# Evolutionary Neural Networks: Designing a NEAT-Inspired Algorithm for Strategic Game Learning

| | |
|---|---|
| Thesis By: | Lucien Kissling 6e |
| Year: | 2025 |
| Supervisor: | Timo Schenk |
| Co Examiner: | Dr. Arno Liegmann |

Danke an meine Eltern Danke an meine Betreuer Danke an Jan

# 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 these 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. I 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 however, a NN is a strongly simplified artificial model of the human brain consisting of an interconnected web of neurons through which information flows and gets computed.

Although the NN I developed two years ago already learned on its own, I still had to provide and label data for it to learn from. This meant that I had to manually scan the RGB values of the lego bricks and assign them the color they represented. In this case, this was the most efficient solution, however there are cases where you train a model where such data is unavailable or you simply want them to develop their own approach for a problem without predefined solutions. That is how I got to my matura thesis, which aims to develop such ML models that learn without labeled data with and use those to learn board games.

## 1.2   Thesis Statement

This thesis explores the field of neural networks (NNs) capable of learning without human-provided data, known as unsupervised learning. The focus is on evolutionary neural networks (ENNs), which combine neural networks with evolutionary competition.

In this thesis, a simplified implementation of ENNs is developed and trained on board games. The research investigates the effects of varying parameters and features on the learning process and evaluates the algorithm's performance in comparison to other machine learning approaches. This project draws inspiration from the NEAT Algorithm, introduced by Kenneth O. Stanley and Risto Miikkulainen in 2002, which uses a minimal initial structure that evolves complexity over time.[1] As a benchmark, the ENNs are trained to play the game Nim, where players alternately remove matches from stacks, and the winner is the one who avoids taking the last match.

In specific, this Thesis addresses following questions:

- How can evolutionary neural networks (ENNs) be developed that learn to play simple games?

- How do different parameters and features of ENNs affect the learning process?

---

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

# 2.  Background

## 2.1  Neural Networks (NNs)

As previously mentioned in the Preface, an artificial neural network (ANN or NN) is a mathematical model for data processing, initially inspired by the structure of the brain. Therefore, a brief overview of how a brain functions is presented next.

### 2.1.1  Neurons in the Brain

Inside the brain, around 86 million neurons[1] form connections to each other through which they activate other neurons with electric and chemical signals. In a neuron, the signals of connected neurons add up and when they reach a certain threshold, the neuron is activated and fires a new signal to its own connections[2]. The neuron then resets after a certain amount of cooldown time.
With this web of neurons inside the brain, animals can process the 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)

So how can these ideas about neural networks learned from the biology of a brain applied to a program that runs on a computer? The first step is to simplify the chaos of neurons in the brain and organize them into layers of neurons. These layers consist of an input layer, an output layer and optional so-called hidden layers in between. As a next step, the neurons in each layer are connected to those in subsequent layers, typically limited to connections with the nearest layer.
Now, the structure of a neural network looks something like this:
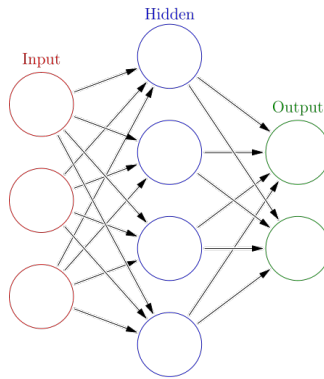


Figure 2.1: A simple Feed Forward Neural Network (FNN) with three layers: Input, Hidden and Output.[3]

The term topology is used to describe the structure of a NN.

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

For the neural network to perform functions, a weight needs to be assigned. This weight value ranges from -1 to 1 and can be thought of as the strength of a connection between neurons.

---

[1]Caruso 2023.
[2]Newman 2023.

In terms of computer science, this structure now resembles a directed, weighted graph, which is why the neurons are called nodes and their connections edges.

> **Nodes/Edges.** *In a neural network, a node receives, computes and sends information, whereas an edge forms the connections between nodes to exchange information.*

The computation process of information by a neural network can be illustrated through the example of a computer vision NN that recognizes digits in a black-and-white image. As a first step, the input is encoded into values that correspond to the nodes of the input layer. In the example of a computer vision NN, the brightness values of the single pixels might directly represent the nodes of the input layer.

Then, the algorithm iterates through all input nodes to identify its connected edges. For each of these edges it multiplies its weight by the value of the input node and adds the result to the node on the receiving end.

After iterating through all the nodes of one layer, it moves to the next layer. At this point, the value of the nodes in the current layer is the sum accumulated by the values of all connected nodes multiplied by the weight of that connection:
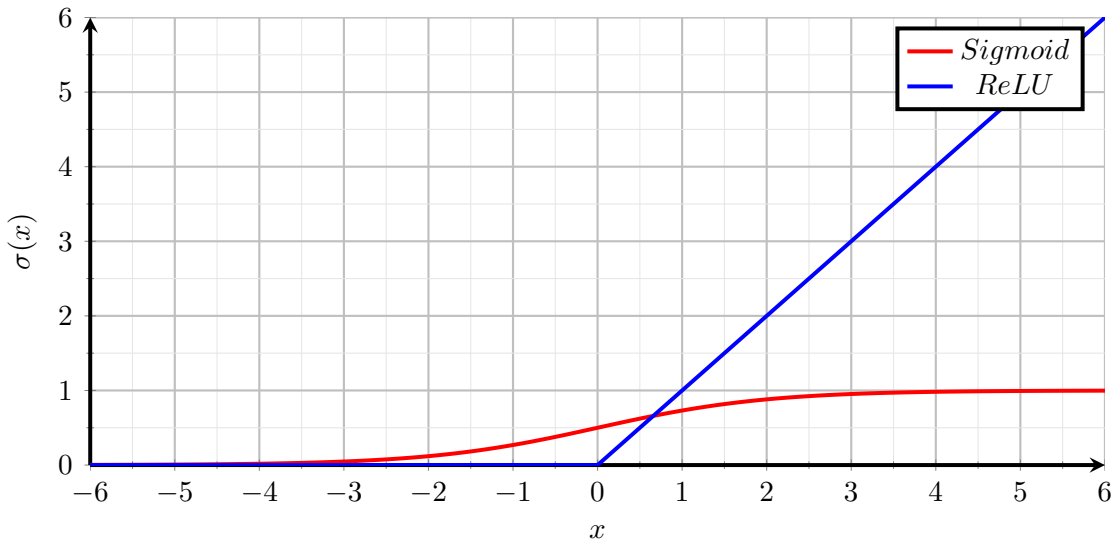
$$v_x = \sum_{i=0}^{N} v_i * w_i \tag{2.1}$$

Where:

- *N is the number of connected nodes*

- $v_x$ *is the value of the current node*

- $v_i$ *is the value of the connected node i*

- $w_i$ *is the weight of the edge connecting the current node to node i*

Additionally, some algorithms add a bias value $r_x$ ranging from -1 to 1 onto the value of the nodes. Finally, an activation function $\sigma(x)$ 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 complete 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

- $\sigma$ is the activation function

Now this value $v_x$ is again sent through its connections to nodes in the upcoming layers. After iterating through all the nodes and layers of the neural network (NN), the process reaches the final layer of the NN, where the computed output can be read out. This output is encoded in node values, as was done with the input, and therefore requires decoding to obtain the final result. In the example of a digit-detecting NN, 10 output nodes could be used, each representing one digit. The result could then be decoded by identifying the output node with the highest value as the result. This kind of output encoding, where all possible results are assigned their own node, is referred to as one-hot encoding. With one-hot output encoding, the values of the output nodes can be interpreted as a the neural networks certainty for a specific result to be correct.

### 2.1.3 Remark: Functioning of NNs

Having established how a neural network operates, it is important to understand why this type of algorithm is considered revolutionary in the field of computer science.

An algorithm processes an input with a set of functions applied in a certain order to calculate an output. Traditionally, a computer algorithm gets 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 of a 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 has 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. In the top layer of abstraction, there are no concrete functions but only neuron activation that approximates a function instead. This means it is hard to prove a neural network to always be accurate and is also the reason why the result of NNs is often called its prediction.

> **NN Prediction.** *The output of a neural network is referred to as its prediction for a given input.*

### 2.1.4 An Example of NN learning: Backpropagation

It is already established how a neural network with specific parameters generates a prediction for a given input. The raises the question: how can parameters encoding the structure, weights, and biases of the neural network be determined? The solution lies in using machine learning, which trains the model to perform a specific task using training data.

> **Machine Learning.** *Machine learning (ML) is a branch of artificial intelligence that focuses on optimizing artificially intelligent models to solve a given problem.*

One machine learning algorithm for NNs is called backpropagation, which is one of the simplest and most efficient methods to train a NN.Backpropagation works using a training and test set containing unlabeled data for a problem.

> **(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 of backpropagation begins with a neural network that has a fixed topology and random weights and biases. First, the NN is given the training problems for which it will generate random predictions. These random predictions are then refined using the sample solutions. This is done by looping back through the NN from the output layer to the input layer, always adjusting the weights and biases in a way that the NN would finally make the right prediction for this problem. However, the NN shouldn't be adapted only to a single problem but make accurate predictions for the whole training set and unknown problems. To therefore prevent overcorrecting a NN for single problems, a learning rate significantly smaller than 1 is applied to the corrections.

The NN is trained on the whole training set for many generations, until the NN starts to make accurate predictions for all problems. To evaluate the performance on unknown problems, a separate test set, which the NN hasn't been trained on, can be used.

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

The focus now shifts to the machine learning technique this thesis focuses on. Once again, nature serves as inspiration:

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

Evolutionary computation (EC) is often employed in situations where the optimal parameters for a function cannot be directly calculated, but the performance of a given parameter set can be evaluated. One example scenario involves a large dataset of points representing an unknown polynomial function with added noise and outliers. To determine the parameters of the underlying polynomial, EC can be utilized to identify the best-fitting parameters. The following shows how EC achieves this optimization.

The process starts with an initial population of agents with random parameter sets. EC then repeats following steps, forming a new generation of the population in each iteration:

- **Fitness evaluation:** First, EC assesses the performance of each agent, defined by its parameters, in the given problem. This performance is referred to as the agent's fitness, which can be evaluated either objectively through a cost function or relatively with a competition between agents. In the given example, the cost function might check the prediction of the NN for the given data points, calculate the absolute differences between the prediction and the values of the data points, and use the total sum of these differences as the cost.

- **Selection:** As a next step, the agents that performed well are selected to be part of the next generation.

- **Reproduction:** The selected agents are then replicated by directly copying their parameters (non-mating) or by merging parameters from different agents (crossover).

- **Mutation:** In each new generation, th parameters of the agents are mutated, either by completely replacing certain parameters with new random parameters or by shifting the existing parameters by a random value.

After a certain number of generations, the EC algorithm will have found best performing set of parameters for a given problem.

### 2.2.1 Gradient Descent & Local Minima

As established in the previous section, EC begins with an initial population of agents with a random set of parameters. The fitness of these agents is determined using a cost function. Agents with relatively low cost (or high fitness) survive the round and are replicated, mutated and again selected by

cost.

This whole process strongly resembles a ball rolling down a hill in some terrain. At any given point on the terrain, the ball moves in the direction of the steepest descent (ignoring the inertia of the ball). Applying this analogy to EC, the parameters for a function can be imagined as the horizontal coordinates for the ball meanwhile the cost of these parameters correspond to the height at that location. Mutations produce parameter sets that are close to the original, resembling neighboring points on the terrain. As the ball rolls down to the lowest neighboring point in any step, EC again chooses the set of parameters with the lowest cost every cycle. This is why the machine learning process is referred to as gradient descent, as the algorithm evaluates all neighboring points and then follows the direction with the lowest cost and therefore steepest gradient.

The ball analogy provides insights into the functioning of the EC learning process, which can be understood by visualizing 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, which have different heights. Therefore, following things can be derived:

- (Minor) differences in the starting position can result in large differences in the end position and the respective cost.

- Once a point with no downwards gradient is reached, innovation halts for that agent, even if there is a point with lower cost somewhere else. Such points are called local minima.

---

**Local Minimum.** *A local minimum is a point with minimal cost compared to its neighboring area and is therefore a halting point for gradient descent.*

---

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 neural networks and evolutionary computation, the two concepts are finally combined to create the algorithm this thesis focuses on.

---

**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. In the context of neural networks, evolutionary computation can be applied as follows:

The parameters encoded by ENNs represent the weights, biases, nodes, and edges of a neural network. ENNs still follow the same core process of evolutionary computation: They start with a random population of NNs that are evaluated and selected based on a fitness function. The selected NNs 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, previously 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 weights are added between the new node and the two other nodes each.

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

## 2.4 Drawing Inspiration from the NEAT-Approach

As already stated in the thesis statement(1.2), neuroevolution of augmenting topologies is an ENN machine learning algorithm developed by Kenneth O. Stanley and Risto Miikkulainen in 2002[4]. NEAT evolves the topology, weights and biases of NNs at the same time, which makes it particularly suitable for tasks where the optimal network architecture is not yet known. One of its key innovations is that the initial population begins with NNs of lowest complexity and the NNs only increase topological complexity as it is useful for the problem. In specific, the NEAT algorithm starts with NNs that only have the nodes of the input layer connected to the nodes of the output layer. This design aims to produce more efficient neural networks by only increasing complexity when it leads to improved performance. The NEAT algorithm therefore only needs the mutations 1. to 4. shown in the last section(2.3) and doesn't need to remove complexity (mutation 5.) as it is already minimal.

## 2.5 Games

The following section offers a brief overview of the games on which the ENNs will be trained.

### 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 with the difference that there are multiple stacks with matches. Now, the player who removes the last match from the last unemptied stack loses. The winning strategy for Nim is much more complex, which forces the ENNs to learn a more complex strategy.[5]

## 2.6 Related Work

The field of AI research in ENNs is largely studied and is often linked to games. The following table provides 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 evaluations 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 derived.

---

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

# 3.  Building my ENN

## 3.1  Algorithm Design

This chapter details how to use the concepts described in the last chapter(1) to build an evolutionary neural network that plays Nim.

### 3.1.1  Overview

As already mentioned in the thesis statement(1.2), this project is written in the Rust programming language[1]. The source code and all test data can be found on this thesis' GitHub page[2]. This project utilizes following Rust libraries: 'rand' (version 0.9.0-alpha.2)[3], 'indicatif' (version 0.17.8)[4], 'csv' (version 1.3.0)[5], 'bincode' (version 1.3.3)[6], 'serde' (version 1.0.209)[7], and 'rayon' (version 1.10.0)[8]. The code has been partially written by the AI coding assistant GitHub Copilot[9]. Every instance of artificially generated code has been thoroughly reviewed.

The codebase is divided into different modules as well as a bin folder containing main files. One module manages the ENN algorithm, while another handles the games that the ENNs are trained to play. The ENN module itself is further 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 data saving

The game modules provide the problems for the NNs, handle the predictions of the NNs, and find the new game state after a move. It further includes an objective performance evaluation function to measure how well the NNs perform.

The following sections provide a bottom-up explanation of all functions, beginning with the neural network. The implementation uses Rust's 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:

```
1    pub struct NeuralNetwork {
2        [...] // redundant side data about the NN
3        pub nodes: Vec<Vec<Node>>,
4    }
```

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:

---

[1]Klabnik and Nichols 2023.
[2]Kissling 2024.
[3]Developers 2024.
[4]Ronacher 2023.
[5]Developers 2023.
[6]Developers 2021.
[7]Tryzelaar and Tolnay 2024.
[8]Matsakis and Stone 2023.
[9]GitHub and OpenAI 2021.

```
1    pub struct Node {
2        pub bias: f64,
3        //edges stored in an adjacency list
4        pub incoming_edges: Vec<Edge>,
5        pub outgoing_edges: Vec<Edge>,
6    }
```

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

```
1    pub struct Edge {
2        input: [usize; 2],
3        out: [usize; 2],
4        weight: f64,
5    }
```

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 the forward propagation algorithm described in 2.1.2.

- `activation_function(x: f64) -> f64`, which computes the neuron activation described in 2.1.2. This implementation uses a variant of the *ReLU* function. The linearity for inputs greater than zero in the *ReLU* function allows for more variety of neuron activations, enhancing the flexibility and efficiency of the neural network. The chosen variant is the *ELU* function, which addresses the issue of dead neurons encountered in *ReLU* by permitting negative output values.

### 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:

```
1    pub struct Agent {
2        pub nn: NeuralNetwork,
3        pub fitness: f64,
4        pub rank: isize,
5    }
```

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 in between of two connected nodes (mutation 4.), the inserted node will be added to a random layer in between of the previously connected nodes. If the previously connected nodes are in neighboring layers, a new layer is created 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 a random sign. This gives us following function:
$v_{new} = shift(v_{old}) = v_{old} + rand(-1, 1) * rand_{float}(0..1)^2$

- The amount of mutations performed per NN is randomly selected in a range defined by adjustable parameters (see section 3.1.5).

- The type of mutation that is performed is randomly determined for each mutation. A weight can be assigned to each mutation as a parameter, which represent its probability to be selected relative to the weights of other mutations (see again section 3.1.5).

### 3.1.4  Evolutionary Computation & inspiration by NEAT

The EC process starts with an initial population of new agents with NNs created by the `NeuralNetwork::new()` function (described in 3.1.2). As per the NEAT algorithm's approach, the new NNs have minimal structure, where all input nodes are directly connected to all output nodes with no hidden layers. Random values area assigned for the weights and biases of the initial NNs. Subsequently, the steps of performance evaluation, selection and mutation are repeated every generation.

## Competition

In the approach examined by this thesis, the fitness of agents is assessed by having them compete against one another in the games. However, allowing every agent to play every other agent is computationally too expensive since the needed time increases quadratically with the population size. To address this, a circular pairing algorithm is employed. This algorithm sets a predefined number of opponents for each agent to play against (see 3.1.5). For each of these predefined games, the algorithm generates a new pairing distance, which it then uses to pair each $agent_i$ with the $agent_{i+distance}$. It also checks that the distance isn't a multiple of any distance it had previously avoiding duplicate pairings between agents. Each game is played twice with both agents taking turns as the starting player while keeping the same initial state.

## Fitness Evaluation

Now, the fitness function evaluates its fitness for each agents. The main idea for the fitness function is to simply count the number of wins an agent has scored during the competition. Of course, there are other possibilities to evaluate the performance of an agent that represent their performance more accurately. Still, the number of wins is used as fitness as this function is generally applicable 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 machine learning.

## Natural Selection

After the fitness evaluation, the performance of the current generation is evaluated (details in section 3.1.6) before the new population is generated. The new population is made up by some portion of each of the 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 selecting all needed agents randomly using the fitness of my agents as their relative probability of being drawn. The fitness can also be raised to some power to adjust the likelihood of survival for 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.

- **Old best agents:** The last part of the new population is made up of best agents from previous generations which help counter populations stuck in a local minimum.

Now that the new population has been generated, the whole process can start over again.

### 3.1.5  ENN Parameters

As already mentioned in the thesis statement(1.2), this research involves testing its 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 the ENN algorithm:

- **General:** Size of the population
- **Game:** Initial state of the game, game function that encodes the state for the NN, decodes its 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, a file is saved with the information about the value of all these parameters.

### 3.1.6 Performance Tests

Some stats and games are tracked to see how the Agents perform in the games. Following data is saved every generation after the competition phase:

- **Fitness of the best agent:** This stat shows us how much better the best agent performs relatively to the average agent.

- **Wins against older generations:** The best agent of the current generation is paired against an evenly distributed set of the best agents from past generations to track relative performance to the past. As long as this number stays above 50%, an improvement over older generations should be expected.

- **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 this algorithm is not used as the fitness function because of the points previously mentioned in section 3.1.4.

- **Number of turns:** The average number of turns the agents play during their competition is tracked throughout each generation. In Nim, low turn count generally means better play.

- **Average amount of hidden layers and nodes per hidden layer:** These stats are tracked to see how complex the topologies of my NNs in the population are.

- **Best agent layer sizes and edges:** These topological stats are tracked to see how complex the best solution is.

- **Best agent games:** The whole games are tracked turn after 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 a previous generation. These game logs offers insight into what moves the agents make in real games which helps derive their tactics and strategies.

### 3.1.7 Nim

For the nim game, there are a few different game modes as well as an objectively perfectly playing dynamic programming algorithm. First, the general concept of how a game is played will be explored.

#### 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 its 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 predicting their moves. On their turn, the active agent reads the current game state as input and predicts its next move. A move indicates the stack where the matches are removed as well as the number of removed matches. Then the game function computes the new game state after the move or ends the game if all stacks are empty.

#### Encoding and Decoding

The 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 each stack.

For output encoding, there is two different implementations.

- **Direct encoding:** The output layer has 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 is 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 removed, 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 values from the initial state parameter except for the configuration *random*, where the stack sizes of the initial state are randomly selected with the corresponding values in the initial state 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, a dynamic programming (DP) algorithm is used to calculate the best move for every state. This algorithm 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 a move, the position is winning, but if all moves lead to winning position for the opponent, the algorithm returns the position with the maximal moves needed 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 it is known for all positions if they're winning, this data can used to grade the moves of agents with a score that is calculated as follows: 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 Findings

Now this implementation of ENNs will be tested with different parameters using the games as a benchmark. The general approach is to start by training the ENNs on easy problems whilst observing the effect of different parameters on the result. Each test is repeated at least three times to try to reduce the impact of randomness on the results. As this thesis outlines the different testing configurations, one test will be discussed for each configuration as long as all three tests showed similar results. However, the full testing data can be inspected on this thesis' GitHub[10].

### 3.2.1 Simple Nim

For the Simple Nim game, each test will be run for 1000 generations, as the ENNs should be able to solve such simple games rather quickly.

---

[10]Kissling 2024.

**Stack: 2, Configuration 1**

In the first configuration, the ENNs need to find the correct move for the initial Nim state with one stack containing 2 matches. This is the easiest problem that can still be considered a game - with one match on the stack there is only one legal move, so it doesn't involve any decision-making. The winning strategy is to only take one match and leave the opponent with the last match on the stack. The first test starts with a relatively small population of 100 agents and completely simplified parameters: `game function:  run_nim_strict`, initial state: `[2]`, population size: `100`, competition 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 one-hot output encoding is used at first, as it keeps the output in a desired range.

The first test shows following results:
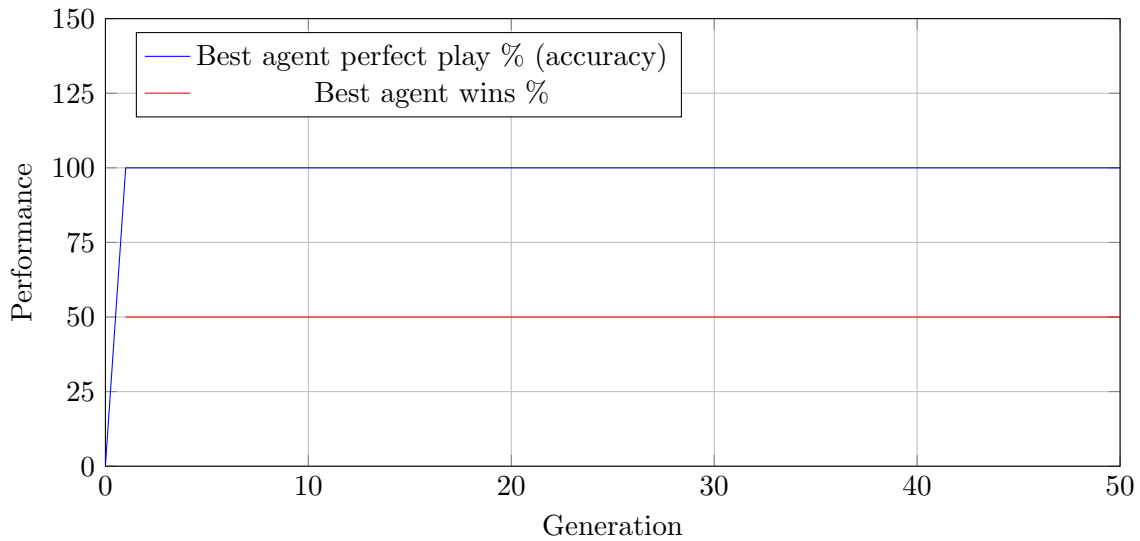


As one can see, the best agent already plays the game perfectly after one generation. One can also observe this in my saved games, where the best agent correctly plays the move [0, 1] (first value is the stack, second value the number of matches removed):

```
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, 1]
6    Game result: [1, 0]
```

The best agent wins percentage in the plot above is a steady 50% since all games are played both ways and the first agent always wins the game.

**Stack: 8, Configuration 1**

Since the stack size with 2 has already worked fine, I will now try the same configuration with an increased initial stack size of 8 matches:
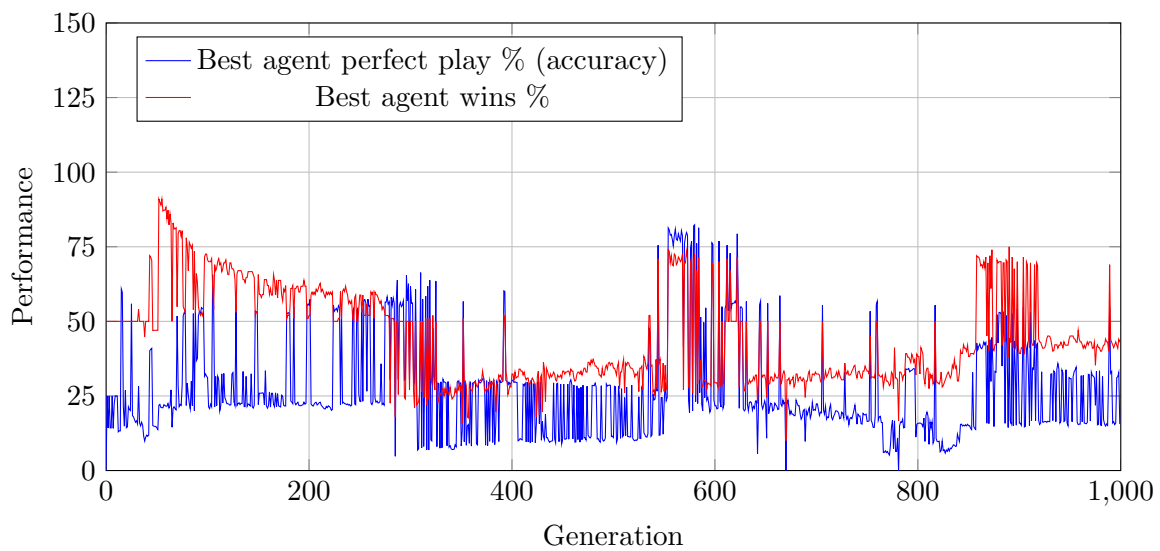
The accuracy of the runs got up to around 30% and if I look at the games recorded in generation 982, one can see why it didn't get to 100%.

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

The best agent only learned to play the moves where it subtracts one or two matches. This seems quite bad since seemingly all the NNs have to do is to play [0, 7] instantly. So why couldn't it learn this one move, when it could do it with stack size 2? The problem is that if the starting player subtracts less than 7 matches, the other player would get in winning position where it has to adapt and play subtract N - 1 matches. There might evolve a population where all agents only subtract 7 matches, however this seems more like a local minimum and my final goal is to develop NNs that can play the whole game. This simplification turns out to be a hindrance, so I will change the stack to have a variable amount of matches.

**Remark: Unnecessary Topology**

If I look at the topology metrics of one of the tests, I realize that complexity of the NNs exploded without any visible improvement in performance:
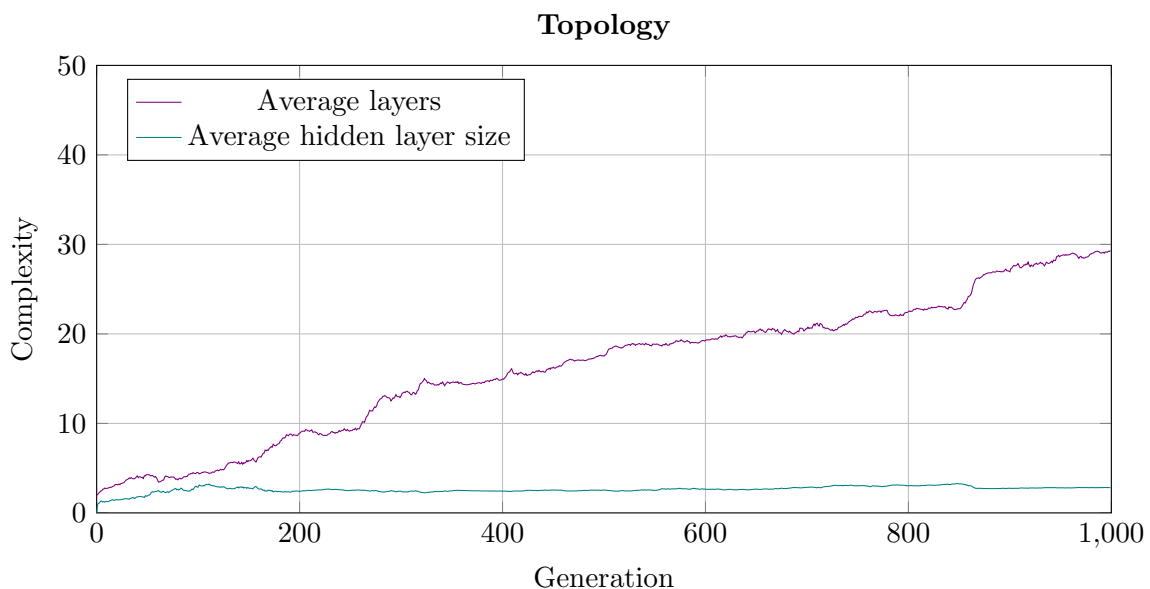


I get an average amount of over 20 hidden layers with on average 2 nodes. It doesn't make sense for the NNs to have such complexity for such a simple problem as it only increases the needed computations.

I will therefore reduce the chance for a new nodes and edges to form by a factor of 20 each. Let's see if this change impacted the performance and topology of the NNs:
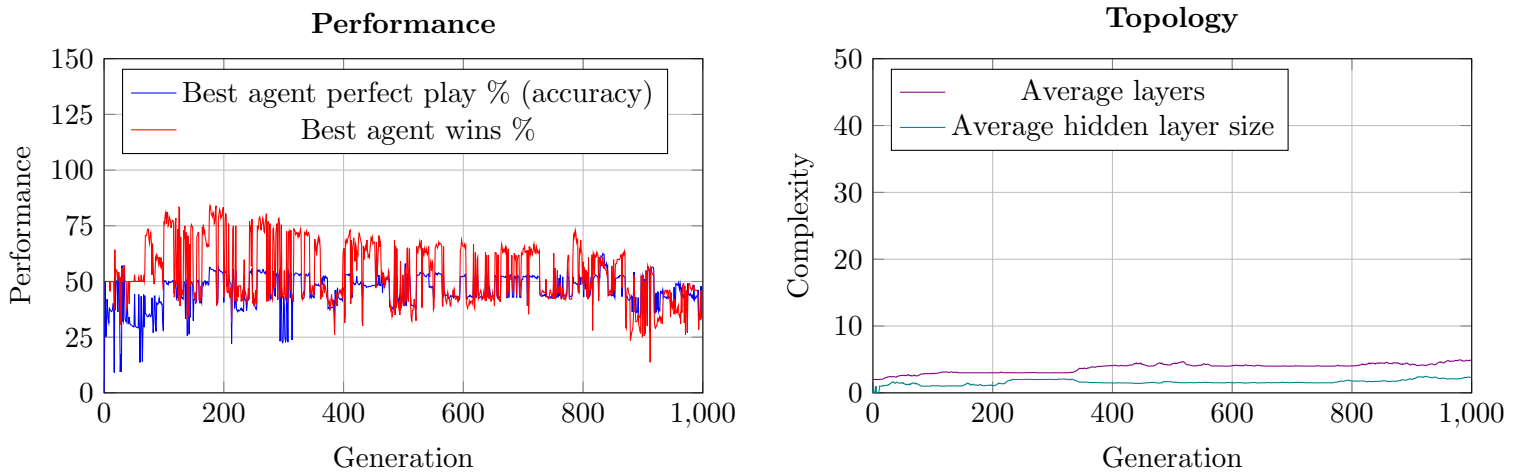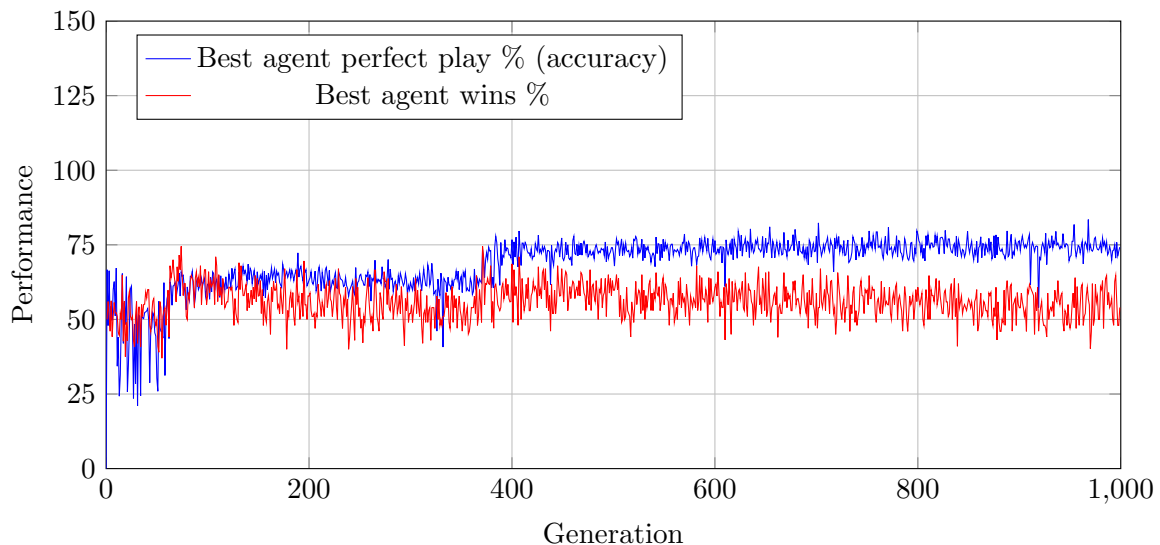


Figure 3.1: Performance metrics over generations in a 2x2 grid.

My modifications didn't impact the performance of the NN, but definitely helped keeping the topological complexity in a reasonable bound. I will keep these modifications moving on.

**Stack 1–8, Configuration 1**

I will repeat the previous test but with a variable amount of matches between 1 and 8:



In my tests, the best agent perfect play performance always reaches a value ranging from 40% to 75% where it remained stable. The NNs were only able to learn the moves where they subtract 1, 2 and 3 matches but no others.

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

The performance of NNs in different tests just depended on how many moves the NNs learned. In any case however, the NNs have got stuck in a local minimum. Such local minima originate in an early generations where the agents aren't highly developed, which means they often only play one move. The agents that only subtracted one match were therefore the most successful and reproduced the most. After reaching that local minimum, there wasn't a competitor that could assert itself, either because of a flaw in the selection process or because none were created in first place. One can also see how homogeneous the NNs across generations have become by looking at the wins of the best agent

against former best agents, which always settles at 50%, revealing that they all use the same strategy. I will therefore try to modify some parameters to see if anything helps against such local minima.

**Stack 1–8, Configuration 2–5**

To overcome local minima, I can use following strategies:

- **Fitness Exponent:** Increasing the fitness exponent to *ex.* 2 might help successful mutations reproduce better and therefore overcome a large population of similar NNs.

- **Population Composition:** I can add other kinds of agents to the new population like best agents from older generations, randomly selected agents from the last generation, or newly generated random agents. This might help the development of new and possibly more complex strategies without the need for instant return in performance.

- **Population Size:** A larger population means more agents with different strategies can exist, evolve and finally help overcoming local minima.

I now test the effect of these parameters with following 4 configurations:

1. Fitness exponent set to 2

2. Old best agents, random agents, and new agents each make up 15% of the population

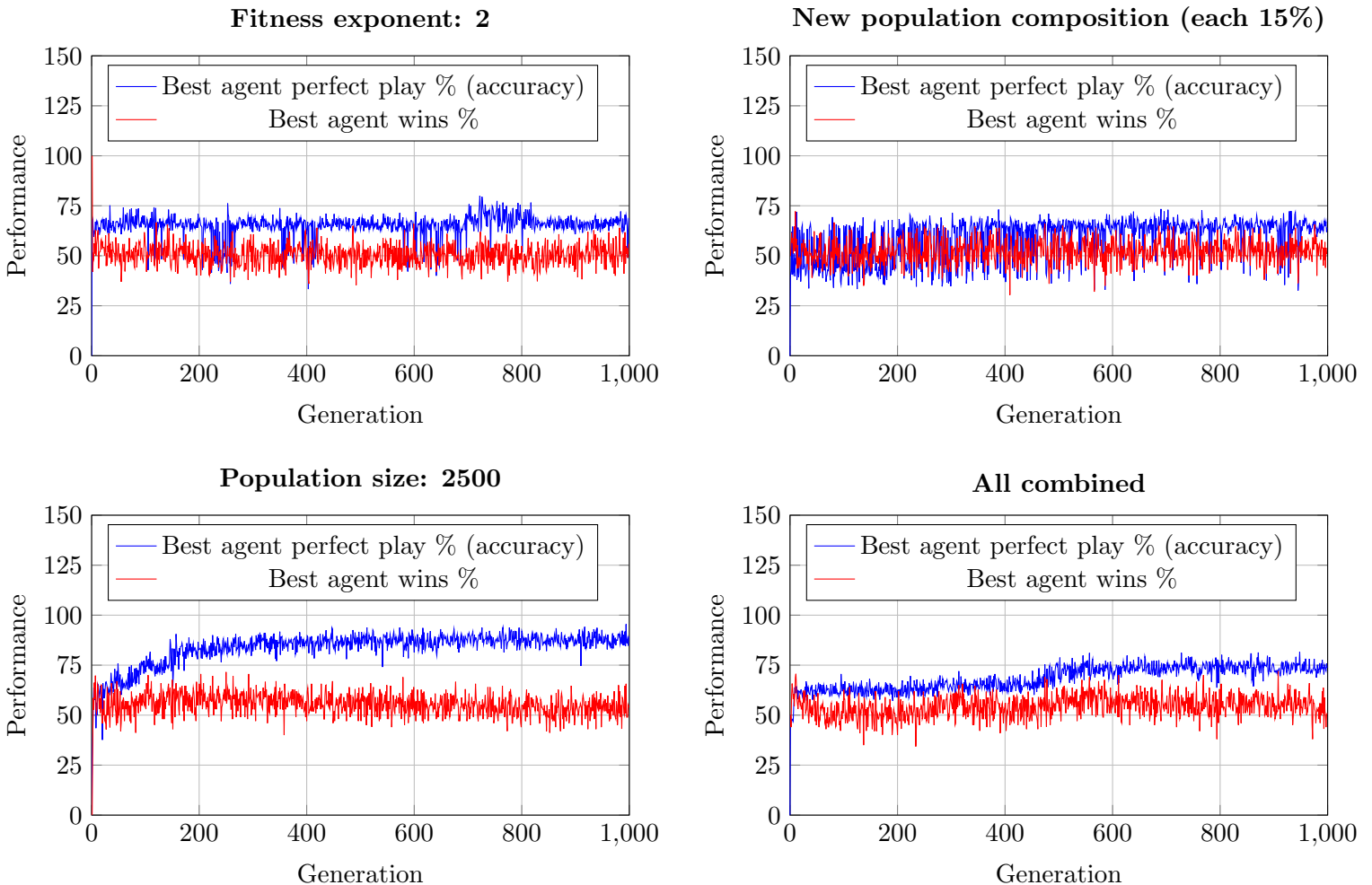3. Population size set to 2500

4. All parameters combined



Figure 3.2: Performance metrics over generations in a 2x2 grid.

As we can see in the graphs, the performance didn't reach 100% in any configuration and the NNs remained stuck in a local minimum with the best agents always winning 50% against previous best agents. However, there is still something I can change.

### Stack 1–8, Configuration 6

I run the simple configuration as in configuration 1, only that I changed the output encoding (explained in chapter 3.1.7) from one-hot to direct encoding:
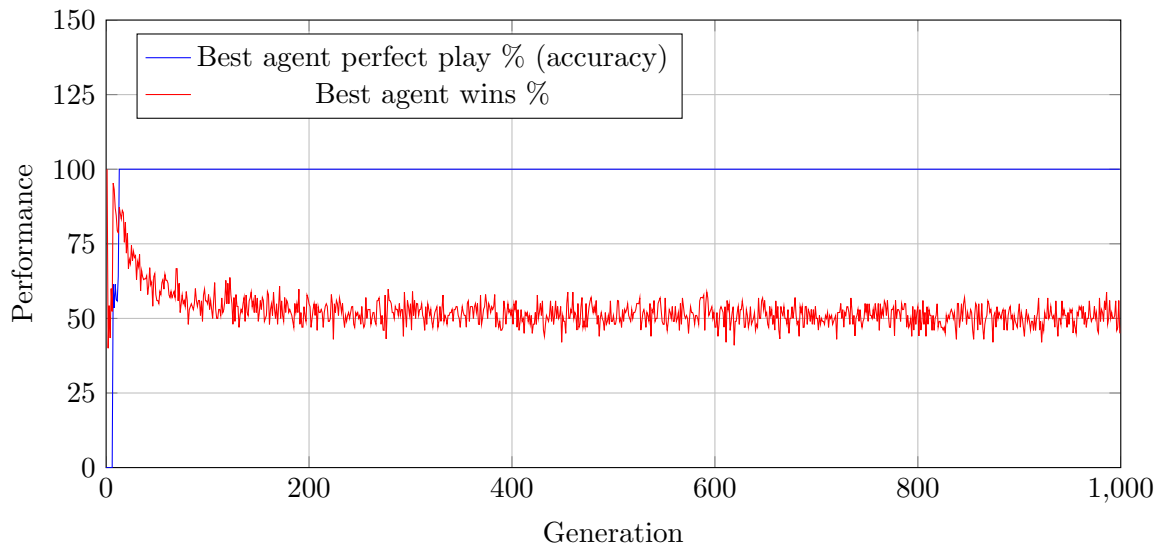


Now finally got it working! The best agent's performance reaches 100% after 13 generations. So why did the change of output encoding affect the result that heavily?

If I look at the NN structure I had a single output neuron for each possible amount of matches I 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 behavimy 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.

Since this test worked fine, I will now a very large stack size.

### Stack: 1–1000, Configuration 1

Now I will increase the maximal stack number to 1000:
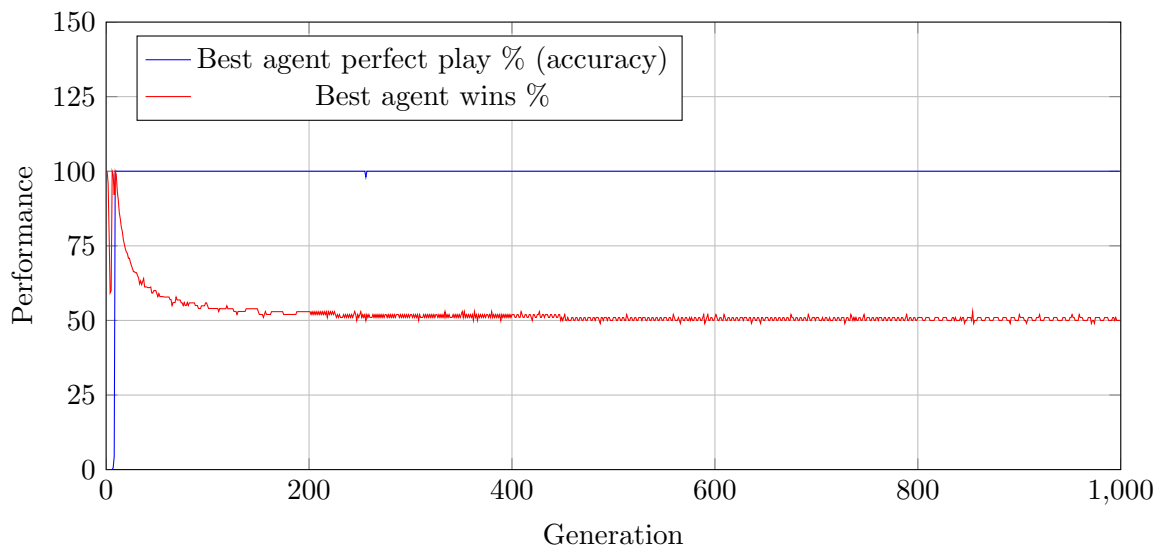


The NNs reached full accuracy in less than 30 generations in both the first and second test run. However, the NNs got stuck in a local minimum in the third run. This is why I also ran other test

runs with different parameters but unfortunately, they showed similar consistency within their runs. Still, the NNs managed to learn the game most of time which means, I can finally start training the NNs on the main game.

### 3.2.2 Nim

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

**Stacks: 2*2, Configuration 1**

I will start with a fairly simple problem where there are two stacks containing a maximum of 2 matches. There are eight possible game states for which the NNs need to find solutions. This shouldn't be a particularly hard task for the NNs so let's see how they do:
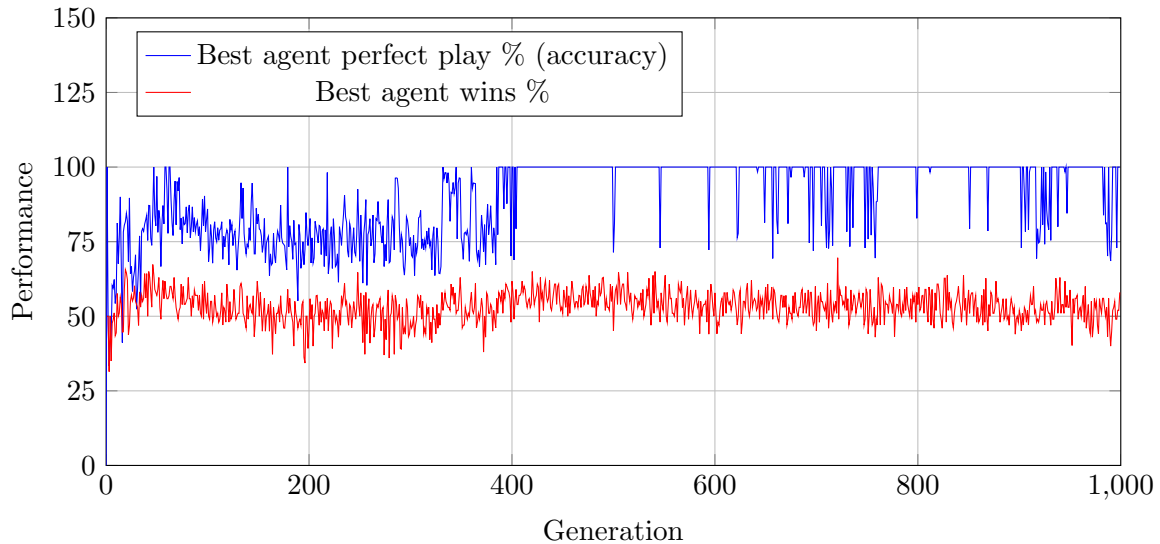


As expected, the NNs had no problem with this task and in my tests, they reached full accuracy in 100 to 400 generations.

**Stacks: 2*4, Configuration 1**

Now I will double the maximal amount of matches per stack which leaves us with 24 possible game states. Let's see, how they do:
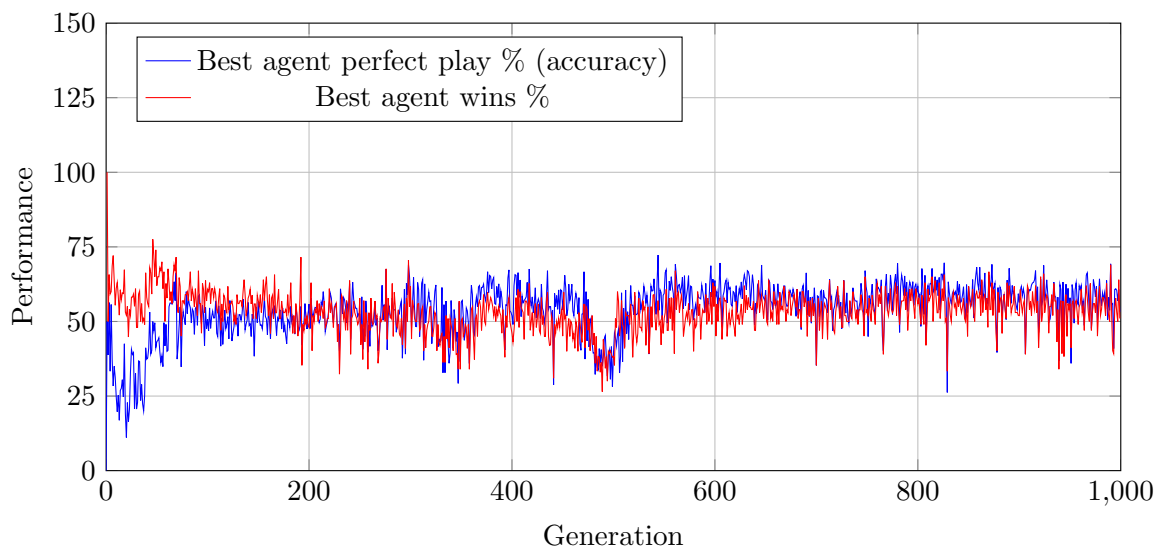


In all tests did the NNs settle at an accuracy of around 57% and if I look at the sample games, the NNs demonstrate a clear lack of simple strategy:

```
1      Agent 994 vs Agent 995:
2      Turn: 0: state: [2, 4], agent_move: [1, 4]
3      Turn: 1: state: [2, 0], agent_move: [0, 1]
4      Turn: 2: state: [1, 0], agent_move: [0, 1]
```

The best agent from generation 994 would have had the possibility to force its opponent into a losing position by subtracting two matches from the second stack. Instead, it empties the whole stack which leaves the opponent to take all but one matches from that stack.

In another game however, the best agent from generation 999 just makes a move that would never be legal in this test:

```
1      Agent 999 vs Agent 239:
2      Turn: 0: state: [4, 3], agent_move: [1, 5]
```

It is hard to tell if this is a result of the NNs being in a local minimum or because of to little training so I will run the tests a bit longer to give the NNs more chances to develop.

### Stacks: 2*4, Configuration 2

I will run the tests for 10'000 generations instead of 1000:

# 4. Wrapping Up

## 4.1 Conclusion

I managed to train neural networks with 100% accuracy in the simple Nim version, where there is only one stack, for stack sizes up to 1000. This success was mostly due to the realization that the output should be encoded directly instead of using one-hot encoding. Although one-hot encoding keeps the output in a safe bound, it finally makes it harder for the NNs to find the correct structure and weights in the case of Nim. Other parameter configurations were explored, however, most of them showed to be similarly effective as the initial configuration, except for the weight reduction of topology increasing mutations which showed to be effective. In the full Nim game, the NNs were able to reach an accuracy of 100% for two stacks with two matches each but only reached an accuracy of 60% for two stacks with 4 matches each. Since the NNs already weren't able to understand this problem, it didn't make sense to train them on even more complex configurations.

There are two possible explanations for the failure to develop accurate NNs in the Nim game. Either, the Nim game is a bad game for NNs to solve since the algorithm to play Nim perfectly isn't easily translatable into neurons, or my implementation and or parameters of the ENN simply wasn't up to the task. To tell which of the two reasons apply, there needs to be done more research in this field, which will be discussed in the last section (4.3).

This thesis also showed sometimes contradictory results within different tests of the same parameters. In an application, one might therefore train several populations to then choose the best performing one.

## 4.2 Auto Review

This research was a large and sometimes painful journey for me to realize but finally resulted in a ENN framework that I proved to be able to learn some functions without human help. Still, there is a few things I would do differently.

I planned to be done with a beta program many months before the thesis end to then have time to test and refine my program. However, I finished the program much later and found many bugs which forced me to repeat the whole testing. The final tests were therefore done in only a few days, which left me with little room to search for optimal parameters. Additionally, my implementation wasn't optimized for speed, which further increased the time problem.

Another thing I realized is that a scientific approach helps a lot with finding a solution. I first had fixed ideas about certain features and parameters and only realized these ideas were flawed after starting with a simple solution and then testing different approaches one at a time.

I would also do some things differently in terms of my implementation. Firstly, I had many confusions with parameters which I had defined all over the place in my project. I would create a struct that handles all parameters and implement automized testing so I could run all tests without my supervision.

However in general, my object oriented programming approach worked fine and my project structure made it easily understandable and editable throughout my research. Still, more consistent commenting and documentation would have further improved clarity of the code.

## 4.3 Future Work

As already mentioned, this thesis leaves much research to be conducted, which might also answer the question whether the Nim game can be played by NNs. In specific, future work might include the following:

### More Testing

Because of the time constraints, this research only tested few configurations with little tests each. More test runs within a set of parameters might therefore increase statistical significance of the results and more tests with different sets of parameters might help find better solutions. One might also use evolutionary computation to optimize different parameters of the ENNs.

### Faster Computation

To enable more tests, one could improve the computation performance of the ENN algorithm. This can be done by converting the NNs into a weight matrix and then compute its predictions with hardware accelerated vector matrix multiplications. However, the complex topologies of the NNs in this approach make it harder to do so than with other NNs. Also many small performance improvements can still be done in this project which might cumulatively make it drastically faster.

### Algorithm Design

There are also a lot of improvements to be made in the ENN algorithm, this thesis developed. For example, the NEAT algorithm, which served as initial inspiration, also uses genetic encoding, tracks genes using historical markings and protects innovation using speciation[1]. All of these and other approaches could improve the ENNs performance and therefore need to be studied.

---

[1]Stanley and Miikkulainen 2002.

# References

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.

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.

Klabnik, Steve and Carol Nichols (2023). *The Rust Programming Language.* 2nd. No Starch Press. URL: https://doc.rust-lang.org/book/ (visited on Dec. 14, 2024).

Kissling, Lucien (2024). *RustENN: Evolutionary Neural Networks in Rust.* Accessed: 2024-12-14. URL: https://github.com/codingZion/RustENN.

Developers, The Rand Project (Nov. 25, 2024). *Rand: A Rust library for random number generation.* Version 0.9.0-alpha.2. URL: https://crates.io/crates/rand (visited on Dec. 14, 2024).

Ronacher, Armin (Nov. 29, 2023). *Indicatif: A Rust library for indicating progress in command line applications.* Version 0.17.8. URL: https://crates.io/crates/indicatif (visited on Dec. 14, 2024).

Developers, The CSV Project (Nov. 29, 2023). *CSV: A fast and flexible CSV reader and writer for Rust.* Version 1.3.0. URL: https://crates.io/crates/csv (visited on Dec. 14, 2024).

Developers, The Bincode Project (Apr. 10, 2021). *Bincode: A binary encoder/decoder implementation in Rust.* Version 1.3.3. URL: https://crates.io/crates/bincode (visited on Dec. 14, 2024).

Tryzelaar, Erick and David Tolnay (Sept. 2, 2024). *Serde: A framework for serializing and deserializing Rust data structures.* Version 1.0.209. URL: https://crates.io/crates/serde (visited on Dec. 14, 2024).

Matsakis, Niko and Josh Stone (Mar. 3, 2023). *Rayon: A data parallelism library for Rust.* Version 1.10.0. URL: https://crates.io/crates/rayon (visited on Dec. 14, 2024).

GitHub, Inc. and OpenAI (2021). *GitHub Copilot: Your AI Pair Programmer*. AI-powered code completion tool. URL: https://github.com/features/copilot (visited on Dec. 14, 2024).