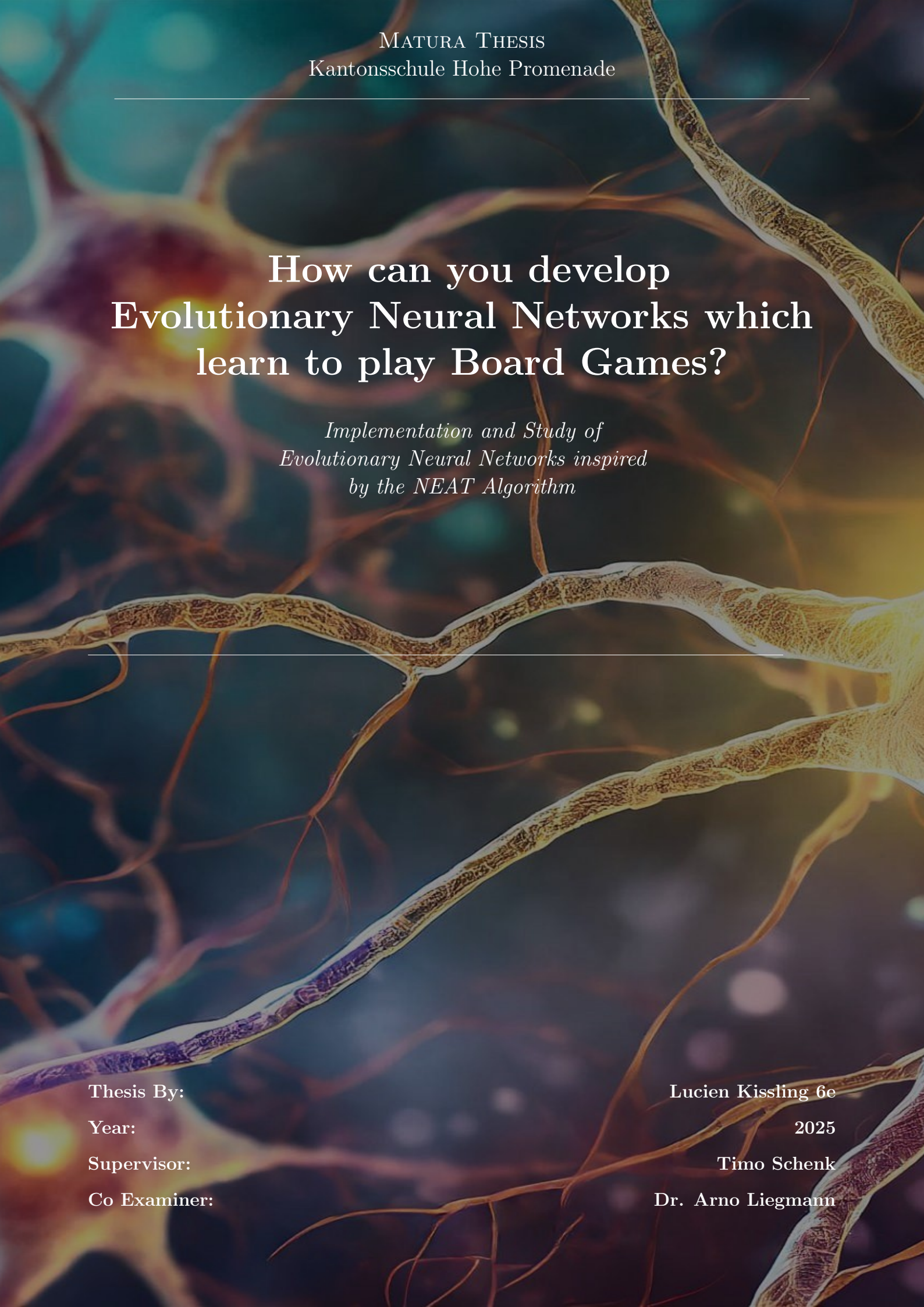
Matura Thesis
Kantonsschule Hohe Promenade

How can you develop Evolutionary Neural Networks which learn to play Board Games?

Implementation and Study of Evolutionary Neural Networks inspired by the NEAT Algorithm



Thesis By: Lucien Kissling 6e

Year: 2025

Supervisor: Timo Schenk

Co Examiner: Dr. Arno Liegmann

Contents

1. Introduction

1.1 Preface

Ever since I got into Computer Science a few years ago, I was fascinated by the Idea of Algorithms that solved various Problems. Therefore, I participated in the SOI (Swiss Olympiad in Informatics) where we were taught everything about developing and programming Algorithms and their Data Structures.

In recent years although, a new field of Computer Science has gained a lot of attention, where those Algorithms are not programmed by humans, but evolved by a computer. This field called Machine Learning immediately got my excitement and two years ago a friend of mine and I had our first practical experience with it. We developed a simple Neural Network, which helped us predict the color of a lego brick in front of a color sensor based on the RGB values in various lighting conditions.

A Neural Network (NN) forms the basis of most Machine Learning Models and I will therefore explain it in much detail in the following Chapters. In simple terms, a NN is a strongly simplified artificial model of the human brain as a NN consists of an interconnected web of Neurons through which information flows and gets computed.

Although the NN we developed two years ago already learned on its self, we still had to provide data for it to learn from. This meant that we had to manually scan the RGB values of the lego bricks and then label them with the color they represented. The aim of my Matura Thesis therefore is to take the idea of self learning a step further by developing Neural Networks, which don't need this kind of data with solutions predefined by humans.

1.2 Thesis Statement

As mentioned in the previous Section, this Thesis will explore the field of Neural Networks (NNs) that learn without data provided by humans, which is called unsupervised learning. The solution this Thesis will focus on are Evolutionary Neural Networks (ENNs), a combination of Neural Networks and Genetic Algorithms.

In this Thesis, I develop my own simplified implementation ENNs and then train them on some Board Games. The research effort consists of testing the ENN algorithm with different Parameters and Features to see how well it can learn to play the games in different configurations. My approach also draws inspiration from the NEAT Algorithm developed by Kenneth O. Stanley and Risto Miikkulainen in 2002, where the NNs start minimally in the first generation and then develop complexity over time.[1] The first game I will train the ENNs on is Nim, a simple game where two players take turns removing matches from different stacks.

In specific, this Thesis aims to answer following questions:

- How can you develop Evolutionary Neural Networks (ENNs) that learn to play Board Games?
- How do different parameters and features of ENNs affect the learning process?
- How does this Implementation of ENNs compare to other Machine Learning Algorithms?

[1] Stanley and Miikkulainen 2002, p.105-106.

2. Background

2.1 Neural Networks (NNs)

As already touched on in the Preface(1.1), an artificial neural network (ANN or NN) is a mathematical model for data processing, initially inspired by the structure of the brain[1]. Therefore, we will first have a brief look at the functioning of a brain.

2.1.1 Neurons in the Brain

Inside the brain, around 86 million neurons[2] form connections to each other through which they activate other neurons. In a neuron, the signals of connected neurons add up and when they reach a certain threshold, the neuron is activated and fires a signal to its own connections[3]. The neuron then resets after a certain amount of cooldown time. With this web of neurons inside the brain, animals can process information from nerve signals from the body and output them again as nerve signals instructing the body.

2.1.2 Feed Forward Neural Networks (FNNs)

Now how do we apply those ideas about Neural Networks learned from the biology of a brain to a program that runs on a computer? The first step is to simplify the chaos of neurons in the brain into layers of neurons. We get an input layer, an output layer and optional so-called hidden layers in between. As a next step, in each layer, we connect its neurons to neurons of following layers, typically exclusively to neurons of the next layer.
Now, the structure of our network looks something like this:

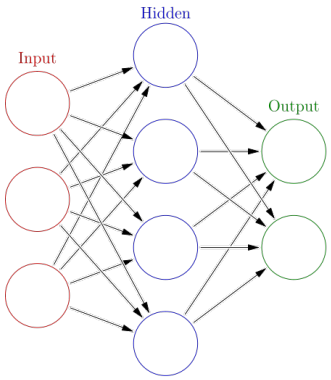


Figure 2.1: A simple Feed Forward Neural Network (FNN) with three layers: Input, Hidden and Output.[4]

The Structure of a NN is also called Topology.

Neural Network Topology. *The Topology of a Neural Network is its distinct arrangement of neurons, layers and the connections between the neurons.*

[1] Chandra 2022.
[2] Caruso 2023.
[3] Newman 2023.

For the neural network to perform functions, we also need to assign a weight to all the connections. This weight value ranges from -1 to 1 and can be thought of as the strength of a connection between neurons.
In terms of computer science, this structure now resembles a directed, weighted graph, which is why we will call the neurons nodes and their connections edges from now on.

Now let's see, how information gets computed by a neural network by using the example of a computer vision NN, that recognizes digits on a black/white image: First, we need a way to encode the information into values for the nodes of the input layer, in our example the Brightness values of the single pixels. We assign these values to the nodes of the input layer.
Then, for each node of the input layer, we look for all edges connected to that node. For each of these edges we multiply its weight by the value of the input node and add the result to the other node connected by the edge.
After iterating through all the nodes of the one layer, we move the next layer. Now, the value of the nodes in this layer is the sum accumulated by the value of all connected nodes multiplied the weight of that connection:

$$v_x = \sum_{i=0}^{N} v_i * w_i \tag{2.1}$$

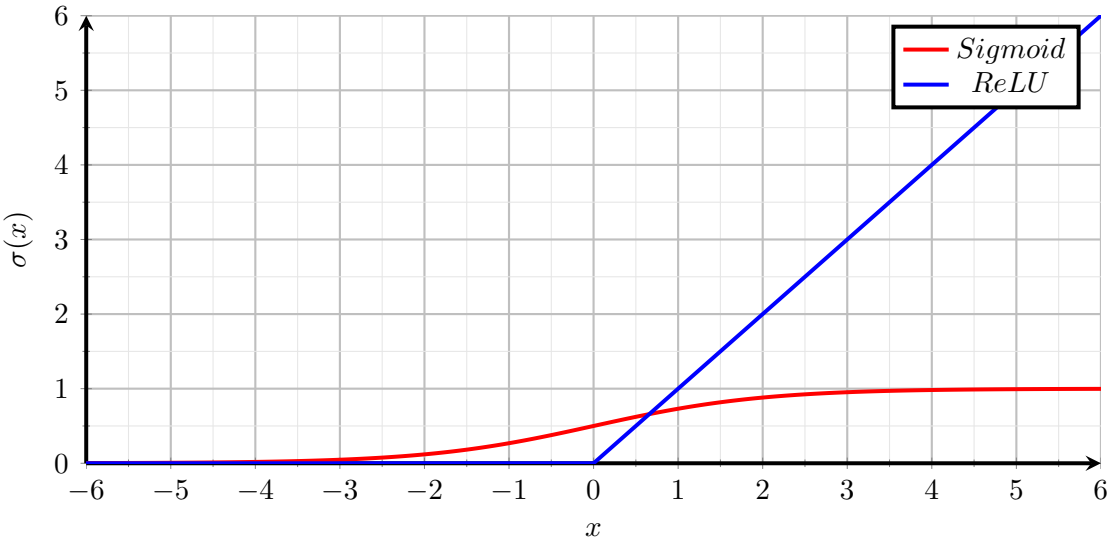
Where:

- *N is the number of connected nodes*
- *v_x is the value of the current node*
- *v_i is the value of the current node*
- *w_i is the weight of the edge connecting the current node to node i*

Additionally, we can add a bias value r_x ranging from -1 to 1 onto the value of the nodes. Afterward, an activation function is applied to the value of the nodes to fit the value of the node inside a preferred range. This activation function can also be thought of as the threshold of stimulation for a neuron to fire. Two examples for activation functions are:

- Sigmoid Function: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Rectified Linear Unit (ReLU): $\sigma(x) = \max 0, x$

Plot of the activation functions:



The final function for the value of one node is therefore:

$$v_x = \sigma\left(\sum_{i=0}^{N} v_i * w_i + r_x\right) \tag{2.2}$$

Where (additionally to equation 2.1):

- r_x *is the bias of the node*
- σ *is the activation function*

Now this value v_x can be used in upcoming layers. When we reach the final layer after iterating through all the nodes and layers of the NN, we can read out its computed output. This output again is encoded in node values like the input and therefore needs to be decoded for the result. In the example of a digit detecting NN, we could encode the output with 10 output nodes, each representing one digit. The result could be decoded by using the output node with the highest value as result. This kind of output encoding, where all possible result get their own node is called One-hot encoding. With this kind of output encoding, the values of the output nodes can be interpreted as a certainty value for a specific result to be correct.

2.1.3 Remark: Functioning of NNs

Now that we have established how a NN works, it is important to understand why this kind of algorithm is revolutionary to computer science.

An algorithm is a set of functions applied in a certain order onto an input to receive a result. Traditionally, this algorithm was programmed with the help of mathematical operations, logic functions, loops, system functions and data structures, which are then converted into binary code for the processor. Of course, this also applies in the context of NNs, however, there is also a third layer of abstraction on top, which simulates the functioning of a brain with neurons. Since the traditional algorithm only enables the neurons in the NN, the third neuron-based layer is what computes the actual function of the NN. Therefore, NNs resemble more the functioning of a brain than a traditional algorithm. This difference results in implications for the functioning of an NN: Traditional algorithms represent actual mathematical calculations and are therefore deterministic. With NNs, the algorithm is based on different parameters for nodes and edges, which make it an unpredictable blackbox. Additionally, NNs don't represent actual functions but try to approximate them instead. This means it is hard to prove a neural network to be always accurate and is the reason why we often call the result of NNs its prediction.

2.1.4 An Example of NN learning: Backpropagation

We already know how a NN can make a prediction for a given input with the help of parameters, which encode for the structure and weights/biases of the NN.But how do we find such parameters for a given problem? The answer is to use machine learning, which trains the model to perform a certain task with the help of training data.

Machine Learning. *Machine Learning is a subset of artificial intelligence that involves the use of algorithms and statistical techniques to optimize artificially intelligent models for a given problem.*

One machine learning algorithm for NNs is called backpropagation, which is one of the simplest and most efficient ways to train a NN.

Backpropagation. *The backpropagation algorithm is an algorithm that can optimize parameters for weights and biases of a NN within a fixed topology for a certain problem with the help of labeled data.*

Backpropagation uses a training and test set containing unlabeled data for a problem. But what does a test set with unlabeled data mean in the context of machine learning?

(Un)labeled Data. *Labeled data for a machine learning problem is a set of data with sample problems and the respective solutions. Unlabeled data instead only contains the sample problems.*

The Machine Learning Process starts with an NN with fixed topology and random weights/biases. Then, you give the NN the training problems, whose predictions will be random at first. However, with the help of the corresponding solutions, you can refine the predictions of the NN by adjusting the parameters in a way that would let the NN make the correct prediction for the given problem. This is done by starting from the prediction of a certain problem its solution and propagating back through the whole NN until reaching the input layer. More specifically, you compute a cost function of The NN shouldn't be adapted only to a single problem but should be able to make accurate predictions for the whole training set as well as for unknown problems. To therefore prevent overcorrecting a NN for a single problem, we have to factor in a learning rate significantly smaller than 1 onto our correction. No we can train the NN on the whole training set for many generations, until the NN starts to make accurate predictions for the whole. To test the performance on unknown problems we can use a separate test set, which the NN wasn't trained on.

2.2 Evolutionary Computation (EC) & Genetic Algorithms (GAs)

Now we will have a look at the Machine Learning Technique, this Thesis focuses on. Again, we will draw inspiration from Nature:

Evolutionary Computation. *Evolutionary Computation (EC) is an Algorithm that optimizes a set of parameters for a problem with the help of natural selection.*

One scenario for EC might be for example if you have a large set of data points of an unknown polynomial function with noise and outliers. If you want to find the underlying polynomial function, you could then employ Evolutionary Computation to find the best fitting parameters for such polynomial. So how do you find these parameters?

You start with an initial population of agents with a random set of parameters. For each generation of your population you will then repeat following steps:

- Fitness: First, you need to find out how good each agent (and its parameters) performs in the function they try to optimize for. The fitness evaluation can be done with an objective cost function or a competition between the agents, which evaluates their relative performances. In our example, the cost function might be the sum of the absolute differences between the data points and the output of the polynomial function.
- Selection: As a next step, you rank the agents performances to then pick the ones that have performed the best.
- Reproduction: The best agents pick is the part of the population that survives. These agents will then be replicated by copying their parameters (non-mating) of by merging parameters from different agents (crossover).
- Mutation: The agents parameters will then be mutated by either completely overwriting certain parameters with new random parameters or by shifting the existing parameters by a random number.

After a certain number of generations, you will then have the best performing set of parameters for a given problem. There is also one popular addition to EC, which draws inspiration from nature again:

Genetic Algorithm. *A Genetic Algorithm (GA) is an implementation of EC that uses a genetic representation of the parameters.*

Its main idea is to have genes that only encode indirectly for the parameters, and that can be on and off.

2.2.1 Gradient Descent & Local Minima

We know from the previous Chapter(3.1.3) that EC starts with an initial population of agents with random sets of parameters, which all have a cost determined by the cost function. Agents with relatively low cost survive the round and then get replicated a few times, mutated and again selected by cost.

This whole process strongly resembles a ball rolling down a hill in some terrain. On any given point on the terrain, de ball will roll down to the next neighboring point that is the lowest of all. Now, in EC the parameters for a function can be imagined as the horizontal coordinates for the ball meanwhile the cost of these parameters is the height at their location. The different mutations have parameters close to the original and can therefore be thought of as points close to the original in our terrain. And as the ball rolls down to the lowest of all neighboring points in any moment, EC again chooses the set of parameters with the lowest cost every cycle. This learning method is called Gradient Descent, as the algorithm evaluates all neighboring points and then follows this direction with the lowest cost and therefore steepest gradient.

So what does the ball example tell us about how EC learning works? Imagine a hill with each a shallow and a deep valley next to it. Depending on which side of the hill you place the ball in the beginning, it will roll in a different direction and end up in either of the two valleys with different height. From this we can take away following things about EC learning:

- (Minor) differences in the starting position can result in large differences in the end position and the respective cost.
- Once in reached a point with no downwards gradient, innovation halts for that agent, even if there is a lower cost point somewhere. Such points are called local minima.

The impact of starting position and local minima often pose a large problem for EC applications. Strategies to reduce the impact of such, are therefore essential for the Algorithms success.

2.3 Evolutionary Neural Networks (ENNs)

After having covered all the basics about NNs and EC, we can now combine those concepts to create the algorithm, that this thesis is about.

Evolutionary Neural Networks. *Evolutionary Neural Networks (ENNs) are Neural Networks that use Evolutionary Computation to optimize for its parameters for the NNs weights/biases and its topology.*

ENNs are useful for complex machine learning problems and also work for unlabeled training data as long as there is a fitness function. Let's bring our knowledge about EC into the context of Neural Networks: The parameters ENNs encode for are for the weights, biases, nodes and edges of a Neural Network. You could either directly encode the NNs parameters as a graph or indirectly as genes, which would make it a Genetic Algorithm. ENNs still start with a random population of agents that are evaluated and selected by a fitness function. Then they are replicated (with crossover or non-mating) and finally mutated in following ways:

1. Change weights/biases: A new random value is set for the weight or bias of a random existing edge or node.
2. Shift weights/biases: The value for an existing weight or bias of a random existing edge or node is altered by a random change but is kept close to the old value.
3. Add edges: A new edge is added between two random unconnected nodes.
4. Add nodes: A new node is added in between of a random existing edge. The old edge is removed and two new edges with random weight are added between the new node and the two other nodes each.

5. Remove nodes/edges: A random node or edge in a way that doesn't cut off the input from the output layer.

2.4 Drawing Inspiration from the NEAT-Apporach

As already stated in the Thesis Statement(1.2), Neuroevolution of Augmenting Topologies is a ENN machine learning algorithm by Kenneth O. Stanley and Risto Miikkulainen in 2002[5]. One of its main innovations is that the initial population starts with NNs of lowest complexity and only increase topological complexity as its useful for the problem. In specific, you start with NNs that have only the nodes of the input layer connected to the nodes of the output layer. This should result in more efficient NNs for the problem, as the complexity is only increased when it improves the NNs performance. The NEAT algorithm therefore only needs the mutations 1. to 4. showed in the last chapter(2.3) and doesn't need to remove complexity (mutation 5.)as it is already minimal.

2.5 Games

Let's also have a quick look at the games, this Thesis will train the ENNs on.

2.5.1 Simple Nim

This game is a simplified version of Nim, where two players take turns removing an arbitrary amount of matches from a single stack with some number of matches. The player who removes the last match loses. Therefore, the winning strategy simply is to remove all matches from the stack but one, which forces the opponent to remove the last match, which makes them lose.

2.5.2 Nim

This game works similarly to Simple Nim (described in the last Section(2.5.1)) with the difference that there are multiple stacks with matches. Now, the player who removes the last match from the last unemptied stack loses. You can read up the winning strategy for Nim on Wikipedia[6]. The winning strategy is much more complex, which forces the ENNs to learn a more complex strategy.

2.6 Related Work

The field of AI research in ENNs is largely studied and is often linked to games. In the following table, we can find some examples of ENN models that have been tested on games:

Author(s) & Year	Model	Game/Benchmark	Computation	Accuracy
Stanley and Miikkulainen 2002	NEAT	Double Pole Balancing With Velocities	3600 evaluations	100%
Qader, Jihad, and Baker 2022	NEAT	Dama	>5000 generations	81.25% (wins against humans)
Richards, Moriarty, and Miikkulainen 1998	SANE	Go	260 generations	>75% (vs Wally, 9×9 board)
Konidaris, Shell, and Oren 2002	Custom ENN	Capture Game (subgame of Go)	100+ generations (distributed)	No significant progress yet
Orlov, Sipper, and Hauptman 2009	Genetic ENN	Backgammon	256 pop, 100–200 generations	62.4% (vs Pubeval)

Note: Although the amount of fitness evaluation is a more accurate representation of computation load than the amount of generations, in many cases the amount of fitness evaluations isn't indicated and cannot be calculated.

[5] Stanley and Miikkulainen 2002, p.105-106.
[6] contributors 2023.

3. Building my ENN

3.1 Algorithm Design

Now, we will have a detailed look at how I used the concepts described in the last Chapter(1) to build an Evolutionary Neural Network that plays board games. First, I will explain the general Structure of my Code:

3.1.1 Overview

As already mentioned in the Thesis Statement(1.2), this Project is written in the Rust Programming Language. If you don't understand the syntax of the code snippets, you can consult the online "The Rust Programming Language" booklet under: https://doc.rust-lang.org/book/

My Codebase is divided into different modules as well as a bin folder with main files, one of which has to be selected to be executed at the beginning. The other modules are divided into one module with the ENN algorithm, the other modules handle the different games, the ENNs learn to play. The ENN module is divided into two files:

- agent.rs: This file handles the NNs and the mutations on the NNs
- population.rs: This file handles the natural selection process and the data saving

The game modules provide the problems for the NNs handle the predictions of the NNs and finds the new game state after a move. It might also include an objective performance evaluation function to measure how well the NNs perform. I will now explain all the functions bottom up, starting with the Neural Network. I work with Rusts Structs which are similar to classes in other programming languages.

3.1.2 Neural Network

The struct *NeurualNetwork* (defined in agents.rs) most importantly contains a two-dimensional vector of nodes, which encodes all the information of the NN:

```
pub struct NeuralNetwork {
    [...] // redundant side data about the NN
    pub nodes: Vec<Vec<Node>>,
}
```

The Vector inside (`Vec<Node>`) represents a layer of the NN and the outside Vector (`Vec<Vec<Node>>`) contains all layers of the NN. A *Node* contains its bias and a vector for the incoming and outgoing edges:

```
pub struct Node {
    pub bias: f64,
    //edges stored in an adjacency list
    pub incoming_edges: Vec<Edge>,
    pub outgoing_edges: Vec<Edge>,
}
```

An *Edge* contains the weight of the edge and its input/output node:

```
pub struct Edge {
    input: [usize; 2],
    out: [usize; 2],
    weight: f64,
}
```

The most important functions of the *NeuralNetwork* struct are:

- `new(input_nodes: usize, output_nodes: usize) -> Self {}`, which initializes the NN with all input and output nodes connected to each other with random weights and biases.
- `predict(&self, input: Vec<f64>) -> Vec<f64>`, which computes the output of the NN for a given input with forward propagation algorithm described in the last Chapter(2.1.2).
- `new(input_nodes: usize, output_nodes: usize) -> Self {}`, which computes the neuron activation described in the last Chapter(2.1.2). I use a variant of the ReLU function, since the linearity for inputs above 0 of the ReLU function allows for more variety of neuron activation, which makes the NN more flexible and efficient. The variant I use is the ELU function, which additionally tackles the problem of dead neurons in ReLU by allowing for negative values.
- All mutation functions for the NN, which we will need in the next Subsection(3.1.3).

3.1.3 Mutation

The mutation of the NNs is handled in the *Agent* struct (defined in agent.rs), which includes a NN, its fitness and its Rank:

```
pub struct Agent {
    pub nn: NeuralNetwork,
    pub fitness: f64,
    pub rank: isize,
}
```

The function `mutate(&mut self, mutations: usize) -> Self {}` has a number of mutations as a parameter which it applies to the NN of the Agent. For each mutation, it randomly selects one of the following mutations: Change weights/biases, Shift weights/biases, Add edges, Add nodes. All of these mutations have already been described in the last Chapter(2.3), although there are some implementation details to mention:

- When inserting a new node between two connected nodes (mutation 4.), the inserted node will be added to a random layer in between the previously connected nodes. If the previously connected nodes are in neighboring layers, we create a new layer in between for the new node.
- The shift mutation adds to the initial value a random float in the range 0.0 to 1.0 squared with random sign: $v_{new} = v_{old} + rand(-1, 1) * rand(0..1)^2$
- random weighted selection

3.1.4 Evolutionary Computation

The EC process starts with an initial population of Agents with new, minimal NNs created by the `NeuralNetwork::new()` function (described in 3.1.2). As we already established before (see 2.2), we now repeat the steps of performance evaluation, selection and mutation.

Competition and Fitness

In my case, the fitness evaluation works by letting the agents of a population compete against each other in the games. However, I can't let every agent play against all other agents, since the needed time increases quadratically with population size. My solution therefore is to use my circular Pairing algorithm. This algorithm, I first define a number of opponents each agent needs to play against. For each of those games, I generate a new distance, where I now loop through my list of agents and pair agent i with agent i + distance. I also check before, that the distance isn't a multiple of any distance we had previously, since this would result in the same agents being paired twice.

Now, we need to evaluate the fitness of the agents with the fitness function. The main idea for the fitness function is to simply count the number of wins an agents has made during the competition. Of course, there are other possibilities to evaluate the performance of the agents that may represent an agents performance more accurately. The reason why I still first try to use the number of wins as fitness is because it is a generally applicable function to any game. Counting the number of wins doesn't require a deeper understanding of the game, which is very useful in games that are indeterministic. This also enables the ENNs to find their own new strategy for the game, which is finally the goal of AI training.

Natural Selection

After the fitness evaluation, we now can test the performance of the generation (in the next section 3.1.6) and then generate the new population. The new population is made up by some portion of each of following:

- Agents from the last generation: The main part of the new population will be drawn from agents from the old generation with high fitness. This works by drawing all needed agents randomly with the probability of an agent being drawn being its fitness. We can also raise the fitness to some power to either allow more or less survival of non top-performing agents.
- Random agents from the last generation: Some fraction of the new population is made up of randomly selected agents from the last generation, which might help counter a population with the top performing agents stuck in local minimum.
- New random agents: Another fraction of the new population is made up of newly generated random agents, which might help counter overly complex NNs and also local minima in a population.

Now that we have generated our new population, we can start over the whole process.

3.1.5 ENN Parameters

As already mentioned in the Thesis Statement(1.2), my research involves testing my implementation with several configurations of different parameters and then evaluate their performance with the stats described in the following section(3.1.6). Here is a list of all possible parameters influencing my ENNs:

- General: Size of the population
- Game: Initial state of the game, game function that decodes the NN prediction and executes the move on the current state
- Competition: Number of opponents per agent
- Selection: Fitness exponent, share of old best agents, random agents from last generation and new random agents making up the new population besides agents selected by fitness
- Mutation: Min and max amount of mutations per agent, weight/probability of the different mutations
- Evaluation: Number of games from the best agent against old best agents

For each test, we then save a file with the information about the value of all these parameters.

3.1.6 Performance Tests

To see how the Agents perform in the games, we need to track some stats and games. We save following data every generation after the competition phase:

- The fitness of the best agent: This stat shows us, how much better the best agent performs relatively to the average agent
- Wins against older generation: We pair the best agent against an evenly distributed set of the best agents from past generations to track relative performance to the past. As long as this number is above 50%, we should expect improvement over older generations.
- Objective grading: This stat is the performance evaluation of the best agent based on some algorithm that either plays mathematically perfect or (in non-deterministic games) is generally accepted to play well. It is important to note that we primarily don't use this algorithm for the fitness function, because of the points previously mentioned in section 3.1.4.
- Number of turns: We track the average number of turns the agents needed while playing their competition games. In Nim, low turn count generally means better play.
- Average amount of hidden layers and nodes per hidden layer: We track these stats to see how complex the topologies of our NNs in the population are.
- Best agent layer sizes and edges: We track these topological stats to see how complex the best solution is.

- Best agent games: We track the whole game turn for turn played by the best agent from the current generation against each the best agent from the last generation and a randomly selected best agent from previous generations. With this game logs, we can see what moves our agents make in real games and try to grasp their tactics and strategies.

3.1.7 Nim

For the nim game, I implemented a few different game modes as well as objectively perfectly playing dynamic programming algorithm. Let's first look at the general idea of how a game is played.

General Approach

Each game starts with a first game state, which is derived from the initial state parameter. A game state is a list where the length is the amount of stacks in that game and each value v_i stands for the amount of matches on the i-th stack. Then, the competing agents take turns playing their moves. In one turn, the agent whose turn it is, reads the current game state as input and then makes a prediction for its move. A move indicates the stack where the matches are removed as well as the number of removed matches. Then the game computes the new game state after the move or ends the game if all stacks are empty.

Encoding and Decoding

Our input is directly encoded into the input layer of the NNs, which means the input layer has the same size as the state. The values of the input nodes are directly copied from the game state list and therefore also represent the amount of matches on the i-th stack.

For the output, I implemented two different ways of encoding.

- Direct encoding: We have two output nodes where the value for the first represents the index of the stack and the second value represents the amount of matches that are removed.
- One-hot encoding: The first N output nodes encode for the index and the following nodes encode for the amount of matches removed. For both the nodes encoding for index and amount the node with the highest value is finally used as output. The output layer therefore has length $l = N + \max(state_{initial})$, where N is the amount of stacks and $\max(state_{initial})$ is the highest possible amount of matches on a stack.

Direct encoding is the more straight forward approach with lower NN topology whereas one-hot encoding ensures that the output stays in a reasonable range.

Game Modes

There are game modes Simple Nim and Nim with some configuration possibilities:
In Nim, each stack of the initial state can hold matches whereas in *Simple Nim*, the value of all stacks but one randomly chosen stack is set to 0.
The stack sizes in the initial state equal the corresponding value from the initial state list parameter except for the configuration *random*, where the stack sizes of the initial state are randomly selected with the corresponding value in the initial state list parameter as upper bound.
Another configuration involves the output decoding. With strict grading, illegal moves lead to a game loss, whereas in safe grading, the closest legal move to the illegal prediction is played.

Objective Grading

To have an exact measure of performance, I implemented a Dynamic Programming (DP) algorithm that knows the best moves for each given state. It works by first defining a base case, which in case of Nim is every stack being empty and the result a loss. Then, it starts with the initial state and tries out all possible moves using recursion. For each state it tries to find a move that forces the opponent into a losing position. If it can find such move, the position is winning, but if all moves lead to winning position for the opponent, the algorithm returns the position with the highest minimum moves for the opponent to win. It also stores the result for each already computed state, so that all other paths

that lead to that position can use the precomputed result.

Now, that we know for all positions if they're winning, we can use this to grade moves of our agents: For each move, an agent gets one point if

- the agent had a winning state and made a move that got the opponent into a losing state,

or if

- the agent had a losing state and made a move that required the maximal amount of moves for the opponent to win.

The sum of all points an agent has collected is then divided by the amount of moves it has made to get the final performance score.

3.2 First Findings

Now we can test our ENN algorithm with the games and different parameters. The general approach is to start by training the ENNs on easy problems whilst observing the effect of different parameters on the result. We therefore start with the game Simple Nim. I will walk you through my tests and provide the most important information. You can gain insight into the full data about parameters and results on my GitHub.

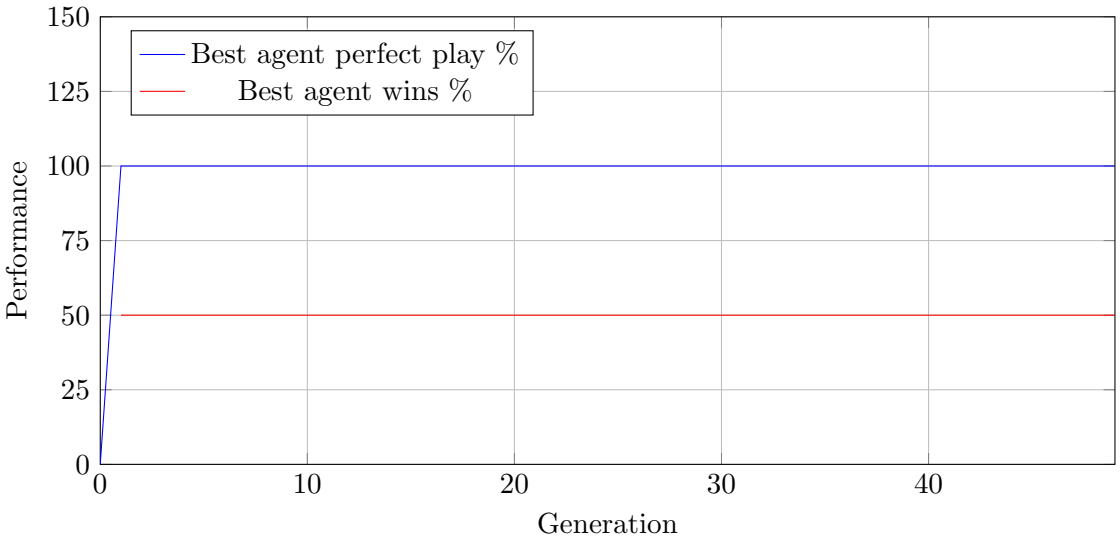
3.2.1 Simple Nim

For the Simple Nim game, I will run each test for 1000 generations, since ENNs should be able to solve this simple game quite quickly.

Stack: 3, Configuration 1

First we test the ENNs on a problem where the only stack contains 2 matches. This is the easiest problem we can still consider a game - with one match per stack the only one legal move, so it doesn't involve any decision-making. The winning strategy of course is to only take one match and leave the opponent with only one match on the stack. We start our first test with a relatively small population of 100 agents and try to simplify all other parameters:
`run_nim_strict`, initial state: `[2]`, population size: `100`, comp games: `50`, mutation min: `0`, mutation max: `1`, fitness exponent: `1`, best agent share: `0`, random agent share: `0`, random old agent share: `0`, best agent tournament games: `50`, add_connection_rand: `1`, add_node_rand: `1`, change_weight_rand: `1`, change_bias_rand: `1`, shift_weight_rand: `1`, shift_bias_rand: `1` It is also important to note that we start with one-hot output encoding, since this means that the output stays in the desired range.

Now, we'll run it and see how it goes:



As we can see, the best agent already plays the game perfectly after one generation. We can also observe this in our saved games, where the best agent correctly plays the move [0, 1] (first value is the stack, second value the number of matches removed):

```
1   Generation: 1
2   Best agent layer sizes: [1, 3]
3   Agent 1 vs Agent 0:
4   Turn: 0: state: [2], agent_move: [0, 1]
5   Turn: 1: state: [1], agent_move: [0, 2]
6   Game result: [1, 0]
```

The best agent wins percentage in the above plot(**??**) is a steady 50%, since all games are played both ways and the first agent always wins the game. We can also see, on line 2 of our output, that the topology didn't evolve for our best agent, which makes sense since it is a very simple task.

Stack: 8, Configuration 1

Now, since the stack size with 2 has already worked fine, we will now try the same configuration but with an increased initial stack size 8. In this configuration, my four tests produced following results:

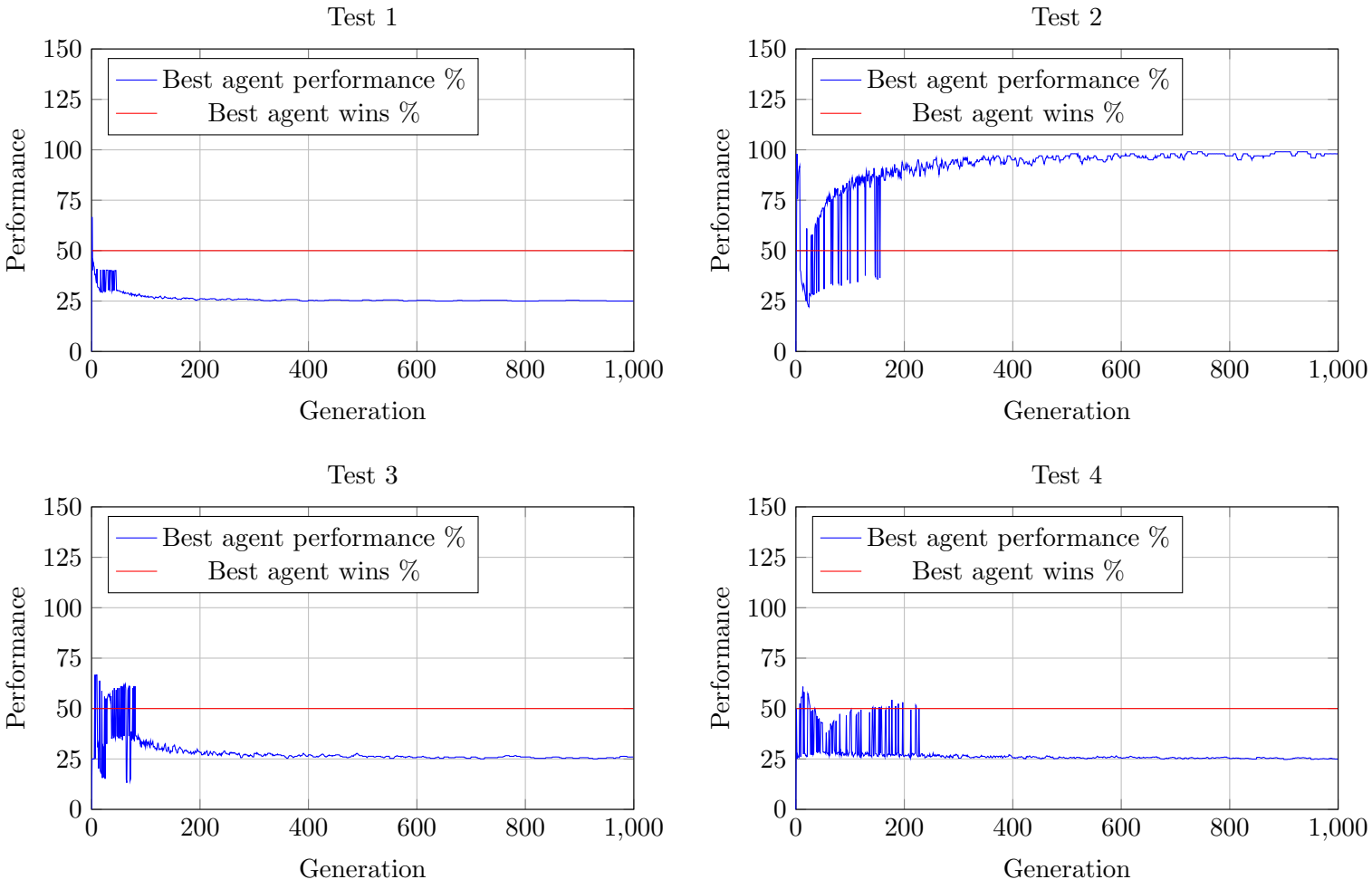


Figure 3.1: Performance metrics over generations in a 2x2 grid.

In only one of four tests did the ENNs learn how to play the correct move and this also only after over 100 generations. The other tests ended in the best agent performance settling on 25%. If we look at the games the NNs played, we also see how we get to this number:

```
1   Agent 999 vs Agent 998:
2   Turn: 0: state: [8], agent_move: [0, 1]
3   Turn: 1: state: [7], agent_move: [0, 1]
4   Turn: 2: state: [6], agent_move: [0, 1]
5   Turn: 3: state: [5], agent_move: [0, 1]
6   Turn: 4: state: [4], agent_move: [0, 1]
7   Turn: 5: state: [3], agent_move: [0, 1]
```

```
    Turn: 6: state: [2], agent_move: [0, 1]
    Turn: 7: state: [1], agent_move: [0, 1]
```

Out of 8 moves, only the last two moves are perfect. Such performance is remarkably bad since all the NN has to do is to always output [0, 7], where the first parameter is set anyway, and it only has to get a hang of the 7. In a sample of 100 agents, we would expect there to be an agent that outputs the correct prediction. All of this seems contradicting, especially if we consider that the NNs were able to find the right move with 2 matches instantly. However, there is one important detail to consider in this test: If the starting agent doesn't play perfectly in the first move (it subtracts fewer than 7 matches), the second agent instantly gains a winning position and would have to play the move N-1 matches. Therefore, we either get lucky and end up with an entire population of agents that always subtract 7 matches, or you need agents that are adapted to all cases. Our final goal is to train the NNs to be like the latter option whereas the former solution seems more like a local minimum which is therefore undesirable. We will now change the stack to have a variable amount of matches, since this simplification turned out to be a hindrance.

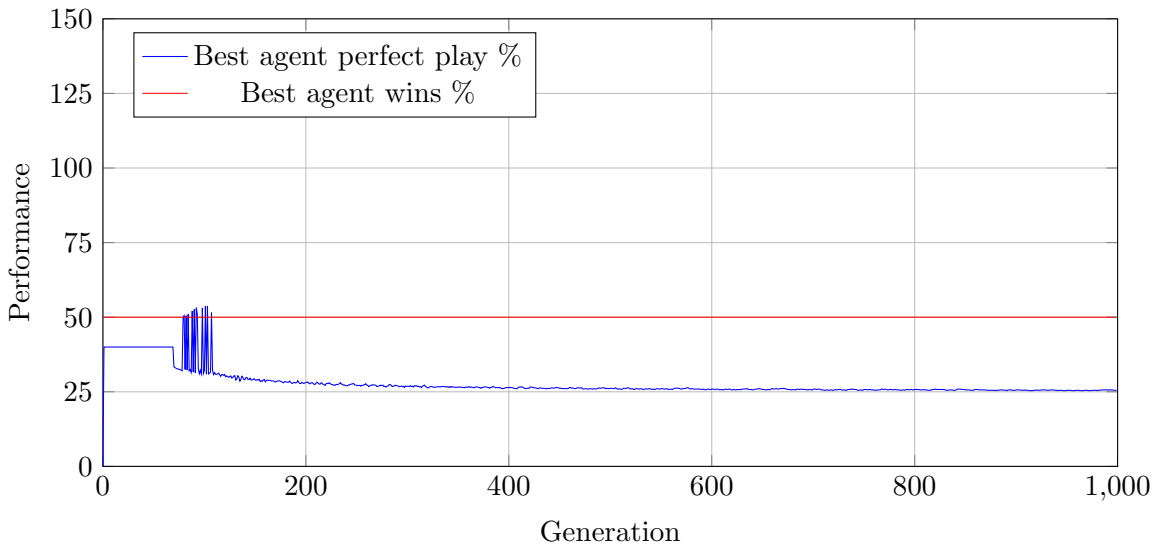
Stack 1–8, Configuration 1

We will repeat the test

HI

Stack 8, Configuration 2

To eliminate agents , we will now try to give better performing agents higher chance of reproduction. We can do this by increasing the fitness exponent from 1 to 2. Let's see how the ENNs do:



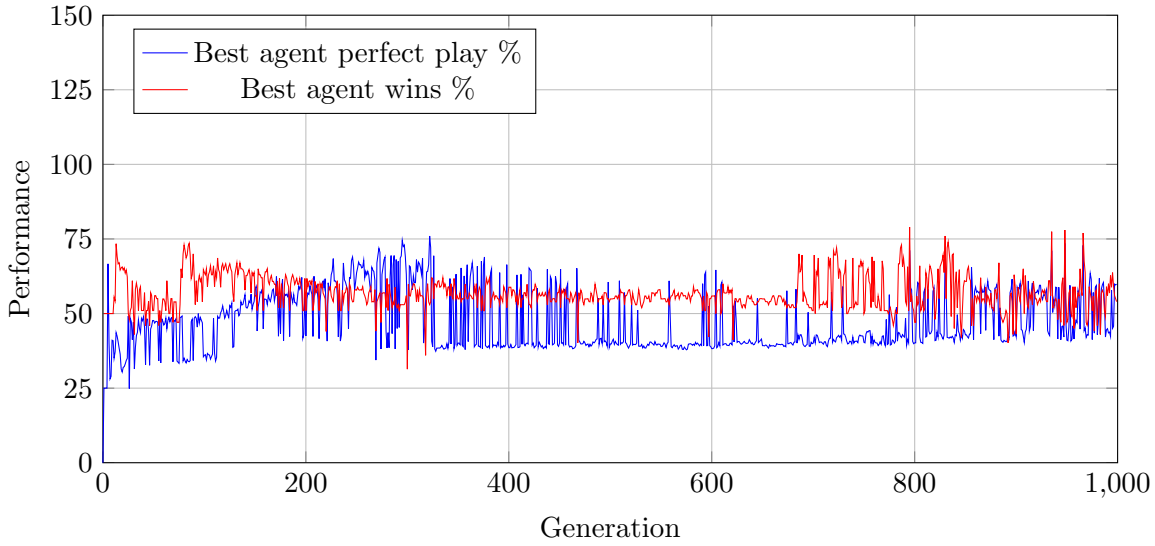
The best agent average performance stalls again at 25%, which means nothing has changed to before. A quick look at the game output shows what happened:

```
    Generation: 997
    Best agent layer sizes: [1, 9]
    Agent 99 vs Agent 997:
    Turn: 0: state: [8], agent_move: [0, 5]
    Turn: 1: state: [3], agent_move: [0, 2]
    Turn: 2: state: [1], agent_move: [0, 2]
    Game result: [0, 1]
```

The best agent seems to have got stuck in a local minimum where the agents play the move [0, 5] when the stack size is 7. This might have originated in an early generation, where an agent with this move pattern and large fitness, which was amplified by the fitness exponent, took over the whole population. Fortunately, we have some strategies ready to use, which lower the risk of local minima. It is also important to note that the other two times I repeated the experiment, the ENN once learned the right move as quickly as in the first test and the other time learned it even faster.

Stack: 8, Configuration 3

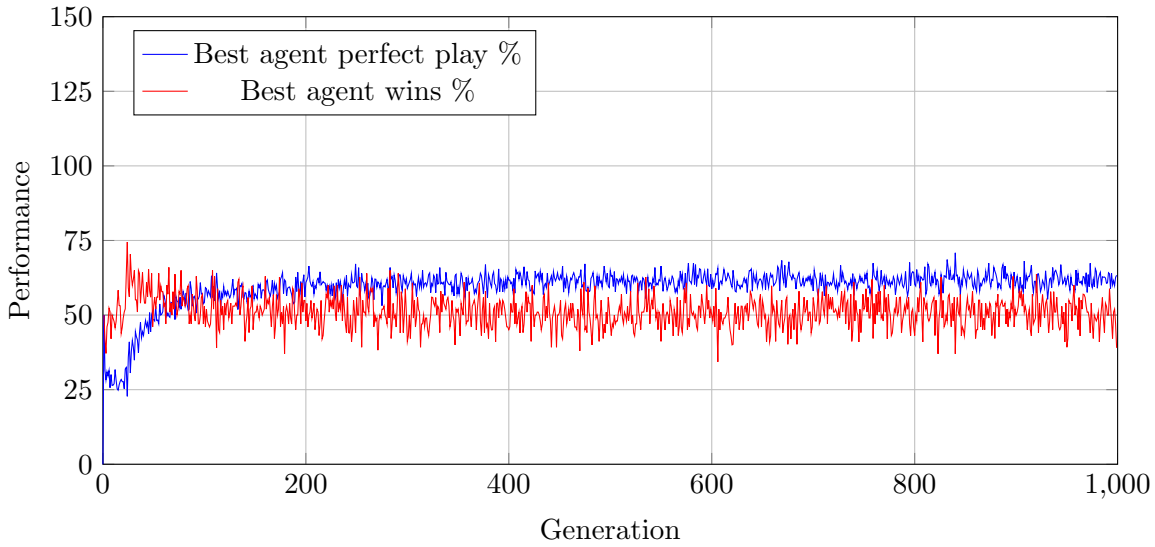
One way to combat the problem of local minima, is to base the composition of new generations not only on old agents with great performance, but also on new random agents, old best agents, and random agents from the previous generation. We will set the share for each of those to 15%.



Et voilà! The ENNs now learned the game in less than 30 generations. An interesting to note is that we get a spike down to 50% twice. These might be due to a bad agent being successful during evaluation since it randomly got paired with other bad agents.

Stack: 1–8, Configuration 1

Now, we'll try the same configuration but with a random amount of matches between 1 and 8 on the stack with in the initial state. The winning strategy would be, to always play the amount of matches on the stack minus one, which forces the opponent to take the last match from the stack.



Now, the performance stalls somewhere above 60%, which isn't optimal. Again, a look at the games themselves helps us understand what happened:

```
Generation: 4
Best agent layer sizes: [1, 9]
Agent 1 vs Agent 4:
Turn: 0: state: [5], agent_move: [0, 1]
Turn: 1: state: [4], agent_move: [0, 1]
Turn: 2: state: [3], agent_move: [0, 1]
Turn: 3: state: [2], agent_move: [0, 1]
Turn: 4: state: [1], agent_move: [0, 8]
```

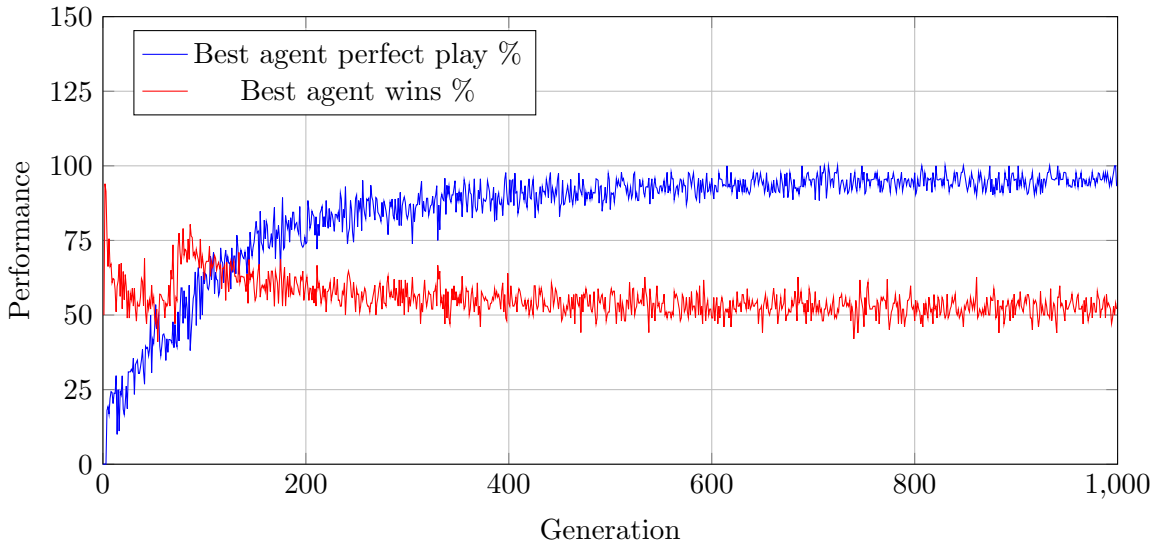
In the 4th generation the NNs learned a first strategy, which was to always subtract one match from the stack. This strategy works optimally in early populations, since it has generally a better chance of winning against agents that always subtract more matches per move. However, this wasn't the final stage for the agents as we can see in generation 993:

```
Agent 993 vs agent: 992
Turn 0: state: [8], agent_move: [0, 2]
Turn 1: state: [6], agent_move: [0, 2]
Turn 2: state: [4], agent_move: [0, 2]
Turn 3: state: [2], agent_move: [0, 1]
Turn 4: state: [1], agent_move: [0, 1]
game_res: [0, 1]
```

The agents now learned to subtract either one or two matches from the stack. Of course, this is still suboptimal and in that case, the local minimum seems to persist. I tried to run the algorithm for more generations, and played with other parameters but the NNs never learn to adapt to predicting moves with more than 2 matches subtracted. Only after I changed the output encoding, did the things start to change.

Stack 1–8, Configuration 2

We run the same configuration, only that use direct encoding instead of one-hot encoding for the output:



The ENNs did indeed learn to play the correct moves, which we can verify in the game logs:

```
agent: 997 vs best_agent
Turn 0: state: [6], agent_move: [0, 5]
Turn 1: state: [1], agent_move: [0, 1]
game_res: [1, 0]
```

So why did this change affect the result that heavily? If we look at the NN structure we had a single output neuron for each possible amount of matches we can subtract in one-hot encoding. During the evolution process, the connections to the neurons for output 1 and 2 got stronger than the other ones since it lead to a competitive advantage. This unfortunately also meant, that the NN was practically unable to adapt to activate the other output neurons. In fact, this behaviour would have been entirely predictable, since there is no mathematical way to model an NN that solves this problem with one-hot encoding except for NNs with large amount of hidden nodes. Yet NNs with large amount of hidden nodes are unlikely to develop with my ENNs since they don't show a competitive advantage instantly.

3.2.2 Nim

We now already have gained a lot of insight about the functioning of ENNs and are therefore ready for the normal version of Nim.

3.3 Complexification

4. Wrapping Up

4.1 Auto Review

4.2 Future Work

5. Appendix

5.1 Code

5.2 Data

5.3 Documentation

5.4 References

Bibliography

Stanley, Kenneth O. and Risto Miikkulainen (June 2002). "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.stack$_8$, pp. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. eprint: https://direct.mit.edu/evco/article-pdf/10/stack_8/99/1493254/106365602320169811.pdf. URL: https://doi.org/10.1162/106365602320169811.

Chandra, Akshay L (2022). *McCulloch-Pitts Neuron - mankind's first mathematical model of a biological neuron*. URL: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

Caruso, Catherine (2023). *A New Field of Neuroscience Aims to Map Connections in the Brain*. Accessed: 2024-11-03. URL: https://hms.harvard.edu/news/new-field-neuroscience-aims-map-connections-brain.

Newman, Tim (2023). *Neurons: What are they and how do they work?* URL: https://www.medicalnewstoday.com/articles/320289#carry_message.

Commons, Wikimedia (2023). *Colored neural network*. Accessed: 2024-11-03. URL: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.

contributors, Wikipedia (2023). *Nim - Wikipedia*. Accessed: 2024-11-03. URL: https://en.wikipedia.org/wiki/Nim.

Qader, Banaz Anwer, Kamal H. Jihad, and Mohammed Rashad Baker (2022). "Evolving and training of Neural Network to Play DAMA Board Game Using NEAT Algorithm". In: *Informatica* 46.5, pp. 29–37. DOI: 10.31449/inf.v46i5.3897. URL: https://doi.org/10.31449/inf.v46i5.3897.

Richards, Norman, David E. Moriarty, and Risto Miikkulainen (1998). "Evolving Neural Networks to Play Go". In: *Applied Intelligence* 8.stacks$_2$x10, pp. 85–96. ISSN: 1573-7497. DOI: 10.1023/A:1008224732364. URL: https://doi.org/10.1023/A:1008224732364.

Konidaris, George Dimitri, Dylan A. Shell, and N. Oren (2002). "Evolving Neural Networks for the Capture Game". In: URL: https://api.semanticscholar.org/CorpusID:1148156.

Orlov, Michael, Moshe Sipper, and Ami Hauptman (2009). "Genetic and Evolutionary Algorithms and Programming: General Introduction and Application to Game Playing". In: *Encyclopedia of Complexity and Systems Science*. Ed. by Robert A. Meyers. New York, NY: Springer New York, pp. 4133–4145. ISBN: 978-0-387-30440-3. DOI: 10.1007/978-0-387-30440-3_243. URL: https://doi.org/10.1007/978-0-387-30440-3_243.