] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. “Real-time neuroevolution in the NERO video game”. In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 653–668. DOI: `10.1109/TEVC.2005.856210`.

[25] Kenneth Stanley. URL: `http://nn.cs.utexas.edu/?neat-c` (visited on 02/03/2021).

[26] Kenneth Stanley. “Compositional pattern producing networks: A novel abstraction of development”. In: *Genetic Programming and Evolvable Machines* 8 (June 2007), pp. 131–162. DOI: `10.1007/s10710-007-9028-8`.

[27] Kenneth Stanley. *The NeuroEvolution of Augmenting Topologies (NEAT) Users Page*. Oct. 2020. URL: `https://www.cs.ucf.edu/~kstanley/neat.html` (visited on 12/14/2020).

[28] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks Through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. URL: `http://nn.cs.utexas.edu/?stanley:ec02`.

[29] Kenneth Stanley, David D’Ambrosio, and Jason Gauci. “A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks”. In: *Artificial life* 15 (Feb. 2009), pp. 185–212. DOI: `10.1162/artl.2009.15.2.15202`.

[30] Kenneth Stanley et al. “Designing neural networks through neuroevolution”. In: *Nature Machine Intelligence* 1 (Jan. 2019). DOI: `10.1038/s42256-018-0006-z`.

Appendices

A Default parametrization

category	parameter	value
setup	seed	42
	population_size	100
	input_dimension	task-dependent
	output_dimension	task-dependent
	connected_input_percent	1.0
	add_to_archive_chance	0.0
	novelty_nearest_neighbors	0
mutation	new_node_chance	0.05
	new_connection_chance	0.1
	connection_is_recurrent_chance	0.1
	change_activation_function_chance	0.05
	weight_perturbation_percent	0.5
	weight_perturbation_std_dev	1.0
	weight_perturbation_cap	3.0
activation	output_nodes	Tanh
activations	hidden_nodes	Linear, Sigmoid, Tanh, ReLu, Gaussian, Step, Sine, Cosine, Inverse, Absolute
reproduction	survival_rate	0.2
	generations_until_stale	10
	elitism_species	1
	elitism_individuals	0
speciation	target_species_count	10
	factor_weights	1.0
	factor_genes	1.0
	factor_activations	1.0

B Compatibility formula with recurrent connections

Compatibility of genomes

$$(I_1, H_1, O_1, F_1, R_1, f_{\alpha_1}, f_{\omega_{F_1}}, f_{\omega_{R_1}})$$

and

$$(I_2, H_2, O_2, F_2, R_2, f_{\alpha_2}, f_{\omega_{F_2}}, f_{\omega_{R_2}})$$

$$\begin{aligned}
\delta &= \frac{c_1 * (\delta_F + \delta_R) + c_2 * (\delta_{\omega_F} + \delta_{\omega_R}) + c_3 * \delta_\alpha}{c_1 + c_2 + c_3} \\
\delta_F &= \frac{|F_1 \triangle F_2|}{|F_1 \cup F_2|} \\
\delta_R &= \frac{|R_1 \triangle R_2|}{|R_1 \cup R_2|} \\
\delta_{\omega_F} &= \frac{\sum\{\, diff_{\omega_F}(\mathrm{f}) \mid \mathrm{f} \in F_1 \cap F_2 \,\}}{|F_1 \cap F_2| * 6 * \sigma} \\
\delta_{\omega_R} &= \frac{\sum\{\, diff_{\omega_R}(\mathrm{r}) \mid \mathrm{r} \in R_1 \cap R_2 \,\}}{|R_1 \cap R_2| * 6 * \sigma} \\
\delta_\alpha &= \frac{\sum\{\, diff_\alpha(h) \mid h \in H_1 \cap H_2 \,\}}{|H_1 \cap H_2|} \\
diff_{\omega_F}(\mathrm{f}) &= |f_{\omega_{F_1}}(\mathrm{f}) - f_{\omega_{F_2}}(\mathrm{f})| \\
diff_{\omega_R}(\mathrm{r}) &= |f_{\omega_{R_1}}(\mathrm{r}) - f_{\omega_{R_2}}(\mathrm{r})| \\
diff_\alpha(h) &= \begin{cases} 0, & \text{if } f_{\alpha_1}(h) = f_{\alpha_2}(h) \\ 1, & \text{otherwise} \end{cases}
\end{aligned} \tag{1}$$