# Neuroevolution of Gameplay Agents in 2-Dimensional Platform Games

Bachelor's Thesis
of

Pascal Süß

University of Passau
Faculty of Computer Science and Mathematics
Chair of Algorithms for Intelligent Systems

Reviewer: Prof. Dr. Dirk Sudholt

# Abstract

Neuroevolution is often used to train gameplay agents in various video games. Platform games are particularly well suited to researching the behavior of AIs in a simulated, physics-based environment. However, most of the existing research was done on platform games of high complexity that require agents to make equally complex decisions. For this reason, a physics-based platform game has been created with a focus on moving through the environment, which is the common objective in all platform games, and aims to make the results more generally applicable to 2-D platform games as a whole.

This thesis examines the effects of varying parameters of the neuroevolutionary algorithm and different game environments on the performance of the game agents. By defining multiple presets for different evolutionary algorithm (EA) and neural network (NN) parameters, a comparative analysis of their performance statistics across multiple runs is conducted. The acquired data shows significant differences in performance when changing parameters of the EA and NN, as well as the performance being highly dependent on the game environment. However, during data analysis, a type of noise was discovered that affected the performance of the game agents. Through a comparison of the agents' performances with and without noise, it was found that performance drastically increased when removing noise from the inputs. Although both with and without noise, the analysis indicates trends that show better performance can be achieved with EA presets that are able to fine-tune the agents better during the search, as well as distance inputs being superior to positional inputs for the NN.

***Keywords* –** neuroevolution, platform game, evolutionary algorithm, neural network

# Contents

# Acronyms

| | |
|---|---|
| **EA** | Evolutionary Algorithm |
| **GA** | Genetic Algorithm |
| **NN** | Neural Network |
| **NE** | Neuroevolution |
| **NEAT** | Neuroevolution of Augmenting Topologies |
| **SLP** | Single Layer Perceptron |
| **MLP** | Multi Layer Perceptron |
| **MBF** | Mean Best Fitness |
| **SR** | Success Rate |

# 1 Introduction

AI has been a hot topic in recent years, especially since the release of OpenAI's ChatGPT to the public. Such systems use Large Language Models (LLMs) to operate. The underlying architecture in these LLMs are neural networks (Min et al. (2023)), which are trained on giant data sets.

Another way to train neural networks (NN) is through neuroevolution (NE), where a population of individual NNs are evolved similar to evolution in nature. One of the fields where neuroevolution has been extensively used is the training of gameplay agents in various games (Risi and Togelius (2017)). Platform games in particular provide a good test bed for the performance of AIs in a simulated, physics-based environment.

## 1.1 Motivation

There has been research in the area of neuroevolution of platform gameplay agents already (Togelius, Karakovskiy, Koutnik, and Schmidhuber (2009)). A large part of this research was conducted on the popular platform game *Super Mario World*[1]. While there has been research comparing different evolutionary algorithms and controller types, the research in certain domains is lacking. Those domains include investigating the impact of varying individual parameters of the evolutionary algorithm or the neural network's inputs and topology. Furthermore, the complexity of many platform games is quite high, which makes it challenging to quickly compare the performance of many neuroevolution configurations. For this reason, a physics-based platform game that has been tailored for low complexity will be the central point of the research. This means the focus lies on the movement in platform games, which is the common objective in all platform games.

## 1.2 Contribution

The aim of this thesis is to investigate the impact of varying NE parameters on the evolutionary algorithm's performance in a platform game. With one main goal being to create an environment that makes comparing different NE parameters as easy as possible. For that purpose, the game focuses on moving through the 2-D space of the platform game and leaves out potential distractions like enemies or intractable objects.

The main research questions are:

1. How do different parameters for the evolutionary algorithm affect agent performance?
2. What are the effects of changing the NN topology and inputs?

[1]`https://www.nintendo.de/Spiele/Super-Nintendo/Super-Mario-World-752133.html`

3. How do the agents perform in various levels of the platform game when varying parameters from questions 1 and 2?

These questions will be examined by defining several presets with parameter combinations and storing the performance data of each preset in a database. This data is then used to compare the average results over multiple runs when changing the parameter presets.

## 1.3 Structure

First, the theoretical background regarding neuroevolution is provided in Chapter 2. Then the project requirements that are necessary to answer the main research questions are explained in Chapter 3. The methods used during this project are shown in Chapter 4 where the implementations for the game environment, neural networks, evolutionary algorithms, and database are demonstrated. Afterwards, Chapter 5 displays the results that have been acquired through those methods, analyzes these results, and evaluates the impact of the findings. Finally, Chapter 6 concludes the thesis by summarizing the main takeaways from this project.

# 2 Theoretical Background

To understand the important concepts behind *Neuroevolution of Gameplay Agents in 2-Dimensional Platform Games*, we first have to look at neuroevolution. NE is a combination of neural networks and evolutionary algorithms. For this reason, we are first looking at NNs in Section 2.1 and EAs in Section 2.2 independently before NE, especially in platform games, is explained in more detail in Section 2.3.

## 2.1 Neural Networks

Artificial neural networks, or just Neural Networks (NN), as they will be referred to in this paper, are structures that process information and can achieve dynamic learning by adjusting themselves at runtime (Bavarian (1988)). This makes them especially well suited for tasks where the objective is to perform different actions depending on the inputs the system receives. Bavarian (1988) also shows NNs are well suited for intelligent control tasks. Further, NNs are capable of "learning from data relationships and generalizing to unseen situations" - Taud and Mas (2018). For this reason, neural networks are a good choice for the game agent controllers in a simulated, physics-based environment. These agents receive sensory inputs, and the NN can map those to the appropriate actions they should take. Furthermore, being able to handle new situations is also important for traversing new environments in 2-dimensional platform games.

To understand how neural networks work, we first look at the model of a perceptron. According to Taud and Mas (2018) the most basic form of the perceptron is a single neuron where a number of inputs are mapped to a single output, which can be seen in Figure 1 on the left side. As Chapter 10 of Shiffman (2012) describes, the process of calculating the output of a single perceptron involves taking the weighted sum of all the inputs $i_1, ..., i_n$ multiplied by the corresponding weights $w_1, ..., w_n$. This makes a single perceptron able to solve linearly separable problems like classification (Taud and Mas (2018)). This means it can differentiate between two classes only if the sets can be split with a linear function. An example of this can be seen in Figure 2 where only the left shows a problem that is linearly separable because there exists a linear function $f_{sep}$ that splits sets A and B. On the right, no such linear function can be found, and $f_{sep}$ must be a non-linear function in order to separate sets A and B.

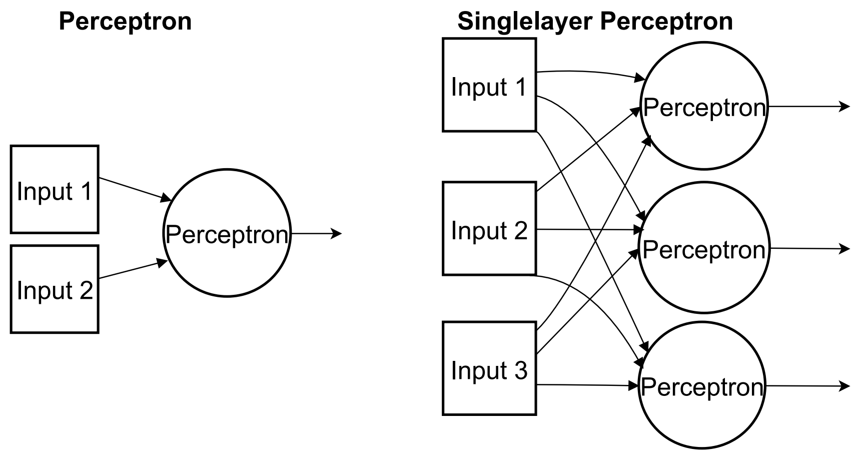


Figure 1: Left: Perceptron with 2 inputs from which an output is calculated
Right: Singlelayer Perceptron with 3 inputs and 3 perceptrons in one layer

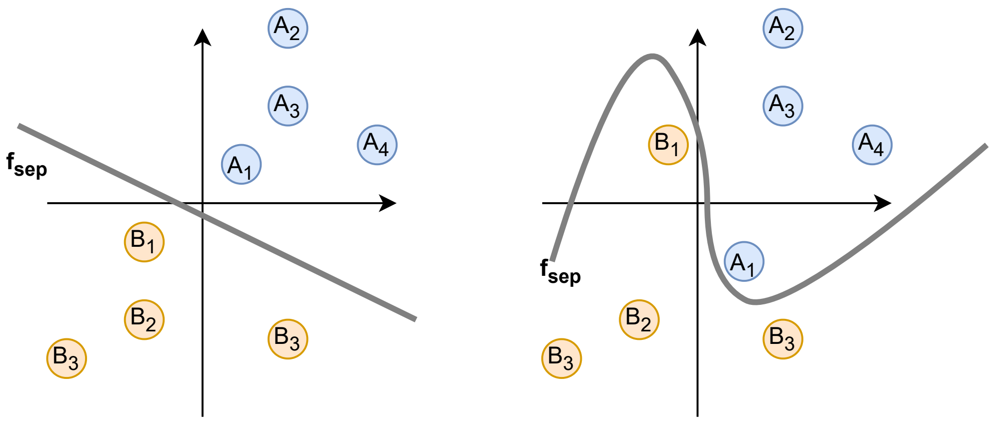


Figure 2: Left: Sets A and B are linearly separable by function $f_{sep}$
Right: Sets A and B are not linearly separable because $f_{sep}$ must be non-linear to separate them

A single perceptron only produces one output, which in many cases is not enough to solve the problem at hand. For example, a controller of game agents needs one output variable for each action they can take in their world. To overcome these issues, perceptrons can simply be organized into one layer where each perceptron receives all of the inputs and has their own individual weights for each of those inputs. This structure is called a Singlelayer Perceptron (SLP) and can be seen on the right side of Figure 1. While the SLP can produce multiple outputs, it is still only able to solve linearly separable problems (Taud and Mas (2018)). But not all problems are linearly separable, especially the more complex and, as a result, more interesting ones. Fortunately, this issue can be overcome and enable perceptrons to solve more complex problems, like taking on the task of acting as a game agent controller.

To address the limitations of the SLP, we can add more layers to it and connect every neuron in one layer with all neurons in the layers that come directly before or after it to create a Multilayer Perceptron (MLP), as seen in Figure 3. Regarding the structure of a MLP, Taud and Mas (2018) shows that usually the MLP's layers are classified as:

- Input layer: This layer directly receives all inputs without any calculations or weights applied.
- Hidden layers: A number of layers between the input and output layers that are responsible for computation and sending the outputs of each neuron forward to the next hidden layer (or, in the case of the last hidden layer, to the output layer).
- Output layer: This is the last layer in the MLP where the outputs of its neurons represent the results of all the computations performed by the MLP's layers.

The connections between all layers are unidirectional, so the outputs are only ever passed in one direction. This makes the MLP a feedforward neural network without any cycles as opposed to a recurrent neural network (RNN). RNNs have the capability to form loops and cycles, which allows them to "keep a memory of past events when applied to control problems", Risi and Togelius (2017). We will focus on MLPs in this thesis, as they will be used to build the game agents' NN. This choice has been made because MLPs are a common way to realize NNs for game agents (Risi and Togelius (2017)). Furthermore, the lower complexity of MLPs makes them well suited to testing different parameters.

The outputs of a single neuron in NNs are usually calculated by taking the weighted sum of all the inputs, which are then passed through an activation function. The activation function is usually a sigmoid function: $S(x) = (1+e^{-x})^{-1}$ (Collobert and Bengio (2004)). But in theory, different types of functions, like the logistic function, could be used[2]. This enables the MLP to solve not only linearly separable tasks but also have the capacity to approximate any function (Hornik, Stinchcombe, and White (1989)) and no longer be limited to only solving linearly separable problems.

As Taud and Mas (2018) describes, the learning in neural networks, especially MLPs, is most commonly done via backpropagation. This approach can be taken in a supervised

[2]`http://www.cse.unsw.edu.au/~billw/mldict.html#activnfn`

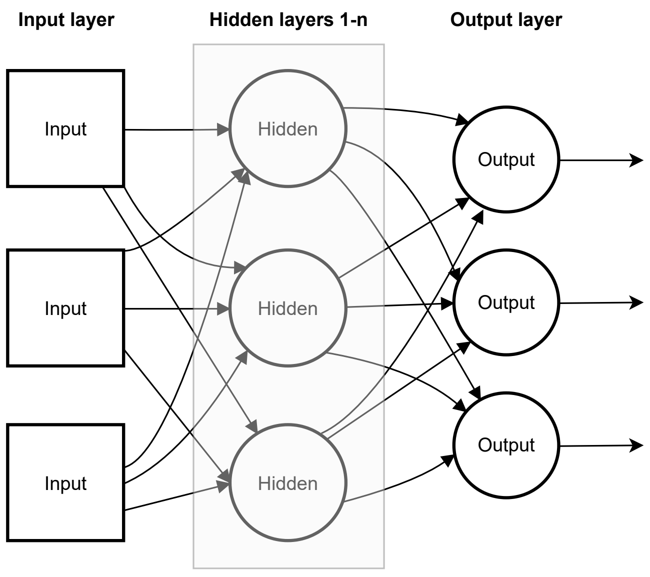


Figure 3: Multilayer perceptron: Feed-forward neural network with 3 neurons on input, hidden and output layers. Each of the connections between neurons has an individual weight $w_i$

learning environment when the expected outputs of the system are known and the difference between expected and actual outputs, also called the error, can be computed. The NN weights are then corrected in accordance with this error, starting from the back (output layer), and the error is propagated back to the other layers for correction (Gurney (2003)).

An alternative way of training NNs that does not require the NN to adjust itself is via evolution. The learning does not happen by correcting weights with backpropagation but by adapting those weights through an evolutionary algorithm. This is called neuroevolution (NE). NE algorithms show performances comparable to backpropagation methods depending on the problem domain (Whitelam, Selin, Park, and Tamblyn (2021)). Also, they have the benefit of being applicable to both unsupervised and supervised learning methods (Risi and Togelius (2017)) which makes them well suited for the application as agent controllers where the correct outputs are largely situation-dependent and not known beforehand.

## 2.2 Evolutionary Algorithms

Evolutionary Algorithms (EA) are part of a bigger algorithm family that Brabazon, O'Neill, and McGarraghy (2015) classifies as "Natural Computing Algorithms". Those algorithms are inspired by phenomena that are observed in nature and try to use the insight we get from fields like biology to profit from them in a computational setting. In particular, EAs use techniques similar to natural evolution in order to solve optimization or modeling problems (Eiben and Smith (2015)). Slowik and Kwasnicka (2020) also further classifies EAs as optimization techniques that use random search because many of the processes in EAs are stochastic and dependent on chance, much like we can observe in natural evolution.

While there are different variants of EAs, they all share the same underlying principles. Brabazon et al. (2015) describes those as population, selection, retention of fit individuals and variation. While Eiben and Smith (2015) further specifies that variation is implemented through recombination and mutation. These principles can all be seen in the general algorithmic sequence that an EA performs, called the "evolutionary cycle" by Bartz-Beielstein, Branke, Mehnen, and Mersmann (2014) and Brabazon et al. (2015). The steps of the evolutionary cycle are the following (Eiben and Smith (2015)):

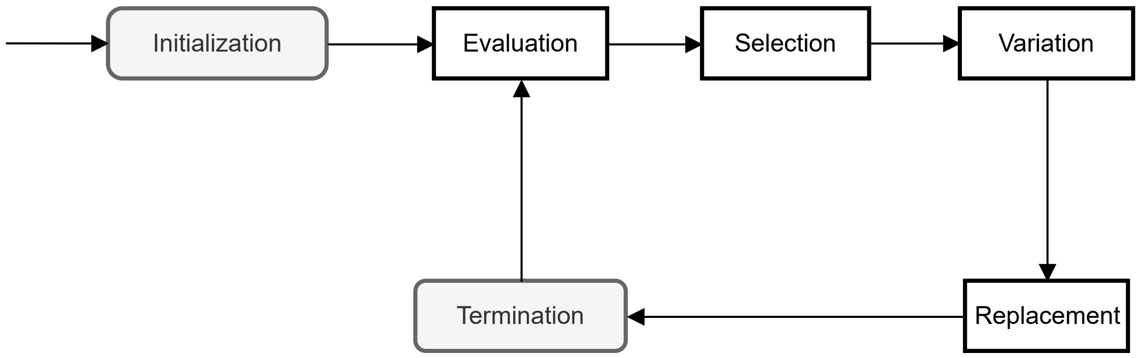


Figure 4: Evolutionary cycle: First, a population of individuals is initialized, evaluated and the best are selected for reproduction. After applying the variation operators, new individuals replace the old ones and start the cycle anew. End if the termination condition is met.

1. Initialization
   At first a population of individuals is (in most cases) randomly generated at the start of the algorithm. The individuals represent possible solutions to the problem (phenotype) and are defined by parameters that make up their encoding (genotype).
2. Evaluation
   A fitness function is used to evaluate each individual in the population. This fitness function establishes the requirements that individuals should meet and measures how well each individual solves the given problem.
3. Selection
   The individuals are then selected from the population based on their fitness values. The higher an individual's fitness value is, the higher is their chance of being selected. Selection methods can vary, but common approaches include roulette wheel selection, tournament selection, or rank-based selection.
4. Variation
   After the selection process, variation operators are applied to the selected individuals in order to create new offspring. Those operators are crossover and mutation. Crossover means a number of parent individuals (usually two) are combined to

create a new individual. Mutation, on the other hand, randomly changes the genotype values of selected individuals. This helps to introduce diversity and explore new regions of the search space for possible solutions.

5. Replacement
   Lastly, the new offspring replace a number of individuals in the current population. Usually, population size remains constant, and individuals with lower fitness are replaced by ones with higher fitness.

6. Termination
   The algorithm continues iterating through steps 2–5 until it meets a termination condition. These conditions can range from time-based ones, like elapsed real time or generation count, to fitness-focused ones, like low fitness improvements over a number of generations.

Those steps make up the general outline of EAs. Figure 4 shows a visualization of the evolutionary cycle's steps. As Eiben and Smith (2015) describes, EAs belong to the family of generate-and-test algorithms because they generate new possible solutions through variation and test them according to the criteria in their fitness function.

Two important terms when talking about EAs are "exploration" and "exploitation" (Eiben and Smith (2015)), which are vital during the search for solutions. Exploration refers to the wider search of the solution space for new regions that might contain better solutions. This is usually accomplished by the variation operators that introduce diversity to the population. The reason exploration is important is that it enables the EA to leave local optima, which are solutions that are only better than their direct neighborhood within the search space but ultimately are sub-optimal compared to solutions in other regions. On the other hand, exploitation describes the improvement of solutions in already discovered and well-performing regions. This means that good solutions are fine-tuned to find better results more quickly. In order to increase exploitation, EA parameters can be adjusted, such as increasing the rate at which strong individuals are picked as parents, so the search is concentrated in areas with good solutions.

Until now, we have looked at EAs in general as a family of algorithms. When using an EA in practice, we are working with a specific variant of the EA and, depending on this variant, different approaches to solving the problems at hand. The individual variants distinguish themselves from the others by varying things like the solution's representations or the variation operators and their applications. In literature (e.g. Sher (2013)) the most important variants of EAs are usually included in the four approaches seen in Figure 5. While those are not all variants, they are building the foundation of evolutionary algorithms.

Out of the different EAs, we will be looking further into the most commonly known variant, the Genetic Algorithm (GA). As Sher (2013) states, the main focus of GAs lies on the

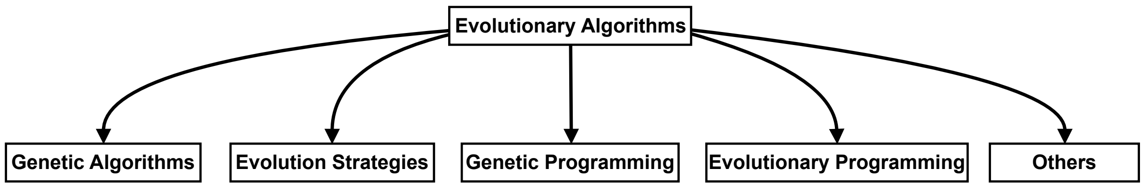


Figure 5: Most notable types of EAs

crossover operator. Because crossover already introduces variation for future generations, the mutation rate is usually kept on the lower end. While early versions of the GA only operated on binary representations, it was not long before real-valued representations were used (Eiben and Smith (2015)). Also, features like elitism, where a portion of the fittest individuals from the last generation are kept alive for the next generation, have shown that they can have a positive effect on the performance of GAs.

GAs have seen many adaptations and variants, so it is difficult to pinpoint a GA instance that encapsulates all of them, but the "simple GA" (or "canonical GA") serves as a basis for how genetic algorithms operate (Eiben and Smith (2015)). In a simple GA, the individuals are represented by bit strings, and mutation is realized by flipping each bit with the chance *mutation rate*. Crossover is done by one-point crossover, as seen in Figure 6 A). That means taking two parents, splitting their bit string in two at a cut-off point, and recombining their genes by swapping and adding them back together. The parent selection is done via roulette-wheel selection, which is a random draw, where the chance of being chosen as a parent is proportional to the individual's fitness. Building on those basic principles, many GA variants have emerged with the aforementioned changes and even more alternative parameters, like different parent selection mechanisms. As Eiben and Smith (2015) states, a popular way to select parents in GAs is via tournament selection, where the individuals compete directly against each other in tournaments and the best are chosen to reproduce. Furthermore, alternative crossover schemes like uniform crossover are applied in GAs, which can be seen in Figure 6 B). Similar to one-point crossover, the gene values are directly taken from the parents' genes, but there is no single cut-off point. Instead, the child is created by randomly picking one of the parent's values at each point. When working with real value representations, even a whole arithmetic crossover operator is possible, where the values of the parents are combined to create the values of the children. For example, it is possible to average the two parent values at each position, like in Figure 6 C). Not only crossover but also mutation can be done in a non-uniform way when the individuals are represented by real values (Eiben and Smith (2015)). For example, uniform mutation can be done by generating random values taken from a Gaussian distribution for each gene that is mutated. These values are then added to the current gene values at their respective positions. This makes mutation take into account the previous values and only make smaller adjustments to each of them. All in all, this means there are various ways to implement GAs and a lot of parameters to tweak and fit the GA to the problem at hand, which makes picking the right variant of GA already a non-trivial task.

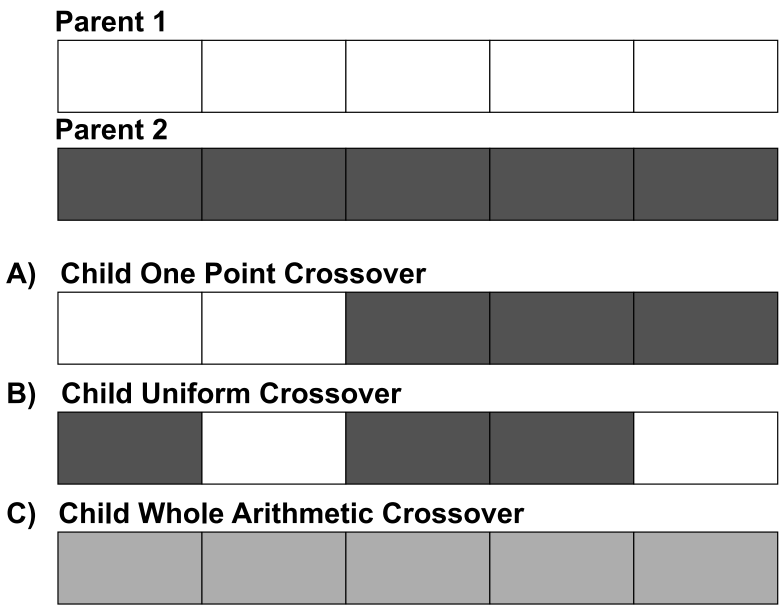

Parent 1
Parent 2
A) Child One Point Crossover
B) Child Uniform Crossover
C) Child Whole Arithmetic Crossover

Figure 6: Two parents and their offspring with different crossover types
A) one point crossover with cutoff point $n = 2$
B) uniform crossover where the child has randomly selected parent weights
C) whole arithmetic crossover with the child having the average values of the parents (for real-value representations)

Evolution Strategies (ES), on the other hand, have a different approach to GAs when it comes to realizing EAs. While ES also have a number of different sub-variants, we are first looking at the (1 + 1)-ES, which was the first ES being researched and acts as a basis for other, more sophisticated ES as presented by Brabazon et al. (2015). In (1 + 1)-ES the biggest focus lies on the selection of individuals for the next generation. At first, one child for every parent is produced. Unlike the simple GA, we don't have any crossover, so variation is only achieved through mutation of a single parent. Then the children are evaluated together with their parents, and only the best move on to the next generation. This developed into more general ES where the number of parents $\lambda$ and children $\mu$ was variable, denoted by ($\mu$ + $\lambda$)-ES. Another common ES is the ($\mu$, $\lambda$)-ES, where children always replace their parents without comparison between them. Furthermore, it is common in ES to evolve the mutation parameters for each individual together with the values that are part of the solution (Eiben and Smith (2015)). All of this introduces more parameters that have to be considered when applying the ES to a practical problem.

Evolutionary Programming (EP) and Genetic Programming (GP) differ from the aforementioned EAs in that they try to evolve more complex controllers. Those controllers are then trained to react to changing environments (EP) or can even show machine learning capabilities (GP). GP even utilizes trees as the form of representation for the individuals, with a focus on evolving a model rather than focusing on optimization, as Eiben and Smith

(2015) explains. This makes these approaches quite similar to neuroevolution, where NNs are evolved and act as controllers for agents in an unknown or changing environment. Because of this, while EP and GP can be applied to NE, they can also be seen as alternatives to it. For this reason, when realizing NE, usually GAs or ES are used to implement the evolutionary part of the system.

Picking the right variant of EA and adjusting the parameters so the given problem can be solved as effectively as possible is one of the big challenges in EAs, as Mirjalili, Faris, and Aljarah (2020) remarks. Not only do these parameters have a big impact on the performance of the EA, but finding the best parameters is also a challenging task. As Eiben and Smit (2012) shows, the topic of finding good parameters has brought forth a lot of scientific research where *parameter tuning* of EAs has been the primary focus. The EA parameters are tuned before the start of the algorithm and remain unchanged, which is not to be confused with *parameter control*, where they are changed at runtime (Karafotias, Hoogendoorn, and Eiben (2015)). While there are certain conventions for what range parameters should usually be in, most of the time "good" parameters are found through experimentation. This often leads to unclear results if more than one parameter is changed at a time, especially since there are interactions between parameter effects. That can be seen in an example Eiben and Smith (2015) gives of how diversity can be achieved by both mutation and crossover.

In consideration of the importance of EA parameter tuning, this thesis asks the question of how changing different evolutionary parameters affects the performance of the individual EA instance in the context of the neuroevolution of game agent controllers. More specifically, a type of neural network, the multilayer perceptron (Section 2.1) is evolved by adjusting the weights of the connections between individual neurons.

## 2.3 Neuroevolution in Platform Games

While it was stated that Brabazon et al. (2015) classifies EAs as Natural Computing Algorithms (NCAs), NNs also belong to the same class. This means neuroevolution combines two separate NCAs into one approach that takes inspiration from biological processes. As Sher (2013) explains, NE is also classified as a machine learning (ML) approach, whose aim it is to find suitable neural network parameters (i.e. weights between neurons) so the NN can solve a given problem.

There are other ML methods for training NNs, most notably gradient descent through backpropagation (as suggested in Section 2.1), which change the weights of NNs in other ways. In the case of gradient descent, it has been shown to have issues leaving local optima (Iba (2018)). Evolutionary algorithms, on the other hand, can introduce large amounts of exploration by varying the EA parameters, which helps to leave these local optima. This and NE's ability to explore the search space deeply through the maintenance of a population, as well as being well suited for parallelization (Stanley, Clune, Lehman, and Miikkulainen (2019)) make NE a good choice for many problem domains.

One of those strong domains of NE is the evolution of gameplay agent controllers. Risi and Togelius (2017) shows a broad overview of practical applications where NE has been used to evolve gameplay agents in various game genres. They further show that the role of NE can range from tasks such as state evaluation over content generation to directly selecting the actions the game agent should take. Platform games are a good fit for direct action selection for two reasons. First, the environment is rapidly changing when moving through 2-D space, and many things are unknown to the game agents, which makes it necessary to quickly react to changes in the environment. And second, a direct mapping of outputs to the actions of a game agent makes sense and is easy to implement because you can create one output for every button that a physical game console controller would have when playing the platform game. As Risi and Togelius (2017) explains, the number of controller inputs in 2-D platform games is usually relatively small, so this approach is often taken in practice.

With NE being a combination of NNs and EAs, there are a number of NE parameters that can be adjusted. Ranging from the NN topology and input types to the different EA parameters like mutation rate, population size, or type of parent selection, various ways to design the algorithm are possible when implementing neuroevolution.

Even the genotype representation can vary. These representations can be differentiated into direct and indirect codings (Iba (2018)). Direct codings are mapped directly to the phenotype of the individuals in the EA, for example, by having a fixed NN topology and saving the weight values of each connection. Indirect codings, on the other hand, code the genotype as a set of "rules" for how the networks should be generated. The table in Risi and Togelius (2017) shows that most of the approaches using direct action selection also rely on direct codings. While different game genres are listed, almost all of them use a MLP as a basis for their NN and either a form of GA, ES, or NEAT to evolve those.

NEAT stands for *NeuroEvolution of Augmenting Topologies* and is a commonly used way to implement NE. In NEAT, the topology of the NN is evolved together with the weights while still representing the genotype as direct codings. On the one hand, NEAT introduces benefits like removing the need to establish a NN topology beforehand by effectively trying to evolve a MLP that is "no more structurally complex than necessary" Brabazon et al. (2015). On the other hand, the fluid topologies NEAT introduces make it harder to compare the effects of changing different EA parameters and remove the option to directly compare the performances of several NN topologies over the course of a single simulation run.

This exact comparison of the effects that different NN and EA parameters have on the performance of the NE of 2-D platform game agents is the main focus of this paper. The existing literature on this topic is quite limited, mainly focusing on a testbed for EA algorithms inspired by the popular video game *Super Mario World* (SMW) as introduced by Togelius et al. (2009). This framework enables the testing of different ML variants to see how well they can evolve game agents for playing the game. The result of comparing different EAs against each other showed that MLPs performed equally well, if not better, than the more sophisticated approaches in terms of the highest level reached on average. Further, the performance of smaller-scale NNs was better (on average) than bigger-sized ones, which was especially true for MLPs. This raises the question of which MLP topologies are most suited for 2-D platform games and how the addition or removal of neurons and layers affects the EA's performance.

The Mario AI testbed provided by Togelius et al. (2009) is also publicly accessible[3] and sparked some other works investigating more aspects of NE in platform games. For example, Ortega, Shaker, Togelius, and Yannakakis (2013) uses the Mario AI to examine how suitable different NE variants are to emulate "human player behavior" and still try to play the game with the goal of completion. But while this paper shows how different NE approaches differ from one another, it does not focus on examining the impact of tweaking each NE instance's parameters. Finley (2015) also uses Mario AI to investigate multiobjective genetic algorithms for the purpose of evolving game agents and states the importance of choosing the right EA parameters in order to guarantee good performance.

While there has been some investigation regarding the changing of parameters in the aforementioned Mario AI papers, most of the time the main focus was not on the comparison between presets with different NN and EA parameters. Because of this, even more insights into the effects of individual parameters for NE in platform games could be gained by systematic testing of different presets against one another while only changing one parameter at a time. Especially the combination of different EA and NN parameters and a deeper investigation of possible connections between the two are areas worth exploring more deeply.

Moreover, the complexity of the Super Mario World video game is quite high. Multiple different blocks can be interacted with. There are moving enemies and platforms, power-ups, and many more things that can have a big impact on what the best move is at any given moment. Lowering the complexity of the game in order to focus more on the movement through a 2-D space, which is the main part of SMW and other platform games, makes it easier to test different parameters and evaluate the results.

## 2.4 Related Work

As stated before, there has been research in the domain of neuroevolution of platform game agents in projects inspired by the Mario AI competition (Togelius et al. (2009)). Going more into the implementation details, we can see how Ortega et al. (2013) sets the

[3]`http://julian.togelius.com/mariocompetition2009/`

NE parameters to tackle the problem of agents learning human-like behavior:

- inputs: a whole 65 inputs are used, including 4x7 grid views of enemies and the environment as well as distances to obstacles, state information about the game agent, and more.
- NN topology: *65 input - 5 hidden - 5 output* neurons brought the best performance, but bigger NNs with topologies like 65-10-5 or 65-5-5-5 have also been tested.
- population: 10
- parent selection: roulette-wheel
- crossover type: two point (like one-point crossover from Figure 6 but with two cut-off points)
- mutation type: Gaussian with a standard deviation of 5.0 and a 0.3 probability
- max generation count: 2000

While using these parameters, the resulting performance of NE was better than alternative methods like backpropagation. Looking at the NE parameters, it might be possible to reduce the number of inputs, especially since the best-performing NN topology was the smallest one. Further, the limitations of training time hindered the selection of larger population sizes.

When looking at Finley (2015) and the approach to evolving agents for Mario AI, the problem of a large input space has been addressed to some degree. Here, the number of inputs has been reduced to 20, with several distance measures to relevant obstacles such as objects below or above the agents. In some cases, even the size of obstacles in front of the game agents was used as an input. These and several other highly preprocessed inputs are fed into the NNs of the agents, which still means the NN has to map the 20-dimensional input space to a total of five output commands. Similar to Ortega et al. (2013) a three-layered NN was later used for testing with a 25-20-5 topology. This begs the question if there are further inputs that could have been left out, as the common trend seems to be that smaller-sized NNs have a performance benefit when it comes to evolving platform agent controllers.

When looking at NE in games with less complexity, there are two projects that are in a similar problem domain as evolving 2-D platform game agents. Both sharing the common goal of jumping, the first of these projects is Dušan Erdeljan's *neuroevolution-flappy-bird*[4]. Here the popular android game *flappy bird*[5] was rebuilt, and controllers were evolved to go as far as possible in the stage. The gameplay concept is quite simple. The game agents automatically move from left to right across the stage, and there are pipes that act

[4]`https://github.com/dusanerdeljan/neuroevolution-flappy-bird`
[5]`https://en.wikipedia.org/wiki/Flappy_Bird`

as obstacles. The aim is to press the jump button at the correct times so the bird passes through the relatively small gaps in the pipes. This means there is only one output present, namely *jumping*. There is also only one pipe at a time that is of interest, and that is the closest one. For this reason, the input space was chosen to have three values: the distance to the closest pipe, the agent's current height, and the height of the gap the agent has to pass through. These three inputs are then fed into a NN with a 3-8-2 topology, and it is determined whether the agent should jump or not. This is simply done by checking if the first output is larger than the second and jumping if it is. Evolution is done via a genetic algorithm with 0.2 elitism and a uniform random mutation that overwrites the values of the genes at a certain position with a probability of 0.1. Crossover is also done uniformly, with a 50% chance for the single child that is produced to inherit one of the two parents' weights at each position. All in all, the project has a simpler input and output space as well as a lower-complexity genetic algorithm, which is fitting for the problem at hand. This is also supported by the number of generations it takes to evolve a controller that can play the game for significant lengths of time without dying, which is usually below 100.

Another very similar project is Sujan Dutta's *jumpinggameAI*[6] project with a description on their website[7]. Here, a bouncy ball is evolved that can control its *x* and *y* velocites and has to jump over approaching obstacles. Very similar to the Flappy Bird project, the NN topology is 6-8-2 with *x* and *y* velocity, *x* and *y* position, and the next obstacle's position and height. But here the positions are given through third person coordinates and not relative distances. Also, the outputs are more complex because we are not dealing with binary decisions like *jump* or *not jump*, but real values for changing the agent's velocities that have to be adjusted. The EA does not possess any crossover and evolves new individuals by small adjustments of the old generations' weights through mutation, with a 2% mutation rate. This is similar to ES, where the focus is not on a crossover operator. But in this EA no strategy parameters have been evolved, which would be common in ES. This approach is very different from the aggressive search for new solutions in Flappy Bird, as all the changes to new generations are only in minor steps and the variation operators are not completely resetting the values.

Lastly, when looking at similar games that are not platformers and do not require game agents to jump, Togelius and Lucas (2005) is quite similar because it requires the agents to be proficient in pathfinding. Here, controllers for a car racing game are evolved to drive through a track. The main point of interest is the different input representations used and the superior performance of distance sensors compared to third-person coordinates. Further, the performance of fixed starting point positions was compared to randomized ones, which resulted in most of the controllers performing worse when positions were changed. But distance sensor-based controllers still performed relatively well overall, so

[6]`https://github.com/Suji04/jumpingameAI`

[7]`https://towardsdatascience.com/neural-network-genetic-algorithm-game-15320b3a44e3`

the effects of changes in starting position may be dependent on the problem at hand and the chosen inputs.

# 3 Project Goals

The main goal of this project is to create a 2-D environment and a NE algorithm that make testing different parameters as easy as possible. For this reason, one of our main requirements for all parts of the implementation is low complexity and readily adjustable parameters. This is also represented by the way the platform game looks, as it is made up of just platforms that the game agents can stand on and does not have any more entities that could affect the behavior of the agents.

When taking into account the aforementioned related work in Section 2.4, one main goal for this project is to combine the problem domain of 2-D platform games with the reduced complexity of some previously stated works. On a fundamental level, when removing all other factors, the most basic problem to solve in our platform game is: *When is the right time to jump?*. This can also be seen in the Mario AI implementations, as they have several inputs that are trying to guide the controllers in the right direction when addressing the issue of when to jump. For this reason, the starting point of this project is to look at similar problem domains with basic implementations like Flappy Bird and jumpingameAI. Of course, there is also the added challenge of teaching the agents to move in the right direction at any given moment, which means the controls of the platform game will be more complex.

This brings us to the question of what type of NE to pick. Because the main goal is to easily test different parameters, the NE implementation aims to be low in complexity so we can easily change the parameters of each test run. For this reason, the NE architecture in Flappy Bird can be taken as a good starting point to explore the basic behavior of neuroevolution on controllers in our own game environment. Furthermore, this means that researching how to pick good parameters for our problem also becomes a main focus.

The NN should have a static topology. If we picked an approach like NEAT, the comparison between individual networks would be very hard, as those are always changing at runtime. This means that we have to keep the NN topology consistent during execution and only change it in between runs. This has the benefit that several different NN presets can be easily compared against each other. Further, most of the related work in Section 2.4 has also been using static NNs, and some have shown good performances even with small sizes, so we can keep the overall complexity of the system lower.

The EA should either be a GA or an ES that works on real numbers. This is because we are directly evolving the NN weights, and a direct encoding of those real values is the most straightforward approach. But there is a lot of freedom in how we can implement the EA. So even fundamental things like the type of crossover and mutation, the parent selection, or the fitness evaluation can be changed. Not only that, but these things are parameters that can be changed and tested against each other. Because of this, we need to establish what the requirements for the EA in this specific problem instance are and what it should prioritize. These requirements for the EA are:

- find a solution that finishes the game
- the solution must reach the finish before a timer runs out
- this solution should be found in as few generations as possible

From these three top-level requirements, we estimate which properties the EA should have in order to fulfill them.

First of all, exploration of the solution space is very important since finding a single solution is the top priority of the EA. This means that even when the rest of the population shows poor performance, a single good individual can make up for it.

But on the other hand, exploitation of already-existing, good solutions is also an important part of the EA. This is for a number of reasons. The difference between solutions that reach the goal and solutions that barely miss it might be very small when looking at the weights of the NNs. But in the end, reaching the goal is a binary check that does not differentiate by how much it was missed. Because of this, there might be candidates that only need to be adjusted a little bit in order to reach the goal, which means exploiting the search space around those top individuals has the chance to be very beneficial for the search. Another argument that speaks for the benefit of exploitation is the type of game we are working with. Since our game world only includes static platforms, they can be seen as a kind of "check point" that has to be reached. Because of this, we can look at reaching a certain platform as part of the overarching problem that has to be solved. In theory, every candidate solution needs to solve the problem of "reaching platform $x$" first before being able to tackle the challenge of "reaching platform $y$", assuming $y$ comes after $x$. So in order to search for solutions that can reach platform $y$ it makes sense to capitalize on the already existing solutions that could reach platform $x$.

On the other hand, when the EA exploits the search space too much, there may be the danger of getting stuck in local optima. Sometimes, slightly changing a solution candidate that performs well may not lead to a solution that can reach the goal. There is also no good way to differentiate between candidates that could lead to a solution and ones that do not. Because of this, the EA needs to have enough exploration to overcome local optima. So both exploitation and exploration are vital to the EA's success, and the EA parameters need to be chosen accordingly.

This chapter provided an outline of how the game, NN, and EA should be designed to meet the main requirements: ease of testing and adjustable parameters, while also ensuring good EA performance. After the design outline is established, we can look into the research that has been done. The main three research questions are:

1. How do different parameters for the evolutionary algorithm affect agent performance?
2. What are the effects of changing the NN topology and inputs?
3. How do the agents perform in various levels of the platform game when changing parameters from questions 1 and 2?

The overarching topics of interest for all of these research questions then become:
**"which of these factors has more or less influence on the results?"** and
**"do trends exist that show how some of the factors can influence each other positively or negatively?"**

# 4 Methodology

This chapter looks at how the neuroevolution of gameplay agents in a platform game has been realized. The specifics of this game are explained in Section 4.1. Then the implementation of the neuroevolution algorithms can be seen in Section 4.2. And lastly, the chapter will look further into how all simulation data is acquired, stored, and evaluated in Section 4.3.

## 4.1 Platform Game

The platform game serves as a test bed for simulating neuroevolution in a physics-based environment. It is coded in Java with the use of the libDGX[8] library. With libGDX, it is easier to manage the different functionalities required for creating a game, like managing the screen, game stage, and input listeners.

The main reason for choosing libGDX is the integrated physics library Box2D[9], which allows the project to create a world and different entities in it that are affected by a basic physics engine. This makes it possible to have a steady downforce from gravity and collisions between physics objects.

The objects that are created in the Box2D world are multiple platforms, which are distributed differently for each level, and bot bodies, which are controlled by the neural networks of each individual gameplay agent. There is also a finish platform, which is slightly larger in size and acts as a finish line for the bots. The game agents have three tools at their disposal to reach the goal: *go left*, *go right*, and *jump*. With these actions, they have to avoid falling off the platforms and aim for the finish. An image of several bot characters trying to reach the goal on the right side can be seen in Figure 7.

[8]`https://libgdx.com/dev/`
[9]`https://box2d.org/documentation/`

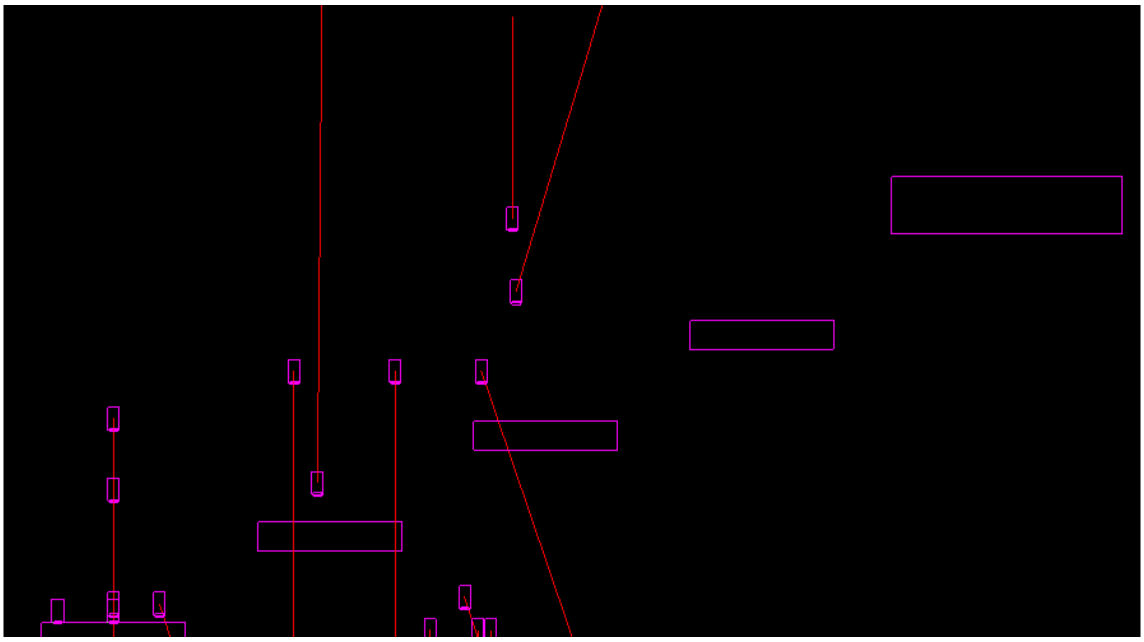


Figure 7: Basic level with four platforms, one goal (big platform to the right) and multiple bots trying to reach the goal. The current velocities which are exerted on the bots are shown by the red vectors.

The drawback of using Box2D is that every physics object has to be created from the ground up by defining all aspects like shapes, collision types, or even density beforehand. While this enables very good control over the bodies, it also takes more time than just creating custom preset objects that would be available in other engines. For example, when creating the physics bodies of the game agents, two fixtures have to be defined, one for the square shaped bot body and one for a collision sensor at the bottom of the bot's feet. The sensor is used to register whether the bot is currently standing on solid ground or not, only allowing it to jump when it is. All properties of the two fixtures then have to be defined, which includes their collision group, so they only collide with platforms and not other bots. After defining the fixtures and adjusting their size and positions, they are added to a Box2D body, which is then assigned to a single game agent.

To speed up the simulation, no sprites (overlay pictures for the physics bodies) have been used, and all objects are rendered with the Box2D debug renderer. That also means there is no discrepancy between the physical bodies and the visual information we see.

To further speed up the simulation, a *physicsSpeedup* variable has been introduced, which affects the physics steps in the world. The value represents the percentage of physics speedup relative to the default Box2d physics steps. So with a standard value of 1.0 (or 100%) there is no speedup, and with a value of 2.0, the physics steps are now double the default speed.

While it may seem that a bigger *physicsSpeedup* is always better, there is a trade-off that has to be considered. When the physics engine calculates in bigger steps, some accuracy is lost because the same in-simulation time is now calculated using bigger and thus fewer steps. This is also problematic because our game agents receive inputs of the world state less often and can make fewer decisions on which actions to take. In practice, a maximum *physicsSpeedup* of 4.0 has still yielded good results without running into issues with the physics engine or game agent behavior.

In total, there are three different levels on which the game agents are tested. The first level has already been shown in Figure 7 and acts as an easy first test to see if the game agents can make a few consecutive jumps without falling off the platforms.

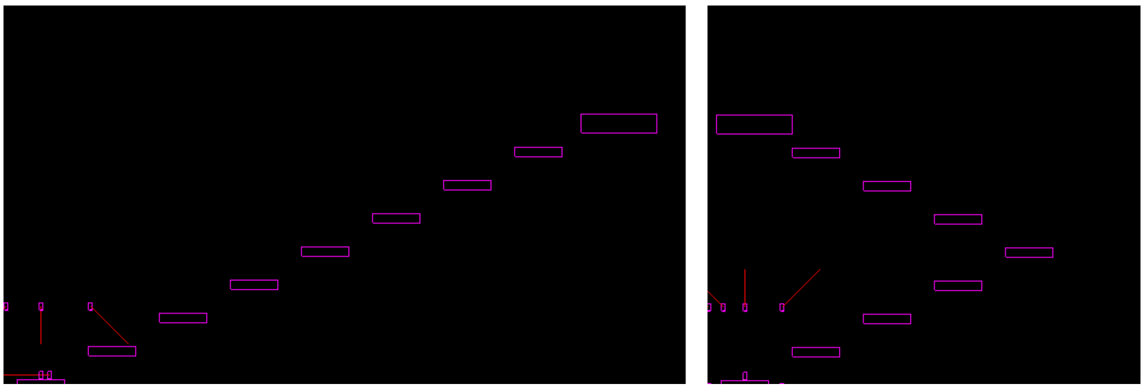


Figure 8: Left: level 2 with many consecutive jumps until the goal is reached.
Right: level 3 with a curve that game agents have to account for

The second level is a longer continuation of the first one and can be seen on the left side in Figure 8. This level was designed to test if agents were able to continue the same pattern from level 1 for a longer time. This means that the margin of error is now much smaller than in the first level. In level 1, only a few jumps are required to reach the goal, but the further you repeat the platform pattern, the greater the chance for agents to fall off. This means a very specific timing for the jumps is required to stay on the later platforms.

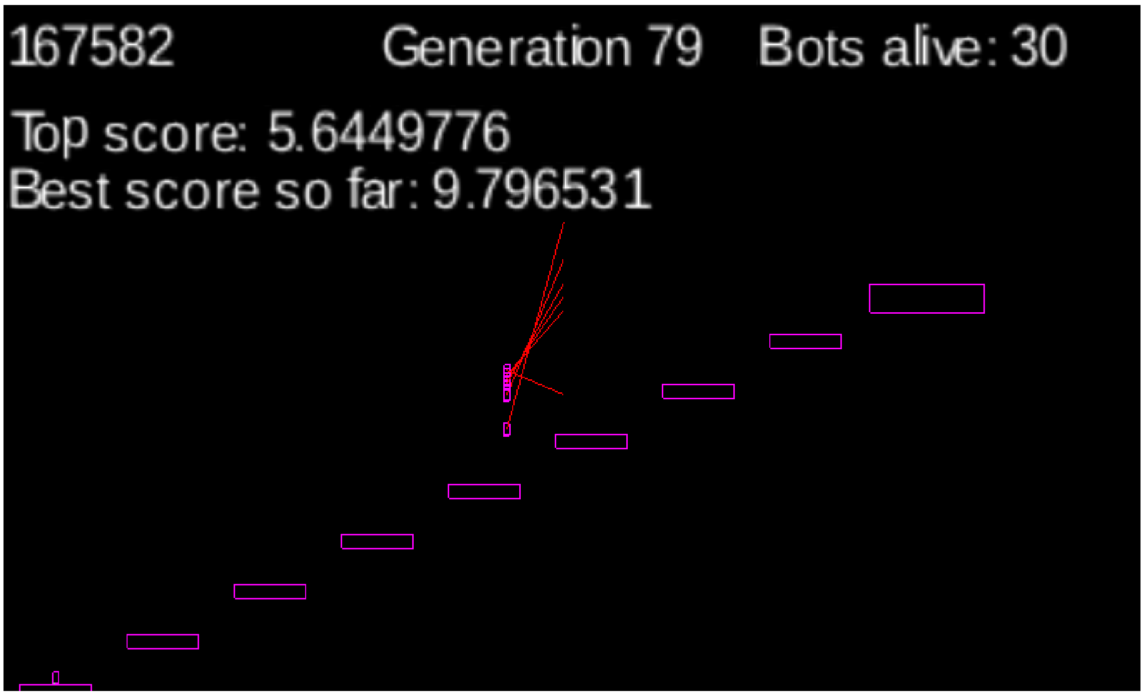
The last level is level three, as seen on the right side of Figure 8. This was added mainly as a test of how the game agents would deal with the sudden curve in the terrain. Because all platforms are solid physics objects, the bots cannot jump through the bottom of them and need to make it all the way to the rightmost platform before they can change direction and climb up further toward the goal.

When simulating the current generation, there are some statistics that are displayed for our simulation run. Those can be seen at the top of Figure 9. They are updated in real time and include:

- Top left: simulation run time in physics ticks
- Top center: current generation number
- Top right: number of bots which that are still alive and have not fallen off the stage yet

- Second row: top score in this generation
- Third row: best overall score across all generations

The top left displays the number of physics steps taken overall since the start of the program's execution. This also covers multiple simulations with different parameter presets and is not reset when another EA instance is started. So this can also be seen as an overall execution time of the program in the sense of physics computations taken until the start. This number is rather high, so for this reason, the physics speedup that has been introduced before is usually kept at 3.0 to 4.0 in order to simulate fewer physics steps in the same real time.



167582 Generation 79 Bots alive: 30

Top score: 5.6449776

Best score so far: 9.796531

Figure 9: Statistics of the current simulation rendered at the top of the simulation window

Usually, these statistics only serve as an indication of how well the current EA instance is doing when looking at the simulation. But all important statistics are also stored persistently in a database, as Section 4.3 shows.

## 4.2 Neuroevolution Implementation

After implementing the physical bodies of the game agents, they need a way to decide which actions they should take. This is done via a neural network, which acts as the bot's "brain" and determines what actions it should take at any given point in the simulation.

### 4.2.1 Neural Network Implementation

The NN implementation closely follows the flappy bird project but uses different (and varying) NN topologies as well as different inputs and outputs. Further, the project's

FlattenNetwork class, which maps the whole neural network into a one-dimensional array, was expanded by methods so we can save the values of this array into a string in the database. This means we can later look up the topology and values of the NN from strong individuals.

The neural network consists of a multilayer perceptron with an input layer, hidden layers, and an output layer. The exact topology is varied in different test runs and is one of the things that is considered when comparing several parameter presets to investigate research question number 2. But all presets share the same basic structure and can be described as follows: The input layer receives information about the current state of the world and itself as inputs. In the first hidden layer, each neuron receives a weighted sum of those inputs and is then activated by the commonly used sigmoid activation function $(1+e^{-v_i})^{-1}$ (Taud and Mas (2018)) where $v$ is the vector of all inputs $v_i$ with $i \in \{1, 2, ..., n\}$ and $n$ as the number of inputs.

This process is repeated for each hidden layer as well as the final layer. This last output layer consists of three neurons, which are directly mapped to the actions a game agent can take. That means, depending on the values of the three neurons in the output layer, the bot takes one of the actions *go left, go right* or *jump*, but can also choose to do nothing. This evaluation of inputs and choice of an output action is done up to 60 times per second. Depending on the simulation speedup, this number is changed in the inverse way, so 50% as many evaluations at 200% speedup.

### 4.2.2 Evolutionary Algorithm Implementation

The game agents' neural networks are then evolved with an evolutionary algorithm. A population of 100 individual bots has shown a good trade-off between population size and the limits of the physics engine. When the number of bots grows even further, the simulation performance starts to dip, and physics steps cannot be executed in the required time any more. This means the execution in real time becomes slower and thus loses some of the speed-up benefit that is gained with larger population sizes.

A common way to implement NE is to evolve the weights of the connections between neurons in the NN, as the examples of video game NE algorithms in Risi and Togelius (2017) show. This can be realized with an evolutionary algorithm on real-valued numbers, as we are looking for a vector of real numbers that represent the weights of the NN connections. This vector should be capable of creating a game agent controller that can successfully reach the goal and complete the level.

The EA implementation loosely follows the neuroevolution-flappy-bird project, as it has been used as a baseline for the structure of a simple EA and the used parameters are serving as an EA instance for value testing. The EA implementation has been greatly expanded upon, however. Multiple EA instances can be tested during one program execution, and the number of variable parameters has been increased by *eaType* and *nnType* which load the EA and NN parameters from a database. Further, the variables *isUniform*, *parentSelection*, *crossoverType*, *scoreEvaluation*, *fitnessCalculation*, and *NN_input* have been introduced to enable a finer adjustment of the EA parameters. There have been many more adjustments, which can be seen in the rest of this chapter.

The underlying implementation of an evolutionary algorithm used in this thesis has a number of parameters that can be seen in Table 1. Those parameters can be changed between runs in a single program execution, and the performance of individual presets can be compared against one another. Because comparison is easier when there is a default, each parameter has been assigned a standard value. This makes it easy to create presets that just change one parameter in isolation and observe how it changes the performance of the EA instance. More information on how parameters are compared can be seen in Section 4.3.2, as this chapter focuses on the EA implementation.

| Variable | Description | Default value |
|---|---|---|
| *population_size* | number of individual bots in each generation | 100 |
| *mutation_rate* | percentage chance that a mutation operator is performed on a single weight of the NN | 0.1 |
| *is_uniform* | if true no step size is used and mutation randomly sets new weights uniformly in $[-1,1]$ | false |
| *mutation_step* | step size for the mutation if it is not uniform. Determines standard deviation of added value with Gaussian distribution | 0.5 |
| *elitism_rate* | percentage of *population_size* how many of the individuals with the best fitness are kept alive for the next generation | 0.2 |
| *randomness_rate* | percentage of *population_size* how many completely randomized individuals are generated each generation | 0 |
| *child_count* | how many children are made during crossover between two parent genes | 2 |
| *parent_selection* | determines how parents are chosen for crossover | deterministic |
| *crossover_type* | the way how the weights of the parents are combined to produce new children | arithmetic |
| *score_evaluation* | how the actors can increase their score during simulation | default |
| *fitness_calculation* | how the score is normalized so fitness proportional selection is possible | unitySum |

Table 1: The variables of the evolutionary algorithm and their default values used

At the very start of each generation, a new population of bots is spawned. During one generation life cycle, the performance of those bots is then evaluated by the *score_evaluation* variable. The higher their score, the better the bot's performance. This score is calculated by using a combination of the minimum distance to the goal and the index of the highest platform reached. The formula used is $(\frac{100*g}{1+d_{\mathbf{min}}} + p_{\mathbf{max}})$ where $d_{\mathbf{min}}$ is the shortest distance to the goal reached, $p_{\mathbf{max}} \in \mathbf{N}$ index of the highest platform touched and $g \in \{1, 10\}$ either 1 if the goal has not been reached or 10 if it has. This makes reaching the goal a top priority, as the score will be a lot higher for bots that reach it. At the same time, this score evaluation encourages going in the goal's direction while also rewarding exploration by taking into account the highest platform reached. $d_{\mathbf{min}}$ also affects the score much more when the bots are closer to the goal. This helps guide the evolutionary algorithm's selection of the bots in the right direction when they are close to the goal.

At the start, the score evaluation function just consisted of the inverse of the minimum distance to goal $(d_{\mathbf{min}})^{-1}$. This was then changed to $\frac{100}{1+d_{\mathbf{min}}}$ so there were no issues with the division when the bots reached the goal, and the values were also mapped to an interval around 0.5 to 100 to get rid of too many leading zeros. To show the range of this distance score, here is a quick example: When starting out in level 2, the bots have a score of around 0.75. This score would then gradually increase the closer the bots got to the goal. Around the time the bots reached the 5th platform, they would have a score of around 1.8 and when reaching the last platform before the goal, the score was in the range of 5 to 7 depending on how good their last jump was. When reaching a point so close to the goal, the score changes rapidly due to the nature of the function used.

One drawback of just using the distances to the goal for score calculation was that it was hard to see which bots had reached the goal and which bots had just come very close and touched its side but ultimately fell off before reaching the top. So in order to better discern bots that reached the goal from bots that have not, the goal multiplier $g$ was introduced. That means when looking at their scores in the database, we can easily see when a bot was able to stand on the final goal platform, as its score would be an order of magnitude higher. Furthermore, this means that picking an individual who has reached the goal is much more likely for fitness proportional selection. And since the main goal of evolution is to produce an individual that can complete a level, the main objective is already complete, and we can search aggressively for mutations or recombinations with the top individuals to maybe even find a better solution. But this also means that the factor of 10 could, in theory, be exchanged for another value that accomplishes a similar function. In practice, values lower than 10 made it not as clear which individuals had reached the goal from their score values, and the top individuals did not produce as many offspring in some EA configurations with fitness proportional selection. And a value greater than 10 seemed to not bring any real improvements to the discernibility of scores or the production of offspring, but the population would start to converge too quickly toward the top individual.

The last variable $p_{\mathbf{max}}$ of the current score evaluation function $(\frac{100*g}{1+d_{\mathbf{min}}} + p_{\mathbf{max}})$ was introduced mainly because agents were having a hard time traversing the layout of level 3. In this level, the goal is directly above the starting point (Figure 8) so the agents start off by having to move away from the goal, and just jumping in place became a strategy that evolved across a few generations. For this reason, exploration was rewarded by adding the index of the highest platform reached to the score, of which there are a total of four in level 1 and eight in levels 2 and 3. This had the effect that, especially at the start of a generation when the distance scores were on the lower end, reaching new platforms was a relatively big improvement to the individual's score. This may seem redundant with the distance part of the function, but in our platform game, an agent that reaches platform two is preferred to one that just jumps at its side and gets an almost equal distance score. Of course, this does not necessarily mean there is not at least some redundancy, and this interaction would need to be investigated further. But in practice, the addition of the platform indices to the score calculation has only been observed to bring a faster orientation of agents in the right direction at the start of a generation.

**Algorithm 1** nextGeneration() method of the EA class evolves its population and handles miscellaneous tasks that do not directly affect the population's evolution

```
1: $alive = populationSize$;
2: $population$.fitnessEvaluation($fitnessCalculation$);
3: DatabaseConnector.saveGeneration($population$, $generation$, $runID$);
4: $population$.evolve($elitism$, $randomness$, $mutationRate$, $mutationRange$, $childCount$,
   $mutationStep$, $isUniform$, $parentSelection$, $crossoverType$, $scoreEvaluation$);
5: $generation \leftarrow generation + 1$;
```

While the *score_evaluation* variable is of interest during the simulation of a generation, most others come into effect after a generation has reached the end of its lifetime. To understand where and when the other variables are used, we look at how the population evolves into a new generation. We can see the overarching nextGeneration() method of the *EvolutionaryAlgorithm* class in Algorithm 1. While this method represents the most top-level overview of an evolution, the following algorithms and explanations will go deeper into the specific implementations of those methods. The nextGeneration() method updates the number of alive individuals as well as saving the current generation to the database. Further, it increments the current generation by one. But the main three steps the method takes are the following:

1. evaluate the fitness of each individual in the old generation and put them in descending order as seen in Algorithm 1 line 2.
2. pick the top individuals according to *elitism rate* and randomly generate new ones in alignment with *randomness rate*.
3. perform crossover and mutation on the old generation and fill up the new generation with newly generated individuals.

The first step normalizes the bots' scores to a fitness value between 0 and 1. This is achieved with the *unity sum*, where we are simply summing up all the score values of the current generation and setting the fitness value to the bot's score divided by the total score in the generation, as provided by neuroevolution-flappy-bird, which can also be seen in Algorithm 2. This not only makes it easy to compare different scoring systems, as regardless of the score values, the fitness will always be normalized. It also guarantees the fitness values of the whole generation add up to 1, which enables fitness proportional parent selection to be used, as Eiben and Smith (2015) explains.

**Algorithm 2** The normalizeFitnessDistribution() method as implemented by neuroevolution-flappy-bird. This guarantees the fitness values of the whole generation add up to 1.

```
1: $sum \leftarrow 0$
2: for each $genome$ in this.$genomes$ do
3:     $sum \leftarrow sum + genome$.getBot().getScore()
4: end for
5: for each $genome$ in this.$genomes$ do
6:     $genome$.setFitness($genome$.getBot().getScore()/$sum$)
7: end for
```

The second and third steps are propagated to the evolve() method in the population class, which can be seen in Algorithm 3. As input arguments, the method gets (*elitism*, *randomness*, *mutationRate*, *mutationRange*, *childCount*, *mutationStep*, *isUniform*, *parentSelection*, *crossoverType*, *scoreEvaluation*) and performs the evolution on the current population, which is stored in a list of genotypes called *genomes*. This list is the single attribute of the Population class, so it can be easily modified with the class methods. Because *genomes* has already been evaluated and ordered by fitness in the aforementioned step 1, we can simply take the top $elitism \times population\ size$ individuals and put them in the next generation to implement elitism. Then we can add $randomness \times population\ size$. individuals with a newly generated neural network that has completely randomized weights between $-1$ and 1. Lastly, we update the genomes list with the resulting next generation from one of the two methods: randomSelectNextGen(*args...*) or deterministicSelectNextGen(*args...*). Depending on the chosen parameter for *parentSelection* a different method for choosing the parents for crossover is used.

The first of the two used parent selection methods is random fitness proportional selection, as seen in Algorithm 4. As a first step, the parents are selected, and crossover is applied to get *childCount* children, which are assigned a unique body number so they can control one of the bot physics bodies. Those children are then added to the *nextGeneration* until the size of the next generation has reached the size of genomes, which is the list of individuals in the old population. At the end, the method returns the new generation, which is now filled with elites, random individuals, and the newly generated children.

How exactly the random parent selection from Algorithm 4 line 3 is implemented can be seen in Algorithm 5. Here, the normalizing of the fitness values in Algorithm 2 as well as the ordering of the old population by fitness can be used to our benefit. The method implements a roulette wheel algorithm as suggested by Eiben and Smith (2015), but instead of using a list with a cumulative probability distribution and checking if *rand1* and *rand2* are smaller, we subtract the fitness value of the individuals from *rand1* and

**Algorithm 3** The evolve(*args...*) method of the Population class. Here the population stored in the *genomes* list is evolved to a new generation

```
Require: elitism, randomness, mutationRate, mutationRange, childCount, mutationStep,
    isUniform, parentSelection, crossoverType, scoreEvaluation
 1: nextGeneration ← empty list of Genotype
 2: eliteCount ← round($elitism \times population\ size$)
 3: for $i \leftarrow 1$ to eliteCount do
 4:     nextGeneration.add(genome with top $i$ fitness)
 5: end for
 6: randomCount ← round($randomness \times population\ size$)
 7: for $i \leftarrow 0$ to $randomCount - 1$ do
 8:     net ← new NeuralNetwork() with established weight topology
 9:     for $j \leftarrow 1$ to (size of $net.weights - 1$) do
10:         $net.weights[j] \leftarrow$ random number $\in [-1, 1]$
11:     end for
12:     nextGeneration.add(new Genotype(net))
13: end for
14: if parentSelection equals "roulette" then
15:     genomes ← randomSelectNextGen(args...)
16: else if parentSelection equals "deterministic" then
17:     genomes ← deterministicSelectNextGen(args...)
18: else
19:     throw RuntimeException("no valid parent selection parameter")
20: end if
```

**Algorithm 4** The randomSelectNextGen(args...) method fills up the *nextGeneration* with children from randomly selected parents

```
Require: mutationRate, mutationRange, childCount, mutationStep, nextGeneration,
    isUniform, scoreEvaluation, crossoverType
 1: outerloop:
 2: while true do
 3:     parents ← randomParentSelection()
 4:     children ← crossOver(args...)
 5:     for each child in children do
 6:         child.assignBodyNumber(size of nextGeneration)
 7:         nextGeneration.add(child)
 8:         if size of nextGeneration $\geq$ population size then
 9:             break outerloop
10:         end if
11:     end for
12: end while
13: return nextGeneration
```

*rand2* until they are $\leq 0$. Then we set *parent1* and *parent2* respectively.

To go through the implementation of the roulette-wheel parent selection in Algorithm 5 let us consider the following example: Assume we had a population of three individuals, $i_1$, $i_2$, and $i_3$, with normalized fitness values of $f_1 = 0.7$, $f_2 = 0.2$, and $f_3 = 0.1$ (which add up to 1), as well as two random values $\in [0,1]$: $rand1 = 0.6$ and $rand2 = 0.9$. In the first iteration of the for-loop, we would subtract $f_1$ from both random values, so now $rand1 = -0.1$ and $rand2 = 0.2$. Then we set *parent*1 to the first genome in our ordered old population *genomes*, which for our example is $i_1$ and we do not set *parent2* yet. In the second iteration, *rand*2 now becomes 0, and we set *parent2* to be $i_2$. This means every individual is chosen with probability $f_i$ to be a parent for the next generation, as we effectively divide the interval $[0,1]$ into smaller intervals that are equivalent to the normalized fitness of the population's individuals and check if the random numbers drawn are in that specific interval. This method of choosing the parents is possible because the fitness values of all individuals add up to 1, and every random number between 0 and 1 can be assigned to exactly one individual.

**Algorithm 5** In the randomParentSelection() method returns two randomly selected parents from the old generation. Selection chance is fitness proportional

```
1: rand1 ← random number $\in [0,1]$
2: rand2 ← random number $\in [0,1]$
3: parent1 ← null
4: parent2 ← null
5: parent1Set ← false
6: parent2Set ← false
7: for each genome in genomes do
8:     rand1 ← rand1 − genome.getFitness()
9:     rand2 ← rand2 − genome.getFitness()
10:    if not parent1Set and rand1 $\leq 0$ then
11:        parent1 ← genome
12:        parent1Set ← true
13:    end if
14:    if not parent2Set and rand2 $\leq 0$ then
15:        parent2 ← genome
16:        parent2Set ← true
17:    end if
18:    if parent1Set and parent2Set then
19:        break
20:    end if
21: end for
22: selectedParents ← empty list of Genotype
23: selectedParents.add(parent1)
24: selectedParents.add(parent2)
25: return selectedParents
```

When looking back at Algorithm 3, the alternative to random parent selection was a deterministic selection, which can be seen in Algorithm 6. Here, the individuals are not chosen proportionally to their fitness but rather according to the rank they take when sorting the whole generation by fitness values. This specific variant of selection closely follows neuroevolution-flappy-bird and implements deterministic selection with very high selection pressure, so the individuals with the highest fitness are selected very often. Effectively, the parent selection starts off by taking the strongest individual of the last generation as the first parent and the second strongest one as the second parent. The algorithm continues to pick pairs of parents in a predetermined order until *nextGeneration* has enough individuals to fill up a new population. In practice, this predetermined order of $(i, j)$, where $i$ and $j$ are the indices of the individuals in the ordered genomes list, would be $(0,1)$, $(0,2)$, $(1,2)$, $(0,3)$,... and so on.

**Algorithm 6** The deterministicSelectionNextGen(args...) method deterministically picks parents for crossover and returns the resulting children in *nextGeneration*

```
Require: mutationRate, mutationRange, childCount, mutationStep, nextGeneration,
    isUniform, scoreEvaluation, crossoverType
 1: maxParentIndex ← 1
 2: outerloop:
 3: while true do
 4:   for i ← 0 to maxParentIndex − 1 do
 5:     parents ← new list of Genotype
 6:     parents.add(genomes[i])
 7:     parents.add(genomes[maxParentIndex])
 8:     children ← crossOver(parents, childCount, mutationRate, mutationRange, mu-
        tationStep, isUniform, scoreEvaluation, crossoverType)
 9:     for each child in children do
10:       child.assignBodyNumber(size of nextGeneration)
11:       nextGeneration.add(child)
12:       if size of nextGeneration ≥ population size then
13:         break outerloop
14:       end if
15:     end for
16:   end for
17:   maxParentIndex ← maxParentIndex + 1
18:   if maxParentIndex ≥ population size then
19:     maxParentIndex ← 0
20:   end if
21: end while
22: return nextGeneration
```

Now that the parents for the recombination are chosen, we can look at how it is handled in the crossover(args...) method, which is depicted in Algorithm 7. First, this method saves the NNs of the parents in the *parentNets* list. Depending on the *crossoverType* a different crossover operation is used to set the weights of the children's neural network weights based on the parent networks. In the case of discrete crossover, each weight is uniformly and randomly selected from one of the parent networks. And if the crossover type is arithmetic, each of the weights is the average of all parent networks combined. After crossover, the resulting *childNet* is mutated according to *mutationRate* and *mutationRange* when applying uniform mutation, and also uses *mutationStep* if the mutation is nonuniform. The mutated child is then added to the *children* list as a new genotype, and the process starting from line 6 is repeated when *childCount* $\geq 2$. Finally, the list of children is returned.

In theory, the crossover can be done between two or more parents, as the method takes *parents* as the list of parents to use for recombination. In practice, this has not been connected to a variable parameter like *parentCount* yet, which would also require more adjustments to the parent selection methods. Currently, they always select two parents for crossover, but Ting (2005) shows that multiparent evolutionary algorithms are a topic worth exploring when trying to optimize the performance of EAs.

**Algorithm 7** The crossOver(args...) method handles the two crossover types, discrete and arithmetic crossover. It also mutates the children after recombination

```
Require: parents, childCount, mutationRate, mutationRange, mutationStep, isUniform,
    scoreEvaluation, crossoverType
 1: children ← empty list of Genotype
 2: parentNets ← empty list of NeuralNetwork
 3: for each parent in parents do
 4:     parentNets.add(parent.bot.getNeuralNetwork())
 5: end for
 6: for ch ← 0 to childCount − 1 do
 7:     childNet ← parentNets[0]
 8:     if crossoverType equals "discrete" then
 9:         for i ← 0 to size of childNet.weights − 1 do
10:             randomIndex ← random integer from 0 to size of parentNets − 1
11:             newWeight ← parentNets[randomIndex].weights[i]
12:             childNet.weights[i] ← newWeight
13:         end for
14:     else if crossoverType equals "arithmetic" then
15:         for i ← 0 to size of childNet.weights − 1 do
16:             childNet.weights[i] ← getWeightAverage(parentNets, i)
17:         end for
18:     else if not crossoverType equals "none" then
19:         throw RuntimeException("invalid crossover type parameter")
20:     end if
21:     if isUniform then
22:         uniformMutation(mutationRate, mutationRange, childNet)
23:     else
24:         nonuniformMutation(mutationRate, mutationRange, mutationStep, childNet)
25:     end if
26:     children.add(new Genotype(childNet))
27: end for
28: return children
```

When looking further into how the mutation is implemented, one of the methods used in the crossOver(args...) method from Algorithm 7 is nonuniform mutation. As Eiben and Smith (2015) explains, this is usually the more common form of mutation for real value representations and is realized by adding a random value from a "Gaussian distribution with mean zero and user-specified standard deviation" while keeping the value restrained in the lower and upper bounds of possible values. This has been implemented by a method that iterates over each weight in the child network and mutates the specific weight with chance $mutationRate \in [0,1]$. By using the Java Random class' nextGaussian()[10] method we can generate a random value $r$ from a Gaussian distribution with a mean of zero and a standard deviation of 1, $r$ ~ N(0, 1). We then adjust this value to the required mutation step size $m_s$ by calculating the adjusted random value $r_s = r * m_s$. This adjusted random value is then added to the current value $v_c$ while making sure it stays within the mutation range $m_r$ of $[-m_r, m_r]$. So we set the new value $v_{\mathbf{new}} = \mathbf{max}(-m_r,\ \mathbf{min}(m_r,\ v_c + r_s))$. At the end, the original *childnet* weight is updated with the new value $v_{\mathbf{new}}$ and the process is repeated for all existing weights of the NN.

The uniform mutation works similarly but is much simpler, as here the new values for the child net are just drawn uniformly randomly from the mutation range $[-m_r, m_r]$.

This concludes the evolution of a generation of individuals. The list of children that results from the crossover and mutation operations is set as the new population, and the generation count goes up by one when looking back at the top-level method that started the evolution process seen in Algorithm 1. Now the same process is repeated until we reach the maximum number of generations that have been configured. How the configuration is done and how the simulation data is stored can be seen in the next section.

## 4.3 Test Data

While running the simulations, we need to keep track of the performance of the individual game agents across multiple generations as well as the evolutionary algorithm parameters used for each simulation run. For that purpose, the simulation data is stored persistently in a SQLite database with the help of the JDBC API[11]. A database connector class establishes a connection with the SQLite DB and handles the storage of simulation data, so it only needs to be called at the appropriate spots in the simulation.

The database connector class also loads the evolutionary algorithm parameter presets that are stored in the DB. This means there are no discrepancies between the presets stored in the database and the EA parameters of the runs, for which the data is stored during execution.

[10] `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`
[11] `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`

### 4.3.1 Database Structure

The SQLite database, which stores all the simulation data, has been split up into four main tables. Those tables are *agents*, *runs*, *NN_parameters* and *EA_parameters*. The database schema can be seen in Figure 10. After a few iterations, this has been found to reduce the amount of redundant data by a large margin. In the early stages, it was quite clear that storing most parameters in the *agents* table would be too much overhead, as this is by far the biggest table of the four. At runtime, every individual from every generation of every simulation run is stored in *agents* so the size of rows quickly goes into the several hundreds of thousands. On the other hand, *runs* only saves one entry per simulation execution until the number of maximum generations is reached and is several orders of magnitude smaller. For example, a standard simulation with 100 population size and 100 generations would have 10,000 times as many agents as runs. Finally, *NN_parameters* and *EA_parameters* are just storages for previously determined NE presets, so no new values are added at runtime at all.

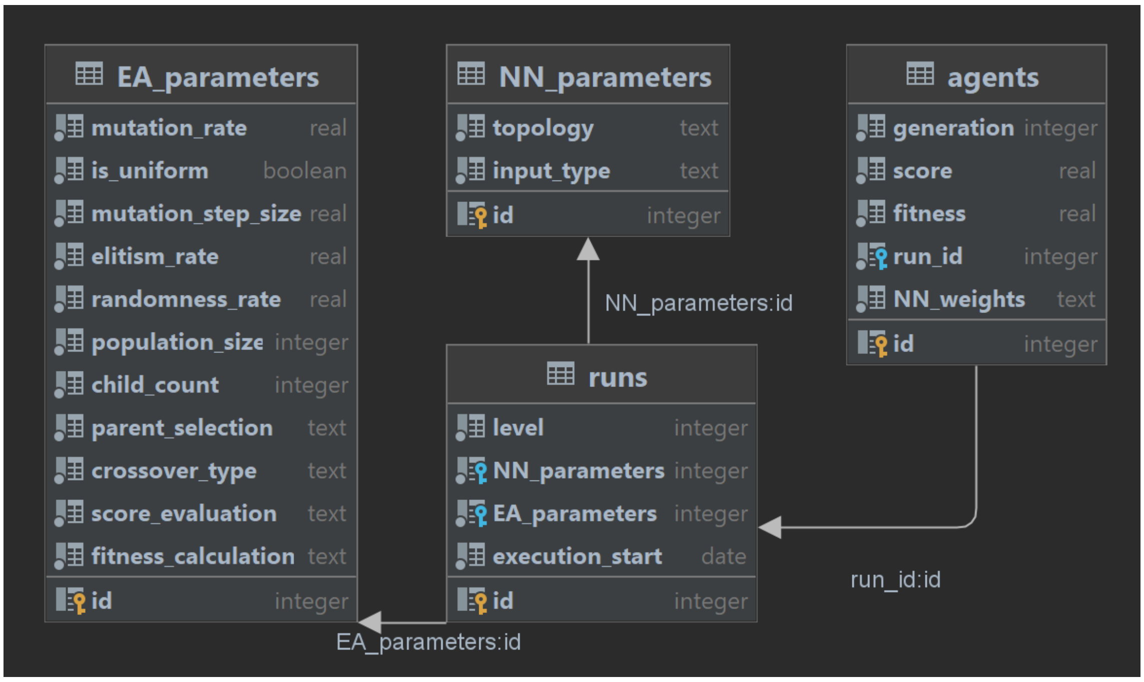


Figure 10: Diagram of the core database architecture. All agents have a foreign key for the specific simulation run they took part in. And every run has foreign keys for the EA and NN parameters it was executed with.

Starting with the *agents* table, *generation*, *score* and *fitness* are stored together with *NN_weights*. This last variable is an exact representation of all the weights in this individual's neural network. This makes it possible to observe well-performing NNs and even load their values for future usage. At the same time, it makes for a rather large text representation, which in turn means more storage space is required. Nevertheless, the option to load certain individuals for later usage provides interesting options, like loading a well-performing agent's NN into another level, although these options have not been explored as of yet.

Every agent has a *run_id* as a foreign key to a specific run. This encapsulates the whole execution of a single EA instance, which was started at the time *execution_start* with a specific NE configuration that uses *NN_parameters* and *EA_parameters*. All presets of NN and EA parameters that are compared to one another are stored in these two tables.

For *NN_parameters* we can vary the NNs' *topology*, so add or remove neurons from one layer or add more layers. We can also change the *input_type*, which determines what kind of input the neural network receives. For example, the NN can receive the coordinates of nearby platforms as input, or alternatively, the distance and angle to those platforms.

For the *EA_parameters* a lot more values can be adjusted. Those have already been explained in Table 1 and the default values of the parameters have been shown. But there are many more presets of EA and NN parameters stored in these tables, which brings us to the next section.

### 4.3.2 Simulations with different Parameters

To enable easy switching between different simulation parameters, they are loaded from the database whenever a new EA instance is initialized. Rather than changing several parameters to many different values, we are looking at predefined parameter presets that are set before the simulation starts. Not only does this make comparing different presets easier, but it also makes sure not too many parameter presets are tested. When having fewer configurations, they can be tested in more runs, which is crucial for comparing EAs since they are inherently stochastic processes and single runs can differ quite a lot from the actual average performance of the EA instance. A main reason for this is that the time constraints of the thesis limit the number of NN and EA presets that can be compared, especially because all combinations of these presets are tested on every level.

| **id** | **topology** | **input_type** |
|---|---|---|
| 1 | 7,5,3 | distances |
| 2 | 7,5,5,3 | distances |
| 3 | 7,7,7,3 | distances |
| 4 | 7,5,3 | positions |
| 5 | 7,5,5,3 | positions |
| 6 | 7,7,7,3 | positions |

Table 2: NN_parameters table values

Starting with the *NN_parameters* table, there are six different NN configurations stored in it, as Table 2 shows. The **topology** column changes the NN layout according to the numbers stored. Each number represents the number of neurons in a layer, going from the input layer (left) to the hidden layers and the output layer (right). It can be seen that, effectively, only the hidden layer sizes and amounts are changed. This is because the number of actions the bots can take is always three (move left, move right, and jump), which means the output layer always has to be of this size. The input layer, however, could be changed, but it stays at size seven for easier comparison between the different presets. Changing the number of inputs would be a form of change to the input type, which is already handled by the values in the second column, and more presets have been avoided as of now. The topology presets were picked in consideration of the results observed in Section 2.4, where often times smaller topology sizes showed superior performances (Ortega et al. (2013)).

With the **input_type** column, we can switch between distances and positions, which completely changes the values the NN receives. This means the neural network effectively "sees" something completely different, as the inputs are the only way for the NN to know what is happening in the environment.

When the NN input type is set to distances, the NN receives the following inputs:

- distance to the goal
- distances to closest three platforms
- angles to the closest three platforms

All of these values have been normalized to the same range by dividing the distances by the maximum distance in the current level. More specifically, the distance from point (0,0) to point (*worldWidth*, *worldHeight*) is taken as a maximum distance, which means the resulting inputs all lie between 0 and 1. This is very important so the NN can function properly, as Risi and Togelius (2017) explains. Otherwise, there would be issues when one input value is much larger than the rest because the MLP implementation works by propagating weighted sums to the next layer, and one of the inputs would have a bigger influence on the output.

The second NN input type still lets the NN receive the distance to the goal but switches to positional coordinates of itself and close platforms. So the values received when the input type is *positions* are:

- distance to the goal
- X and Y position of the agent
- X and Y positions of the closest two platforms

Here we switch away from an egocentric view of the game agent, as now we don't calculate the distance or angle to the platforms beforehand but only relay the raw positional coordinates. This also means information about only the two closest platforms can be fit into an input array of size seven, as opposed to the three closest ones with input type distances. Again, all input values have been mapped to the same interval of [0, 1] by dividing the X coordinates by *worldWidth* and the Y values by *worldHeight* respectively.

In order to investigate the first research question "*How do different parameters for the evolutionary algorithm affect agent performance?*", suitable parameters have to be chosen that will be investigated further. To identify suitable parameters, a good baseline algorithm has to be picked beforehand. This was achieved through preliminary research, where several different combinations of EA parameters have been tested. After arriving at a satisfactory baseline for comparison, it was determined to further investigate the effect of six parameters that either had a significant impact on performance in the initial tests or were not varied at all at the start and were hence a good choice for further analysis.

When looking at the *EA_parameters* table (Table 3) we can see the exact EA presets that have been used. It is important to note that every preset has an ID, and we are only looking further into the ones listed in the table. More EA presets have been stored in the database and have been briefly tested, but a number of them have not produced any solutions at all. For this reason and also because running multiple simulations for different combinations of NN parameters, EA parameters, and levels would take too long, with about 10 minutes per simulation of population 100, the simulation results will be limited to those presets.

| id | m_rate | uni | m_step | elit | rand | pop | child | p_select | cross | s_eval | f_calc |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 0.1 | 1 | 0 | 0.2 | 0.2 | 100 | 1 | deterministic | discrete | std | unitySum |
| 20 | 0.1 | 0 | 0.5 | 0.2 | 0 | 100 | 2 | deterministic | discrete | std | unitySum |
| 21 | 0.1 | 0 | 0.5 | 0.2 | 0 | 100 | 2 | deterministic | arithmetic | std | unitySum |
| 22 | 0.1 | 1 | 0 | 0.2 | 0 | 100 | 2 | deterministic | arithmetic | std | unitySum |
| 23 | 0.2 | 0 | 0.5 | 0.2 | 0 | 100 | 2 | deterministic | arithmetic | std | unitySum |
| 24 | 0.1 | 0 | 0.5 | 0.2 | 0 | 100 | 2 | roulette | arithmetic | std | unitySum |
| 25 | 0.1 | 0 | 0.5 | 0 | 0 | 100 | 2 | deterministic | arithmetic | std | unitySum |
| 26 | 0.1 | 0 | 0.5 | 0.2 | 0 | 100 | 1 | deterministic | arithmetic | std | unitySum |

Table 3: EA_parameters table values, written out column names from left to right: mutation_rate, is_uniform, mutation_step, elitism_rate, random_rate, population_size, child_count, parent_selection, crossover_type, score_evaluation, fitness_calculation

The default EA preset, which has also been shown in Table 1, has id 21. This combination of EA parameters has been shown to produce individuals who reach the goal at a relatively high rate. Starting with this as a base, only one parameter is changed at a time for all presets with id $\geq 20$. This allows us to make direct comparisons between those EA instances and their performance. After determining these presets, the first research question can be divided into six specific sub-questions, where each of them lines up with one of the presets $EA \in \{20, 22, 23, 24, 25, 26\}$. These sub-research questions will be referred to as questions 1(a) to 1(f) and can be stated as follows:
"What effects on agent performance can be observed when changing

(a) *crossover_type* from arithmetic to discrete crossover?" (preset 20)

(b) the mutation from non-uniform to uniform mutation, so *is_uniform* = TRUE and *mutation_step* = 0?" (preset 22)

(c) *mutation_rate* from 10% to 20%?" (preset 23)

(d) *parent_selection* from deterministic to roulette?" (preset 24)

(e) *elitism_rate* from 0.2 to 0?" (preset 25)

(f) *child_count* from 2 to 1?" (preset 26)

The only preset left is the one with ID 10, which represents an older standard preset where some tests have been made by altering its parameters one by one and comparing the results. Preset 21 has taken its place after adding some more configurations to the EA parameters, but the old preset will be used to have one cross-reference of a preset where more than just one variable has been altered. This preset is quite similar to the basic parameters used in neuroevolution-flappy-bird and the only one in this list that uses the *randomness_rate* to generate completely randomized individuals.

### 4.3.3 Processing and Visualizing the Data

While storing as much data as possible for every individual in every population in every run of the simulation may seem like a good thing at first, it also brings the challenge of making sense of the very large amount of data. In fact, most of the time, we do not really care about most individuals from a generation and are mainly interested in only the best one. For that reason, a visualization table called *top_gen* has been created in the database, where all the data about only the best individuals of each generation is gathered. This already reduces the number of entries by 99%, as the standard population size used in our presets is 100.

Using the *top_gen* table as a starting point, we can now determine the performance of each EA. More specifically, there have been two performance measures used to analyze how well one EA instance performed. Following the explanations in Eiben and Smith (2015), the success rate (SR) and mean best fitness (MBF) have been determined.These measures are then stored in the table *performance_measures*. The values are grouped by *EA_parameters*, *NN_parameters*, *level* and *generation*, so we get the average performance of all different NE configurations for each level across the generation lifetime of the EA.

Using the *performance_measures* we can now visualize the progression of our two performance measures. To do this, the data has been exported to a .csv file and imported to a *Google Sheets*[12] table, which provides nice filtering and diagram options for data visualization. An example visualization can be seen in Figure 11, where we can see the progression of our two performance measures' averages across all runs with the specified preset of $EA = 21$, $NN = 1$, $level = 2$.

## 4.4 Program execution

Before program execution, the adjustment of the parameters is done via a *config.properties* file in the source folder of the project. This file allows us to change which of the predefined

[12]`https://www.google.com/sheets/about/`

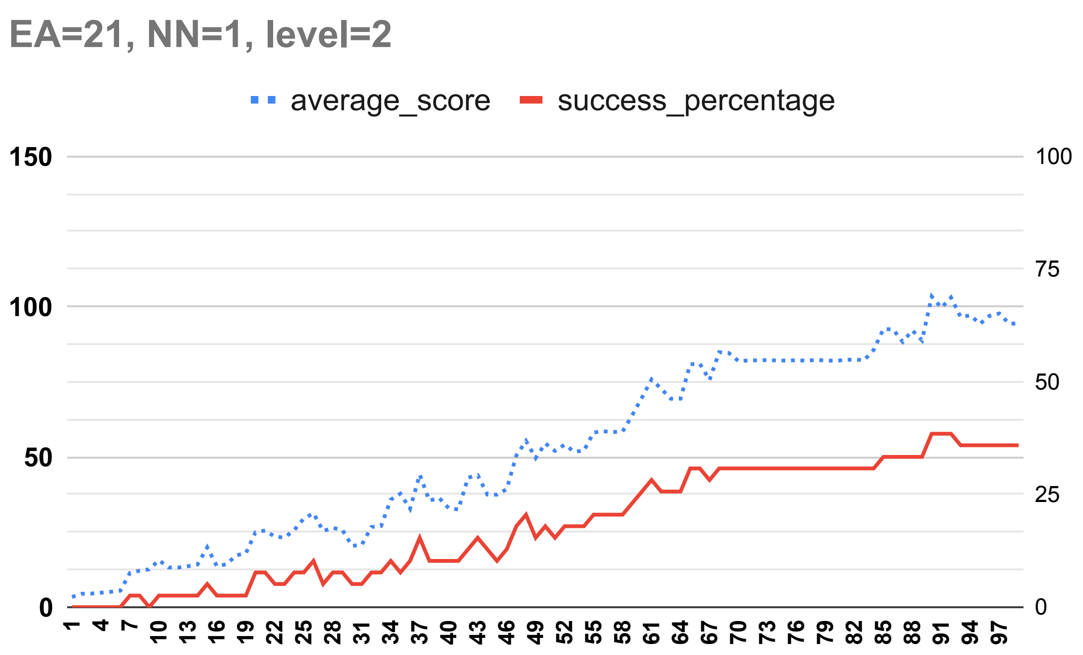


Figure 11: MBF and SR averaged across all simulations with parameters EA=21, NN=1, level=2
*x*-axis: generation, left *y*-axis: MBF, right *y*-axis: SR in %

EA and NN presets will be loaded and used during simulation. In addition to that, we can change the simulation level, physics speedup, and maximum generation number. So the config file is our central point for interacting with the system and choosing which EA and NN instances we want to compare later.

When the program is executing, it starts a run with the chosen parameters that continues until the maximum generation is reached. Then, both the EA and NN instances are reset, and a new configuration is loaded. This does not necessarily have to be the same configuration, as there is a class that handles the input from the config file in such a way that multiple EA and NN parameters can be loaded when they are separated by a comma. Those parameters will then be executed in order by cycling through all of them to ensure each combination of EA and NN parameters is picked. So for example, this could be the inside of our config.properties file:

- ea.config = 21,22,23,24
- nn.config = 1,2
- level = 2
- maxGen = 100
- physics.speedup = 4.0

On the first run of our program execution, this loads the parameters for $EA = 21$, $NN = 1$ and simulates for 100 generations on level 2 with a 4.0-time speedup. After the first reset, the parameters $EA = 22$, $NN = 1$ will be loaded and used for another 100 generations, the same with 23 and 24. Then after that, the EA parameter loops around and a new NN parameter will be picked, so the next in line would be $EA = 21$, $NN = 2$. This ensures an equal distribution of the number of times each preset combination is simulated. Furthermore, this enables longer runs since, at some point, the benefit of simulating more runs of a commonly picked preset combination is not that big. Or rather, the benefit of more simulations with seldom-picked presets is way higher, since the higher the number of test runs, the more confident we can be about the results of evaluating these inherently stochastic algorithms. Because simulations take a lot of real time, it is vital that the program can execute for long periods of time and store the performance data of multiple preset combinations.

# 5 Results & Discussion

The data that is collected during simulations is then examined, and the effects of varying the parameters are analyzed. This is explained in Section 4.3.3 and gives us the two measures of success rate (SR) and mean best fitness (MBF). Because SR is the rate at which agents reach the goal, it is represented as a percentage value between 0 and 100. MBF, on the other hand, displays the average best score of the agents and can go up to 250. The MBF measure makes improvements in the agents' progression more clear even if none of them has reached the goal yet, and SR would be 0. In the following sections, we will look at those performance measures in more detail when changing:

- simulation levels
- EA parameters
- NN parameters

And then we consider if there is an interaction between individual parameters when more than one type is changed.

All of the preset values used in the following sections are shown in Table 2 for NN presets and Table 3 for EA presets, respectively.

After sections 5.1 to 5.4 were written, a bug was detected that caused the inputs that the agents received to be unexpected in some cases, effectively creating noisy inputs. This means the results of these sections have to be considered in light of the issue found. The exact complications and the process of searching for their origins are explained in Section 5.5. Then a comparison between the data before and after resolving the issue is made in Section 5.6.

The reason why the noise in the inputs was not detected earlier was that the performance of the NE algorithms was still relatively good, with many presets showing high success rates in completing the levels. Only after plotting all performance measures into line charts were some discrepancies discovered that did not stand out before deeper data analysis. Considering the number of agent entries in the database had already exceeded 10 million at that point, it was very hard to locate the issue before aggregating the data in such a way that it became easier for humans to analyze.

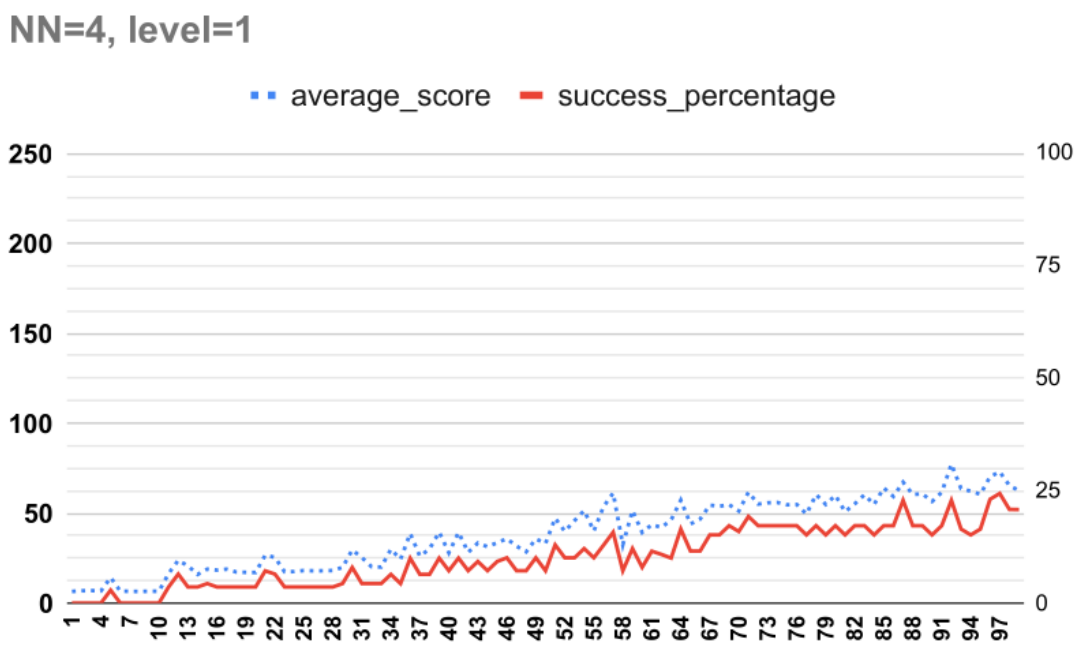
That being said, the data acquired until that point can still be analyzed in order to answer the research questions. It was collected in over 1400 runs, with most runs having 100 generations of 100 individuals. This required many simulations, taking up a lot of real time. The number of simulations given the time constraints of the thesis was limited, but the speedup factor of 4.0 helped to decrease the time an average run with 100 generations took to under 10 minutes, which was a big improvement to the time without speedup.

## 5.1 NN Parameter

The first simulations when implementing the program have been done with coordinates as the input type for the NN, as previously explained in Section 4.3.2. After some time, it became clear, however, that this was not the best way to feed information to the NN in such a way that it could be used to create good game agent controllers. No matter the topology changes made, the game agents were not able to consistently reach the goal in level 1 within 100 generations. The best results with distances as an input type were obtained with NN preset 4, which was around 25 SR after 100 generations in level 1, as shown in Figure 12 (top). With only one fourth of all runs finding a solution with those parameters on average, it was clear that there had to be things changed other than the NN topology in order to perform better at this first, relatively simple level. For this reason, another input type was explored to improve the performance of the bots.

This new input type fed the NN the distances and angles to the platforms instead of their x and y coordinates. Risi and Togelius (2017) already shows several examples of NE in games where relative position sensors and angle sensors have been used for the NN of game agent controllers. Compared to the old coordinate input type, the new one performed quite a bit better on the initial tests on level 1. The improvements were so drastic, in fact, that reaching the goal in level became the norm for each run with NN preset number 1. This can be seen in Figure 12 (bottom), as now the SR was around 85% and the agents had already reached a SR $\geq$ 75% starting from generation number 45.

Because of this vast difference in performance between coordinates and distances as input types for the NN, the second part of research question number 2, "*What are the effects of changing NN topology and inputs?*" can already be answered. Changing the NN input type has a very big impact on how many instances of EAs are able to find solutions for the given problem of completing level 1 and also how quickly those solutions can be found.



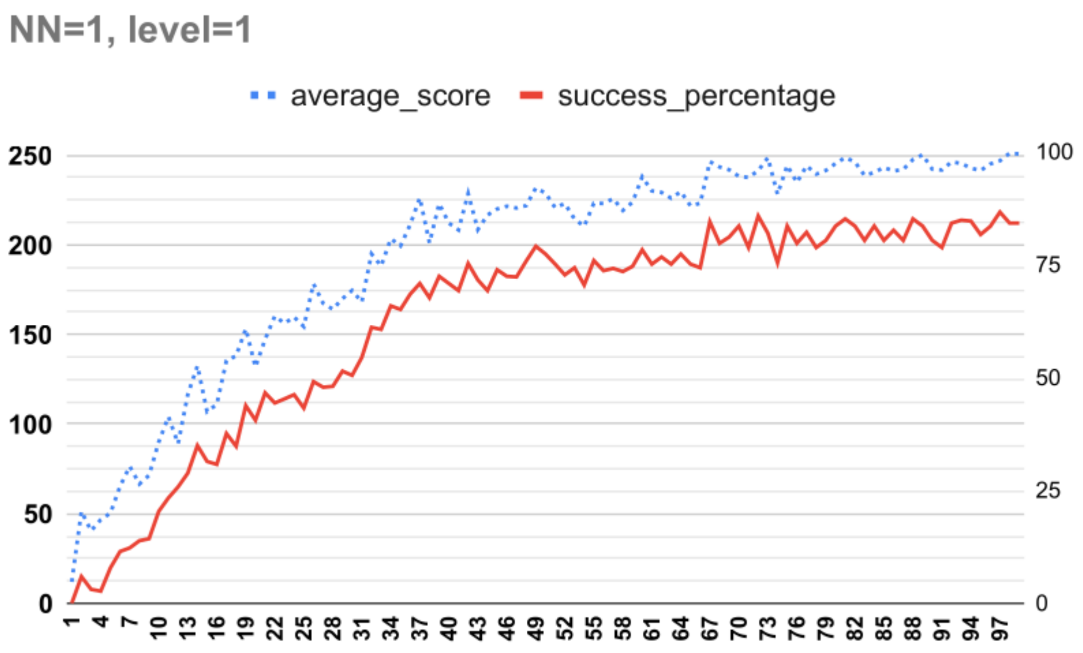


Figure 12: MBF and SR for level 1 and NN preset 1 (top) vs NN preset 4 (bottom)
*x*-axis: generation, left *y*-axis: MBF, right *y*-axis: SR in %

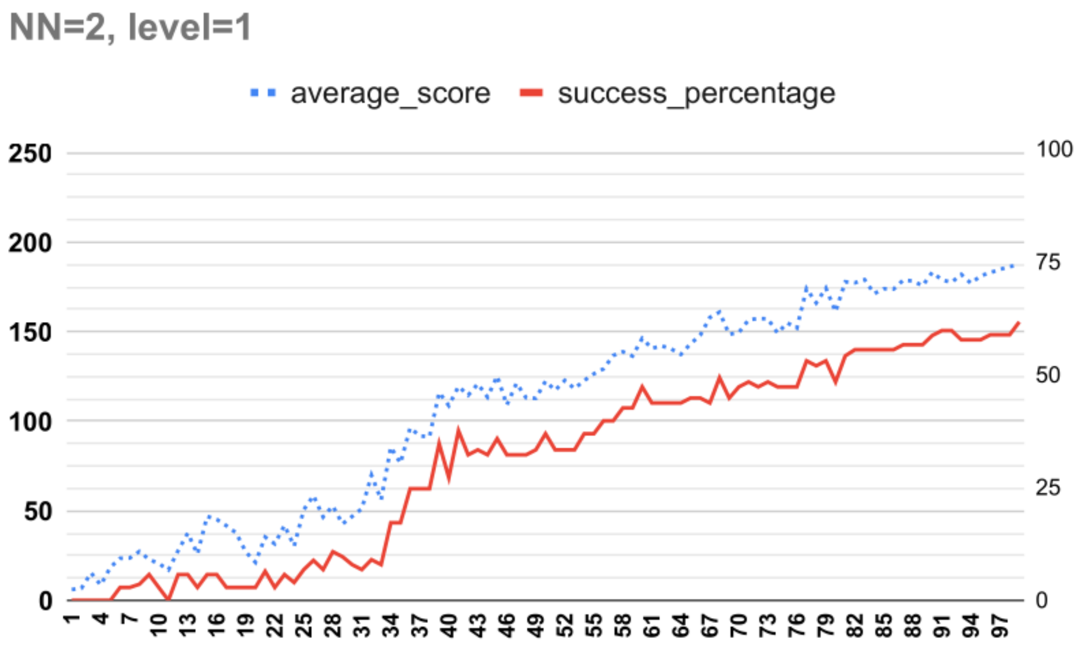
This can also be generalized to more than just the platform game that has been researched in this thesis. There has been research where the performance comparison between these two input types came to the same conclusion. This can be seen in Togelius and Lucas (2005), where the results suggested a big improvement from positional inputs to egocentric distance inputs. Further, they made the hypothesis that mapping the coordinates to the actions that are taken by individuals is probably non-linear. In that case, it would explain why game agents struggle with making connections between their own positions and the positions of obstacles instead of just being given the relevant distances as inputs.

After looking at the NN input types, the second NN parameter that was varied during the simulations was the topology. The three topology presets that have been tested range from a smaller topology with one hidden layer, over a medium topology with two hidden layers, and lastly, two bigger hidden layer sizes (Section 4.3.2).

When we looked at the performance of NN presets 2 and 3 in level 1, those two had noticeably worse MBF and SR compared to the aforementioned NN preset 1. But surprisingly, the difference between NN presets 2 and 3 was not big at all. This can be seen in Figure 13. While preset 3 shows a rapid early increase in MBF and SR at around generation 17, after that point they are only slowly climbing upward. Preset 2, on the other hand, takes a bit longer to increase its MBF and SR but improves its performance during generations 34 and 40 by a big margin. Despite those differences, both presets end up at around 60% SR and just below 200 MBF after 100 generations, which means they ultimately show close performances after the default number of generations. But compared to preset 1, these results are clearly inferior, as it took that preset only 45 generations on average to reach 75% SR, and by that, it already had 15 percentage points more SR than those two had after 100 generations. These findings line up with the results of related work in Section 2.4, where smaller NN topologies showed better performance when evolving game controllers.

Similarly to the distance input NN presets, the performance measures for presets with the input type coordinates only decreased when increasing the size of the NN topology. Only here the performance measures were not that high in the first place with preset 4, as seen in Figure 12. This meant that almost none of the runs with NN parameters 5 and 6 even produced individuals that could finish the first level. As with presets 2 and 3, both of the performance measures of the bigger topologies were relatively equal but did not even reach SR $\geq 10$ after 100 generations, so they were quickly deemed unsound choices for good NN presets in the first level.

The fact that the smaller topologies that have been tested show better performance lines up with the findings in Togelius et al. (2009), researching NE in the Mario AI environment, where small MLP topologies outperformed medium and large ones. In the paper, it is assumed that the bigger the NN input spaces are, the more NE struggles to evolve fit controllers because of the high number of inputs to consider. Personally, I agree with that assessment, and it is also one of the main reasons why the input spaces in this project



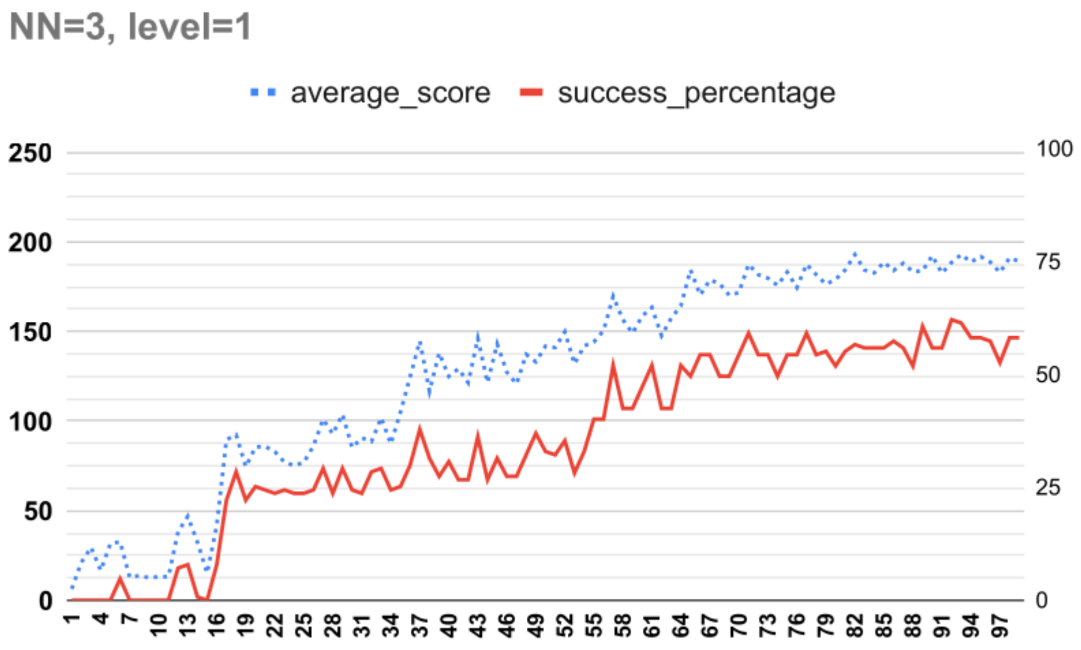


Figure 13: MBF and SR for level 1 and NN preset 2 (top) vs NN preset 3 (bottom)
*x*-axis: generation, left *y*-axis: MBF, right *y*-axis: SR in %

are only seven inputs big. But the number of inputs is clearly not the only part of the NN topology that affects its performance, because all of the presets tested in this project had the same number of inputs, and only the number of neurons in the hidden layers was adjusted. These additional neurons allow for more complex behaviors but also increase the complexity of relatively simple steps. For example, if the aim is to jump when we are about to leave a platform and already know which direction to move in, in theory we only need the distance to the closest platform and maybe the next platform to perform the simple action of *jump* without falling off. This means that adding more neurons or hidden layers may enable more sophisticated controllers, but at the same time, it impedes the quick learning of simple actions. It is probably for this reason that controllers did not perform as well in this project because the game world, input spaces, and output actions were specifically designed to be of low complexity. This means NN topologies that perform well when it comes to learning relatively simple tasks have an advantage, especially in level 1, which is the most straightforward level.

This answers the second part of research question 2. Changing the topology of the NN also has a big impact on the quality of the controller that can be evolved with NE. In particular, smaller topologies have an advantage over bigger ones in level 1.

To summarize, NN preset 1 did show the best performance in level 1 by far. This preset is a combination of the distance input type, which proved superior to positions, and a small topology that also had vastly better performances than bigger ones. The performance of this preset was so good, in fact, that its SR almost reached 100% after 100 generations.

The high SR of NN preset 1 raised the question of how well EA presets can be compared when a solution is found on every run anyway. For this reason, two new levels have been introduced to further examine how well the NE can evolve good controllers in harder levels. Particularly interesting is the question of how applicable the aforementioned results are when looking at different game environments.

## 5.2 Levels in the Simulation

At first, we can look at the general statistics for the levels over all EA and NN parameters. As Figure 14 shows, there is a significant difference in the average scores in each level. The goal of creating harder levels has certainly been reached. Because of this, we can compare runs with well-performing presets like $NN = 1$ against each other without the MBF and SR being close to the maximum values and missing potential changes.

It can be seen just how much easier level 1 is in comparison because of the short length of the level. Level 2 was designed to be a longer continuation of level 1. While this may look like a problem quite similar in difficulty to solving the first level, as the distance between platforms is the same as in level 1, it is quite a challenging task for the game agents. This is because the margin of error is a lot smaller than the first level allows. So even game

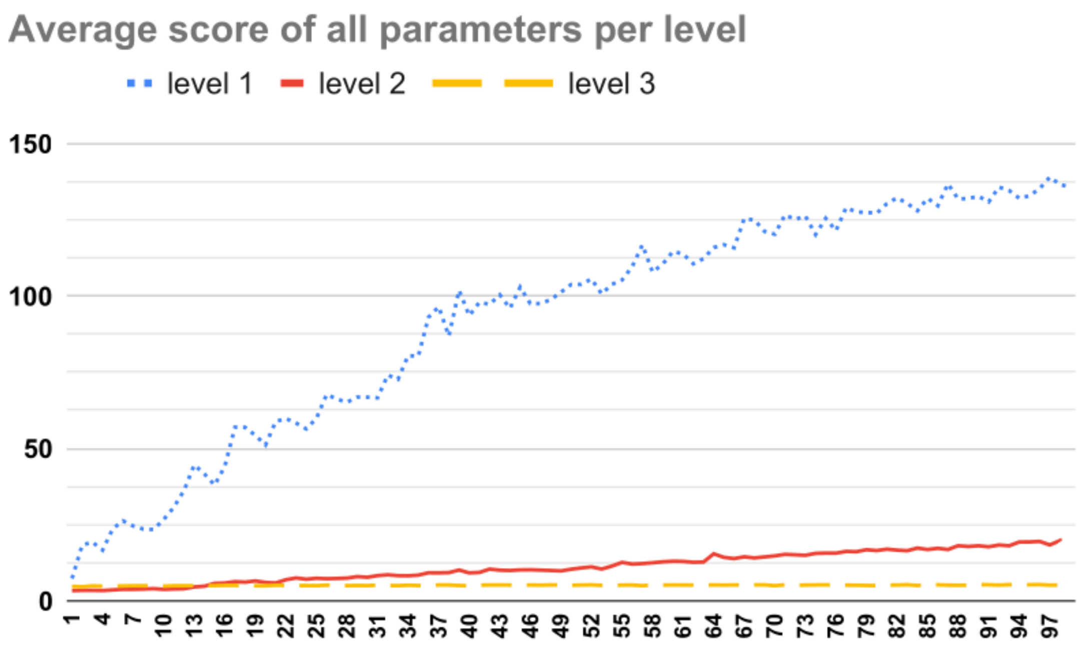


Figure 14: Average MBF of all EA and NN parameter presets tested on each level
*x*-axis: generation, *y*-axis: MBF

agents with a moderately good NN can solve the first level, but the further you repeat the platform pattern, the more agents cannot reach the next platform. For this reason, a direct comparison between the performance of individual presets in levels 1 and 2 is of interest in order to get insights into the topic of the third research question. This can also show if the relative performance difference between the presets stays the same or if it changes in level 2.

The performance data for all NN presets in levels 1 and 2 can be seen in Table 4. We are specifically only looking at generation 100 to focus on the most important performance measure that the presets show after the default last generation. From this table, we can see both MBF and SR in each level, as well as a relative comparison of the two values between levels. This is simply done by dividing the performance measures in level 2 by the ones in level 1 and storing the result as a percentage value. If we look at the data, we can see that all NN presets had worse performance in level 2, but some displayed a bigger decline in MBF and SR than others. The highest scoring preset, $NN = 1$ for example, showed SR and MBF in level 2, which were around 20% as high as those in level 1. Presets 2 and 3 fell off more drastically with $\text{MBF}_2$ being less than 7.5% of $\text{MBF}_1$ and $\text{SR}_2$ less than 5.2% of $\text{SR}_1$ for both. On the other hand, the performance of preset 4 did not decrease by quite as much, with both SR and MBF in level 2 being around one-third of that in level 1. Technically, presets 5 and 6 showed the smallest decrease in MBF and SR, but that is probably due to their bad performance in level 1 to begin with. With SR being 0 in both levels 1 and 2, the decrease in MBF is the only way to measure a difference in performance between the two levels.

When we interpret the data from Table 4, there are two things that stand out. The first is that the ranking between presets when looking at SR and MBF changes from level 1 to level 2. While $NN = 1$ stays the best preset, in level 2 $NN = 4$ is in second place, with a considerable difference to $NN = 2$ and $NN = 3$ that were superior in level 1. This is quite surprising, as the preset with a simpler topology seems to be better at solving more complex levels. One reason for this might be that, given the limit of 100 generations, good solutions can evolve more quickly when the number of neurons is smaller. It is quite possible that with higher maximum generation counts, bigger topologies will overtake the smaller ones at some point. The second part that stands out is that the presets that used distances as inputs showed a smaller decrease in performance in level 2. Especially the almost 7% SR of preset 4 is surprising, considering it only achieved slightly more than 20% in level 1. This would hint toward positional inputs hindering the initial evolution of agents toward good NNs that can solve the first level but ultimately helping to find agents that can clear long levels after those NNs have been found.

| NN | SR level 1 | SR level 2 | $\mathbf{SR_2/SR_1}$ | MBF level 1 | MBF level 2 | $\mathbf{MBF_2/MBF_1}$ |
|---|---|---|---|---|---|---|
| 1 | 84.20 | 16.99 | 20.17% | 250.76 | 50.52 | 20.15% |
| 2 | 61.70 | 2.83 | 4.58% | 187.18 | 13.87 | 7.41% |
| 3 | 58.15 | 2.96 | 5.10% | 189.12 | 13.32 | 7.04% |
| 4 | 20.73 | 6.89 | 33.25% | 62.53 | 23.08 | 36.91% |
| 5 | 0 | 0 | - | 7.29 | 3.67 | 50.38% |
| 6 | 0 | 0 | - | 5.94 | 2.25 | 37.87% |

Table 4: NN preset performance measures averaged over all NN presets for generation 100 in level 1 and 2 as well as a comparison between SR and MBF in those levels with the percentages of $\mathbf{SR_2/SR_1}$ and $\mathbf{MBF_2/MBF_1}$

In contrast to levels 1 and 2, there was not a single preset combination tested that was able to generate agents capable of completing level 3. In fact, none of the agents ever got a score higher than 5.6 because they were not able to get past the curve in the level (Figure 8). It was certainly expected that level three would be the hardest of the levels by far, and the challenge of evolving controllers to suddenly reverse the direction in which agents go had been added for exactly this reason of increasing difficulty. But the complete lack of a single successful controller in almost 9000 generations with 100 individuals each surpassed the initial expectations of difficulty by far.

There were several attempts to change the algorithm in order to produce controllers that cleared level 3. At first, the variable of highest platform reached was added to the score evaluation function to encourage the agents to explore new platforms (Section 4.2.2). While this had the effect of making agents reach the last platform before the curve more quickly, it did not yield any higher scores than before. Most of the time, agents would get stuck under the platform just before the rightmost one. That is because the agents evolved to try and jump through the top platform and skip the one on the right, which can be seen in

Figure 15. The problem with that approach is that the platforms are solid physics objects, and there is no way for the agents to pass through the bottom. This limitation is given by the Box2D engine, and no way could be found to make a platform *semi-solid* of some sort so that agents can pass through from one side but not the others. Some agents that did not permanently get stuck there would evolve to go right, reach the rightmost platform, and then jump off. But an agent that both tried to reach the right platform and then went left never evolved. It seemed that because the platforms were so close together, it hindered the agents from progressing. This was especially problematic because the NN inputs fed the closest platform to the NN, which kept changing between the top and the right one, depending on the current jump height of the agent. So with the tested input type, level 3 seemed near impossible for the agents to complete.

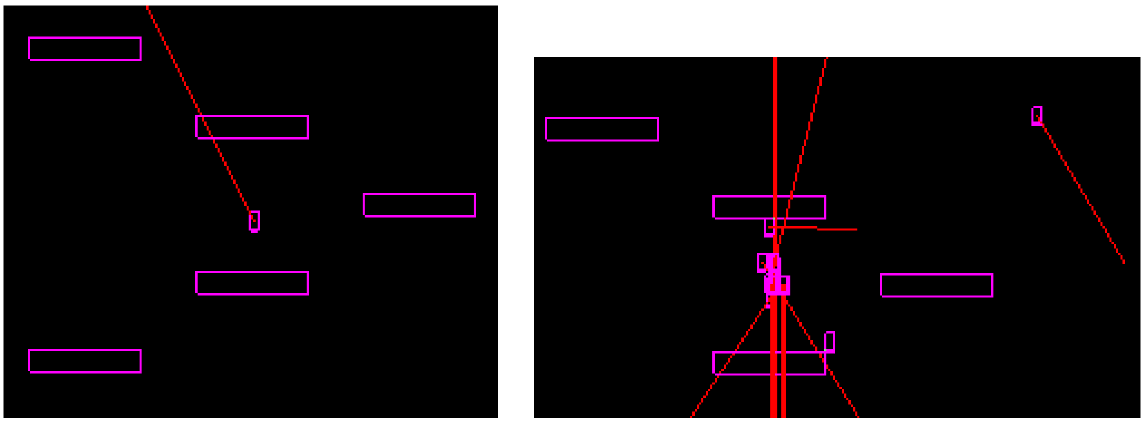


Figure 15: Point where agents get stuck in level 3.
More and more agents try to jump "through" the platform on top

It is possible that other input types would have been able to handle the curve in level 3 better. One possible way to change the inputs would be to feed the platform information to the NN in a predetermined order. So assuming we had an ordered platform list, the first input could come from the current highest platform reached, and the next platform in the list would be fed to the NN second. While this could overcome the issue of agents not being able to progress, it also requires the inputs to be highly processed before the NN even gets them. So a part of the logic from the agent's brain is outsourced into code that makes the game world easier to interpret for the agent. At this point, the inputs are no longer merely sensory and also have the disadvantage that a clear path with a fixed next platform may not be possible in some cases. Levels with more than one way would be a problem and would need even more logic to decide between paths outside of the agent's NN so inputs could be processed.

Another possibility for changing the NN inputs would be to use a grid of sensors similar to the ones used in Ortega et al. (2013). This would remove the need for platform distances or positions to be fed into the NN. Further, it would tell the agents not only the distance but also the angles to the platforms with one input because the distance and position of the grid sensors relative to the game agent are fixed, and a sensor provides meta information about where objects are when it is triggered. This would most likely mean controllers are able to handle the curve better than NNs where two inputs are used for distance and angle, like NN presets 1, 2, and 3. The biggest problem with this approach is a limitation of the physics engine. While there is a way to implement sensors (in fact, jump detection is solved via a sensor at the bottom of the agent's feet), adding more sensors to the body fixtures of the Box2D bodies is easier said than done. It would require a large number of sensors, which would not only slow down the simulation considerably but also obstruct the view of what is happening in a level, as the debug view shows all sensor hit boxes. That being said, if there is a way to add a number of sensors to the static bodies of the agents without impacting the simulation performance too much, it would be a promising option to explore in the future in order to complete level 3.

We can now answer research question number 3: *How do the agents perform in various levels of the platform game when changing parameters from questions 1 and 2?* First of all, the performance differences between each level are (unexpectedly) very big. The difficulty ranges from easy to nearly impossible to complete for the agents. But some presets showed better performance in one level while having worse MBF and SR in another level when compared to other presets. For now, this could be observed when comparing NN presets in levels 1 and 2, and the performance of EA presets in different levels will be compared in Section 5.3.

Because no agents were able to complete level 3, it will not be used for performance comparisons between individual NN and EA presets. Especially when looking at EA presets, level 2 is the best suited for these comparisons, as agents were able to consistently complete it, but at the same time, there are no NN presets that show almost 100% SR, like $NN = 1$ in level 1. In fact, the first NN preset is of particular interest when comparing EA performances. This is because this preset had the best performance in level 2 overall and is expected to be a part of a preset combination that shows the best SR and MBF in level 2. For this reason, we are also researching the question of "*what is the highest SR in level 2 with a specific NN and EA preset combination?*" in Section 5.4.

## 5.3 EA Parameter

After looking at the results of changing NN presets, we can do the same for EA presets. Most of the research on EA parameters has been done after testing different NN presets and their performance in the levels. This meant there was already a baseline expectation for the SR and MBF using individual NN presets in each level. Considering the fact that we are only ever changing a single EA parameter per preset compared to the standard preset 21 (Table 3), the assumption was that there would be observable differences, but not by very big margins. Before testing EA presets, when looking at the big impact that changing NNs had, the initial impression was that changing EA presets would not have as much of an influence on agents reaching the goal. But the more EA parameters were tested, the more it became clear just how much of an impact each single parameter could have. This can be seen quite clearly in Table 5, which shows the average performance measures of the main EA presets in levels 1 and 2 for generation 100.

| **EA** | **SR level 1** | **SR level 2** | $\mathbf{SR_2/SR_1}$ | **MBF level 1** | **MBF level 2** | $\mathbf{MBF_2/MBF_1}$ |
|---|---|---|---|---|---|---|
| 10 | 38.33 | 1.39 | 3.62% | 132.80 | 8.43 | 6.35% |
| 20 | 75.00 | 13.17 | 17.57% | 202.79 | 38.35 | 18.91% |
| 21 | 40.61 | 11.99 | 29.52% | 121.84 | 34.08 | 27.97% |
| 22 | 36.14 | 7.00 | 19.37% | 136.54 | 26.15 | 19.15% |
| 23 | 55.67 | 1.43 | 2.57% | 170.44 | 9.73 | 5.71% |
| 24 | 25.00 | 0.74 | 2.96% | 75.89 | 7.31 | 9.64% |
| 25 | 20.63 | 1.82 | 8.81% | 84.72 | 11.22 | 13.25% |
| 26 | 53.50 | 7.16 | 13.37% | 160.25 | 22.32 | 13.93% |

Table 5: EA preset performance measures averaged over all NN presets for generation 100 in levels 1 and 2, as well as a comparison between SR and MBF in those levels

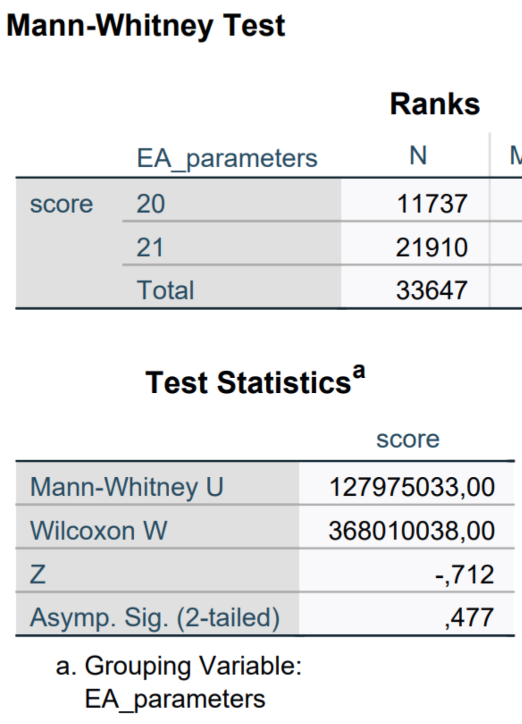
There is a big difference between the highest and lowest SR in both levels. With a highest SR of 75% and a lowest of barely over 20% the difference is comparable to the gap between the best and worst NN parameters. The same can be said for level 2. While the best performance in level 1 is clearly taken by EA preset 20 by a large margin, in level 2, the best two SR measures are quite close. Considering EA preset 21 did not show a very high SR in level 1, the good performance is certainly an outlier, resulting in almost 30% as much SR in level 2 as in level 1. On the other hand, the worst performances in level 1 with only 20% SR ($EA = 25$) and in level 2 with even less than 1% ($EA = 24$) were achieved by two different presets. This is primarily because preset 25 did not show as much decline in SR and MBF as preset 24, which was second to last in level 1. With less than 3% SR in level 2 compared to level 1 from preset 24, only preset 23 showed a bigger decline in performance. But compared to the SR, the MBF in preset number 24 did not decline as much, with a more than 3 times higher relative MBF (9.64%) than the SR (2.96%). One possible reason for this might be that preset 24 changes the parent selection method to roulette wheel selection, which has more exploration and less exploitation of the search space compared to deterministic parent selection. This means roulette wheel selection is able to consistently produce moderately well-performing individuals after 100 generations but struggles to evolve the top agents toward a solution that reaches the goal.

When directly comparing EA presets, the main focus is on their performances in level 2. Looking at the SR data in Table 5, these can be divided into three categories:

- top two presets with 13 and 12 SR
- middle two with around 7 SR
- bottom four with $< 2$ SR

First, we are looking at the two best-performing presets, numbers 20 and 21. The good performance of $EA = 21$ confirms the initial impression that this particular combination of parameters was performing well on level 2. Because of this, preset 21 has been used as a basis for all other EA presets $\geq 20$ to examine the effects of changing only a single parameter at a time. But at the time of deciding on preset 21 as a baseline for comparison, no data analysis had been done apart from watching several runs and looking at which presets performed well. The fact that most other presets based on this one show considerably worse performances implies that most of the changed parameters have a big impact on the performance of the evolution.

This brings us to the first of the six sub-research questions, 1(a), where the impact of changing from arithmetic to discrete crossover is questioned. Unexpectedly, the parameter that seems to have the least impact on the performance in level 2 is the crossover type. When looking at the top two presets, they only differ by $NN = 21$ having arithmetic crossover and $NN = 20$ having discrete crossover. The performance of those two presets is relatively close as preset 20 has a SR of only 10% more than preset 21. This makes preset 20 not only the closest to the baseline preset 21, but also means it actually performs *better*, as opposed to all other presets. This is probably one of the most interesting findings in the whole experiment for two reasons. First, when considering that arithmetic and discrete crossover are quite different approaches to how recombination is implemented, it seems unintuitive that the performance measures are this close. And second, those two presets also vastly outperform all other presets, with SRs more than 65% higher than the third-best EA preset. This raises the question of what exactly is responsible for this discrepancy in scores. There are two different possibilities that could explain this outcome. The first possibility is that all other parameters have a big impact on the SR and are set to "good" values in presets 20 and 21, while these two particular presets do not have a large impact on the SR at all. The other possibility is that both presets are able to search for solutions with comparable success. Considering the two crossover approaches are very different, this would be surprising. Since we are comparing all EA presets, it might be easy to dismiss the relatively small difference between presets 20 and 21 in level two. But when looking at the absolute SR, preset 20 still shows more than 10% more SR than preset 21. To confirm if these two presets show a statistically significant difference in performance, a Mann-Whitney U test on the average MBF of both presets was performed. The data points that are compared are all entries in the *top_gen* table (Section 4.3.3) which contains the agents with the highest score value in their generation for the run they took part in. This means we are comparing the distribution of MBF from the top individuals for every run with $EA = 20$ and $EA = 21$. The reason why the MBF has been used instead of SR, is that the progression of the MBF over generations is smoother. With the SR being 0 in the first few generations and only slowly increasing, it is harder to draw conclusions from the SR progression over time than from MBF. The results of the test can be seen in Figure 16. With a p-value of almost 0.5, it suggests there is no significant difference in the distributions.


**Mann-Whitney Test**

**Ranks**

| | EA_parameters | N | Mean Rank | Sum of Ranks |
|---|---|---|---|---|
| score | 20 | 11737 | 16875,44 | 198067090,00 |
| | 21 | 21910 | 16796,44 | 368010038,00 |
| | Total | 33647 | | |

**Test Statistics**[a]

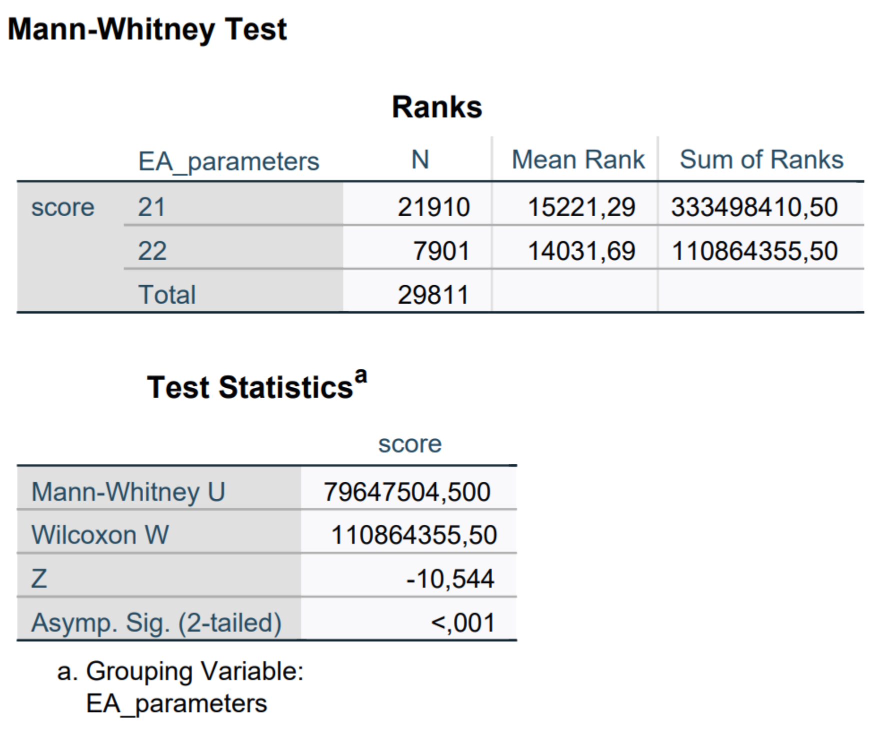
| | score |
|---|---|
| Mann-Whitney U | 127975033,00 |
| Wilcoxon W | 368010038,00 |
| Z | -,712 |
| Asymp. Sig. (2-tailed) | ,477 |

a. Grouping Variable: EA_parameters

Figure 16: Man Whitney U test between EA parameter 20 and 21
data points are from the *top_gen* table and contain the highest scoring individuals in each generation from every run with the chosen parameters

The next EA presets we are looking at are the two with moderately good SR, which would be presets 22 and 26. To answer research question 1(b), the mutation operator is changed from non-uniform to uniform mutation for $NN = 22$. This means the mutated weights are now randomized without consideration of what the previous weights were. For this reason, the initial expectation had already been that discarding the weights of strong individuals would result in a decrease in exploitation and negatively affect performance. The data confirms this expectation. With a 7% SR compared to the 12% of preset 21, the results clearly showed that nonuniform mutation performed better than uniform mutation. The Mann-Whitney test between EA presets 21 and 22 also shows a highly significant difference between the distributions with a p-value $< 0.001$ (Figure 17).

The second preset with around 7% SR was $EA = 26$, which is part of research question 1(f). Here, the child count is decreased from the standard 2 children per crossover to 1. Because of this, twice as many parent selections need to occur until enough crossovers are performed to fill up the next generation. Due to the deterministic selection process for parents, the top individuals, who are picked first with this selection method, only produce half as many children. This means individuals with lower fitness can participate in reproduction due to the increased number of parent selections. Because of this, exploration increases, but at the same time, exploitation decreases. This is likely the main reason why



**Mann-Whitney Test**

**Ranks**

| | EA_parameters | N | Mean Rank | Sum of Ranks |
|---|---|---|---|---|
| score | 21 | 21910 | 15221,29 | 333498410,50 |
| | 22 | 7901 | 14031,69 | 110864355,50 |
| | Total | 29811 | | |

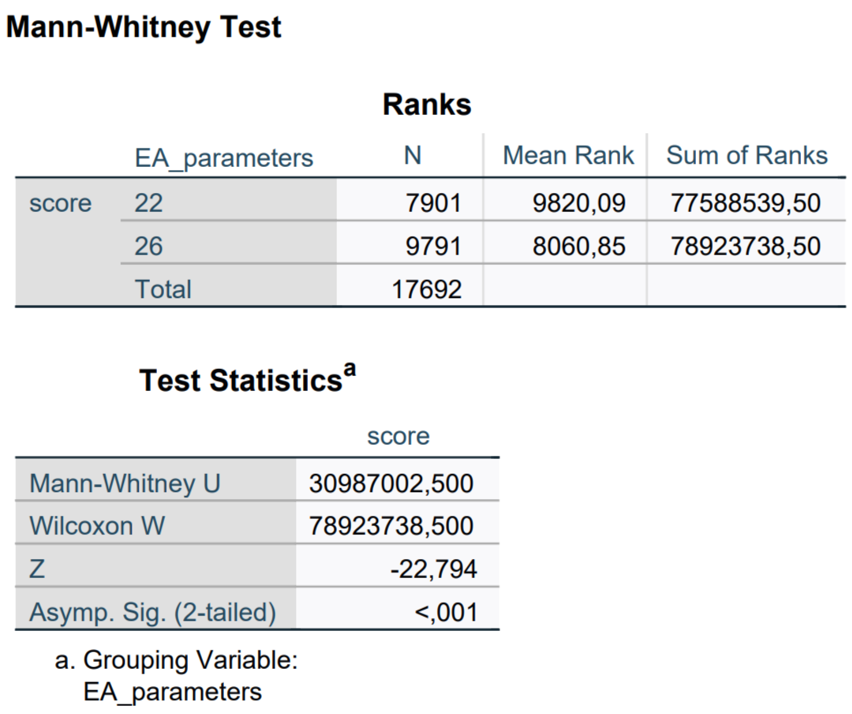
**Test Statistics**[a]

| | score |
|---|---|
| Mann-Whitney U | 79647504,500 |
| Wilcoxon W | 110864355,50 |
| Z | -10,544 |
| Asymp. Sig. (2-tailed) | <,001 |

a. Grouping Variable: EA_parameters

Figure 17: Man Whitney U test between EA parameter 21 and 22
data points are from the *top_gen* table and contain the highest scoring individuals in each generation from every run with the chosen parameters

decreasing the number of children per crossover affects the SR negatively. With roulette-wheel parent selection, lowering the number of children might not have as big of an impact as it can be seen here, as the chance of picking good-scoring individuals is the same even when more parent selections occur. But when comparing the decrease in performance measures after the parameter change in $EA = 26$ with the worst four presets, it still shows moderately good SR. So the evolution's progress is hindered by decreasing the number of children to 1, but it still stays competitive to some degree. The Mann-Whitney U test between EA presets 21 and 26 showed the same results as the comparison between 21 and 22 (the other moderately well performing preset), with a p-value $< 0.001$. But when comparing presets 22 and 26 directly, there was also a significant difference found (Figure 18). It shows that the mean rank of preset 22 is considerably higher than that of preset 26. This finding can be backed up by the data in Table 5, where only the performance measures of agents at generation 100 are shown. Despite the close SR, the MBF shows more than a 10% difference. So while preset 22 is the better-performing preset of the two on average over 100 generations, its overall success rate at generation 100 is slightly lower than that of preset 26. The main reason for this is likely to be the ability of nonuniform mutation to explore a lot of the search space and produce moderately good agents relatively quickly while having problems fine-tuning good individuals and finding solutions within the generational time limit given.



**Mann-Whitney Test**

**Ranks**

| | EA_parameters | N | Mean Rank | Sum of Ranks |
|---|---|---|---|---|
| score | 22 | 7901 | 9820,09 | 77588539,50 |
| | 26 | 9791 | 8060,85 | 78923738,50 |
| | Total | 17692 | | |

**Test Statistics**[a]

| | score |
|---|---|
| Mann-Whitney U | 30987002,500 |
| Wilcoxon W | 78923738,500 |
| Z | -22,794 |
| Asymp. Sig. (2-tailed) | <,001 |

a. Grouping Variable: EA_parameters

Figure 18: Man Whitney U test between EA parameter 22 and 26
data points are from the *top_gen* table and contain the highest scoring individuals in each generation from every run with the chosen parameters

Lastly, we can look at the worst-performing presets and which parameters were varied in each one. In preset 23, the mutation rate (MR) was increased from 10% to 20% to answer research question 1(c). Just by doubling the MR, the SR of the evolutionary algorithm in level 2 plummeted by nearly 90%. The variance in new children was likely too high when increasing the mutation rate too much. Because of this, the preset struggled to evolve agents that were able to complete level 2, where the margin of error is small and the exploitation of good individuals is required. On the other hand, the preset performed relatively well in level 1, where a bigger margin of error means quickly reaching moderately good solutions with good exploration is an advantage. This is likely the reason for the large difference in performance between levels 1 and 2.

In preset 24, the parent selection method was changed from a deterministic selection to a roulette wheel selection (research question 1(d)). In the deterministic parent selection algorithm, there is a very high selection pressure as the top individual is picked for recombination many times more than an individual just a few ranks below it. This makes it very hard for lower-ranking individuals to even be picked as parents at all, so new strategies that would take a few generations to emerge with good scores have problems developing. This was part of the reason why roulette wheel selection was introduced, to give the option for less aggressive parent selection. That being said, the problem at hand, namely that of the traversal of a 2-D environment, was expected to require a balance of both exploration and exploitation, so a high selection pressure is not necessarily a bad thing if it is balanced out by other EA operations that introduce variety and stop the population from converging in a local optimum. But surprisingly, this preset showed the worst performance in level 2 of all EA presets. The performance in level 1 was also relatively bad, so changing this parameter must have a big impact on finding agents that can complete the game. The reason this preset performed so badly is probably similar to the reason for the decrease in performance when the child count was set to 1 in preset 26. Both times, fitter individuals are effectively not as often picked for reproduction, and in turn, individuals with lower fitness can partake. While the expectation before testing had already been that this preset would not perform as well as 21 in level 2, the falloff in SR is more drastic than anticipated.

On the other hand, the preset where a drastic performance falloff was already expected beforehand is preset 25, where elitism has been removed. In fact, the expectation was that research question 1(e) would lead to this preset showing the worst performance, or at least worse performance than presets 23 or 24. This indicates that the variance introduced by the removal of elitism hindered but did not completely stop the algorithm from finding any solutions to level 2. But in the end, this preset was one of the worst-performing ones with the chosen parameters. The most likely reason for this is that removing elitism altogether takes away a large amount of exploitation by completely discarding all individuals of the old generation after each run and only relying on the new children. It could be interesting to experiment with different parameter combinations without elitism in the future, but it is strongly suspected that an elitism rate of at least greater than 0 is beneficial to the performance of the search.

Lastly, the only preset in the list that is not based on $EA = 21$ is preset number 10. This could be called a legacy preset of the neuroevolution-flappy-bird project that was used way back when the first test runs had been done. It was mainly kept in as a preset to see how a particular EA instance that performed well in another type of game would perform in a new problem domain. When looking at the flappy bird project, the search was quite successful with these parameters, finding solutions quickly the majority of the time. For that reason, it was expected the preset would perform relatively well compared to others, but it was one of the worst-performing ones (Table 5). This might be because it combines two of the inferior parameter choices that have been previously explored. With a child count of 1 and a uniform mutation, it has both of the EA presets 22 and 26 that were categorized as moderately well-performing when only changing one of these parameters at a time. But with both detrimental parameter choices, its performance is on the lower end at 1.39 SR. It is also worth noting that preset 10 has a randomness rate of 20%, which makes it the only preset in this list with randomness. This was kept as it was in the original preset, but it was ultimately deemed not useful to introduce completely randomized individuals to other presets when exploration of the search space is already accomplished by mutation and crossover. Further, it seemed like a detriment to the performance of the algorithm most of the time when a portion of the population is completely randomized without regard for already existing individuals. While there is theoretical value in being able to leave local optima when the search gets stuck, this can also be accomplished by the classic EA variation operators. And in the few cases where the operator might be beneficial, it is questionable if the underlying EA is not at fault, if such cases can even occur in the first place, where the search would be unable to leave local optima without pure randomization.

Coming back to the first research question, *How do different parameters for the evolutionary algorithm affect agent performance?*, this question can now be answered. Even changing a single EA parameter can have a large influence on the EA's performance. The biggest impact was observed when varying mutation rate, parent selection, and elitism rate. Moderate differences in performance measures were observed when changing child count and mutation type. And the smallest difference in SR and MBF came from switching between arithmetic and uniform crossover.

Looking at the sub-research questions 1(a) to 1(f), the questions of the effects each parameter has on agent performance can be answered as follows:

(a) changing *crossover_type* had the smallest impact on performance measures

(b) *nonuniform mutation* performed better than *uniform mutation*

(c) increasing *mutation_rate* from 10% to 20% drastically diminished the performance

(d) changing *parent_selection* from deterministic to roulette produced the worst results of any parameter change

(e) removing *elitism* caused a big falloff in performance measures

(f) reducing *child_count* from 2 to 1 lowered performance by a moderate amount

Most of the results had one thing in common, and that was the importance of exploitation. When comparing presets that showed different amounts of exploitation, the presets that caused more exploitation almost always performed better. This shows the importance of fine-tuning good individuals in order to complete levels, especially level 2, where most of the research was focused. A main reason is likely the length of the level as well as the consistent pattern of platform distances, which requires agents to perform precise jumps several times in a row.

## 5.4 Interactions

After looking at the effect of changing NN and EA parameters separately, it also has to be considered if there are interactions between them that affect performance. For that reason, data for the best scoring preset combinations has been gathered and can be seen in Table 6. Specifically, the top performance measures in level 2 ordered by SR can be seen. Every preset combination also has an entry count that shows how many full simulations with 100 generations and 100 individuals each were performed. All combinations that are not

| NN | EA | SR* | MBF* | Entry count |
|---|---|---|---|---|
| 1 | 21 | 35.90 | 94.29 | 39 |
| 1 | 20 | 35.14 | 95.20 | 37 |
| 1 | 26 | 28.57 | 77.04 | 14 |
| 1 | 22 | 25.00 | 68.95 | 16 |
| 4 | 20 | 23.08 | 60.64 | 13 |
| 3 | 21 | 15.39 | 41.80 | 13 |
| 4 | 21 | 15.38 | 41.95 | 13 |
| 2 | 20 | 12.50 | 37.58 | 16 |
| 4 | 22 | 10.00 | 43.57 | 10 |
| 1 | 25 | 9.09 | 39.09 | 11 |
| 1 | 10 | 8.33 | 27.95 | 12 |
| 3 | 20 | 8.33 | 27.70 | 12 |
| 2 | 26 | 7.69 | 24.65 | 13 |
| 1 | 23 | 7.14 | 26.05 | 14 |
| 4 | 26 | 6.67 | 20.51 | 15 |
| 2 | 21 | 5.26 | 19.66 | 19 |
| 1 | 24 | 3.70 | 18.25 | 27 |

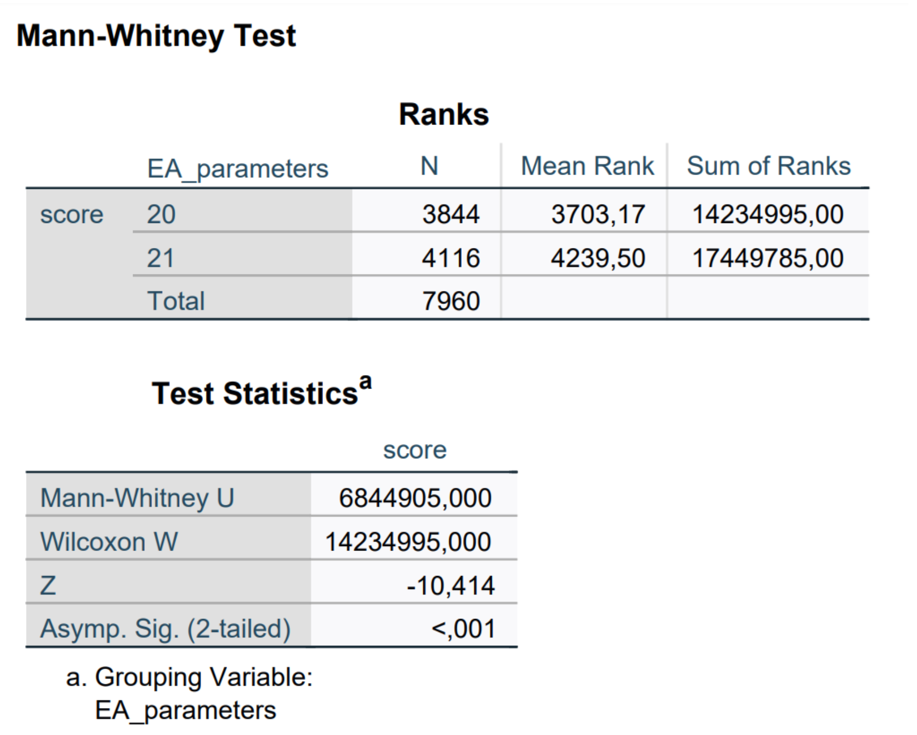
Table 6: best performing combinations of NN and EA presets in level 2, ordered by SR
*values are averages for the presets at generation 100

on this list had a SR of 0 or close to it and are considered unsuccessful in completing the second level consistently.

As Table 6 shows, the top preset combinations were $(NN = 1,\ EA = 21)$ and $(NN = 1,\ EA = 20)$. They show very similar MBF and SR. Because of this, several test runs have been performed to increase confidence in which combination is the best choice for having as high a SR/MBF as possible. In the end, the best SR was reached by $(NN = 1,\ EA = 21)$ even with a slightly lower MBF than $(NN = 1,\ EA = 20)$. The Mann-Whitney U test in Figure 19 also shows a significant difference in the mean rank of all scores with these parameters. That means while the performance measures are relatively close, when looking at the scores from all individuals, it can be seen that EA preset 21 shows significantly better scores than preset 20 with $NN = 1$ in level 2.

Because EA presets 20 and 21 already showed similar average performances and close top scores in the previous section, we can look at how they compare when combined with other NN presets. $(NN = 4,\ EA = 20)$ performs quite a bit better than $(NN = 4,\ EA = 21)$ with a SR rating that is 50% higher. This indicates that uniform crossover was better suited to evolve NNs when they were receiving positions as inputs, as opposed to arithmetic crossover. On the other hand, large NN topologies seem to be favored by arithmetic crossover, as $(NN = 3,\ EA = 21)$ performed way better than $(NN = 3,\ EA = 20)$ with almost twice the SR.

When looking at EA preset 20, the performance of NN presets with those EA parameters decreases the bigger the topology gets. Not only are the three distance presets ordered by



**Mann-Whitney Test**

**Ranks**

| | EA_parameters | N | Mean Rank | Sum of Ranks |
|---|---|---|---|---|
| score | 20 | 3844 | 3703,17 | 14234995,00 |
| | 21 | 4116 | 4239,50 | 17449785,00 |
| | Total | 7960 | | |

**Test Statistics**[a]

| | score |
|---|---|
| Mann-Whitney U | 6844905,000 |
| Wilcoxon W | 14234995,000 |
| Z | -10,414 |
| Asymp. Sig. (2-tailed) | <,001 |

a. Grouping Variable: EA_parameters

Figure 19: Man Whitney U test between the best two preset combinations.
NN parameter 1 with EA parameters 20 and 21 on level 2
data points are from the *top_gen* table and contain the highest scoring individuals in each generation from every run with the chosen parameters

topology, but preset 4 also scores higher than preset 2, showing that changing to positional inputs does not decrease performance measures of EA preset 20 as much as increasing topology size does.

Compared to that, preset 21 does not show a strict order when looking at NN sizes. In fact, NN preset 3 with the biggest topology scored over 15 SR, which is almost three times as much as the second NN preset with just over 5 SR. Because of this, $(NN = 3,\ EA = 21)$ and $(NN = 4,\ EA = 21)$ have very similar performance measures, and the second-best choice for a NN preset with $EA = 21$ can be considered a tie. Compared to those presets, $(NN = 2,\ EA = 21)$ did not score very well in contrast to EA preset 20, where $NN = 2$ performed better than $NN = 3$.

When looking at the NN parameters, the first notable thing is that all of the top four spots are taken by $NN = 1$, making it the clear best choice for NN presets. It is, in fact, the best-scoring preset for all EA parameter combinations. But when looking at the second-best NN preset, things are not quite as clear. Spots five to eight are taken by NN presets 2, 3, and 4, and all of them either have EA preset 20 or 21. This shows again that those EA presets are the best performing presets not only when combined with the best-performing NN presets but also in all other combinations.

To summarize, it can be said that there are certainly presets that show good performance no matter which other presets they are combined with. On the other end of the spectrum, there are also some presets that are not suited for the problem at hand and did not show any combinations with good performance, like NN presets 5 and 6. But there are also interactions between EA and NN presets that show some EA parameters are better suited to evolve NNs with bigger topologies, and some can handle positional input types better. Because of this, future work should consider the impact NN parameters have when trying to tune the EA. This means that not every well-performing set of EA parameters is able to handle different NNs equally well, and there is a need to test if certain combinations of parameter choices might be able to perform unexpectedly well.

## 5.5 Noise

The aforementioned results stem from EA parameters that all include elitism except for EA preset 25, where it was specifically disabled. But when looking at SR and MBF over time in diagrams like Figure 14, we can see that performance measures are not monotonically increasing. In fact, there are some short-term decreases in SR and MBF from one generation to another. Coupled with the fact that the performance measures only look at the mean best fitness and success rate, they only account for the top individual in each generation. This means that if we have at least one elite, the top score from the previous generation should never be higher, as the next generation must include the previous best individual. Considering this, there must exist an underlying reason that can explain this discrepancy. The search for exactly this reason is the topic of this section.

First of all, the EA implementation was checked to see if the elitism had been implemented correctly and if the top individuals were correctly added to the next generation. This was quickly confirmed by looking at the agents table in the database and seeing that the NN weights of the top agents were part of the next generation. But agents with the exact same NNs did not always show the same score. While the scores were identical most of the time, there were several occurrences where they differed by a lot.

This phenomenon implies that there must be some sort of noise affecting the agents that influences their performance. But at the same time, this noise is small enough that all test runs still show an increase in score over time despite the small, temporary dips in performance. This would also mean that the algorithm is robust enough to withstand the noise introduced and can quickly recover from those temporary setbacks. Of course, this assumes the noise is introduced externally and not part of the algorithm itself. But this has been deemed unlikely, as there are many factors in the physics environment that have the potential to be responsible for this. Because of this, more research toward the origin of the noise has been conducted, starting with the main simulation logic.

After confirming the elitism worked correctly, the most likely theory as to where the noise came from was how physics steps are handled in the Box2D simulation. The main simulation loop is handled by the act(*delta*) method of the GameStage class. Here, all interactions between the physical game world and all actors in it are handled. These actors can be human players or, in the case of our simulation, bots. The *delta* is a float value with the time since the last frame. In our case, the frame rate is set to 60, so the image that is rendered to display on screen is updated 60 times per second. Under normal circumstances, the time between frames is the same, but there might be frames that take more or less time than the usual $\frac{1}{60}$ seconds. Because of this, we have to take into account how much real time has elapsed since the last frame when handling the simulation logic, since the simulated time should be consistent with real time. For this reason, the world is simulated by calling the step(*delta*) method of the Box2D World class in the act(*delta*) method and simulating a physics step in the world. This means we have to simulate the same *delta* time in the Box2D world, but we have to take into account that simulating physics steps that are too big can have negative effects on the simulation. To solve this issue, a physics step method (Section 5.5) as recommended by libGDX was implemented[13], where each simulated step has a fixed time *TIME_STEP* that it simulates. If there ever is a case where *delta* between frames is larger than *TIME_STEP*, more steps are made so the simulation can catch up to the real time. All this is made possible by a global *accumulator* variable that keeps track of the difference between real time and simulated time. There is also a maximum *frametime* that makes sure not too much time is simulated in one physics step to avoid the performance of the simulation spiraling toward more and more time between frames because more steps have to be calculated.

But with the way the physics steps are implemented, the time between bot actions is af-

[13]`https://libgdx.com/wiki/extensions/physics/box2d#stepping-the-simulation`

**Algorithm 8** The doPhysicsStep(*delta*) method

```
Require: $delta$
1: $frameTime \leftarrow \min(delta, 0.25)$
2: $accumulator \leftarrow accumulator + frameTime$
3: while $accumulator \geq TIME\_STEP$ do
4:     world.step($TIME\_STEP$, $VELOCITY\_ITERATIONS$, $POSITION\_ITERATIONS$)
5:     $accumulator \leftarrow accumulator - TIME\_STEP$
6: end while
```

fected by how long frames take. If the accumulator is big enough, there is the possibility of doing two or more physics steps before the bot can take its next action. While this physics step method may be very good for enabling smooth gameplay for human players, it also makes the simulation inherently non-deterministic, and two runs with the exact same bots could differ by the time when the bots can perform their actions. To confirm these suspicions, the delta values were observed during simulations that might have caused variances in times between bot actions and thus the noise. This showed that most of the time delta was very close to $\frac{1}{60}$, but when a new generation is evolved, the frame takes between 0.5s and 0.6s. Even with the maximum *frametime* in place, this meant that the first frame in each generation had multiple Box2D world steps.

Because of the observed differences in *delta* values, the doPhysicsStep() method was changed to always simulate a single fixed time step. The now deterministic simulation would show if the noise originated from the physics step method. The method was now implemented in a way that was not recommended by libGDX because the simulation would not be as smooth and sync up with the real time, but this would only have a real impact on the experience of real players and not our bots, who can give inputs on every frame.

But even after changing the simulation to constant time steps, agents with the same NN weights still showed different behaviors in different runs. This meant the cause of the issue could not have been the time steps themselves. Even a variance in frame times does not affect the simulation anymore because, no matter how long a frame takes, the physics simulation still only progresses by a single time step.

After these results, the only hypothesis about the origin of the noise that was left was the order of the bots in the list of actors. The bots were stored in a list in the GameStage class so they could interact with the world every time the act() method of the class was called. Then the current inputs were given to each bot in order of their positions in that list, and they used those inputs to decide what actions to take. That means some bots are able to output their actions earlier than the ones behind them in the list. Depending on the time between those actions, it could have an impact on their performances and create noise due to differences in the timing of giving the agents their inputs.

In order to investigate the impact the bots' order had on them, the fact that the NN weights were stored in the DB for each bot was utilized. Because of this, a bot that was able to complete level 1 could be quickly found, and the array of float values representing the NN of this bot was extracted. Then a test was created, where instead of evolving new populations, a fixed number of bots were simply created with those exact NN weights. If there was a connection between bot index in the list and outputs, it could be found by setting all bot NNs to the same values and comparing their results. Several thousands of generations were tested with 100 bots that had these NN weights, and not a single one ever showed any difference to the movements of all other individuals. Initially, the assumption was that this meant there was no connection between the order of the bots and their performance. That would also make sense, because when further looking into the physics steps, it became clear that the results of the bot's actions would only take effect the next time the world step() method was called. And despite there being real-time differences in the bots' action outputs, they would all be processed at the same time for the simulation.

Even though the order of the bots did not directly cause the noise, when investigating the impact of the bot order, a few inconsistencies were found that could not be explained immediately. As stated previously, all bots performed equally well when every single bot had the same NN, but when giving only a few bots the same NN, they did sometimes show different behaviors. Not only that, all bots with the same NN showed the same behavior in the same run, but in runs where the same bots were in different positions in the list, their performance differed greatly from what has been observed before. After testing many combinations of bot positions in the list, it was found that as long as the first bot in the list had the same NN as the others, all of them showed the expected behavior and never did when the first bot's weights were randomized. These results meant that there was some way the order of the bots in the list influenced their behavior, but only when they were in the first position.

After a long time of debugging, the issue was located. The bot's inputs include information about the closest platforms. For that reason, a getClosestXPlatforms(*x*, *bot*) method was implemented that returns the *x* closest platforms to the given *bot*. In this method, a priority queue is created that orders all platforms by the distance to the *bot* and returns the *x* top entries. But the creation of new priority queues had a bug where only the first bot in the bot list would be picked when instantiating a new queue. This meant that this method only ever returned the current closest platforms to the first bot for all other bots as well. But distance and angle calculations were still done correctly as they were handled in another method, the only issue was that the calculations were done on platforms that were not the closest to the bots with list index $> 1$ but the current closest platforms to the first bot.

This finding would certainly explain the inconsistencies observed when looking at bot scores. Further, it fundamentally changes the input all agents with index $> 1$ receive. This has implications for the whole neuroevolutionary algorithm and the results that were obtained until the error was found. Further, this could explain why exploration was so strongly discouraged when testing the presets. This was because the inputs from the closest platforms stemmed from the bot on position 1, which is always the best-performing bot when elitism is enabled since it is the first to be added to the new generation. That means the more the actions of the other bots differed from the first bot, the bigger the discrepancy between the received closest platforms and the actual closest platforms was. So exploitation was a far superior way to achieve progress for two big reasons:

1. Bots with NNs that resembled the best bots NN took similar actions, which means the inputs they received were closer to the actual inputs they would have received normally.
2. If one bot evolves that performs better than the current best one, it did so while receiving information stemming from platforms closest to the first bot. So for the next generation, it is picked as the new top individual and becomes the first bot, which means it now correctly receives its own closest platforms. How much these inputs differ from the original inputs it evolved with influences how well it can perform.

In fact, the second reason explains why so many "zig-zag" lines can be observed in the score development across generations, where the score would increase in one generation and immediately go down in the next one (Figure 11). This is likely caused by new individuals evolving with better performance than the current top individual and then being unable to perform equally well with the new inputs received.

All in all, it can be said that this causes the whole evolution to operate under the constraint of discouraging the search for new individuals in regions that are not close to the current best individual. While it is not impossible for well-performing new individuals to evolve behaviors that greatly differ from the current best, it makes the success of individuals more unlikely the farther their NN weights are apart.

Most surprising is the fact that the algorithm was able to evolve solutions at all within 100 generations, since the limitations of the search space were a big hindrance in arriving at a solution. If anything, this shows the robustness of evolutionary algorithms to search for solutions even with external factors hindering the search. Further, it shows the ability of neural networks to handle changes in the environment (or the perceived environment through bad inputs).

## 5.6 After the fix

Since the problem of noise was only identified relatively late in the research process, there was only limited time to run new simulations with the correct closest platform information being given to the agents. Also, the previous results have to be considered while keeping the influence of noise in mind, but they still show the impact of varying presets on the search process of the EA. This means we can compare the data acquired before and after the fix. For this reason, this chapter will focus on the following research questions regarding the effects of the fix:

F1 *Is the noise gone after implementing the fix?*

F2 *How does the fix affect individual presets and what parameters show the biggest changes in performance measures?*

F3 *Are there conclusions that can be drawn from the comparisons before and after the fix?*

The difference between the following new results and the old ones is not only that of the bug fix. While searching for possible origins of the noise, the pseudo-random number generation in all parts of the EA has been updated. Because the java.util.random package shows inherent flaws[14] when it comes to pseudo-random number generation, the MersenneTwister[15] java package was used to overcome those limitations and guarantee a good uniform distribution. Because the MersenneTwister does not have a way to generate random values from a Gaussian distribution, a RandomGenerator class has been implemented that provides the method nextGaussian(). This method utilizes an implementation in java.util.Random nextGaussian() but generates the two random double values needed for the algorithm by MersenneTwister.nextDouble().

Because of time constraints, test runs on level 1 have been skipped, and results were gathered starting from level 2. There have also been tests on level 3 in the hopes that the fix would enable agents to clear it, but that did not happen in any of the new test runs. This indicates that other inputs need to be explored to clear the level, as currently none of the agents can get past the curve in level 3. So the results after the fix are focused on level 2, which was also the level where the most research had been conducted previously.

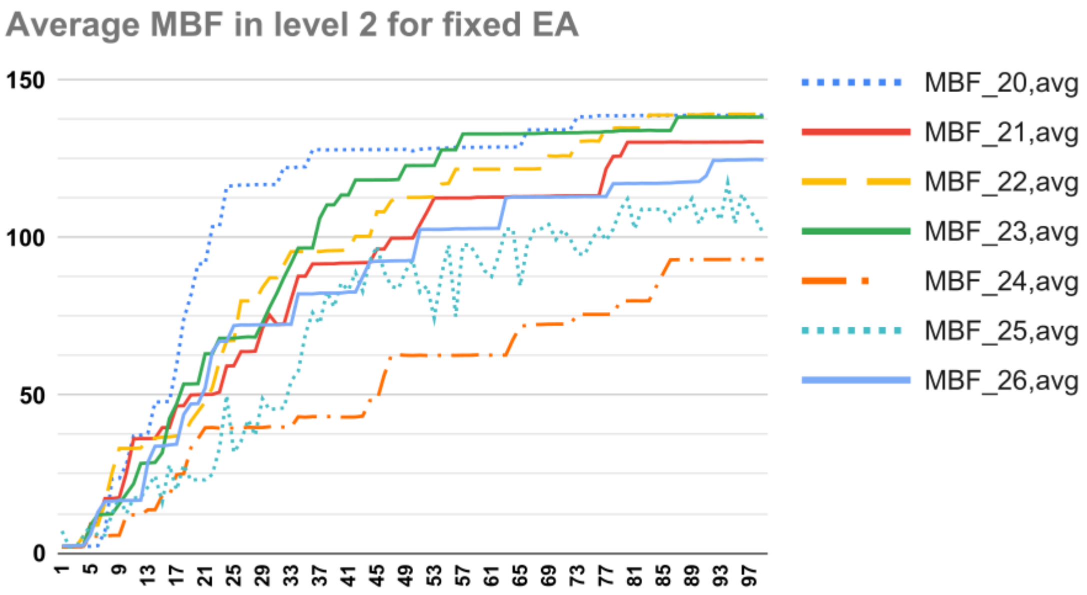
The first thing that needs to be answered for this section is research question F1, regarding whether the noise is gone after the fix. The short answer to this question is "yes, the unexpected score fluctuations were not observed after the fix". This can be seen in Figure 20 where the performance measures of all EA presets in level 2 can be seen, averaged over the six NN presets. The "zig-zag" in the graphs is now gone in all presets but number 25. This is an expected result, however, because EA preset 25 removes elitism and is the only preset where a score decline is expected.

So the underlying issue that caused scores of presets with elitism to fall off has been identified and fixed. This now enables us to investigate the newly acquired data after the fix.

### 5.6.1 EA parameters

Starting with the data shown in Figure 20 we can see that there are three presets that all share the top spot after 100 generations with around 140 MBF and 50 SR. EA presets 20,

[14]`https://commons.apache.org/proper/commons-rng/userguide/why_not_java_random.html`
[15]`https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/engine/MersenneTwister.html`



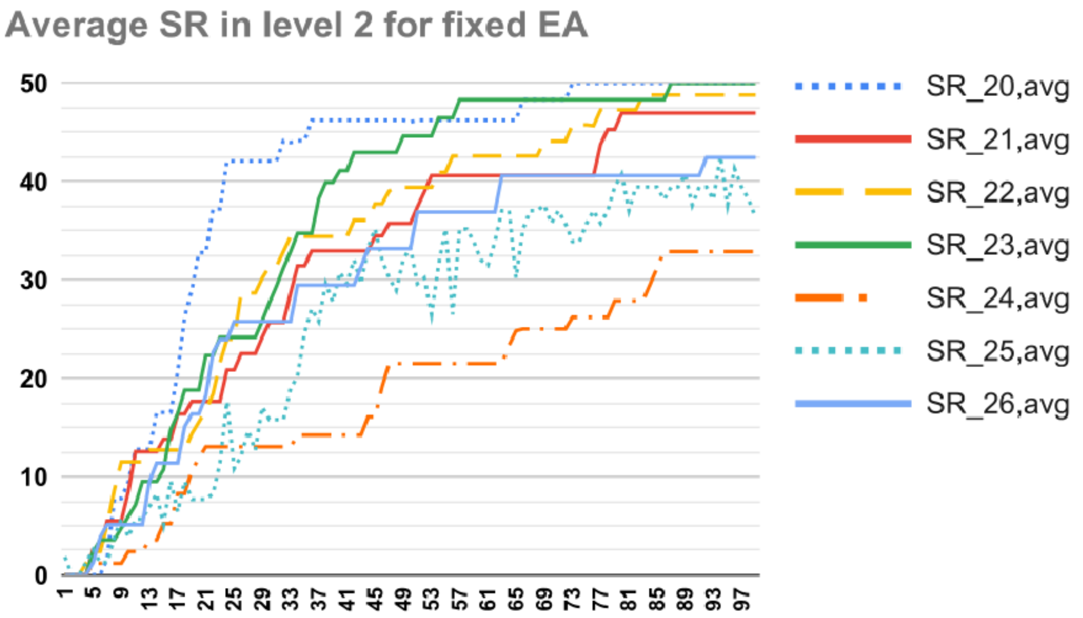


Figure 20: MBF (top) and SR (bottom) of all EA parameter presets in level 2 averaged over all NN presets

22, and 23 not only show the best performance measures at generation 100, they also never fall below the preset at 4th position after generation 100 in both MBF and SR. This would suggest that the three parameters that were changed in those presets, namely crossover type (20), mutation type (22) and mutation rate (23) were beneficial to the search. But preset 21 is in a close fourth place in terms of performance measures, so the improvement over the baseline preset was not by a lot. The other presets showed worse performance when compared to baseline preset 21. Not too far behind is preset 26, where the child count was reduced to 1. Then preset 25 with no elitism is second to last. Here we can also see how the MBF and SR keep going up and down due to no elitism, which even causes the performance measures to decrease right before generation 100. Lastly, preset 24, which switched parent selection from deterministic to a roulette wheel selection, showed the worst performance by far. After generation 30, it showed far worse performance than all other presets, which all had twice the SR from generation 30 to 40. Even after generation 40, it always stayed far below all other presets.

We can now use the new data to answer research question F2, regarding a comparison of performance measures before and after the fix. For this purpose, Table 7 shows the performance measures of individual EA presets pre- and post-fix. It displays the MBF and SR for generation 100 in level 2 for a particular EA preset averaged over all NN presets. Further, it shows the factors by how much the MBF and SR have increased from pre- to post-fix for each preset.

| EA | MBF pre fix | SR pre fix | MBF post fix | SR post fix | $\frac{\mathbf{MBF_{post}}}{\mathbf{MBF_{pre}}}$ | $\frac{\mathbf{SR_{post}}}{\mathbf{SR_{pre}}}$ |
|---|---|---|---|---|---|---|
| 20 | 38.35 | 13.17 | 138.78 | 50.00 | 3.62 | 3.80 |
| 21 | 34.08 | 11.99 | 130.30 | 46.97 | 3.82 | 3.92 |
| 22 | 26.15 | 7.00 | 139.13 | 48.81 | 5.32 | 6.97 |
| 23 | 9.73 | 1.43 | 138.23 | 50.00 | 14.21 | 34.97 |
| 24 | 7.31 | 0.74 | 93.06 | 32.88 | 12.73 | 44.43 |
| 25 | 11.22 | 1.82 | 101.10 | 36.32 | 9.01 | 19.96 |
| 26 | 22.32 | 7.16 | 124.4 | 42.45 | 5.57 | 5.93 |

Table 7: MBF and SR of all EA presets before and after the fix for generation 100 in level 2 averaged over all NN presets
and the factor by how much the performance measures increased from pre to post fix by calculating $\mathbf{MBF_{post}/MBF_{pre}}$ and $\mathbf{SR_{post}/SR_{pre}}$

The first thing that can be said is that every single EA preset showed better performance after the fix. While not all presets showed the same amount of relative increase in performance measures, all of them improved by a large amount. Even for presets 20 and 21, which showed the smallest increase, the MBF and SR are almost four times as high as before the fix. This means that all presets were able to evolve solutions more effectively, which hints toward a big improvement in the way inputs are handled now. This is certainly not unexpected, as the noise was hindering evolutionary progress.

When looking at the individual presets, we can start by looking at research question 1(a), which asks about the effects of switching from arithmetic to discrete crossover and how the new results compare against the old ones. The old data already showed a slight increase in performance measures from preset 21 to 20 before the fix, so the results for research question 1(a) are similar pre- and post-fix. Both of those presets showed comparable performance increases with close to 4 times as much MBF and SR, so the crossover type still seems to have a relatively small impact. But this is only true when looking at generation 100. In Figure 20 it can be seen that the MBF and SR of preset 20 climb at a very fast rate, far outpacing the other presets, at least until generation 20. After that point, the performance measures increase more slowly, allowing the other presets to catch up after 100 generations. These results would hint toward discrete crossover, causing the search space to be quickly explored for good solutions but then having problems with fine tuning agents in later generations.

The answer to research question 1(b) shows there are presets with even higher performance increases than presets 20 and 21 showed after the fix. Here preset 22 is observed, where mutation is set to uniform. Previously, this preset showed worse performance compared to 21. But after the fix, MBF and SR are among the best-performing presets, even higher than preset 21. This is also reflected by the increases in MBF by 5.32 times and SR by 6.97 times. Because of this, the scores of preset 22 and 21 are relatively close after the fix, making the impact of changing the mutation type relatively small. One more takeaway from this data is that the effects noise had on this preset were bigger than with the baseline preset. This is probably due to uniform crossover showing higher exploration, which was heavily discouraged by the noise created.

Even bigger was the impact of the fix on the answer to research question 1(c), where the mutation rate was increased from 10% to 20%. While the old data indicated that increasing the mutation rate worsened performance by a lot, after the fix, preset 23 was one of the best-performing presets. With a 14-times higher MBF and almost 35-times as much SR, it showed one of the biggest improvements of all presets. This increase in performance was bigger than expected, especially since a 20% mutation rate is quite high in the context of evolutionary algorithms. Usually, the recommendation is to keep the mutation rate on the lower end so as not to introduce too much variation and compromise the exploitation. But in this case, a high mutation rate seems to help the search find good individuals relatively quickly, which can be seen in Figure 20 where the MBF and SR of preset 23 keep climbing at a high rate that only starts diminishing around generation 50.

Preset 24 was investigated for research question 1(d), where parent selection was changed to roulette-wheel selection. As shown in Figure 20 this preset had the worst performance after the fix. But it also had the worst performance before the fix, so the factors by which the MBF and SR increased after the fix are the highest of all presets. After all, it could neither perform well before nor after the fix, indicating that roulette wheel parent selection is an inferior method to deterministic selection for the given problem. At the same time, this shows that parent selection has a big impact on the performance of the algorithm.

There are different approaches to parent selection, like Stochastic Universal Sampling (Eiben and Smith (2015)) that might perform better than the roulette wheel algorithm and could be explored in the future.

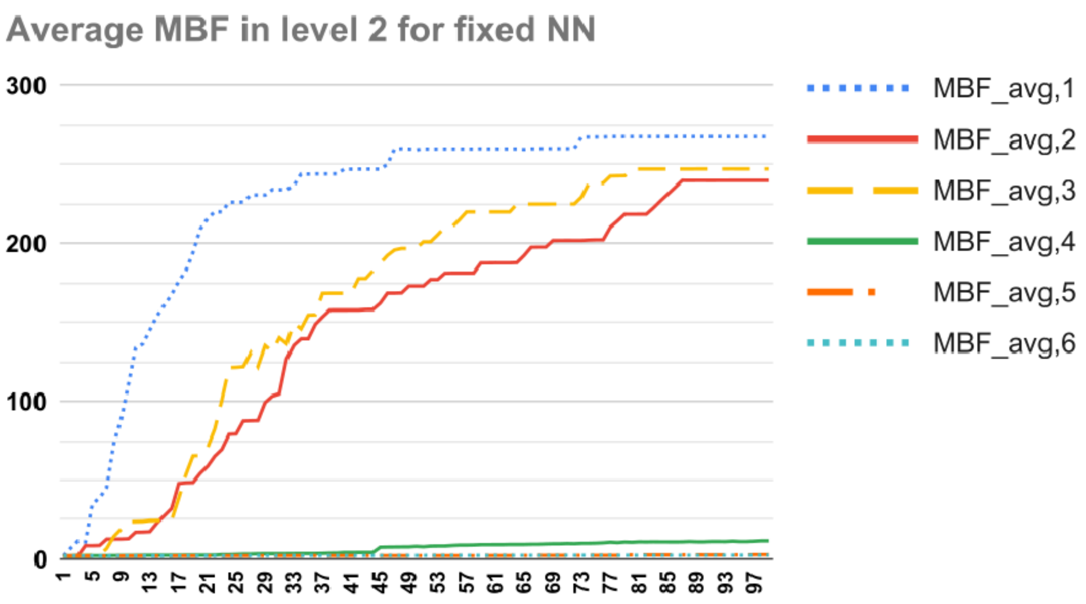
The second-worst preset after the fix was number 25. As part of research question 1(e), the effects of no elitism showed a decline in MBF and SR compared to preset 21. A reason for this is probably the falloffs in performance measures that can be seen in Figure 20 that are caused by discarding the best individuals of the old generation. Both of the worst-performing presets (numbers 24 and 25) reduce exploitation by either removing strong individuals or selecting them less often for reproduction. This means that even after the fix, exploitation still remains an important factor, even if there is more room for exploration now, as the good performance of preset 23 shows.

The last sub-research question 1(f) observed the impact of lowering the child count to one in preset 26. Here, the performance measures are slightly worse than those of baseline preset 21, with only around 124 MBF and 42 SR. The increase in performance with 5.57 times the MBF and 5.93 times the SR is close to the increase of preset 22, which it was already very close to before the fix in terms of performance. But in contrast to preset 22, it does not quite reach the same performance after the fix, as preset 22 performed better than the baseline preset post-fix. A reason for this might again be the decrease in exploitation due to fewer children of the top individuals being produced. But it seems the act of lowering the child count to one is not as impactful as removing elitism or roulette wheel parent selection, so the performance stays higher than with those presets.

As a summary of the research on EA parameters after the fix and an answer to research question F3, the following can be said: All presets showed better performance, with all EA performances now being closer. The worst SR is more than 50% of the best SR after the fix as opposed to less than 10% before the fix. Exploitation still seems to be important, as most presets with less exploitation performed worse. But the outlier is preset 23, where one of the best SRs was reached with an increased mutation rate. In the future, it could be of interest to further examine the effects of changing mutation rate and research why the preset shows results that differ from the common trend. Further, it would be of interest to explore more parent selection methods, as changing the parent selection had the biggest impact on the performance measures. And lastly, researching a way to benefit from the fast initial increase in MBF and SR that discrete crossover showed without the falloff thereafter would be of interest.

### 5.6.2 NN parameters

After looking at the results of changing EA presets, we are looking at the impact different NN presets had after the fix. A direct comparison of all NN presets in level 2 can be seen in Figure 21 where the performance measures are averaged over all EA presets.



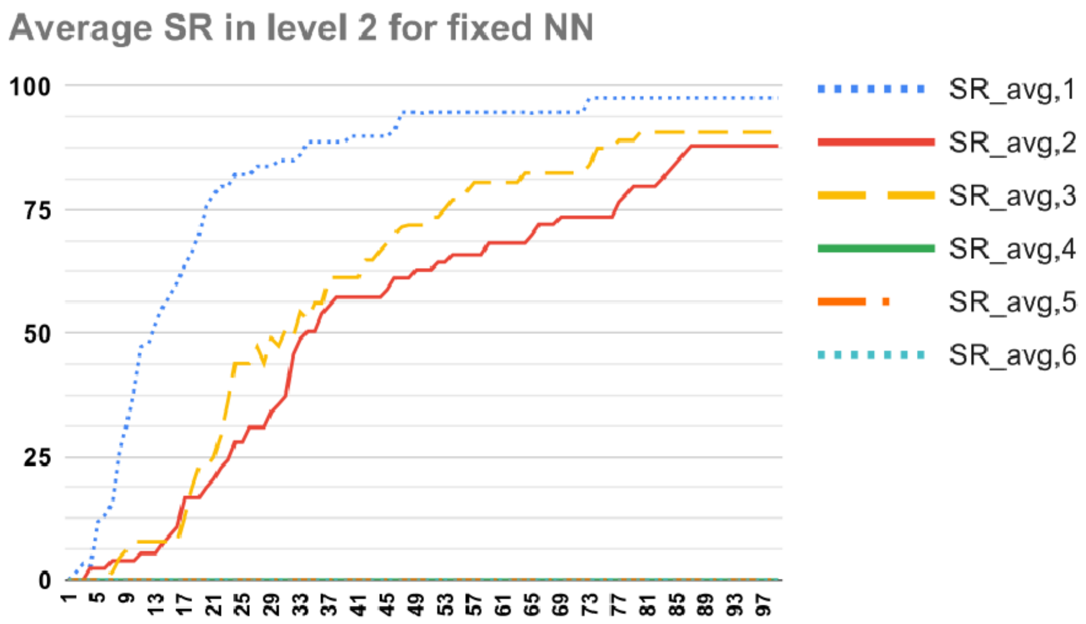


Figure 21: MBF (top) and SR (bottom) of all NN parameter presets in level 2 averaged over all EA presets
*x*-axis: generation, *y*-axis: MBF (top), SR (bottom)

The first thing that can be observed is the big difference in both MBF and SR between the top and bottom presets. While the first three NN presets show very good performance, the second half shows a very bad one. The performance is so bad, in fact, that presets 4, 5, and 6 showed a SR of 0 for all 100 generations in all simulation runs. On the other hand, preset 1 almost reached 100 SR after 100 generations, which would mean it is close to finding a solution within 100 generations every time. But the performance of presets 2 and 3 is also very good after 100 generations. Only in terms of the initial increase in MBF and SR, those two presets are far inferior to preset 1, which has about twice the MBF and SR for a few generations around generation 20.

When coming back to research question F2, we can compare this new data to the performance measures observed before the fix. This comparison can be seen in Table 8 with MBF and SR before and after the fix as well as a factor by how much they differ.

| NN | MBF pre fix | SR pre fix | MBF post fix | SR post fix | $\frac{\text{MBF}_{\text{post}}}{\text{MBF}_{\text{pre}}}$ | $\frac{\text{SR}_{\text{post}}}{\text{SR}_{\text{pre}}}$ |
|---|---|---|---|---|---|---|
| 1 | 50.52 | 16.99 | 257.68 | 93.41 | 5.10 | 5.50 |
| 2 | 13.87 | 2.83 | 236.98 | 86.77 | 17.09 | 30.66 |
| 3 | 13.32 | 2.96 | 235.15 | 86.11 | 17.65 | 29.09 |
| 4 | 23.08 | 6.89 | 13.11 | 0 | 0.57 | 0 |
| 5 | 3.67 | 0 | 2.65 | 0 | 0.72 | - |
| 6 | 2.25 | 0 | 2.34 | 0 | 1.04 | - |

Table 8: MBF and SR of all NN presets before and after the fix for generation 100 in level 2 averaged over all EA presets
and the factor by how much the performance measures increased from pre to post fix by calculating $\mathbf{MBF_{post}/MBF_{pre}}$ and $\mathbf{SR_{post}/SR_{pre}}$

Starting with the input type, which was compared as the second part of research question 2, the differences were already clear from Figure 21. Presets 1, 2, and 3 with distance input types vastly outperform the three positional presets after the fix. While this was already the case before the fix to some extent, now the difference is even more clear. All distance presets showed increases in performance measures, with preset 2 even reaching up to 17.67 times the SR before the fix. The positional presets, on the other hand, even showed a decrease in performance measures, with as low as 0.57 times the MBF before the fix for preset 4. This result is quite interesting, as it would imply the positional presets performed better with the noise in place. It was certainly expected that the positional presets would not show as much of a performance increase compared to the distance presets. The reason for this is that when the bots receive the wrong distance inputs based on their first-person point of view, it becomes more difficult to navigate through the 2-D space since the distances change very much depending on which platforms are used for comparison. On the other hand, receiving the wrong platforms for the positional coordinate inputs, which are third-person information, seemed to be less problematic since the platform coordinates are the same for all bots. But while it was expected that positional inputs would not be impacted that much after the fix, it was also expected that the performance of all NN presets, including positional input types, would increase after the fix. The reason why positional input types show a decrease in performance is not quite clear and would certainly be of great interest to further explore in the future.

The other part of research question 2 can be answered when looking at the NN topology. Before the fix, the data showed preset 4 performing second best, with presets 2 and 3 coming in third and fourth place, respectively. This indicated that smaller topologies have an advantage over big ones. This advantage was so big that it even outweighed the disadvantage of having a positional input type. But when looking at the data after the fix, preset 4 showed the biggest decline in MBF of all presets and was not able to even get an SR $> 0$. This means the new data suggests that the input type has more impact than the topology on the performance of the presets.

That being said, there is still a difference between the smallest topology and the bigger two topologies even after the fix. Both preset 1 and preset 4 outperform the bigger NN variants with the same input type, even if only the MBF can be compared for the latter. Despite that, presets 2 and 3 had a big increase in performance measures, especially in SR, with around 30 times more SR after the fix. Those two presets showed close performances before and after the fix, similar to presets 5 and 6. Considering the three topologies compared were 7-5-3 as the smallest, 7-5-5-3 as medium, and 7-7-7-3 as the biggest topology (Table 2), this shows that the performance differs more strongly when adding another hidden layer to the topology as opposed to adding more neurons in one layer.

# 6 Conclusions

This thesis researched Neuroevolution of Gameplay Agents in 2-Dimensional Platform Games, especially the impact that varying different parameters of the NE algorithm has on the performance of agents in the platform game. The central questions for this research were:

1. How do different parameters for the evolutionary algorithm affect agent performance?
2. What are the effects of changing the NN topology and inputs?
3. How do the agents perform at various levels of the platform game when changing parameters from questions 1 and 2?

In order to investigate the research questions, a platform game with multiple levels was created with a focus on low complexity, so game agents could learn to move through the environment without distracting factors like enemies or interactable objects. Then neuroevolution was realized by implementing a type of neural network, the Multilayer Perceptron (MLP), and an Evolutionary Algorithm (EA) with adjustable parameters. This made it easy to test and directly compare parameter presets, which means the main focus in the early design phase on easily adjustable and testable parameters paid off. Another focus of the implementation was on performance and speedup in the simulation, which enabled more test runs within the time constraints of the thesis. The simulation data of these runs was then stored in a database, and data analysis on the run's performance measures was performed in order to compare the effects of changing individual parameters.

The data showed significant differences in performance when changing any of the parameters from the three research questions. All in all, the data showed that EA parameters were favored the more exploitation they introduced to the algorithm, while NN parameters tended to favor small topologies and distance sensor inputs. Different levels also had a big impact on the ability to reach the goal, ranging from levels with an almost 100% completion rate after 100 generations to levels that were not possible to complete for the agents.

However, during data analysis, a type of noise was discovered that affected the performance of the game agents. Locating the noise was made possible by storing as much simulation data as possible in the database for easy data analysis. This allowed for spotting irregular score progressions and also enabled tests where the agents' NNs were reconstructed by using the NN weights and topology stored for each agent. Because of this, the noise could be traced back to agents receiving the platforms in order of distance to the best-performing bot instead of to themselves. This meant newly evolved individuals had an advantage when their movements were close to those of that bot, and the search for new individuals with vastly different behaviors was discouraged. After removing the noise factor, a comparison between the performances with and without noise was made. This caused a big increase in performance for all EA presets and NN presets with distance inputs. But positional input types showed a decrease in performance after removing the noise. Further research is needed to determine the causes of the better performance of positional inputs with noise and the implications for the choice of inputs for agents in platform games.

Furthermore, the impact noise had on the NE not only shows how much the inputs influence performance but also that some input types unexpectedly perform better with noise. While existing research on neuroevolution in platform games is aware of the importance of NN inputs, these results show that even seemingly detrimental inputs can have a positive effect under certain circumstances, and a wide variety of inputs should be considered and tested in order to achieve good performance. Future research could further explore the impact of noise on different NN input types as well as consider not yet explored avenues for the types of inputs that are fed to NNs when evolving neural networks in platform games.

Lastly, EA presets with more exploitation were favored not only because of the noise but also after removing it. As the focus of the platform game was to have challenging jumps with small windows for input timings, it can be said that evolutionary algorithms benefit from exploitation when evolving agents that require precise outputs in these types of games. Since all platform games share the common objective of moving and jumping through the world, no matter what other kinds of objectives or mechanics might be available, these findings might be of benefit for neuroevolution in platform games as a whole. It has to be considered, however, that the performance of individual EA instances depends on the environment that they are tested in, and while these results can make claims about the importance of exploitation for movement in platform games, agents in other games may have secondary objectives to consider.

All in all, it could be shown that the performance of neuroevolution in platform games greatly depends on the chosen parameters of the evolutionary algorithm, the neural networks, and the specific game environment. The task of choosing the right parameters for the algorithm is not a trivial one and must be thoroughly researched for each individual platform game. In the future, more research is needed on platform games that focus on the movement of game agents and give general insights for how to choose the parameters of the NE algorithm to guarantee good performance.

# References

Bartz-Beielstein, T., Branke, J., Mehnen, J., & Mersmann, O. (2014). Evolutionary Algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *4*(3), 178–195. doi: 10.1002/widm.1124

Bavarian, B. (1988). Introduction to neural networks for intelligent control. *IEEE Control Systems Magazine*, *8*(2), 3–7. doi: 10.1109/37.1866

Brabazon, A., O'Neill, M., & McGarraghy, S. (2015). *Natural Computing Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-43631-8

Collobert, R., & Bengio, S. (2004). Links between perceptrons, MLPs and SVMs. In C. Brodley (Ed.), *Twenty-first international conference on machine learning - ICML '04* (p. 23). New York, New York, USA: ACM Press. doi: 10.1145/1015330.1015415

Eiben, A. E., & Smit, S. K. (2012). Evolutionary Algorithm Parameters and Methods to Tune Them. In Y. Hamadi, E. Monfroy, & F. Saubion (Eds.), *Autonomous Search* (pp. 15–36). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-21434-9_2

Eiben, A. E., & Smith, J. E. (2015). *Introduction to evolutionary computing* (Second Edition ed.). Berlin and Heidelberg and New York and Dordrecht and London: Springer.

Finley, I. P. (2015). *Evolving Intelligent Multimodal Gameplay Agents and Decision Makers with Neuroevolution* (Master's thesis, Oklahoma State University). Retrieved from `https://hdl.handle.net/11244/48976`

Gurney, K. (2003). *An Introduction to Neural Networks*. Hoboken: Taylor and Francis. Retrieved from `https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=182103`

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. doi: 10.1016/0893-6080(89)90020-8

Iba, H. (2018). *Evolutionary Approach to Machine Learning and Deep Neural Networks*. Singapore: Springer Singapore. doi: 10.1007/978-981-13-0200-8

Karafotias, G., Hoogendoorn, M., & Eiben, A. E. (2015). Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation*, *19*(2), 167–187. doi: 10.1109/TEVC.2014.2308294

Min, B., Ross, H., Sulem, E., Veyseh, A. P. B., Nguyen, T. H., Sainz, O., … Roth, D. (2023). Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*. doi: 10.1145/3605943

Mirjalili, S., Faris, H., & Aljarah, I. (2020). *Evolutionary Machine Learning Techniques*. Singapore: Springer Singapore. doi: 10.1007/978-981-32-9990-0

Ortega, J., Shaker, N., Togelius, J., & Yannakakis, G. N. (2013). Imitating human playing styles in Super Mario Bros. *Entertainment Computing*, *4*(2), 93–104. doi: 10.1016/

j.entcom.2012.10.001

Risi, S., & Togelius, J. (2017). Neuroevolution in Games: State of the Art and Open Challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, *9*(1), 25–41. doi: 10.1109/TCIAIG.2015.2494596

Sher, G. I. (2013). *Handbook of Neuroevolution Through Erlang*. New York, NY: Springer New York. doi: 10.1007/978-1-4614-4463-3

Shiffman, D. (2012). *The nature of code* (Vers. 1.0 ed.). Lexington KY. Retrieved from `https://natureofcode.com/book/`

Slowik, A., & Kwasnicka, H. (2020). Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, *32*(16), 12363–12379. doi: 10.1007/s00521-020-04832-8

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*(1), 24–35. doi: 10.1038/s42256-018-0006-z

Taud, H., & Mas, J. F. (2018). Multilayer perceptron (mlp). In M. T. Camacho Olmedo, M. Paegelow, J.-F. Mas, & F. Escobar (Eds.), *Geomatic Approaches for Modeling Land Change Scenarios* (pp. 451–455). Cham: Springer International Publishing. doi: 10.1007/978-3-319-60801-3_27

Ting, C.-K. (2005). *Design and analysis of multi-parent genetic algorithms* (Doctoral dissertation, Universität Paderborn). Retrieved from `https://digital.ub.uni-paderborn.de/hsmig/content/titleinfo/3764`

Togelius, J., Karakovskiy, S., Koutnik, J., & Schmidhuber, J. (2009). Super mario evolution. In *2009 IEEE Symposium on Computational Intelligence and Games* (pp. 156–161). IEEE. doi: 10.1109/CIG.2009.5286481

Togelius, J., & Lucas, S. M. (2005). Evolving Controllers for Simulated Car Racing. In *2005 IEEE Congress on Evolutionary Computation* (pp. 1906–1913). IEEE. doi: 10.1109/CEC.2005.1554920

Whitelam, S., Selin, V., Park, S.-W., & Tamblyn, I. (2021). Correspondence between neuroevolution and gradient descent. *Nature communications*, *12*(1), 6317. doi: 10.1038/s41467-021-26568-2

# Eigenständigkeitserklärung

Hiermit versichere ich, Pascal Süß,

(a) dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe.

(b) Außerdem erkläre ich, dass ich der Universität ein einfaches Nutzungsrecht zum Zwecke der Überprüfung mittels einer Plagiatssoftware in anonymisierter Form einräume.

Passau, den 28. Juli 2023 Süß Pascal

Pascal Süß