FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Meistern von unterschiedlichen Computerspielen mittels Generation Based Learning

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Manuel Esberger
Matrikelnummer 01525631

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Allan Hanbury
Mitwirkung: Pretitle Forename Surname, Posttitle
Pretitle Forename Surname, Posttitle
Pretitle Forename Surname, Posttitle

Wien, 30. September 2018

Manuel Esberger	Allan Hanbury

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mastering Divserse Computer Games using Generation Based Learning

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Medical Informatics

by

Manuel Esberger
Registration Number 01525631

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Allan Hanbury
Assistance: Pretitle Forename Surname, Posttitle
Pretitle Forename Surname, Posttitle
Pretitle Forename Surname, Posttitle

Vienna, 30th September, 2018

Manuel Esberger

Allan Hanbury

Erklärung zur Verfassung der Arbeit

Manuel Esberger
Preßgasse 11/2 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. September 2018

Manuel Esberger

Danksagung

Zu erst möchte ich meinen Großeltern danken. Sie haben mir angeboten bei ihnen zu wohnen, als ich mein Studium angetreten habe.
Ich werde mich vermutlich mein Leben lang daran erinnern, dass ich bis spät in die Nacht in die Tastatur hämmerte, um Tätigkeiten zu erfüllen, die das Studium von mir verlangten, während mein Großvater versuchte im Nebenzimmer Schlaf zu finden. Er hat es sich nie anmerken lassen, dass mein Lernen nicht nur mich wach gehalten hat. Auch erinnere ich mich an die vielen grantigen Diskussionen mit meiner Großmutter, wenn es im Studium mal nicht so rund lief. Sie hat mir jedes zornige Wort verziehen und sie grüßt mich weiterhin Willkommen in ihrem Heim.
Hätten sie mir nicht ihre Türen offen gehalten und mir einen Platz zum Lernen angeboten, wäre das Studium vermutlich nicht möglich gewesen.

Weiteres möchte ich meiner Freundin und baldigen Mutter meines Sohnes Danken. Auch wenn es abseits vom Studium viel zu Tun gab, wie zum Beispiel Möbel kaufen, dem Job oder den nicht enden wollenden Arztbesuchen, erinnerte sie mich immer wieder daran an meiner Bachelorarbeit zu schreiben. Ebenso, wenn sie manchmal fragte, ob wir etwas Zeit für uns haben wollen und ich sie aufgrund der Arbeit abwies, zeigte sie sich mit Verständnis.

Zu guter Letzt möchte ich auch allen Professorinnen, Professoren und Universitätsangestellten danken, die nicht nur an der Wissensvermittlung interessiert waren, sondern die auch aktive Schritte gesetzt haben, um interessierten Studierenden bei ihren Lernprozessen zu unterstützen. Ich denke, die Universität würde nicht ohne ihnen funktionieren und ich habe sie auch für mein Studium schätzen gelernt.
Vielen Dank!

Acknowledgements

First of all, I want to thank my grandparents. They offered me to live with them when I started to study.
Probably, I will remember all my life that I pounded into the keyboard to solve all the tasks that studying demanded to solve. Meanwhile, my grandfather tried to catch some sleep in the adjoining room. Still, he never mentioned that my studies not only kept me awake. Also, I remember the many times I grumpily argued with my grandma when University was rough. She forgave me every angry word and still welcomes me to her home.
If they wouldn't have opened their doors for me, most likely I wouldn't have been able to study.

Furthermore, I want to thank my partner, who also happens to become the mother of our son in a few months. When there was much to do apart from studying, like buying furniture, working a job or the never-ending visits at the doctor's place, she still reminded me to take time for my bachelor work. On the other hand, when she asked to spend some time together and I dismissed her proposal because of the bachelor work, she always showed herself understanding.

Last but not least, I want to thank all the professors and employees of the University, who not only tried to transfer their knowledge but also took active steps to support students who were interested. I think that the University would not work without them and I came to appreciate them while my studies.
Thank you!

Kurzfassung

Im Alltag sind viele komplexe Aufgaben enthalten, die teilweise oder gänzlich noch nicht automatisierbar sind. Oftmals gibt es gute Ansätze dafür, jedoch bietet der künstlich generierte Automatismus noch zu viele Unbekannte. Um etwas näher an die Automatisierung von zurzeit manuell gesteuerten Prozessen zu kommen, befasst sich diese Arbeit mit NeuroEvolution of Augmenting Topologies(NEAT), einem Neuroevolutionsalgorithmus, der sein neuronales Netz mittels einem genetischen Algorithmus erweitert. Um eine breit gefächerte, dennoch überschaubare Testumgebung zu haben, wende ich den Algorithmus auf zwei unterschiedlichen Spielen an, nämlich Super Mario World und Flappy Bird. Ich werde zeigen, wie sich das Ergebnis verändert, wenn eine unterschiedlich große Initialpopulation gewählt wird, während die Simulationsläufe pro Populationsklasse annähernd konstant bleiben (n=2828 bis 5329 bei Super Mario World; n=627 bis 1684 bei Flappy Birds). Weiteres werde ich zeigen, dass die Ergebnisse nur teilweiße vergleichbar sind und woran das liegt. Es wird in der Arbeit beleuchtet, welche Parametereinstellungen für welche Umgebungen bessere Ergebnisse erzielen und ein Ausblick für zukünftige Arbeiten wird geboten.

Abstract

Everyday's life contains many complex tasks. This task can be automated only partially or jet not at all. However, there are good rudiments for this, still, the generated automatism holds many unknowns. In order to get closer to automating manually controlled processes, this work is contributed to NeuroEvolution of Augmenting Topologies(NEAT). NEAT is a neuroevolution algorithm, which builds up its neuronal network through a genetic algorithm. In order to have a broad, but still manageable test-environment, I use this algorithm on two different games. These games are Super Mario World and Flappy Bird. I will show how the results change if different initial population sizes will be used, although the number of simulations is staying constant to some degree (n=2828 to 5329 at Super Mario World; n=627 to 1684 at Flappy Birds). Furthermore, I will show that the results are comparable only partially and I will give the reasons for this. This work will point out what parameter settings to use for what environments to get better results. Last but not least I will give an outlook for future works.

Contents

CHAPTER 1

Introduction

"Some people worry that artificial intelligence will make us feel inferior, but then, anybody in his right mind should have an inferiority complex every time he looks at a flower."

— Alan Kay, *(Computer Scientist)*

1.1 Motivation and Problem Statement

In the last decade may different solutions for neuronal networks (NN) have been implemented, whereas these implementations propose various changes like the amount and distribution of connections between neurons, the weight calculations between neuronal connections or the number of neuronal layers of the network as well as other structural decisions. The most popular subject of this questions is the XOR affair, which was proven to be unsolvable by a single layer NN[New69] The efficiency of these algorithms depends on the problem space and the environment in which they were tested. Two popular fields of using Artificial Neuronal Network (ANN)s are in image recognition and forecasting.[KAV15, MNS+17]
One popular method of adopting a neuronal network is via Genetic Algorithm (GA) since GAs offer a way to find new and possibly enhanced patterns in a reasonable (but not necessarily fast) time. In the case of NNs, GAs are used to find new connections between neurons or different structures inside the network. One popular implementation of this combination is Neuro-Evolution of Augmenting Topologies (NEAT), among others.[SM02] Since it not trivial to decide how the NN architecture should look like NEAT builds up it's architecture autonomously and in a minimalistic way.
These NN implementations are used in various fields as mentioned before. One field with rather clear boundaries is games, compared to real-world applications. Still, many types of games with different complexities exist[RT14]. Therefore this work analyses two different games which are played by autonomous NEAT implementations for these games. The first NEAT implementation is MarI/O for the game Super Mario World, made by

a popular YouTube-uploader called SethBling[1]. Since Super Mario World is a rather complex game, the results are later compared to a NEAT implementation for Flappy Bird developed for a coding challenge called NEAT_FlappyBird [2,3].
Super Mario World and Flappy Bird are two different games when considering their achievements. A level of Super Mario World has a finite game map but still offers a high level of complexity compared to the input possibilities of Flappy Bird. However, Flappy Bird has an infinite and self-generating map. Flappy Bird is quite challenging to humans because of the unexpected map and fixed game speed.
Still, it is expected that the game solving implementation for Super Mario World takes longer to complete a level than to find a solution for Flappy Bird that can exceed a certain threshold score because of the many possibilities of solving a level in Super Mario World.

1.2 Results

The NEAT algorithm finds a solution for both environments. MarI/O finds a solution in the majority of runs, namely 7 times out of 9. NEAT_FlappyBird exceeded a defined threshold in all runs.
Interestingly, many correlations that could be found in the environment Super Mario World, could not be confirmed in Flappy Bird. The results seemed more random in Flappy Bird, which can be caused by the random environment generation of Flappy Bird, wheras Super Mario World has predefined levels.
However, what could be concluded consistently, was that the average distance from the score of the majority of runs (calculated by the median) to the best runs of each generation tend to rise when using a higher population. Since there where fewer generations when the population count increased, the average fitness increase from generation to generation increased as well in both games.

[1] `https://www.youtube.com/channel/UC8aG3LDTDwNR1UQhSn9uVrw`, last accessed on 30th of October 2018

[2] `https://github.com/llSourcell/neuroevolution-for-flappy-birds`, last accessed 30th October 2018

[3] `https://github.com/rsk2327/NEAT_FlappyBird`, last accessed 30th October 2018

CHAPTER 2

Related Work

2.1 NEAT

NEAT stands for NeuroEvolution of Augmenting Topologies and is a method of constructing generation based NN with the use of GA. [SM02] Over the time many implementations in many programming languages were created[1]. Furthermore many extensions and amendments exist[2,3,4] that try to solve different aspects of different problems more efficiently than the basic implementation.[KM11]
Still NEAT has proven to provide solutions to 3 common problems [SM02]:

1. **Competing Conventions** In ordinary GAs it can happen that genomes which hold similar solutions but are differently encoded create worse children than their parents have been.
   In NEAT historical markings are introduced, namely the innovation number. When a new structure within the genomes is created, this structure will be assigned with an incremented innovation number. So whenever two individuals are chosen to mate, their genes with the same innovation number (therefore it is a historical marking) are aligned and the different genes, which don't align with the ones from the partners, are exchanged.

2. **Protecting Innovation through Speciation** When new genomes are created through crossover they often end up worse than before since they need time to adapt and specialize. However usual genetic algorithms are not very tolerant to this type of trainings. That's why NEAT introduced the concept of specification,

[1] `http://eplex.cs.ucf.edu/neat_software`, last accessed on 31st October 2018
[2] `https://www.cs.ucf.edu/~kstanley/neat.html`, last accessed on 31st October 2018
[3] `http://eplex.cs.ucf.edu/hyperNEATpage/HyperNEAT.html`
[4] `http://eplex.cs.ucf.edu/ESHyperNEAT`, last accessed on 31st October 2018

like it happens in nature. Genomes (the population) then get divided into species groups that protect genomes that still have to optimize. For the genomes it is less likely to mate with individuals from other species even when their fitness is equally high. Still this behavior has proven to support diversity because a species with many genomes shared a higher fitness, allowing some individuals inside the species to differentiate.

3. **Topological Innovation** Last but not least, NEAT keeps the topology of the network minimal. In the paper [SM02] the authors state that random initializations of the network cause many problems, like inefficient networks or no paths from input to output neurons. It takes time to sort out the problems caused by random initial topologies. By keeping the network as simple as possible these problems don't come into weight and the search space is as minimal as possible, which enhances performance.

2.2 Tools

For the completion of this work, some other work was taken and further analyzed. Of course these works used programs, scripts and other tools for their work as well, still, I want to give an entry reference, so others can reproduce my work at wish.

MarI/O A popular YouTuber called SethBing published his project MarI/O on YouTube and explained rough details of it shortly[5]. However, he also published the code he produced[6], which was used for this work. MarI/O is an implementation of the NEAT-algorithm mentioned prior in this chapter(see 2.1) used for the video game Super Mario World. The program is written in LUA-script and can be used with the Blitzhawk emulator[7].

NEAT_FlappyBird The second NEAT implementation is called NEAT_FlappyBird which is written in Python for a coding challenge[8,9]. This implementation uses a Python-framework for NEAT called NEAT-Python[10]. The game Flappy Bird was rewritten in Python as well with the pygame-framework[11].

Python for statistics For this work the used implementations were manipulated, so they write the results into a text file. This text-file was later analyzed using Python-

[5] `https://www.youtube.com/watch?v=qv6UVOQ0F44`, last accessed on 31st October 2018
[6] `https://pastebin.com/ZZmSNaHX`, last accessed on 31st October 2018
[7] `http://tasvideos.org/BizHawk.html`, last accessed on 31st October 2018
[8] `https://github.com/rsk2327/NEAT_FlappyBird`, last accessed on 31st October 2018
[9] `https://github.com/llSourcell/neuroevolution-for-flappy-birds`, last accessed on 31st October 2018
[10] `https://neat-python.readthedocs.io`, last accessed 30th October 2018
[11] `https://www.pygame.org`, last accessed on 31st October 2018

scripts as well. For the statistics the default python statistics framework was used. For the plots, I used the framework matplotlib.

CHAPTER 3

Generation Learning in Computer Games

In this chapter, the algorithm NEAT (see chapter 2.1) will be applied on the two environments Super Mario World and Flappy Bird. These environments are very different from their inputs and their goals. In Super Mario World many input possibilities are given, whereas Flappy Bird allowed only one input. These inputs should be calculated by the Artificial Intelligence (AI) implementation, namely the NEAT algorithm. The outcome of the algorithm should result in a certain score called the fitness. Later in this chapter, the fitness progress over the generations will be shown on a graph. The result of this simulations is to see how the algorithm behaves with this different challenges.

3.1 MarI/O

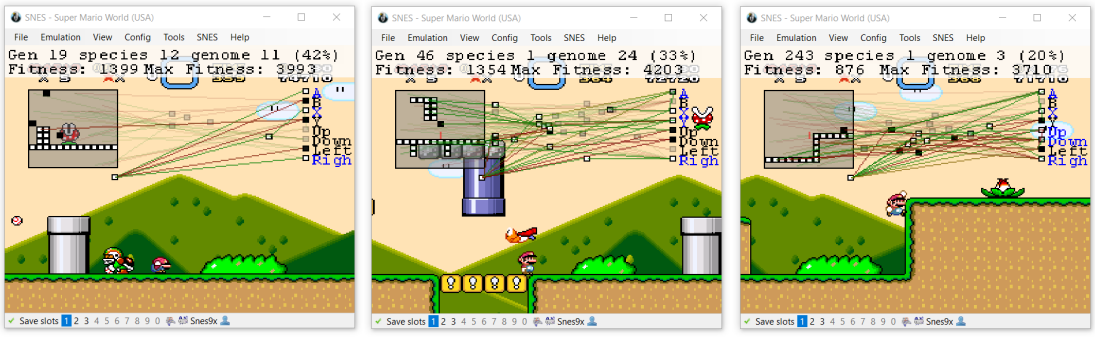

Figure 3.1: MarI/O simulation

As mentioned in section 2.2 MarI/O (see figure 3.1) is an implementation of NEAT-algorithm written in LUA. It provides a solution for automatic learning of the game Super Mario World. In Super Mario World a level is a two-dimensional map with steady as well as moving obstacles. Some of them block the path to the goal and others cause damage to Mario's health. A few of them give health upgrades to Mario or add coins to the player's account, although the coins are ignored in the implementation of the AI. Since there are many positions in which Mario can stay and the speed of the game depends mostly on the player and his/her/its decisions the environment of Super Mario World is rather complex when compared to the second game Flappy Bird 3.2.
This complex world leaves the expectation that many hundred thousand runs are necessary to learn how to complete a level. However, MarI/O implementation reached to goal after approximately 2664.29 runs on average in the simulations described later in this section. Still, 2 of the 9 simulations didn't reach the goal once.

For the selection of the fittest genomes a fitness function has to be defined. In the LUA-script following lines of code indicate the fitness:

```
local fitness = rightmost - pool.currentFrame / 2
```

whereas rightmost is the furthest point reached so far. It was defined prior as:

```
getPositions()
if marioX > rightmost then
        rightmost = marioX
        ...
end
```

Basically the goal is reached at a fitness score of around 3900 to 4000. For the means of completeness, it should be mentioned that there is also a bonus programmed that is calculated like the following:

```
if gameinfo.getromname() == "Super Mario World (USA)" and rightmost > 4816 then
        fitness = fitness + 1000
end
```

Still, in none of the 37957 runs made, the bonus was applied.

Later in this section, 3 figures with 3 similar graphs will be shown. The three different figures display the success of the algorithm in different classes of initial population size. Since the NEAT-algorithm used does not produce a deterministic amount of populations after the first generation (in general: Generation 0), the initial population size defines these classes. There were three classes chosen with a scaling factor of 5 between them. These initial population sizes are 10, 50 and 250. Whether or not the initial population sizes are well-chosen will be discussed shortly in the conclusion section (see 4) of this chapter. Depending on the evolution of the NN, there are a certain amount of generations evolved. Every generation contains their own set of species. And on the other hand, the species contain the genomes. In generation 0 every species contains only one genome each. The sum of all genomes in all species of a generation is called the population. In the cases where the initial population size is 10 or 50 over time to many generations were

created to show a viewable graph in the end. That's why only 30 generations were picked in the display with even distances between them. Still, a continuous line with the best run of a population is showed above all generations, even the skipped ones.
In the later descriptions of the population classes, there are two types of runs introduced. First is the "plot-run" which indicates the simulation and the graph. Inside this graph, there were many "runs" which represent the runs of the population (genomes) of each species. In figure 3.1 there are 3 individual runs displayed. On average one plot-run consists of $4217.\overline{2}$ runs, whereas population 10 has 2828 runs on average, population 50 consists of $4494.\overline{6}$ runs on average and population 250 of 5329 runs averaged.
In order to understand the fundamental differences of these simulations, the population classes are examined in more detail:

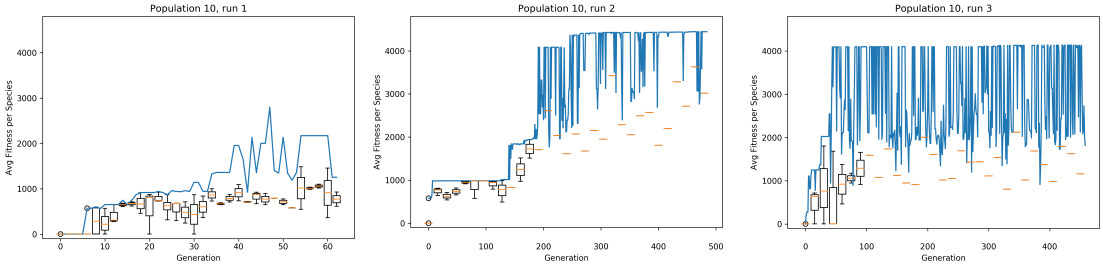

Figure 3.2: MarI/O Population 10

Population 10 / Generation 500 As it is visible in figure 3.2 the vertical axis shows the fitness score average of the genomes within a species. The horizontal axis portraits the generations containing the species. Each generation contains up to 10 populations which are divided into species and genomes within species. This species division was made based on the NEAT algorithm described in section 2.1. The best run of the genomes grouped by each generation is marked with a blue line. Therefore the blue line indicates the best overall run within a generation. Since the boxplot portrays the species's average score of each generation and the blue line shows the best run per genome (population), the boxplot and the blue line rarely meet. Still, the average population score is closer to the best run than in the next two population variants (see later in this section population 50 3.1 and population 250 3.1). This can be calculated by taking the median of the species fitness and subtracting that number from the best run of the genomes: $average\_distance = \frac{\sum_{g_i \in generations} max(g_i.genomes) - median(g_i.species)}{|generations|} \approx 1107$ whereas $g_i.genomes$ and $g_i.species$ are lists of the respective fitness.
In the three plot-runs on average $334.\overline{6}$ generations were created, which results in a skipping of generations inside the graphics of around $11.1\overline{5}$ generations averaged, between two displayed generations. Unfortunately, the first run crashed after generation 60. Still, because of the long runtime of the simulation, the plot-run was kept. However, indicated by plot-run 2 and 3, the population growth started after this generation. As it can be seen in the 3rd plot-run of figure 3.2, sometimes runs over 3000 fitness score could be

achieved even after the 30th generation. In plot-run 2 the average fitness of the single species left tends to rise, however, more and longer plot-runs would be needed to test this hypothesis.
In each generations, there are up to 10 populations. In the first generation (Gen 0) no mating was done. So in the first generation there were 10 species spawned with one genome each. In the 10th generation on average only $4.\overline{3}$ species where left. After generation 50 maximal 3 species where left in all runs and after generation 190 in plot-run 2 and after generation 91 in plot-run 3, respectively, only 1 species was left for mating. The mating results into the crossover of species.
All runs except plot-run 1 reached the goal (the end of the level) multiple times which can be seen by the fitness score being over 4000. However, plot-run 3 reached the goal the earliest with runs over 4096 starting from generation 44. Still, there was the most overall regress made in plot-run 3. This can be calculated by adding the differences between the best runs of each generation if the difference was negative: $average\_regress = \frac{\sum_{g_i \in generations} min(max(g_i.genomes) - max(g_{i-1}.genomes), 0)}{|generations|} \approx -348$ again whereas $g_i.genomes$ is a list of the fitness of each genome inside the generation. The regress of plot-run 1 was -88.27 approximately and of plot-run 2 was around -109.98.
In plot-run 1 the $average\_fitness\_increase = \frac{\sum_{g_i \in generations} max(g_i.genomes) - max(g_{i-1}.genomes)}{|generations|} \approx 19.87$ was the biggest of the three plot-runs since the first plot-run ended early and plot-run 3 had many drawbacks. The average fitness increase of plot-run 2 was around 7.97 and of plot-run 3 was only 3.95 approximately. Since it is only slightly possible to extend the maximum score above the score of 4000 and plot-run 1 has never reached this ranking, plot-run 1 pointed out to have the best score increase per round. Every successful round, whereas Mario reached the goal will only minimize the fitness increase when averaged with the generation count. In other words, for an infinitely large amount of generations the $average\_fitness\_increase$ is expected to converge to 0 since the game has an end-state in contrast to the game Flappy Bird, as it can be seen in section 3.2. In mathematical terms: $\lim_{n \to \infty} average\_fitness\_increase(n) = 0$, whereas the $average\_fitness\_increase(n)$ is defined as the average fitness increase of a set of n generations.

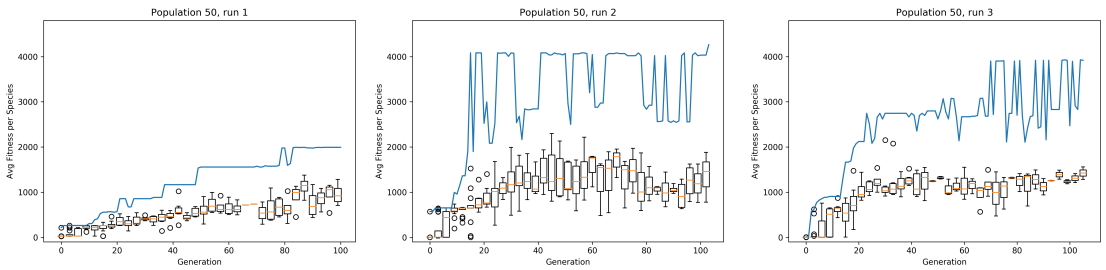

Figure 3.3: MarI/O Population 50

Population 50 / Generation 100 In this setup, the population count is up to 50, again distributed into species and genomes within species according to the MarI/O NEAT implementation. The *average_distance* between the median of the species of each generation to the best genome run of this generation is bigger than that of the simulation with its initial population size of 10 but it is smaller than in the last case. The *average_distance* was calculated as described in the previous simulation (population 10 3.1) and the value is approximately 1406.
In this simulations the plot-runs where executed until there were over 100 generations (101 generations in plot-run 1, 104 in plot-run 2 and 106 in plot-run 3). This results in an average skipping of 3.456 generations between the display of two generations.
In generation 0 there were 50 species spawned, again, with one genome each. In the 10th generation, there were 15 species left on average. At the end of generation 100, on average $3.\overline{3}$ species were left from the initial 50 generations.
Interestingly the plot-run 1 couldn't learn to reach the goal. From this data, it is not trivial to predict if the breakthrough would have started within the next 50 generations or if this plot-run would have stayed low in its fitness score since there are no clear patterns to find in the graphical representation of these runs. In order to answer this question more profoundly, further and longer plot-runs have to be made and the big jumps between the fitness scores of each neighbor generation would have to be analyzed.
Plot-run 2 and 3 had more luck in reaching the end, however, plot-run 3 had more stability in its high score results between generations. Still, after generation 70 plot-run 3 also shows stronger differences between its generation's high scores. Nevertheless, plot-run 2 reached the goal the earliest. The first time plot-run 2 achieved a fitness-score over 4000 was in generation 15 (it reached a score of 4082.5), whereas plot-run 3 reached a maximum score of 3928 in generation 98. Still, plot-un 3 reached to goal with a score of 3902 the first time in generation 70. The 3rd plot-run has the highest $average\_fitness\_increase \approx 36.92$ of the three plot-runs. Plot-run 1 has an $average\_fitness\_increase$ of 17.58 approximately and plot-run 2 of 35.5 precisely.
Plot-run 2 and 3 have similar *average_regress* values with about -158.02 for plot-run 2 and -152.37 for plot-run 3. Because of the early end of a general low performance of plot-run 1, the *average_regress* is also the lowest with -6.30. Still, there were only 15 cases where the succeeding generation performed worse than the previews in plot-run 1 whereas there were 29 of these cases in plot-run 2 and 30 in plot-run 3. This indicated a certain stability in the first plot-run although the maximum score remained far lower than 3000.

Population 250 / Generation 30 In figure 3.4 the population size is up to 250 in generation 0. In the first generation (Gen 0) 250 species are born with one genome each. The *average_distance* for this plot-runs is the biggest with approximately 1827 when compared to the plot-runs with an initial population size of 10 and 50. In this plots, no generations had to be skipped in order to portray a descriptive graph since the maximum generation count is 30 in plot-run 2 (25 generations in run 1 and 29 generations in run 3). Already in the 6th generation, on average only 94.8 species were left. At the end of

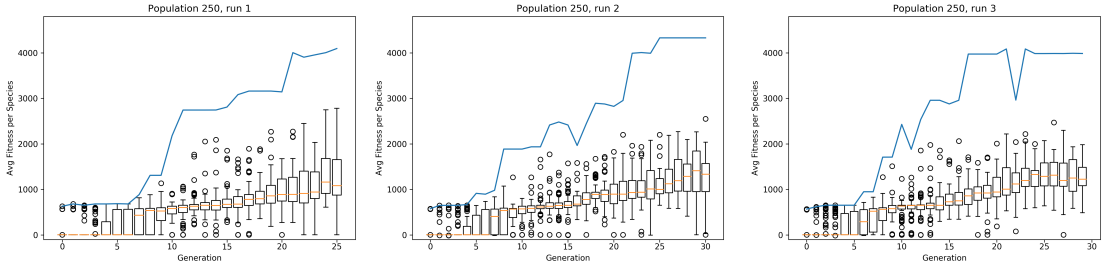

Figure 3.4: MarI/O Population 250

generation 25 there were 31 species left on average.
Compared to the other two population classes, there are at least 7 times more species left at the end of the simulations which results in longer whiskers of the boxplot. The whiskers even contain bad starts with fitness-scores lower than 100 in plot-run 1 and 2. Interestingly the best runs are always exceptions after generation 6 (in plot-run 1 and 2 even earlier).
Further, it is to mention that the plots are rather uniform compared to the plots of population 10 and 50. Therefore the *average_fitness_increase* has similar values with a low variance which are 133.25 for generation 1, around 121.08 for generation 2 and 113.95 for generation 3. The *average_regress* is the lowest in plot-run 1 at -5.87 approximately. This is because the maximum value of the succeeding generation is smaller than the previous generation in only 4 cases. The other two plot-runs have an *average_regress* of about -19.97 in plot-run 2 and -61.92 in plot-run 3. All of the plot-runs reached the end of the level even though plot-run 3 reached the end at generation 17, whereas plot-run 1 reached the end at generation 23 and plot-run 2 at generation 22.

MarI/O	avg. runs /σ	avg. fitness score /σ	avg distance /σ	avg. regress /σ	avg. fitness increase /σ
Population 10	2828 /2055.44	1231.42 /531.37	1107.09 /534.5	-182.03 /144.01	10.6 /8.28
Population 50	4494.$\bar{6}$ /176.09	960.96 /321.34	1405.96 /664.75	-105.56 /86.01	30 /10.78
Population 250	5329 /656.74	776.31 /57.88	1826.32 /81.79	-29.25 /29.16	122.76 /9.76

Table 3.1: MarI/O Population Comparison Overview

Comparison of the results In order to compare the results, 5 distinct values (see table 3.1) of the plot-runs where calculated, as three of them were introduced in more detail earlier in this section 3.1. The first observations indicate that the average fitness score of each generation drops when establishing a bigger initial population. However, the standard deviation tends to drop as well.
Also, the distance of the median of the species to the best run of the generation seems to become greater with a greater population count in generation 0. However, the average regress (if present) becomes lower with bigger population sizes and fewer generations, as well as it's deviation. Since there are fewer generations in the simulations with an initial

population of 250 and these simulations having similar achievements, the average fitness increase is higher than in the other two simulation classes. The standard deviation of the average fitness increase is relatively similar.
It is interesting to see how the fitness increase compares to the average distance value. Even though the fitness increase of population class 250 is higher than the fitness increase of population class 10, the distance remains large which indicates that the majority of runs stayed low and the average score of population class 10 is higher than in the other two population classes. Still, the other two classes remained more stable when taking the average regress into account.
Another interesting point of view is the reaching of the end of the level (the goal). In all population classes, the goal was reached even though population class 1 and 2 didn't reach the goal in one plot-run each. Population 10 reached the goal in the first 24.48% on average, only including the cases where the goal was reached. Population 50 reached the goal in the first $40,24\%$ on average and population class 250 in the first $69,47\%$.
To summarize the results roughly it can be said that an initial population size of 10 promises faster and better results in a complex environment like Super Mario World, however, more stability can be reached when increasing the initial population size.

3.2 NEAT_FlappyBird

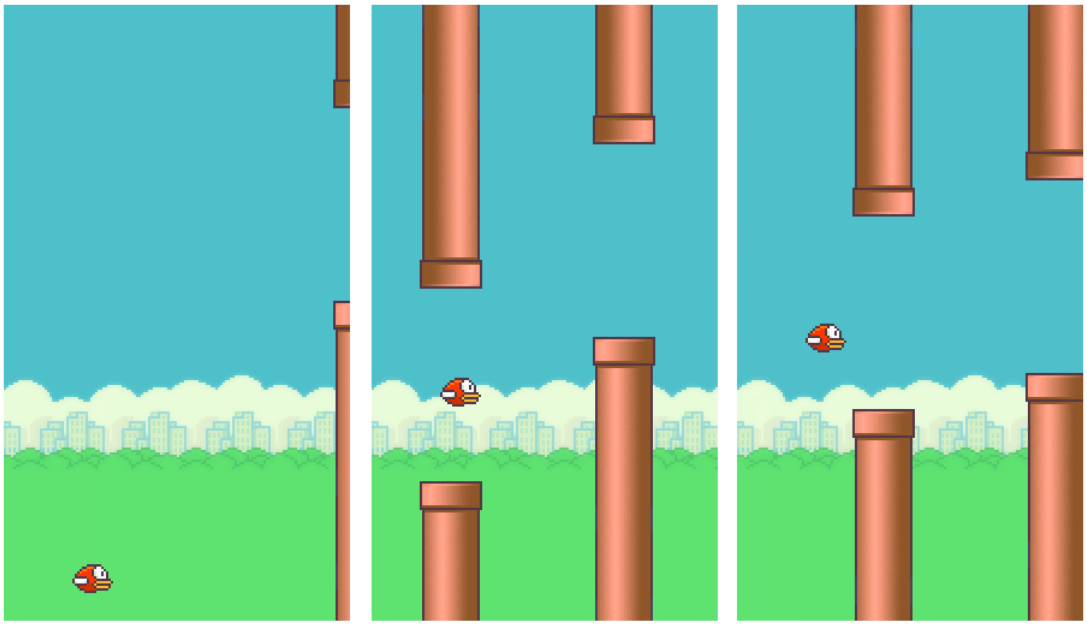

Figure 3.5: Flappy Birds simulation

As mentioned in section 2.2, NEAT_FlappyBird (see figure 3.5) is an implementation of NEAT-algorithm written in Python using the NEAT framework NEAT-Python [1]. It provides a solution for mastering the game Flappy Bird which was also rewritten in Python. In Flappy Bird, a level is a two-dimensional, infinite map with obstacles along the way. The player, in the representation of a bird, moves at a constant speed to the right, where new obstacles are generated. These obstacles are pipes which create gaps, through which the bird has to fly (see figur3.5).
Therefore the player has only one input, namely if he/she/it wants to fly up, or if not then the player automatically falls down. Since there is only one decision to make the game is rather simple compared to Super Mario World (see section 3.1).

For the algorithm, a fitness function was chosen that takes the passed pipes as well as the vertical distance into account:

```
pipe1Pos = pipe1.move()
if pipe1Pos[0] <= int(SCREENWIDTH * 0.2) - int(bird.rect.width/2):
        if pipe1.behindBird == 0:
                pipe1.behindBird = 1
                SCORE += 10

pipe2Pos = pipe2.move()
if pipe2Pos[0] <= int(SCREENWIDTH * 0.2) - int(bird.rect.width/2):
        if pipe2.behindBird == 0:
                pipe2.behindBird = 1
                SCORE += 10
```

[1] https://neat-python.readthedocs.io, last accessed 30th October 2018

```
vertDist = (((bird.y - centerY)**2)*100)/(512*512)
time += 1
fitness = SCORE - vertDist + (time/10.0)
```

This simple set-up creates the expectation that the NEAT algorithm should be able to find a solution with a score above a certain threshold (which is defined later in this section). In fact, all simulations exceeded this threshold and the generations used where far lower the maximum generation defined for this simulations. NEAT_FlappyBird reached the fitness threshold after $1047.\overline{4}$ runs on average.
In this section, 3 figures similar to the ones used in section 3.1 are shown. However, this time these figures are split into two ranges since the details of the graph are too far apart. Therefore a simulation of a population class contains 3 plot-runs whereas one plot-run is displayed in 2 graphs of different ranges. For the size of the population classes, the same 3 sizes were chosen as used in the MarI/O simulation, which are 10, 50 and 250.
Again a maximum amount of 30 generations are displayed in the graphs when the generation size was bigger than 30. Luckily, this was only the case in the population 10 simulations. Also, again, the generations contain the species, which by themselves contain the genomes (the population). As in MarI/O a "plot-run" indicates the simulation and the graph which was plotted based on the simulation. Inside this graph, there were many "runs" executed which represent the runs of the population (genomes) of each species.
Now the simulations are examined in more detailed categorized by their population size:

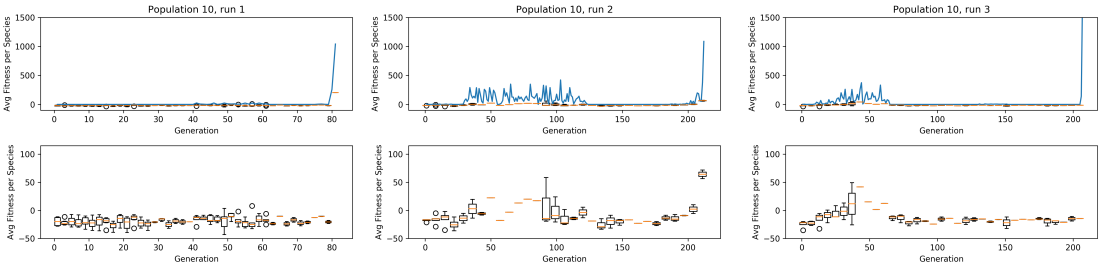

Figure 3.6: Flappy Bird Population 10

Population 10 / Generation 500 As it is with the graphical representations of MarI/O (see figure 3.2 for example) the vertical axis shows the fitness score average of the genomes within a species. Again, the horizontal axis portrays the generations containing the species. Each generation contains up to 10 populations which are divided into species and genomes within species. In this NEAT-implementation a fixed size of generations was specified.
As in MarI/O the best runs of genomes per generation is marked with a blue line. In the Flappy Bird simulations the best runs and the median of each species (or the *average_distance* described in section 3.1) are to far apart to be able to display them in one graph of linear scale. That's why two graphs are shown with different fitness ranges. The upper part displays a range from -100 to 1500 and the upper plot shows values in

the range of -50 to 115 fitness score.
Although the *average_distance* is big in general, this plot-run has the smallest *avergage_distance* of the three simulation classes with an average value of 51.3. In this three plot-runs the upper boundary was 500 generations, however, there was an fitness-threshold implemented as well which ended the simulation when a fitness score over 600 was reached. Plot-run 1 launched 81 generations, plot-run 2 launched 307 generations and plot-run 3 had 367 generations before this threshold was exceeded. This results in an average skipping of $8.3\overline{8}$ generations inside the graphs, between two displayed generations.
Interestingly, plot-run 2 and 3 managed to enhance their score by generation 30 but dropped again latest at generation 130, however, all three plot-runs managed to reach a score beyond 600. Plot-run 1 reached this goal the earliest in generation 81 and therefore needed less than a third of the generations plot-run 2 and 3 spawned.
In comparison to MarI/O (see section 3.1) wherein generation 0 there were as many species spawned as configured with the population size, whereas every species contained only 1 genome, this implementation of the NEAT algorithm spawns the configured amount of genomes first and after the first simulation run assigns them into species. Moreover, the generation number starts from 1 and not from 0 as in MarI/O. So after the first run on average $5.\overline{3}$ species where classified. In generation 80 on average $1.\overline{6}$ species where left. Since there was a setting configured which reset the species if a total extinction (no species left) has occurred. These values have to be considered carefully.
In plot-run 2 and 3 there was a significant *average_regress* made of approximately -24.54 in plot-run 2 and -11.38 in plot-run 3. Plot-run 2 regressed 67 times and plot-run 3 41 times. Plot-run 1 kept the *average_regress* far lower with an approximate value of -1.42.
Since plot-run 1 has the fewest generations the $average\_fitness\_increase$ is the highest with a value of around 16.04, whereas plot-run 2 has a value of ≈ 6.86 and plot-run 3 of ≈ 10.54. However, the average score of each generation is the lowest in plot-run 1 (≈ -15.87). Plot-run 2 has a value of about -4.22 and plot-run 3 of -10.62.
Since the game environment is open-ended the expectations from MarI/O (see 3.1) that $\lim_{n\to\infty} average\_fitness\_increase(n) = 0$ does not hold here. However, the opposite is true that $\lim_{n\to\infty} average\_fitness\_increase(n) = \infty$ is to be expected. Therefore there was the fitness-threshold introduced to end the simulation before an infinite flight is expected.

Population 50 / Generation 100 These simulations have an average *average_distance* of 270.25. In this instance, the standard deviation should be taken into account, which is also quite high with a value of 192.24. In the plot-runs of this population class, the upper boundary was 100 generations, however, again the fitness-threshold of 600 was exceeded in every plot-run. Plot-run 1 launched 4 generations, plot-run 2 launched 9 generations and plot-run 3 had 24 generations when the threshold was exceeded. Therefore no generation had to be skipped in the graphical display.
After the first run, in generation 1, the genomes were divided into $9.\overline{3}$ species on average. At the end of generation 4, plot-run 2 and 3 kept their species and plot-run 1 lost one

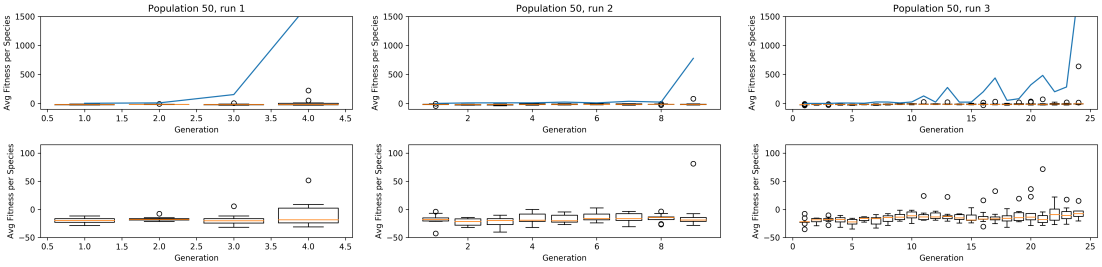

Figure 3.7: Flappy Bird Population 50

species. The species-reset did not had to be used at this point. In plot-run 3, 7 species were left at the end (generation 24). In plot-run 1 no regress occurred and in plot-run 2 only one time a regress was made which results in an average-regress of -0.07. Plot-run 3 had many potential outbreaks which resulted in 9 regresses with an *average_regress* of ≈ -45.84.
The *average_fitness_increase* is the highest in plot-run 1 since the generation count is the smallest. Its value is 461.52. The *average_fitness_increase* values are similar in plot-run 2 and 3 with approximately 87.93 in plot-run 2 and 92.79 in plot-run 3. The average score of each generation remains negative in this simulations for all plot-runs.

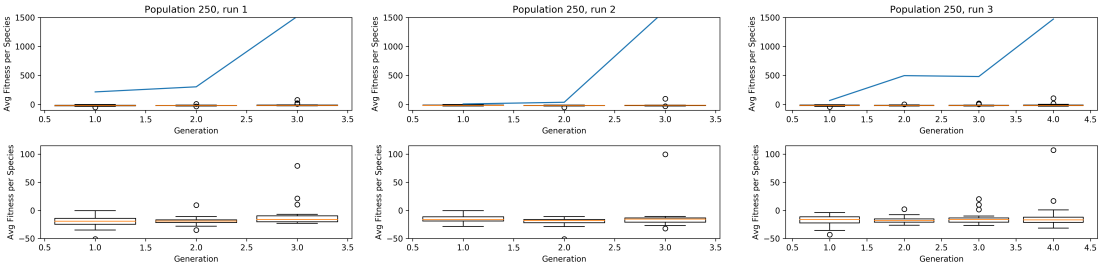

Figure 3.8: Flappy Bird Population 250

Population 250 / Generation 30 In three plot-runs have an average *average_distance* of 636.5. Compared to the population 50 instances the standard deviation is lower with a value of ≈ 63.14. In the plot-runs of this population class, the upper boundary was 30 generations, however, again the fitness-threshold of 600 was exceeded in every plot-run. Plot-run 1 and 2 launched 3 generations, and plot-run 3 had 4 generations when the threshold was exceeded. Again, no generation had to be skipped in the graphical display. After the first run, in generation 1, the genomes were divided into $22.\overline{3}$ species on average. At the end of generation 3, all plot-runs contained more than 10 species each.
Interestingly all plot-runs could avoid regress within the next generations.
The *average_fitness_increase* was high in general with a value of 433.86 in plot-run 1, 537.57 in plot-run 2 and 347.07 in plot-run 3 since there were few generations spawned.

Since the NEAT algorithm had no problems with this game it is more interesting to see how this population classes compare to one another.

NEAT_FlappyBird	avg. runs /σ	avg. fitness score /σ	avg distance /σ	avg. regress /σ	avg. fitness increase /σ
Population 10	1684 /714.01	-10.23 /5.84	51.3 /18.93	-12.45 /11.59	11.15 /4.62
Population 50	626.$\overline{6}$ /531.71	-9.77 /2.95	270.24 /192.24	-15.30 /26.45	214.08 /214.30
Population 250	831.$\overline{6}$ /139.72	-13.81 /1.38	638.5 /63.14	0 /0	439.5 /95.38

Table 3.2: Flappy Bird Population Comparison Overview

Comparison of the results In order to compare the results, the same 5 distinct values (see table 3.2) of the plot-runs where calculated, which were taken into consideration in section 3.1.
The most obvious observation is that the average fitness increase rises with the number of populations used since the generations remain low in the count when the threshold is reached. Secondly, the average distance (from the species median to the best genome run) tend to rise as well with a bigger population size.
The other measurement values leave little to no conclusions since the values don't rise or fall with growing/shrinking population sizes. The average regress, for example, tends to rise between population class 10 and 50 but is 0 in population class 250. The question is if the population class 250 would have a greater regress than the other two population instances if there would be any (future) regress. The average fitness score is non-rising/shrinking as well, however, the standard deviation seems to shrink with greater population size. Probably, the negative values of the fitness score depend on the environment of Flappy Birds, where only partially well-learned birds can make it through the first obstacle (namely the pipes).
Since there is no defined goal, the fitness-threshold can be taken into consideration when deciding how good simulations have been. Since the threshold was exceeded in every plot-run, the last generation holds the best run of a plot-run. The fewest average runs were done by population class 50 with averaged 626.$\overline{6}$ runs. However, population class 250 had the smallest standard deviation of their running length, which indicates that the goal can be more consistently reached around the 831.$\overline{6}$th run compared to population count 10 and 50.

CHAPTER 4

Comparison and Meta-Analysis

Much data was gathered and now we want to find out what this data indicates and how it can be used for future projects, maybe even real-world applications.

Ignored Parameters In order to find a straight line for analyzing the data, many parameters were set by their default value. Other parameters have abstract meanings or simply define boundaries that didn't have to be considered. For example, in the configuration file of the NEAT framework used in NEAT_FlappyBird there are 63 lines of configuration.
This complex set-up of the games leaves many wheels to turn and also give space to study their effect on the results systematically in future works. One starting-point can be the NN parameters that were pre-specified for this simulations.
Furthermore, the NEAT implementations were not equal. First of all, they were written in two different programming/scripting languages. Furthermore, MarI/O was written from scratch and NEAT_FlappyBird used a NEAT-framework that allowed a high level of set-up-configurations and even self-implementations for various aspects of the algorithm. In this application only the default implementations were used as indicated by the following config initialization:

```
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
neat.DefaultSpeciesSet, neat.DefaultStagnation, "config")
```

However, despite the different set-up and configuration some similar results could be achieved using the NEAT algorithm:

4.1 Comparison of the different game environment

However, one parameter that was manipulated for this analyses is the initial population size. In MarI/O the initial population size spawned the defined amount of species with

one genome each. In flappy bird, however, there were as many genomes launched as defined by the population size and after the run, they got assigned to their species.
In order to display the differences in an overview, a table is presented containing the data trends. Despite their similarity, different fitness-functions have been taken to evaluate the success of a genome. This results in different scales in their values. That's why a more abstract indicator of the trend was chosen instead of the numbers. An arrow up ($\uparrow$) indicates that the value was rising with a bigger initial population size and an arrow down ($\downarrow$) indicates the opposite. When there is an $\times$ shown, it means that no trend could be found within the three population classes. At first, the differences are pointed

Data Trend Comparison	avg. runs /σ	avg. fitness score /σ	avg distance /σ	avg. regress /σ	avg. fitness increase /σ
MarI/O	$\uparrow$ /$\times$	$\downarrow$ /$\downarrow$	$\uparrow$ /$\times$	$\downarrow$ /$\downarrow$	$\uparrow$ /$\times$
NEAT_FlappyBird	$\times$ /$\downarrow$	$\times$ /$\downarrow$	$\uparrow$ /$\times$	$\times$ /$\times$	$\uparrow$ /$\times$

Table 4.1: Date Trend Comparison of different games and their NEAT implementation

out which are the following: In MarI/O the average fitness score dropped with a bigger initial population. In NEAT_FlappyBird there couldn't be a correlation drawn between the population size and the average fitness score.
However, the standard deviation of the average fitness score dropped in both environments. This outcome is reasonable since there are fewer generations when the initial population size is larger.
Further, in MarI/O the average regress (if present) becomes lower with bigger population sizes, as well as it's deviation. Interestingly, in NEAT_FlappyBird no trend could be found at all since the population class had a higher average regress than the population class 10 but population class 250 didn't have any regress at all.
The standard deviation of the average fitness increase had no trend in the NEAT_FlappyBird implementation, however, it was quite similar in MarI/O's simulations. Still, the trend of the average fitness increase is the same in both simulations.

When looking at the similarities of the results of the two environments there are two trends visible: One not so obvious correlation is that the distance of the median of the species to the best run of each generation is becoming greater with a greater population count. However, the standard deviation of this data is quite high in some cases. Still, in population class 250, the standard deviation was low when compared to the average regress value in MarI/O as well as NEAT_FlappyBird.
A more reasonable trend is that the population class 250 has a bigger average fitness increase than the other two simulation classes since the number of generations are smaller, although the goal/threshold was reached as well.
When comparing the average distance with the average fitness increase, it can be seen that there are only a few genomes in the runs of population class 250 that reached a higher score, however, the majority of runs remained low. This division becomes larger the bigger the initial population size is. In MarI/O the average regress became lower with

a higher population count, which indicates a certain stability. However, this stability could not be found in Flappy Bird. A reason for this can be the known bad performance of neat in extreme situations, since Flappy Bird has a randomly generated and therefore unknown world, whereas Super Mario World is deterministic in its level behavior[KM11].

CHAPTER 5

Conclusion

"By far, the greatest danger of Artificial Intelligence is that people conclude too early that they understand it."

— Eliezer S. Yudkowsky*, (Artificial Intelligence Researcher)*

The NEAT algorithm successfully provided a solution for both environments. Still, the outcome in relation to the population size is more predictable in the rather static environment of Super Mario World when compared to Flappy Bird. As mentioned in the paper [KM11], the usual NEAT algorithm doesn't perform so well in an environment that has abrupt and unexpected changes. The concept of the game Flappy Bird might be clear, however, the generation of the world is random, which might influence the behavior of the algorithm as it can be seen in chapter 3.2. Possible future studies might show if the proposed algorithms SNAP-NEAT of the paper [KM11], enable more predictable results.

5.1 Future Work

In this work, some open questions can be found, which require future analyzes.
One of them is to see how the genomes behave directly compared to the generations when ignoring the species. How would this influence the plot?
In this work, I mostly compared the trends of the best runs of each generation. How do the average fitness increase and the average regress behave when applied to the majority of runs and not only the best runs of each generation.
In this analyzes, there were many parameters ignored as mentioned in chapter 4. How do this parameters influence the results, or do they lead to similar outcomes with different paths (details)?

As it was mentioned earlier in this conclusion, the NEAT algorithm doesn't allow a trend prediction in all cases when it comes to dynamic worlds. It would be interesting to see how the learned NEAT algorithm reacts in other levels of Super Mario World after it

completed a previous level multiple times. Would it take long to adapt to the new world or would the problems stated in the paper [KM11] come to display, since NEAT results into a form of over-fitting?
Further it would be interesting to see how other algorithms perform compared to MarI/O. For example, in an unofficial paper[1] there was a solution proposed where a lexicographic ordering was used.

[1] https://www.cs.cmu.edu/~tom7/mario/mario.pdf, last accessed on 2nd November 2018

List of Figures

List of Tables

Glossary

LUA "Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description."[2]. 5

[2] https://www.lua.org/about.html

Acronyms

AI Artificial Intelligence. 6

ANN Artificial Neuronal Network. 1

GA Genetic Algorithm. 1, 3

NEAT Neuro-Evolution of Augmenting Topologies. 1–3, 5, 6, 11–15, 17, 23

NN neuronal networks. 1, 3, 6

Bibliography

[KAV15] Ina Khandelwal, Ratnadip Adhikari, and Ghanshyam Verma. Time Series Forecasting Using Hybrid ARIMA and ANN Models Based on DWT Decomposition. *Procedia Computer Science*, 48:173–179, January 2015.

[KM11] Nate Kohl and Risto Miikkulainen. An Integrated Neuroevolutionary Approach to Reactive Control and High-level Strategy. *IEEE Transactions on Evolutionary Computation*, 2011.

[MNS+17] M. M. Mehdy, P. Y. Ng, E. F. Shair, N. I. Md Saleh, and C. Gomes. Artificial Neural Networks in Image Processing for Early Detection of Breast Cancer. *Computational and Mathematical Methods in Medicine*, 2017.

[New69] Allen Newell. Perceptrons. An Introduction to Computational Geometry. Marvin Minsky and Seymour Papert. M.I.T. Press, Cambridge, Mass., 1969. vi + 258 pp., illus. Cloth, $12; paper, $4.95. *Science*, 165(3895):780–782, August 1969.

[RT14] Sebastian Risi and Julian Togelius. Neuroevolution in Games: State of the Art and Open Challenges. *arXiv:1410.7326 [cs]*, October 2014. arXiv: 1410.7326.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, June 2002.