TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

MASTER THESIS

Investigating the Generalization Capabilities of Deep Reinforcement Learning in the Domain of Jump'n'Run Games

Author:
Johannes AUBART

Supervisor:
apl. Prof. Dr. habil.
Marcus LIWICKI

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Computer Science Department

November 29, 2017

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Declaration of Authorship

I declare that this thesis titled "Investigating the Generalization Capabilities of Deep Reinforcement Learning in the Domain of Jump'n'Run Games" and the work presented in it are my own and I have documented all sources and material used.

This thesis was not previously presented to another examination board and has not been published.

Signed:

Date:

"Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world."

Albert Einstein

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstract

Computer Science Department

Master of Science

Investigating the Generalization Capabilities of Deep Reinforcement Learning in the Domain of Jump'n'Run Games

by Johannes AUBART

Deep reinforcement learning is a state-of-the-art machine learning approach. However, the often used Arcade Learning Environment lacks the fine-tuning possibilities required to investigate the influence of small changes to a game on the learning behavior. This work presents a simple, highly customizable jump'n'run game. Several AIs are trained to play the game with deep Q-learning, using three reward functions, several training sets with varying game parameters, and two different neural networks. Some combinations produce agents capable of playing the game pretty well, while others can't beat the game consistently. The different behaviors are analyzed and ideas for improvement and future research are given.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Zusammenfassung

Fachbereich Informatik

Master of Science

Investigating the Generalization Capabilities of Deep Reinforcement Learning in the Domain of Jump'n'Run Games

von Johannes AUBART

Deep reinforcement learning ist eine State-of-the-Art Technik für maschinelles Lernen. Allerdings mangelt es der bekannten "Arcade Learning Environment", mit der KIs für Atari 2600-Spiele trainiert werden können, an Möglichkeiten, Spiele geringfügig zu verändern um die Auswirkungen dieser Änderungen auf die KI untersuchen zu können. In dieser Arbeit wird ein einfaches, auf Anpassbarkeit ausgelegtes Jump'n'Run-Spiel vorgestellt. Mehrere KIs werden mit deep Q-learning darauf trainiert, dieses Spiel zu spielen. Dafür werden drei verschiedene Belohnungsfunktionen, mehrere Sets von Trainingsdaten, die sich jeweils in spielspezifischen Parametern unterscheiden, und zwei verschiedene neuronale Netze benutzt. Einige der so trainierten KIs sind in der Lage, das Spiel gut zu spielen, während andere nur in Ausnahmefällen ein Level schaffen. Die verschiedenen Verhaltensweisen der KIs werden untersucht und Ideen zu ihrer Verbesserung und weiterer Forschung werden vorgestellt.

Acknowledgements

I wish to thank Professor Marcus Liwicki for supervising me and giving me the opportunity to write my thesis about artificial intelligence and games, a combination which fascinated me for a long time, and Jasmin Mankad for his help with understanding the Super Mario Project.

I'd further like to thank my family and friends for their support throughout the creation of this work.

Contents

Declaration of Authorship i

Abstract iii

Zusammenfassung iv

Acknowledgements v

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Outline . . . 2

2 The Game 3
- 2.1 Blocks . . . 3
  - 2.1.1 Air . . . 4
  - 2.1.2 Ground . . . 4
  - 2.1.3 Spawn . . . 4
  - 2.1.4 Goal . . . 5
  - 2.1.5 Enemy . . . 5
  - 2.1.6 Coin . . . 5
- 2.2 Movement . . . 5
  - 2.2.1 Left/Right . . . 5
  - 2.2.2 Jumping . . . 6
- 2.3 Technical Background . . . 7
  - 2.3.1 Requirements . . . 7
  - 2.3.2 AIConnector.py . . . 7
  - 2.3.3 game.py . . . 8
  - 2.3.4 paths.txt . . . 9
  - 2.3.5 worldGenerator.py . . . 10
  - 2.3.6 worldSaver.py . . . 10
  - 2.3.7 worldViewer.py . . . 10
- 2.4 The Worlds . . . 10
  - 2.4.1 World File Format . . . 11
  - 2.4.2 The World Generator . . . 12
  - 2.4.3 Restrictions and Prospects . . . 15

3 Experiments: Basic 17
- 3.1 Training Data . . . 17
- 3.2 The Network . . . 17
- 3.3 The Learning Algorithm . . . 19

3.4 Evaluation 21
3.5 The Reward Functions 22
3.5.1 Reward Function: classic 22
3.5.2 Reward Function: goaldist 24
3.5.3 Reward Function: nox 27
3.6 Comparison 29

4 Experiments: Randomized Graphics 33
4.1 Training Data 33
4.2 Network and Learning Algorithm 34
4.3 The Reward Functions 35
4.4 Comparison 35

5 Experiments: Atari 39
5.1 Training Data 39
5.2 Network and Learning Algorithm 39
5.3 The Reward Functions 40
5.4 Comparison 40

6 Experiments: Bidirectional 44
6.1 Training Data 44
6.2 Network and Learning Algorithm 44
6.3 The Reward Functions 45
6.4 Comparison 45

7 Experiments: Double Jump 49
7.1 Training Data 49
7.2 Network and Learning Algorithm 51
7.3 The Reward Functions 51
7.4 Comparison 51

8 Testing 56
8.1 Results 56

9 Future Research 62

10 Conclusion 64

A Figures from Training 65
A.1 Experiments: rndgfx 65
A.2 Experiments: atari 69
A.3 Experiments: bidirectional 72
A.4 Experiments: doublejump 75

B Figures from Testing 79

Bibliography 87

List of Figures

2.1 A screenshot from the game. 3
2.2 The default graphics. 4
2.3 The player's jump pattern. 7
2.4 Coin directly above enemy on ground level. 7

3.1 Network structure. 20
3.2 The classic loss. 23
3.3 The classic move count. 23
3.4 The achieved score for classic. 24
3.5 The classic coin count. 24
3.6 The classic level count. 24
3.7 The classic death count. 24
3.8 The goaldist loss. 25
3.9 The goaldist move count. 25
3.10 The achieved score for goaldist. 26
3.11 The goaldist coin count. 26
3.12 The goaldist level count. 26
3.13 The goaldist death count. 26
3.14 The nox loss. 28
3.15 The nox move count. 28
3.16 The achieved score for nox. 28
3.17 The nox coin count. 28
3.18 The nox level count. 28
3.19 The nox death count. 28
3.20 Averaged loss for classic, goaldist, and nox. 30
3.21 The averaged level count for classic, goaldist, nox, and random. 31
3.22 The averaged death count for classic, goaldist, nox, and random. 31
3.23 The averaged coin count for classic, goaldist, nox, and random. 32
3.24 The averaged score for classic, goaldist, nox, and random. . . 32

4.1 Averaged loss for classic, goaldist, and nox for basic and rndgfx. 36
4.2 The averaged level count for classic, goaldist, and nox for basic and rndgfx. 37
4.3 The averaged death count for classic, goaldist, and nox for basic and rndgfx. 37
4.4 The averaged coin count for classic, goaldist, and nox for basic and rndgfx. 38
4.5 The averaged score for classic, goaldist, and nox for basic and rndgfx. 38

5.1 Averaged loss comparison for basic and atari experiments. . . 40
5.2 The averaged level count comparison for basic and atari experiments. 41
5.3 The averaged death count comparison for basic and atari experiments. 42
5.4 The averaged coin count comparison for basic and atari experiments. 42
5.5 The averaged score comparison for basic and atari experiments. 43

6.1 Averaged loss comparison for basic and bidirectional experiments. 45
6.2 The averaged level count comparison for basic and bidirectional experiments. 46
6.3 The averaged death count comparison for basic and bidirectional experiments. 47
6.4 The averaged coin count comparison for basic and bidirectional experiments. 48
6.5 The averaged score comparison for basic and bidirectional experiments. 48

7.1 Averaged loss for classic, goaldist, and nox for basic and doublejump. 52
7.2 The averaged level count for classic, goaldist, and nox for basic and doublejump. 53
7.3 The averaged death count for classic, goaldist, and nox for basic and doublejump. 54
7.4 The averaged coin count for classic, goaldist, and nox for basic and doublejump. 54
7.5 The averaged score for classic, goaldist, and nox for basic and doublejump. 55

8.1 Averaged amount of beaten levels during testing on the basic test set. 59
8.2 Averaged amount of beaten levels for the basic and rndgfx agents on their respective test sets. 59
8.3 Averaged amount of beaten levels for the basic and bidirectional agents on their respective test sets. 60
8.4 Averaged amount of beaten levels for the basic and doublejump agents on their respective test sets. 61
8.5 Averaged amount of beaten levels for the basic and atari agents on the basic and long test sets. 61

A.1 The rndgfx classic loss. 65
A.2 The rndgfx classic move count. 65
A.3 The achieved score for rndgfx classic. 66
A.4 The rndgfx classic coin count. 66
A.5 The rndgfx classic level count. 66
A.6 The rndgfx classic death count. 66

A.7 The rndgfx goaldist loss. 66
A.8 The rndgfx goaldist move count. 66
A.9 The achieved score for rndgfx goaldist. 67
A.10 The rndgfx goaldist coin count. 67
A.11 The rndgfx goaldist level count. 67
A.12 The rndgfx goaldist death count. 67
A.13 The rndgfx nox loss. 67
A.14 The rndgfx nox move count. 67
A.15 The achieved score for rndgfx nox. 68
A.16 The rndgfx nox coin count. 68
A.17 The rndgfx nox level count. 68
A.18 The rndgfx nox death count. 68
A.19 The atari classic loss. 69
A.20 The atari classic move count. 69
A.21 The achieved score for atari classic. 69
A.22 The atari classic coin count. 69
A.23 The atari classic level count. 70
A.24 The atari classic death count. 70
A.25 The atari goaldist loss. 70
A.26 The atari goaldist move count. 70
A.27 The achieved score for atari goaldist. 70
A.28 The atari goaldist coin count. 70
A.29 The atari goaldist level count. 71
A.30 The atari goaldist death count. 71
A.31 The bidirectional goaldist loss. 72
A.32 The bidirectional goaldist move count. 72
A.33 The achieved score for bidirectional goaldist. 72
A.34 The bidirectional goaldist coin count. 72
A.35 The bidirectional goaldist level count. 73
A.36 The bidirectional goaldist death count. 73
A.37 The bidirectional nox loss. 73
A.38 The bidirectional nox move count. 73
A.39 The achieved score for bidirectional nox. 73
A.40 The bidirectional nox coin count. 73
A.41 The bidirectional nox level count. 74
A.42 The bidirectional nox death count. 74
A.43 The doublejump classic loss. 75
A.44 The doublejump classic move count. 75
A.45 The achieved score for doublejump classic. 75
A.46 The doublejump classic coin count. 75
A.47 The doublejump classic level count. 76
A.48 The doublejump classic death count. 76
A.49 The doublejump goaldist loss. 76
A.50 The doublejump goaldist move count. 76
A.51 The achieved score for doublejump goaldist. 76
A.52 The doublejump goaldist coin count. 76
A.53 The doublejump goaldist level count. 77

A.54 The doublejump goaldist death count. 77
A.55 The doublejump nox loss. 77
A.56 The doublejump nox move count. 77
A.57 The achieved score for doublejump nox. 77
A.58 The doublejump nox coin count. 77
A.59 The doublejump nox level count. 78
A.60 The doublejump nox death count. 78

B.1 Averaged death count on the basic test set. 79
B.2 Averaged amount of taken coins on the basic test set. 80
B.3 Averaged score on the basic test set. 80
B.4 Averaged death count for the basic and rndgfx agents on their respective test sets. 81
B.5 Averaged amount of taken coins for the basic and rndgfx agents on their respective test sets. 81
B.6 Averaged score for the basic and rndgfx agents on their respective test sets. 82
B.7 Averaged death count for the basic and bidirectional agents on their respective test sets. 82
B.8 Averaged amount of taken coins for the basic and bidirectional agents on their respective test sets. 83
B.9 Averaged score for the basic and bidirectional agents on their respective test sets. 83
B.10 Averaged death count for the basic and doublejump agents on their respective test sets. 84
B.11 Averaged amount of taken coins for the basic and doublejump agents on their respective test sets. 84
B.12 Averaged score for the basic and doublejump agents on their respective test sets. 85
B.13 Averaged death count for the basic and atari agents on the basic and long test sets. 85
B.14 Averaged amount of taken coins for the basic and atari agents on the basic and long test sets. 86
B.15 Averaged score for the basic and atari agents on the basic and long test sets. 86

List of Tables

3.1 World generator parameters for standard training data. 18
3.2 The classic reward function. 23
3.3 The goaldist reward function. 25
3.4 The nox reward function. 27

4.1 The amount of different graphic files per block type. 34

7.1 World generator parameters for doublejump training data. . . 50

8.1 World generator parameters for the different test sets. 57

List of Abbreviations

AI	**A**rtifical **I**ntelligence
ALE	**A**rcade **L**earning **E**nvironment
CSV	**C**omma-**S**eparated **V**alues
FPS	**F**rames **P**er **S**econd
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
LSTM	**L**ong **S**hort **T**erm **M**emory
MSE	**M**ean **S**quared **E**rror
RL	**R**einforcement **L**earning

Chapter 1

Introduction

In the last decades, artificial intelligence (AI) in general and neural networks in particular have been big topics in computer science research. Neural networks have been shown to be useful for a wide variety of tasks, such as optical character recognition [14], face recognition [10, 20], and speech recognition [5]. They gained further popularity in 2012, when the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) was won by the now famous "AlexNet", with a remarkable lead before the second place in terms of error rate [9, 15]. Since then, many different neural networks were used successfully in the ILSVRC, further lowering the error rate [15, 18]. In 2017's contest, the winners from all categories were based on neural networks, although ensembles or combinations with other technologies were used instead of pure neural networks [15].

One technology that has been combined with neural networks successfully is *reinforcement learning* (RL). In 2013, Mnih et al. proposed a new deep learning model called *deep Q-learning*, that uses RL with a deep neural network as a function approximator to estimate the maximum achievable reward [12]. Using this method, their agent learns to play seven *Atari 2600* games provided by the Arcade Learning Environment (ALE) [2, 12]. By now, the method has been applied to 49 games from the ALE and the agent was able to reach a superhuman level in 29 of them [11]. Using policy distillation, it was also possible to train one single network to play ten different Atari games [16]. Recently, *AlphaGo* has beaten the world champion of Go [4] using a combination of tree search, deep neural networks, and reinforcement learning [17] - a challenge that has been thought to be impossible to beat due to the incredible large state space of Go.

Games - digital as well as physical ones - are useful for exploring the power of an AI. They are usually designed to be challenging for human players and they provide a large diversity of challenges. While the successes mentioned above show that RL in combination with deep neural networks is indeed powerful, the games in the ALE lack the ability to be fine-tuned to better investigate the capabilities of deep reinforcement learning. To cope with this, this thesis introduces a simple jump'n'run game that is highly customizable and easy to modify. Using deep Q-learning [7, 12], several versions of an AI are trained on this game. The versions differ in either changes to the game,

the network, or the training parameters. The performance of each version of the AI is evaluated and compared to other versions to better understand the capabilities of deep reinforcement learning.

1.1 Contributions

This thesis has two main goals. The first one is to provide a new learning environment for AI in form of a jump'n'run game. The ALE offers a lot of different games, but lacks flexibility regarding the fine-tuning, as the games cannot be modified easily. The game proposed in this work isn't as versatile regarding genres as the games of the ALE, but new levels can be created easily with just a text editor. The layout of the worlds has a huge influence on the difficulty, so the game is fitted to test AI agents of different power on it. The interface between the game and the AI is reduced to a single Python file, making it possible to exchange the AI without much rewriting of code. Additionally, the game is kept simple to allow for easy code modification.

The second main goal is to investigate the capabilities of deep reinforcement learning. To do so, multiple agents are trained to play the game using the deep Q-learning algorithm from [12]. The agents are trained using different neural networks, game levels, and reward functions and compared to each other afterwards. This provides interesting results on how each of these changes affects the performance of the AI and additionally proves the game's ability to serve as a flexible learning environment.

The results will show that already small changes to the worlds or the reward function can have a huge impact on the agent's performance. Using only the limited options of the world generator, it is possible to create worlds which can be learned to play easily by the AI, as well as worlds which significantly increase the difficulty of this task. Statistics logged directly by the game show good agents and bad agents and provide interesting ideas for future research in the field of deep reinforcement learning. It can thus be concluded that both goals are met.

1.2 Outline

Chapter 2 explains the game created for this thesis. It mentions the aspects and features of the game and provides a documentation and technical information necessary to run the game and plug an AI into it. Chapters 3 to 7 describe the conducted experiments and their results. In Chapter 8, the agents trained in the experiment chapters are tested on independent test sets and again compared to each other. This thesis then finishes with some thoughts on future research (Chapter 9) and a conclusion (Chapter 10).

Chapter 2

The Game

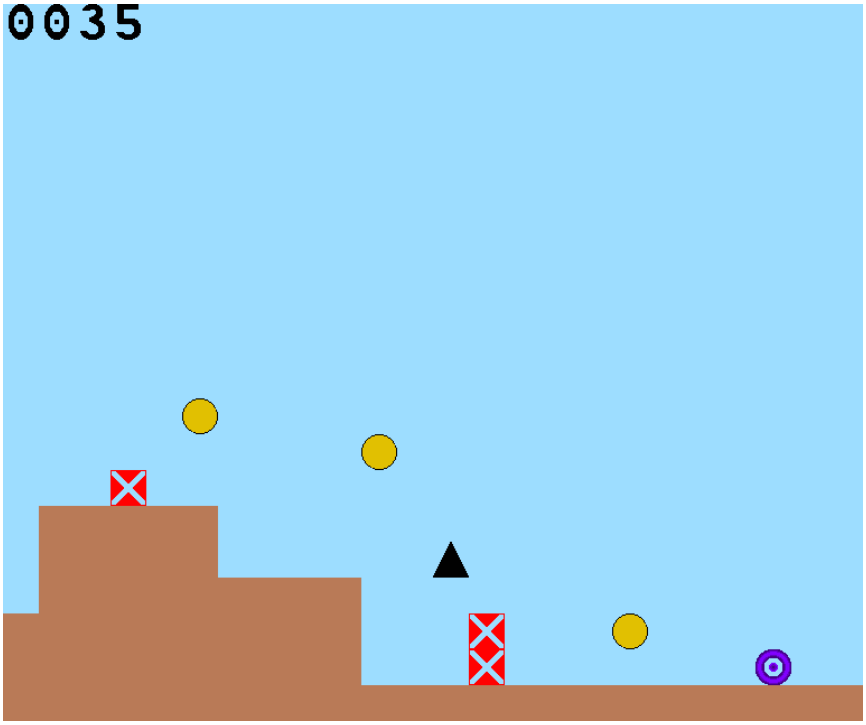

FIGURE 2.1: A screenshot from the game.

This chapter describes the game that was used for the experiments. It has been created with the purpose of beeing able to simulate several different but similar games while still being simple. The game is basically a platformer jump'n'run game: the world is shown from a side view and the player mainly moves horizontally, similar to usual Super Mario levels.

2.1 Blocks

Each level (also called "world") is built from blocks. Every block has the same size of 32x32 pixels, although the game could be easily adapted to support other block sizes. There has to be a block at every position of the world and no game object will ever be between two blocks. By default, a world consists of the following blocks:

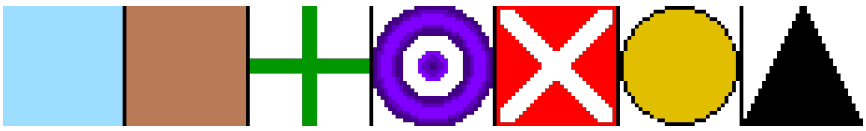

FIGURE 2.2: The default graphics for the blocks, separated by black lines. From left to right: air, ground, spawn, goal, enemy, coin, and the default player graphic. Each graphic is 32x32 pixel and everything that appears white here is actually transparent.

air the default background block

ground the player walks on ground blocks

spawn where the player spawns when the world is loaded

goal the player has to reach this block to finish the level

enemy the player will die if it runs into this block

coin can be picked up by the player to increase the score

At every position in the world, there is exactly one block, although the graphics of the *air* block is drawn behind every other block except for *ground* to allow for partly transparent block graphics.

The player may be on the same position as any block, although game mechanics make it impossible to be "inside" *ground* blocks or reach *air* blocks further away from the ground than the player can jump.

2.1.1 Air

Simply put, *air* is where no other block is. This block is a filler block without any special effect.

2.1.2 Ground

This is the block the player walks on. While not jumping or falling, the player will usually be at the position of an *air* block directly above a *ground* block.

2.1.3 Spawn

The *spawn* block marks the position the player will spawn when the world is loaded or after it died. There is not really a need for this block to be visible, although it is by default.

2.1.4 Goal

The goal of the game is to reach the *goal* block. If the player hits it, the score will increase (by 100 by default) and a new world will be loaded. This block is usually located at the opposing end of the world regarding the *spawn*.

2.1.5 Enemy

The *enemy* block serves as an obstacle the player has to jump over. If the player runs into an *enemy*, the player will die causing it to respawn at the *spawn* and the score will decrease by a set amount (10 by default). In the current implementation, *enemies* are always stationary and located directly above *ground* or *enemy* blocks.

2.1.6 Coin

Coins can be collected by the player to increase the score (by 5 by default). Upon hitting a *coin*, the player is rewarded the score increase and the block is replaced with an *air* block. *Coins* are only created when the world is loaded - if the player respawns due to death, only the *coins* that were not collected until then will be there.

2.2 Movement

When played by a human, the game is controlled with the arrow keys: left arrow key for movement to the left, right arrow key for movement to the right, and up arrow key for jumping. When run in AI mode, the AI can choose between the actions *left*, *right*, *jump*, and *no action* (see 2.3 for further information on the AI).

2.2.1 Left/Right

Moving to the left and right works as intuitively expected: pressing the corresponding key (or choosing the action in case of AI mode) moves the player one block in that direction. The game tries to keep the player in the center of the screen. While in the middle of a world, instead of the player moving left or right, it will actually stay at its position and the world will move in the opposite direction. To prevent showing things "outside" the world, this behavior will stop at the world's borders and then just move the player instead. This applies not only to horizontal movement, but also vertical one. The latter one is not visible in the experiments, because all worlds that were

used are exactly as high as the game window can show at once, so no vertical world movement will occur.

2.2.2 Jumping

The jumping behavior of the player is controlled by three variables which are either specified in the world file or set to default values:

- jump

  This variable specifies how often the player can jump before it needs to touch ground again. While this might sound confusing compared to real-world physics, the concept of *double jump* is quite common in jump'n'run-like games. The game is not restricted to single or double jump, any arbitrary positive integer could be used here. A value of 0 is rarely useful, as it renders the player unable to jump at all, thus preventing it from ascending height levels as well as jumping over enemies. It is not required for the player to be on ground when jumping - if it is falling (either from higher ground or a previous jump) and hasn't used all of its jump charges, it is able to jump midair. The default value is 1.

- jump_height

  This variable defaults to 3 and specifies how many blocks the player is actually moved upwards while jumping.

- jump_width

  When the jump action is executed, the player is moved upwards one block per frame until it has ascended *jump_height* amount of blocks. Then it will "hover" at this height for a few frames before gravity will kick in and move it downwards again. *jump_width* controls how long the player will hover at the turning point (default value of 3). The name is a little confusing as the maximum horizontal distance that the player can travel while in air is actually larger than this value, because the player can move horizontally while ascending or descending, not only while hovering. It can't move sideways during the first upward movement of a jump, however, since it can only take one action per frame and needs the *jump* action to initiate the jump. See Figure 2.3 for a visualization of the jumping behavior.

This jumping behavior feels somewhat realistic while being quite simple at the same time.

If multiple jumps are allowed, an additional jump action can be taken at any time during the previous jump. This will immediately initiate a new jump independent of the current jump's state - this means with double jump and a *jump_height* of 3, issuing a second jump directly after the first one (player has only ascended 1 block so far) will cause the player to ascend only a total of 4 blocks, not 6.

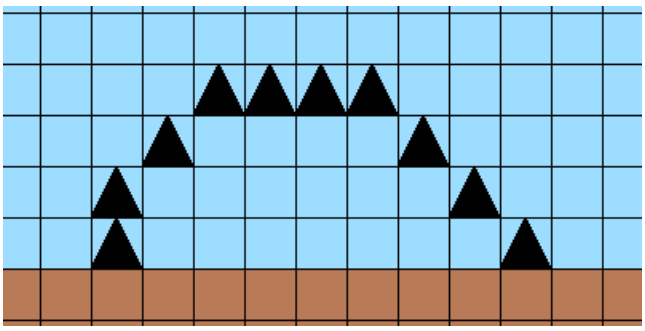

FIGURE 2.3: Jump pattern for the player (from left to right) for jump_ height=3 and jump_ width=3. This pattern is produced by a single jump action followed by consecutive right actions. Raster added for better understanding.

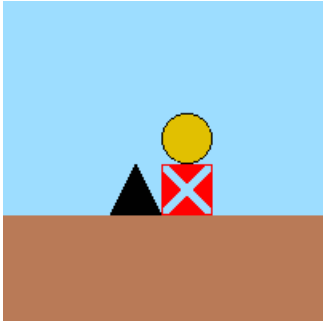

FIGURE 2.4: Coin directly above enemy on ground level. This coin can only be taken while descending, as the player can't jump and move right in the same action.

A side effect of the inability to initiate a jump and move horizontally at the same time is that coins which are one block above the player as well as one block to the left/right can only be collected during the descending part of the jump (see Figure 2.4).

2.3 Technical Background

This section focuses on the Python files belonging to the game and other issues necessary to reproduce the experiments or to use the game for further research. It shortly explains the important Python modules and functions and mentions some technical details about the implementation. The neural network (or the AI, respectively) is not part of this section, as that can be exchanged and is described in the experiment chapter (Section 3.2).

2.3.1 Requirements

The game requires *Python 3.5* and the *Pygame* library (version 1.9.3 is used here). Further requirements may be needed to attach the neural network - the Lua code used in the experiments is hooked in using the *Lunatic Python* fork from Bastian Bechtold [1, 13].

2.3.2 AIConnector.py

In order to make it as easy as possible to exchange the AI for the game, communication between game and AI was reduced to one interface module - the AI connector. This file needs to be adapted when the underlying AI is

changed. While ideally only this file needs to be changed to plug in a new AI, the disadvantage of this design is that every communication between game and AI has to be routed over the AI connector, which creates a slight overhead of primitive functions that mostly just transfer information. The most important function is *getAction()*, which prompts the AI to compute the next action the player should take and returns this decision to the game. The AI is only called when the player has an action to take, not in between. This provides a low interlocking between game and AI but has some side effects. For example, when the player reaches the goal, it is rewarded a large score increase and then a new level is loaded. Since there is no action to take in between these two steps, the AI will not notice the score increase - when it is called again, the new world has been loaded and the score has been reset. Similarly, when the player dies, the game will show a black screen for a short time, which is also not forwarded to the AI. To provide game-relevant values (e.g. score), the *setParams(params)* function allows to pass a Python dictionary to the AI. In the given implementation, the game forwards the score, x position of the player, distance to the goal, amount of deaths, amount of beaten levels, the name of the world, and a value containing information about the player's last few movements in x direction to the AI using this function.

2.3.3 game.py

This is the main module of the game and also the one that has to be started to run the game. It has many functions and a lot of global variables and explaining all of them would go beyond the scope of this section. However, the file is well commented, making all functions (mostly forming the blocks from which the final game loop is built) and most of the variables (primarily keeping track of the game state) quite self-explanatory, so this section will focus on the important variables that can be used to modify the game.

- WORLD

  This stores the world, as a list of lists of block ids. It is created during the init_world function. Since the world is saved in the file in a row-wise manner and also loaded in this way, the first dimension specifies the y coordinate and the second one the x coordinate - this is different from the other variables coping with positions, where the first value or dimension usually belongs to the x axis.

- SCREEN_RESOLUTION

  This is where the resolution of the game window can be set. It is advised to choose values that are a multiple of the block size, so there won't be any "half" blocks on the screen.

- MODE

  This can be either 0 ("AI mode") or 1 ("manual mode") and determines whether the game is played by a human or by an AI. A value of 0 will

load the *AIConnector* module and have the AI choose the actions. A value of 1 will disable the AI and let a human play. Setting this to 0 will not disable the arrow keys - pressing a key will overwrite the action chosen by the AI. It is not recommended to make use of this feature though, as this would interfere with the training of the AI.

- FPS

  Frames per second. Determines how often the main game loop is executed and thus the overall speed of the game. Values between 10 and 15 make the game playable for human players. Because the game waits for the AI to choose an action, the AI has no problems regarding reaction time and the maximum game speed is already limited by the computing power of the machine. Setting the FPS to 0 will remove the limit and make the game run as fast as possible.

- MAX_TRAINED_FRAMES

  If the game runs in AI mode and this value is greater than 0, the game will stop after that amount of actions. This was implemented to simplify training several AIs for the exact same amount of frames for better comparability.

- SCREENSHOT_...

  SCREENSHOT_DIRECTORY specifies where screenshots are stored, but this should be set in the *paths.txt* file (see 2.3.4).

  SCREENSHOT_CURRENT_SIZE and SCREENSHOT_OLD_SIZE specify the resolution to use for the screenshot of the current frame and the older frames respectively. The image will be scaled to this value (possibly distorting it).

  SCREENSHOT_OLD_COUNT contains the length of the screenshot history - how many older screenshots should be saved. The screenshot history also contains the current frame: a value of 0 will only save the current frame in the specified resolution, a value of 1 will save the current frame two times (once in the "current" resolution and once in the "old" resolution), and greater values will additionally save screenshots of previous frames.

  SCREENSHOTS_ACTIVE can be set to 0 to completely deactivate taking screenshots.

2.3.4 paths.txt

To make it easier to run multiple instances of the game (training different networks, for example) at the same time without always having to change *game.py*, some options have been moved into a separate file. This file is expected to contain the following information in that order:

- Path to the level folder. The game will search for world files in this folder.

- Pattern for the world name. Instead of loading all world files in a certain order, the game first filters all files in the specified folder (see above) according to this pattern (using python's *fnmatch* module) and then randomly chooses one. A specific world can be loaded by giving the exact name of the world file here.

- Path to the statistics file. The game logs several values for the purpose of evaluating an AI's performance and dumps these values into a csv file. The given path should contain the file extension. The file will be created if it doesn't exist and it will be appended to otherwise.

- Path to the screenshot folder. If taking screenshots is activated, they will be stored in this folder (naming of the screenshot files is still specified in *game.py*). The folder will be created if it doesn't exist, its parent folders won't though.

2.3.5 worldGenerator.py

Due to its importance, this module is described thoroughly in Section 2.4.

2.3.6 worldSaver.py

This module handles the saving and loading of worlds. The format the worlds are saved in is explained in Section 2.4.

2.3.7 worldViewer.py

An auxiliary tool that loads a world from a file (using the *worldSaver* module) and draws it on the screen in the same way the game does. The arrow keys can be used to move the world. It is basically a preview program for world files.

2.4 The Worlds

The game was designed to simulate many different (although somewhat similar) games. To achieve this, the game was made customizable through the levels. While it is easily possible to design worlds by hand with nothing but a text editor (see Section 2.4.1), the amount of levels needed to properly train a neural network without overfitting makes this option inapplicable. The *world generator* is the solution to this problem: it can generate arbitrary amounts of worlds and has many parameters to control what the generated worlds look

like. This section explains the world format as well as the possibilities and parameters of the world generator.

2.4.1 World File Format

In order to make worlds human readable, they are saved in simple text files using a easy to understand format. The file consists of several sections, each started by a line containing an "!" followed by the name of the section. The world loader (in *worldSaver.py*) interprets lines differently depending on last seen section indicator. All lines above the first section indicator are ignored, as well as all lines starting with an "#" to allow for commenting.

A mandatory section is the "blocks" section. It maps block names used by the game - see Section 2.1 for details on the block types - to ids used in the world. In addition to the standard blocks that have to appear here, a value for "player" can be specified. This value shouldn't be used in the "world" section, but it can be used in the "blockgfx" section to define a graphics file for the player. An arbitrary amount of blocks can be defined here, although all of the non-standard blocks will behave similar to *air* - a background block without any effect - unless *game.py* is adapted to implement the desired effect.

The other mandatory section is the "world" section. Every line in this section stands for one row of blocks in the game, each block specified by its id as defined in the "blocks" section. The block ids are separated by a space and all lines need to have the same amount of ids in it. A world needs exactly one *spawn* and one *goal* and has to be at least big enough to fill one screen of the resolution set in *game.py*.

The "blockgfx" section can be used to specify graphic files for the blocks. The blocks are identified by their id set in the "blocks" section. The paths may be relative to the execution path of the game and have to specify the file extension. If not specified, the game searches for the default graphics. For every block without a graphics file set here, the game loads "gfx/<block name>/primitive.png". The block names are as mentioned in section 2.1 and "player" for the player graphics, so for example the default graphic file for coins is at "gfx/coin/primitive.png".

Every information that can be saved in a world file but doesn't belong in one of the other sections can be put in the "*info*" section. The game reads the following parameters frome here:

- blocksize

  Width and height of a block. This is to be handled with caution, as the game expects all blocks to have exactly this size. Also, the resolution of the game window should be set to a multiple of this value. It defaults to 32 if not specified.

- jump, jump_height, jump_width

These variables control the jumping behavior of the player. See Section 2.2.2 for details.

- starting_score

  With what score the player will start this world. 0 by default.

- coin_worth

  How many points will be rewarded for taking a coin. The default value is 5.

- death_worth

  How many points the player will get for dying. This is supposed to be a negative value (otherwise the player will get points for dying) and defaults to -10.

- goal_worth

  The score reward for hitting the goal. 100 by default.

- score_position

  Where the score will be displayed. -1 for upper left corner, 0 for upper center, and 1 for upper right corner. The default is -1.

Only the sections "blocks" and "world" are mandatory. The order of the sections is irrelevant, but there has to be a line break after the last line of the file. All assignments are expected to have the form "<key>=<value>" (without spaces), so for example to enable double jump there should be a line "jump=2" in the "info" section.

2.4.2 The World Generator

The world generator creates worlds in several steps. At first, the ground is generated. Currently the world generator can create ground only as a continuous mass growing from the bottom - there will never be a ground block above a non-ground block. The terrain height of the first row is chosen randomly between the minimum height of 1 block and the maximum height specified as parameter. After that, the height of the next column is generated based on the height of the current column.
Relevant parameters for ground generation are:

- MAX_HEIGHT_DIFF

  The maximum amount of blocks the terrain height can change within two neighboring columns. Setting this to 0 will make height changes impossible and values greater than the maximum height the player can jump could create unbeatable worlds.

- MAX_HEIGHT_TERRAIN

  The upper bound for terrain height. There won't be any ground blocks above this height. Without this boundary, terrain could be as high as the level, blocking the player from crossing it. Furthermore, worlds with more air than ground feel more intuitive for human players, although this is not relevant for the AI. This specifies a position in an array, therefore there can be one block of ground more than specified by this value due to the array counting offset.

- HEIGHT_CHANGE_PROBABILITY

  The probability that the height will change at all. Setting this to 0 will create flat levels while a value of 1 will generate height changes at nearly every column. The generator will try to change the height with every column, but this height change might be reduced or completely negated if it would violate the maximum/minimum height boundaries of the world.

- HEIGHT_UP_PROBABILITY

  The probability for a height change to go upwards instead of downwards. Values other than 0.5 cause an increase (or decrease, respectively) of the averaged terrain level towards the right side of the world. This will probably create plateaus if the upper or lower height boundary is reached.

- HEIGHT_DOWN_FACTOR

  This value will be multiplied by the current terrain height and subtracted from the HEIGHT_UP_PROBABILITY for deciding the direction of a height change. It can be used to reduce the probability of plateaus at high terrain levels.

After the ground has been generated, spawn and goal are placed in the world. Both are positioned directly above a ground point and at opposing ends of the level.
Parameters controlling the spawn and goal placement:

- SPAWN_AREA_PERCENTAGE

  This value specifies in what percentage of the world (measured from the ends) spawn and goal will be placed. So, for example, a setting this to 0.2 means that spawn or goal will be positioned within the leftmost 20% of the world while the other one will be placed within the rightmost 20%. A small enough value (or 0) forces spawn and goal to be at the first and last column. When set to 1, spawn and goal can be placed anywhere in the world, including next to each other in the middle of the world.

- SPAWN_LEFT_PROBABILITY

  The probability of having the spawn left of the goal as opposed to the other way around. When set to 1, the spawn will always be left and

the goal will always be right. For 0, it's the other way around. A value of 0.5 will generate levels with an equal chance for both variants. If SPAWN_AREA_PERCENTAGE is greater than 0.5, the areas in which spawn and goal can be placed overlap, which impairs the influence of this variable.

Next, the enemies will be generated. Enemies are only placed on the ground or on each other. If stacked on each other, they are seen as one "taller" enemy. There are several constraints preventing the generator from creating enemies that cannot be jumped over.

- ENEMY_SPAWN_PROBABILITY

  On what percentage of valid positions enemies should be placed. Set this to 0 to create worlds without enemies. This variable only influences enemies placed directly on a ground block, so setting it to 1 will create as many enemies as possible, but the height of these enemies - whether enemies will be stacked on each other - is controlled by other parameters.

- ENEMY_MAX_HEIGHT

  The maximum height of enemies. If it is higher than the player can jump, generated levels could be impossible to beat.

- ENEMY_GROW_PROBABILITY

  The probability with which an enemy will grow one block taller. Setting this to 1 will cause all enemies to have the maximum height possible at their respective position while a value of 0 will create no enemies taller than one block.

- MAX_HEIGHT_DIFF

  An enemy that has the same height the player is able to jump can be jumped over - if it is on even ground. Enemies on terrain "peaks" could make the level unbeatable. Thus, MAX_HEIGHT_DIFF not only influences the ground generation, but also restricts placement and height of enemies.

Last, the coins will be generated. The idea here is that a coin can be created at any position that is currently *air* and can be reached by the player. The last part is somewhat problematic: which positions the player can theoretically reach is highly dependent on the jumping variables, which are tried to be kept separated from world generation. Thus coin generation uses MAX_HEIGHT_DIFF instead, but this renders exact computation of reachable blocks impossible.

The coin generation takes the maximum terrain height of the current column and the neighboring ones. A block is considered to be a possible coin position if it is not further from this height level than MAX_HEIGHT_DIFF + 1 (the player can reach this height as it can jump over obstacles of height MAX_HEIGHT_DIFF). Columns where something is above the ground - an

enemy, the spawn, or the goal - are ignored for this computation, since the player can't jump from that place if an enemy or the goal are there. This means that there won't be any coins in columns without "free" ground in either this or one of the neighboring columns. That's a simplification for the sake of complexity - there are most probably some reachable positions in this columns, otherwise the player wouldn't be able to cross it. Also, there might be coins that the player can't take directly as they are too high, but has to jump up a terrain elevation and then jump back from there to get it. This generation uses the assumption that the player can reach coins that are up to MAX_HEIGHT_DIFF + 1 higher than the current ground and one block to the side. If the *jump height* is set to MAX_HEIGHT_DIFF and *jump width* is set to 0 (see Section 2.2.2 on jumping), the player will be able to reach the necessary height - because jumping measures from the player's position which is one block higher than the ground, so the "+ 1" is not needed here - it can't move sideways on that height level though, as it will start descending immediately, thus some coins might be unreachable with this configuration.

- MAX_HEIGHT_DIFF

  Used as an indicator of how high the player can jump, see above.

- COIN_SPAWN_PROBABILITY

  The probability with which a coin will be generated at a suitable position. Setting it to 0 will create worlds without coins while a value of 1 will generate worlds with coins at every position meeting the requirements explained above.

2.4.3 Restrictions and Prospects

While the game's possibilities are enough for this thesis, there are many more features that could be added to enhance it for future work.

First of all, it should be mentioned that the world generator is somewhat limited compared to the potential of the game itself. Currently, it can only generate horizontally oriented levels where the terrain is one big mass at the lower half while the upper half of the level is more or less empty. Some kind of "floating" blocks could add a whole level of complexity: worlds could have a second "layer" of floating blocks the player can walk on. There could be a higher concentration of coins on the upper layer and the player can go there to gather points. Or the goal is on the upper layer so the player has to go there. To increase the difficulty, the floating blocks could be too high to jump to for most of the world, leaving only a few transition points where the player can actually reach the upper layer. This means that it would have to actually go back to one of these transitions if it falls down. Another challenge for the AI could be to distinguish possible and impossible paths: the world could have two layers, but one layer is always blocked, for example by an enemy that is too high to be jumped over, forcing the AI to detect which path is actually possible. Vertically oriented worlds where the player starts at the bottom

and has to go to the top are also possible. The game also supports "abysses", meaning the player can fall off the world (causing it to die), although the world generator currently doesn't use this feature. Using abysses or vertical orientation, worlds can be created where the main difficulty are not enemies, but that are more like "jumping puzzles" and falling down resets the player to an earlier part of the level. It is also possible to create huge worlds where the spawn is somewhere in the center and the main difficulty is to locate the goal.

For more options, it is also possible to modify the game itself. For example, new blocks could be added. Examples could be different enemy types (moving enemies, destroyable enemies, ...), jumping pads, power-ups, blocks that are destroyed after walking on them, checkpoints that become the new spawn after walked on once, and many more. Another option to enhance the game could be alternative game modes: instead of just having to reach the goal, there could be a time limit, or instead the player has to collect all coins or destroy all enemies to beat the level. Most of these enhancements should be relatively easy to implement due to the simplistic nature of the game.

Lastly, the game was created with the purpose of serving as an environment for training and testing AIs. Although it is possible to play it via keyboard, this is not the intended use and the game lacks several aspects that would make it enjoyable for human players. The controls feel really clunky, sometimes pressed keys aren't recognized, and there is no sound (it is usually not forwarded to the AI and thus not needed). The *manual mode* should therefore be seen as a kind of debugging feature and not as an alternative game mode.

Chapter 3

Experiments: Basic

The experiments described in this chapter were conducted using a very simple version of the game: single jump, default graphics only, the score is always displayed in the upper left corner, and the goal is always on the right side of the world.

These experiments will be called *basic* experiments in the following chapters.

3.1 Training Data

The training set consists of 1200 levels that were created with the world generator. 200 levels were created with "standard" parameters that lead to levels which were thought to be interesting for novice and advanced beginner human players. Expert players would probably be bored soon, as the levels aren't too hard. While it would be possible to train an AI on these levels, it would probably take a long time before the AI would reach the goal for the first time. To make it easier for the AI to reach the goal and learn about its relevance, another 1000 worlds were created with enhanced parameters: some of the parameters were not set to standard values, but instead chosen randomly from an interval ranging from the standard value (roughly) to a value likely to create an easier world. These worlds tend to be shorter and flatter, and to have fewer enemies. Table 3.1 shows the parameters used for world generation. Refer to Section 2.4 for an explanation of the world generator and its parameters. All worlds were created with single jump, jump width and height of 3, and the score in the upper left corner.

3.2 The Network

The neural network used is an adaptation of the network from [7]. It was successfully trained to play the first level of Super Mario from the SNES console in the original project and since Super Mario is quite similar to this work's game in many aspects, the network appeared to be a good choice. The network consists of three separate paths that are later merged. The first path is

TABLE 3.1: World generator parameters for standard training data. Intervals in the *Enhanced value* column denote the range the value is randomly chosen from, empty entries mean no change from the standard value.

Parameter	**Standard value (200 worlds)**	**Enhanced value (1000 worlds)**
jump	1	
jump_height	3	
jump_width	3	
score_position	-1	
world_ size_ x	75	[50, 75]
world_ size_ y	20	
max_ height_ diff	3	[2, 3]
max_ height_ terrain	10	
height_ change_ probability	0.2	(0.1, 0.2]
height_ up_ probability	0.5	
height_ down_ factor	0.01	[0.01, 0.1)
enemy_ spawn_ probability	0.15	[0.1, 0.15)
enemy_ max_ width	1	
enemy_ max_ height	3	
enemy_ grow_ probability	0.25	[0.1, 0.25)
coin_ spawn_ probability	0.05	
spawn_ area_ percentage	0.2	
spawn_ left_ probability	1.0	

an action history that gets the last four chosen actions (as one-hot-vectors) as input. The second path is a screenshot history with a tensor containing the last four screenshots (greyscale 32x32 pixel) as input. The third path takes the current frame's screenshot (greyscale 64x64 pixel) as input and contains - among several convolutional layers and one max-pooling layer - a spatial transformer network based on [6].

Figure 3.1 shows the original network. It was only slightly changed for this project: the agent can choose exactly one of four actions each frame. The actions are *left*, *right*, *jump*, and *no action*. This is different from the Super Mario project, where the agent could choose one action button and one arrow button at a time. So, for this project, the amount of output neurons was changed from eight to four and the action history branch gets one-hot-vectors of size four instead of two-hot-vectors of size eight. Apart from that, the network is unchanged.

3.3 The Learning Algorithm

The network is trained using the algorithm from [7], which is an implementation of *deep Q-learning* [12], including experience replay and minibatch training. Again, a few changes were made to adapt the code to this project.

There are several aspects that could be investigated to evaluate the capabilities of deep reinforcement learning. One of these is the input to the neural network. This project uses only the pixel values as input for the neural network (and the last chosen actions, which are no new information and could also be directly remembered by the network). Giving game specific information - e.g. the player position - directly to the network would probably increase performance. However, as deep reinforcement learning has already been shown to be able to learn from pixel values only [11, 12], which is a more complex problem, giving the network more information is not really interesting for a capability study. The network itself could be changed too. Chapter 5 uses a different network for comparison. However, due to the vast amount of possible neural networks - only limited by computational feasibility - and their complexity as well as the time it takes to properly train a network, this thesis focuses on other aspects of deep reinforcement learning. There are many hyperparameters in deep Q learning, for example the ϵ for the ϵ-greedy policy, the learning rate, the gamma value, and many more. Most of these are fine-tuning tools that can improve the performance of an agent, but won't help much if e.g. a bad network is used. Instead, the experiments in this and the following chapters focus on the effects of changes to the reward function and to the game. The reward function gives feedback to the AI on how good a decision was. Ideally, the AI receives only the same feedback a human player would get - that is positive rewards for score increase and finishing levels and negative rewards for score decrease and deaths. This means that to successfully play the game, the AI needs to figure out that there is a goal reachable through movement actions in the first place,

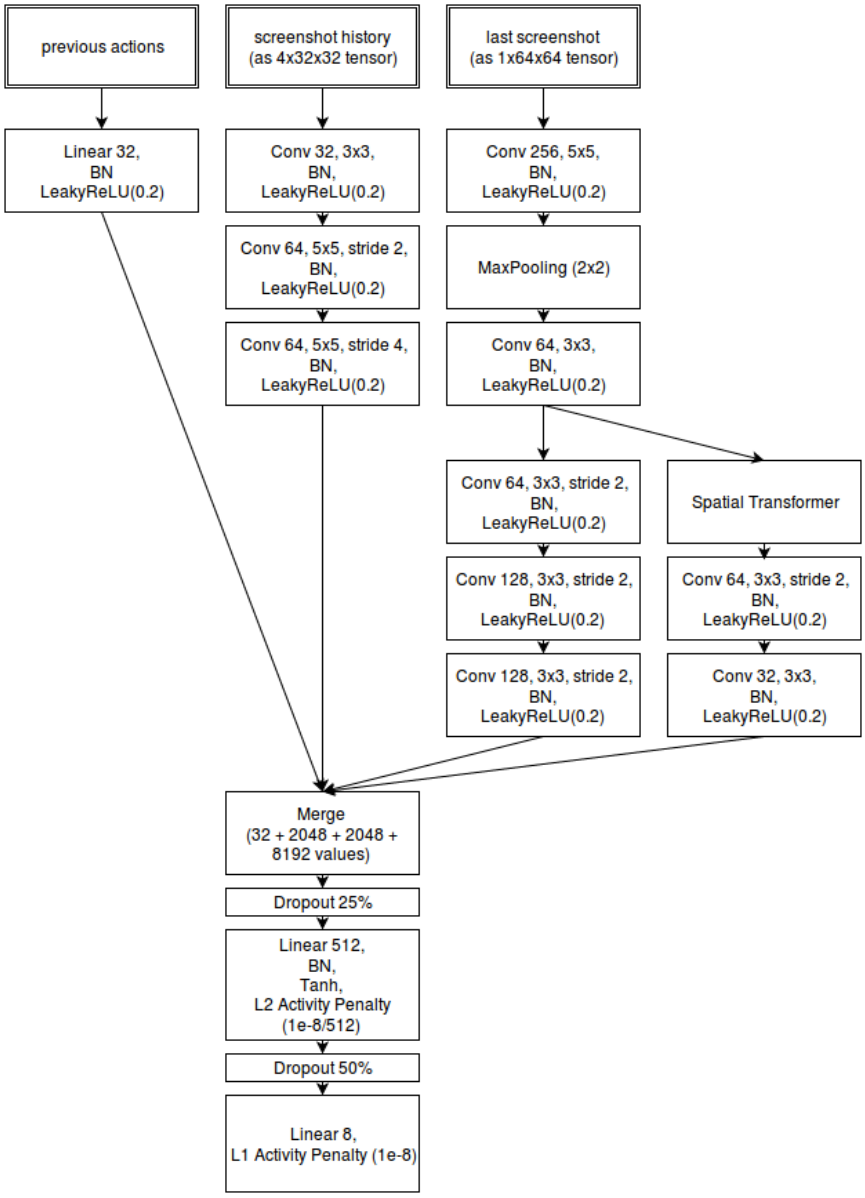

FIGURE 3.1: Structure of the original network, taken from [7].

which is quite hard. To make the task easier for the AI, the reward function could include rewards for moving in the correct direction. These changes to the reward function can have a huge impact on the performance of the AI. As mentioned in Chapter 2, the game is quite flexible and can be modified in different ways that also influence the task of learning. The amount of enemies, position of the goal, existence of dead ends, length of the level, jumping behavior, and used graphics also strongly influence the difficulty of the task, to mention only some of the game's possibilities.

Most of the hyperparameters were set to the same values as in [7], because they worked for Super Mario and were expected to also work for this project. ϵ decays from 0.8 to 0.1 over 400,000 actions, the learning rate is 0.0001, *gamma* is 0.9. All games are simulated for 1,000,000 actions total. A new, randomly chosen level is loaded every 1000 actions and when the current level is beaten. The 1000 actions are accumulated over the levels, so if a level is beaten after 920 actions, the AI will only have 80 actions in the next level before another world is loaded. A death will cause the player to respawn at the spawn position and lose points. Coins will not respawn when the player dies. States are saved to the replay memory every 1000 actions and training occurs every 5000 actions. The batch size is 16.

In [7], a selective mean squared error (MSE) loss function is used, that only propagates gradients of the actions that were chosen and sets all other gradients to 0, because the actual reward is known only for the chosen action. The target reward is used as an indicator which action was chosen. This works for the Super Mario project, as the reward function is designed to be (almost) never exactly 0. This thesis, however, uses different reward functions which will return 0 in many cases. As the experiments show, it is still possible to train an agent capable of playing the game. The loss function hasn't been further investigated therefore, but future research might want to have an eye on this.

3.4 Evaluation

The next section presents the trained agents using several statistics that are logged by either the AI or the game itself. For a better understanding of the graphs depicted in this chapter and in the following ones, the different values that they show are shortly explained in this section.

The *loss* is the result of the MSE function and logged by the AI during training. It measures the distance between the network's reward predictions and the actual rewards. Training the network aims at minimizing the loss, so small values are better here and the loss typically decreases over time.

The *move count* shows how often the AI has chosen which action. For all experiments except the ones described in Chapter 6, the goal is always to the right side of the spawn, so the AI is supposed to take mainly *right* and *jump* actions.

Level count is the amount of levels the AI has beaten. A level is considered beaten if the AI manages to move the player on the same position as the goal. As reaching the goal is the main task of the game, this is probably the most important performance measure.

Similarly, the *coin count* is the amount of coins collected by the AI. To collect a coin, the AI has to be on the same position as the coin after it has performed its action. The coin will then disappear and not respawn if the player dies, so each coin can only be collected once between first spawning in a world and reaching its goal.

The *death count* is the number of times the player died. As the training and test sets do not feature any abysses where the player could die by falling out of the world, the only way to die in the experiments is to hit an enemy. Analogous to coins and goal, an enemy is hit when the player is on the same position as the enemy. Deaths are penalized by all reward functions and the AI is expected to learn to avoid dying.

Deaths, collected coins, and beaten levels are all merged together in the *score*. Dying will cause the player to lose points while collecting coins and reaching the goal will increase the score. The score is only logged when a new world is loaded and it is set to 0 directly afterwards. As negative values are not allowed, dying while having a score of 0 won't change it. Furthermore, since a new world is loaded when the goal is reached, points granted for beating the world will be immediately logged and then disappear due to the score reset, so the player can't lose these points through dying. Due to this and the comparatively high point reward for beating a level, the score graph usually looks quite similar to the one for the level count belonging to the same experiment.

The graphs show the actual values, as well as an iteratively computed average. There are two exceptions to this in order to preserve readability: the move count graphs don't show an average and when data from multiple experiments is plotted into one graph for comparison, only the averages are shown. There are no comparing move count graphs. All statistics except for the loss are logged by the game itself and their graphs are normalized to show one data point per 1000 frames.

3.5 The Reward Functions

In this first experiment, three different reward functions were compared to each other.

3.5.1 Reward Function: classic

The *classic* reward function is a slightly modified version of the reward function used in [7]. It grants a positive reward for moving to the right (as the

TABLE 3.2: The classic reward function.

Event	Reward
movement to the right	+0.5
movement to the left	-0.5
score increase	+1.0
goal	+5.0
death	-3.0

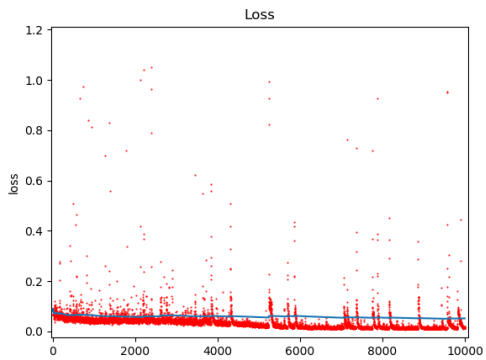

FIGURE 3.2: The classic loss. One data point per trained batch. This graph is cropped at 1.2, cutting off a few higher loss values.

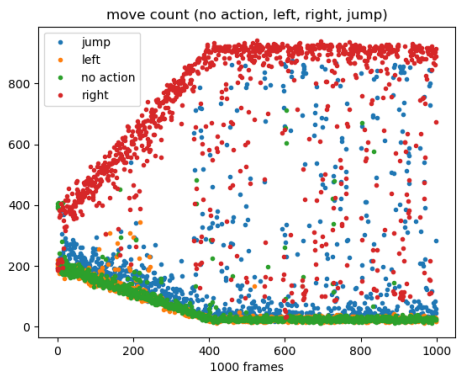

FIGURE 3.3: The classic move count (how often the AI has chosen which action). This includes randomly chosen actions from the ϵ-greedy policy.

goal is always to the right in the training data), score increases, and reaching the goal, and a negative reward for movement to the left and deaths. See Table 3.2 for the values.

This reward function works quite well. The agent learns to walk to the right (see Figure 3.3) and to die less (see Figure 3.7). Observing it during training shows that it doesn't seem to take coins actively, but it doesn't dodge them either. Another problem that has already been mentioned on [7] is that the agent learns to move to the right, not towards the goal. This means that if the goal is not on the very right column of the world (which happens quite often), it is likely that the agent will jump over the goal and then continue to walk to the right, running into the world's border until a new level is loaded. Also, if there is an enemy close to the spawn (on the right side of it), the agent usually runs into that enemy many times before jumping over it, accumulating more deaths than in other levels.

Interestingly, *classic*'s performance regarding coins (Figure 3.5) peaks somewhere around 300,000 actions and decreases again afterwards, which means that the agent takes fewer coins than before. The reason for this could be that it learns to go right all the time, which makes it take less coins that would need other actions than *right* to take. This is, to some extent, also true for the beaten levels, see figure 3.6. Another possible explanation could be the decreasing ϵ, that reaches its minimum after 400,000 actions and could have

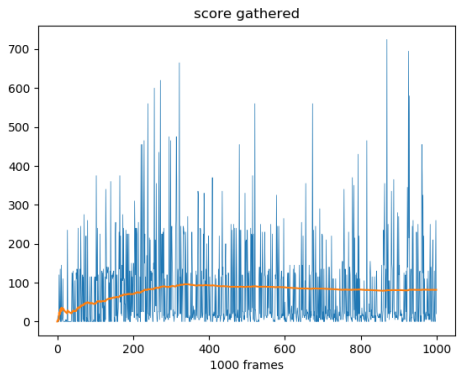

FIGURE 3.4: The achieved score for classic.

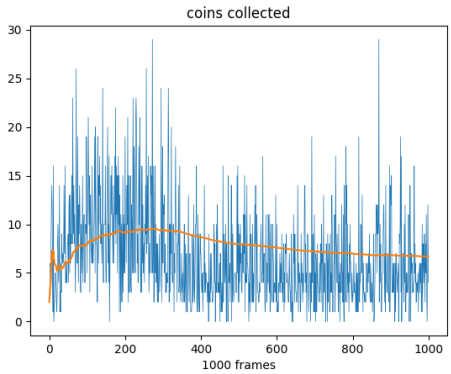

FIGURE 3.5: The classic coin count (how many coins the AI has taken).

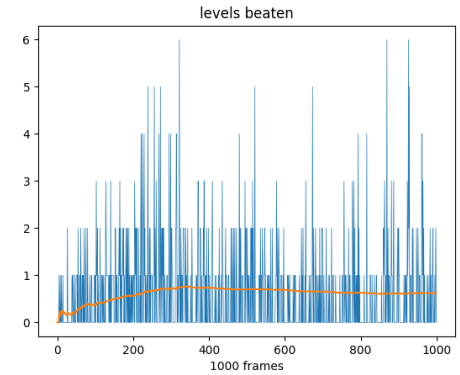

FIGURE 3.6: The classic level count (how many worlds the AI has beaten).

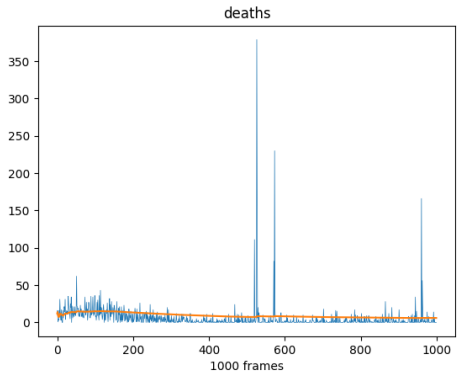

FIGURE 3.7: The classic death count (how often the AI died).

caused the agent to take more coins (and reach the goal more often) by chance before.

3.5.2 Reward Function: goaldist

Due to the above mentioned problem with the agent jumping over the goal, the *classic* model has a lot of unused potential - quite often the agent has more or less beaten the level by reaching the right end of it, but isn't able to hit the goal. To solve this, the *goaldist* variant doesn't give rewards for moving in a specific direction, but instead computes the distance to the goal and rewards moving closer to the goal and punishes moving further from the goal. This way the AI will get negative rewards for jumping over the goal. Apart from that, the reward function is identical to the one used for *classic*, see Table 3.3.

The agent trained with this reward function also dies quite often when there is an enemy close to the spawn and it also jumps over the goal sometimes. However, contrary to the *classic* version, it doesn't run into the right end wall of the world, but instead it moves back and forth a few blocks before

TABLE 3.3: The goaldist reward function.

Event	Reward
goal distance decrease	+0.5
goal distance increase	-0.5
score increase	+1.0
goal	+5.0
death	-3.0

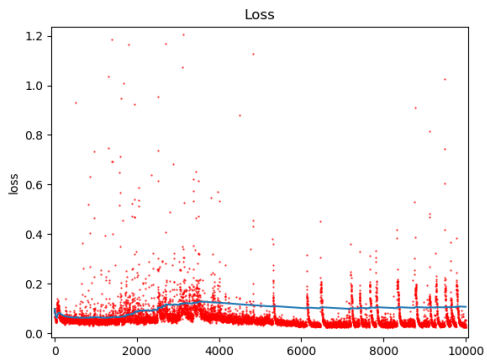

FIGURE 3.8: The goaldist loss. One data point per trained batch. This graph is cropped at 1.2, cutting off a few higher loss values.

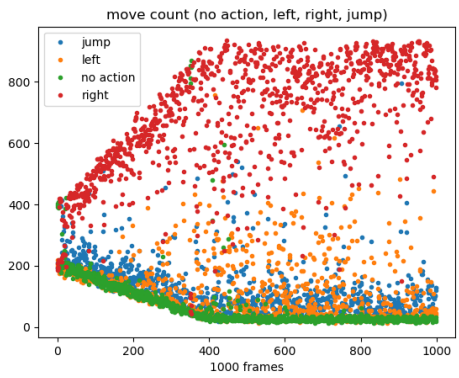

FIGURE 3.9: The goaldist move count. This includes randomly chosen actions from the ϵ-greedy policy.

the world's end. It sometimes dies doing this, but it reaches the goal more frequently than with the *classic* reward function.

Another difference to *classic* is that no graph shows a peak in performance with a decrease afterwards. Instead, Figures 3.11, 3.12, and 3.13 display a constant improvement in performance.

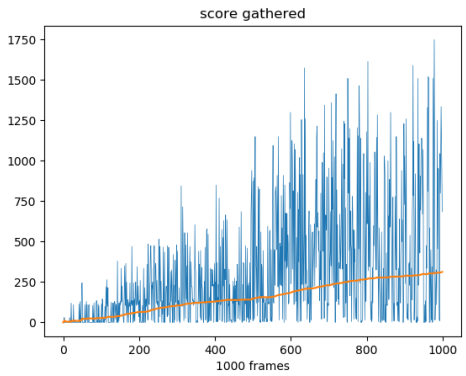

FIGURE 3.10: The achieved score for goaldist.

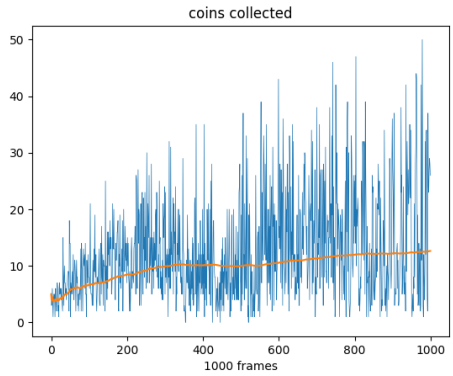

FIGURE 3.11: The goaldist coin count.

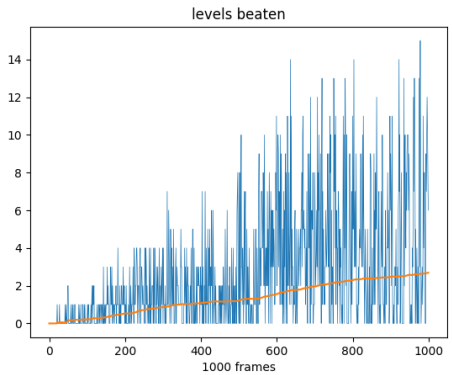

FIGURE 3.12: The goaldist level count.

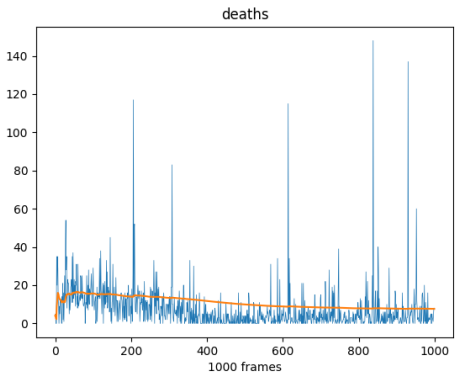

FIGURE 3.13: The goaldist death count.

3.5.3 Reward Function: nox

The *goaldist* reward function still uses information a human player doesn't have, namely the distance to the goal. To see whether the AI would still be able to reach the goal without getting any positive reward for movement, another version of the agent was trained. However, a missing reward for movement could lead the agent to just choosing the *no action* action all the time, as it won't die this way. To prevent it from doing that, there is a very small negative reward for not moving in x direction. This punishment is purposely small, because there are situations in which the agent has to take this punishment since it either can't (while taking a *jump* action) or shouldn't move sideways, e.g. while in between two enemies.

The *nox* version displays a behavior completely different from *classic* or *goaldist*. The agent "flickers" back and forth one block constantly to avoid the punishment, though it also moves through the level sometimes. It "hesitates" when coming close to an enemy, to avoid the negative reward for dying. It might hit the goal occasionally, but compared to the other two variants it seems just way less straightforward or goal-oriented.

Figure 3.15 looks more colorful than the equivalents for *classic* and *goaldist*. This shows that the agent doesn't really learn to move to the right. Instead it learns the mentioned flickering behavior, which results in significantly more *left* actions than the other two agents use. The loss (Figure 3.14) also doesn't really decrease in the way it should if the agent was learning properly. It decreases for some time and then suddenly rises before slowly decreasing again. There is a huge peak shortly after 6000 in Figure 3.14, which corresponds to the part after 600,000 actions in Figure 3.15. The agent has taken a visibly high amount of *no action* and *jump* actions, completely changing its behavior and resulting in a high punishment for not moving. It is, however, not obvious why the agent did this.

Figures 3.16, 3.18, and 3.19 show that the agent is actually improving. The improvements are really small though.

TABLE 3.4: The nox reward function.

Event	Reward
no movement in x direction	-0.01
score increase	+1.0
goal	+5.0
death	-3.0

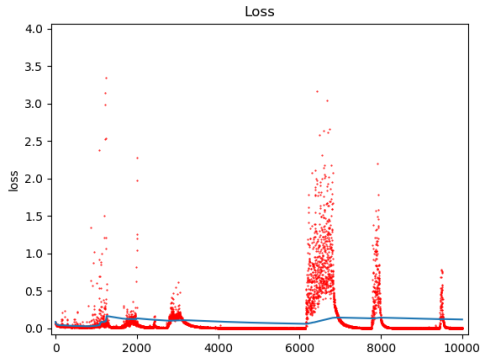

FIGURE 3.14: The nox loss. One data point per trained batch. This graph is cropped at 4.0, cutting off a few higher loss values.

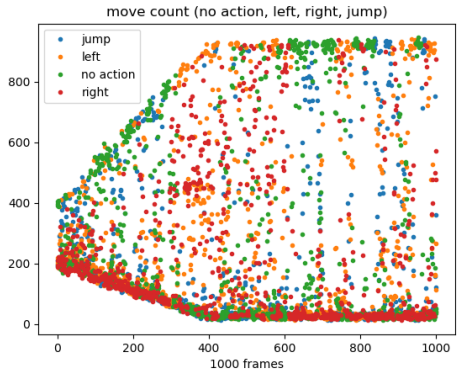

FIGURE 3.15: The nox move count. This includes randomly chosen actions from the ϵ-greedy policy.

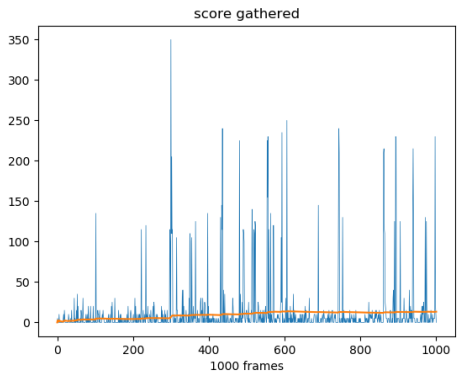

FIGURE 3.16: The achieved score for nox.

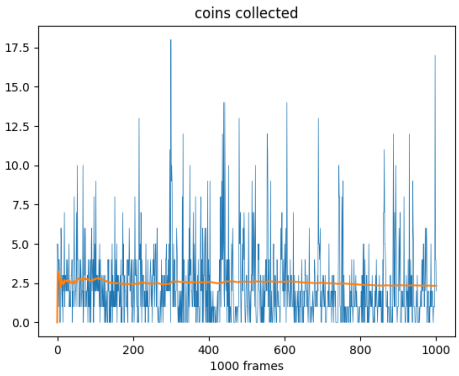

FIGURE 3.17: The nox coin count.

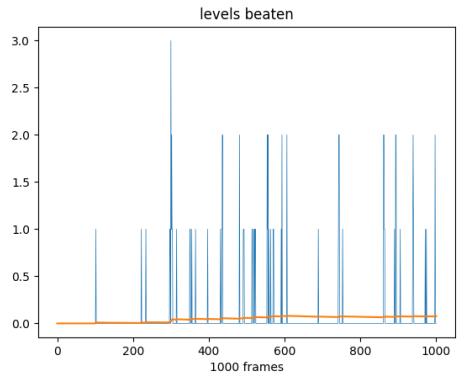

FIGURE 3.18: The nox level count.

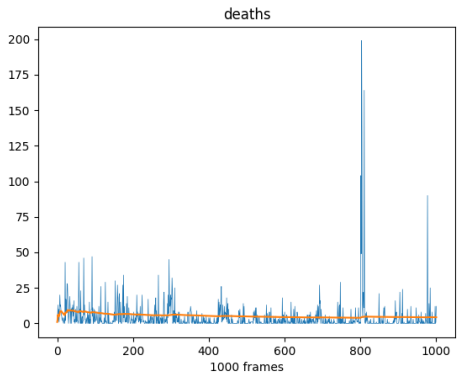

FIGURE 3.19: The nox death count.

3.6 Comparison

For a better comparison, the graphs for the variants have been merged together, displaying the averages of all three variants in one graph per aspect. A random agent (choosing a random action in every step) has been used to generate a reference value.

While the loss for *classic* looks at least somewhat like expected, the curves for *goaldist* and *nox* show that the learning doesn't work as planned. For *nox*, this fits the poor agent behavior. One of the main reasons for this is probably the sparse reward. As long as the agent keeps moving, it won't get any reward most of the time. Especially the reward for hitting the goal is hard to get, as the agent needs to cross the whole level without dying, and without the goal reward, there is no motivation for the agent to do so at all. Another problem might be that jumps are punished: it is not possible to move sideways when taking the jump action, which means that for every jump the agent gets the negative reward for not moving on the x axis. Though this negative reward is really small, it still could lead the agent to jump less, which will further hinder it from jumping over enemies and reaching the goal. The *goaldist* loss is interesting, as it doesn't fit the agent's performance at all. Of all three variants, *goaldist* outperforms the other two in most of the game's aspects (see below). There is no curve for the random agent, as the agent isn't trained and thus doesn't produce any loss values.

As can be seen in Figure 3.21, *classic* seems to learn a little bit faster than *goaldist*, but reaches its maximum performance - in terms of beaten levels - after around 300,000 actions and more or less stops improving afterwards, while *goaldist* surpasses *classic*'s performance after ca. 200,000 actions and keeps improving until the end, leading to a significantly higher amount of beaten levels than the other two variants. *nox* manages to beat some levels and it actually improves over time, but its performance is far from *classic*'s or *goaldist*'s.

The deaths are interesting, as the ranking is the inverse for this aspect: *nox* performed best while *goaldist* dies the most. This is not completely unexpected, however. As mentioned before, *nox* rarely reaches the goal and hesitates to jump over enemies. It doesn't try to reach the other end of the world most of the times - which drastically decreases the chance of dying. Both *classic* and *goaldist* will cross the level, resulting in a higher death rate. The latter one moves back and forth shortly before the end of the level in order to find the goal. While this leads to an increased amount of beaten levels, *goaldist* will also hit enemies while doing this, which is probably the cause for bad performance regarding deaths.

Figure 3.23 and Figure 3.24 are somewhat similar, due to the correlation between coins and score. They both show results similar to Figure 3.21: *classic* learns quite fast and peaks at roughly 300,000 frames, but is soon outperformed by *goaldist*, while *nox* is increasing very slowly and can't keep up with the other two. It even takes fewer coins than the random agent. This is

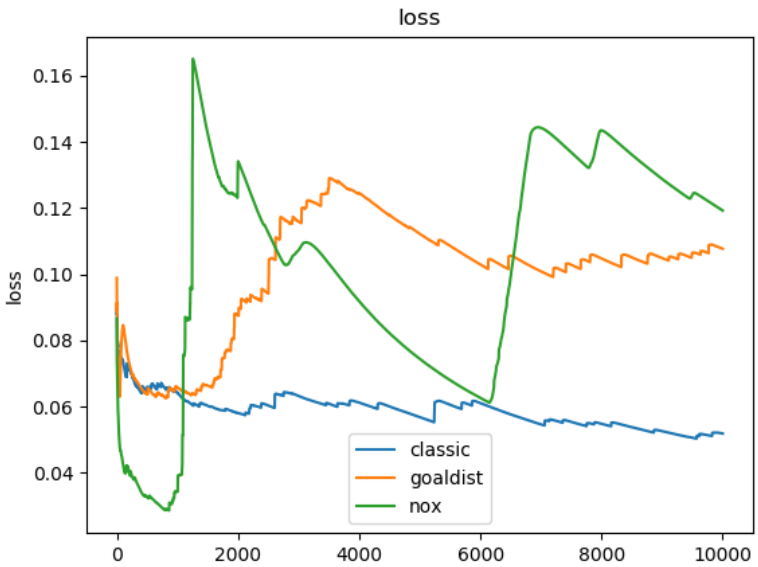

FIGURE 3.20: Averaged loss for classic, goaldist, and nox.

probably due to the fact that most coins can only be taken with jumps and the agent learns to avoid jumping as it can't move sideways during the first frame of the jump and is punished for that. The positive reward from the score increase that comes with taking a coin is far greater than the negative reward for not moving (see Table 3.4), but most of the time there won't be a coin above the player, which results in only the punishment and is probably the reason the agent learns to avoid jumping. However, *nox* will sometimes beat a level and therefore still achieves a higher score than the random agent.

The *classic* and *goaldist* versions of the agent were able to play the game quite well. The *nox* agent had trouble learning, which results in poor overall performance. It mostly moves back and forth, this way it will die rarely but also doesn't reach the goal most of the time. The *goaldist* agent performed the best by far, except for deaths, where *nox* had the fewest. The *classic* agent learns the fastest, but stops improving at some point and is quickly surpassed by *goaldist*.

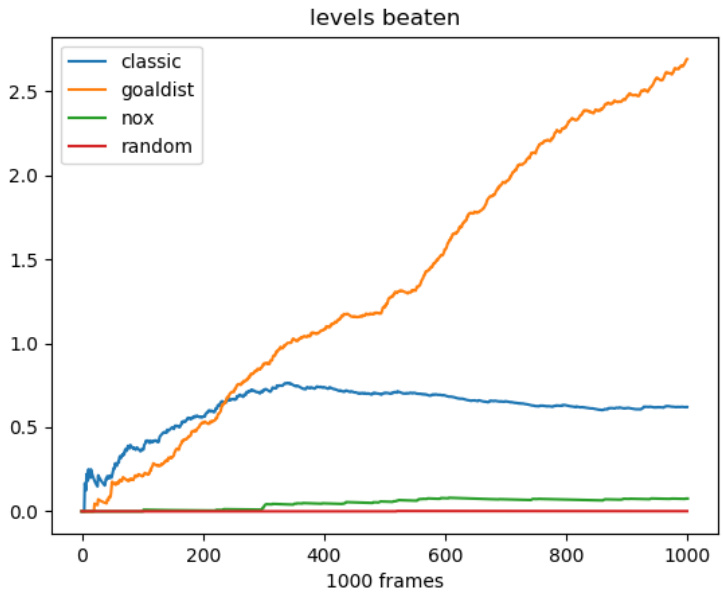

FIGURE 3.21: The averaged level count for classic, goaldist, nox, and random.

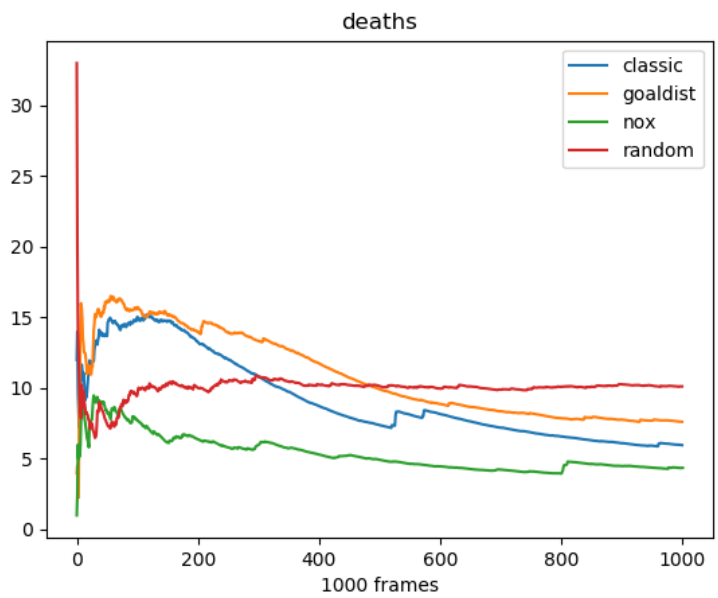

FIGURE 3.22: The averaged death count for classic, goaldist, nox, and random.

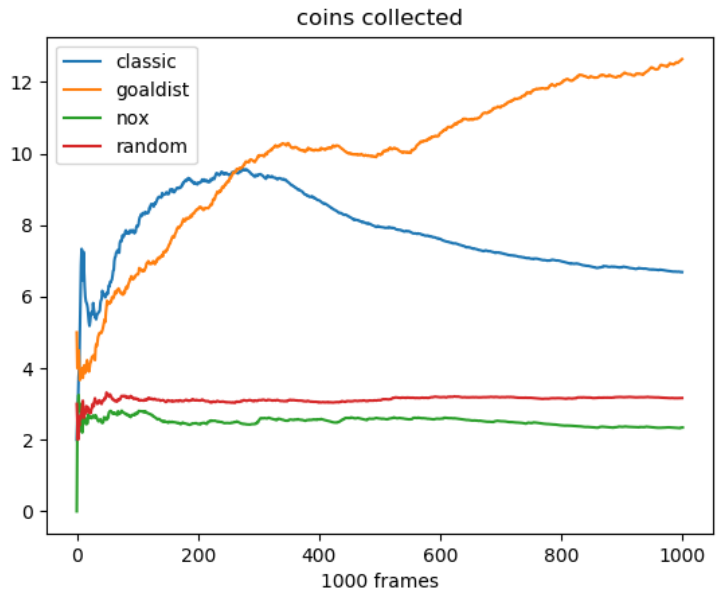

FIGURE 3.23: The averaged coin count for classic, goaldist, nox, and random.

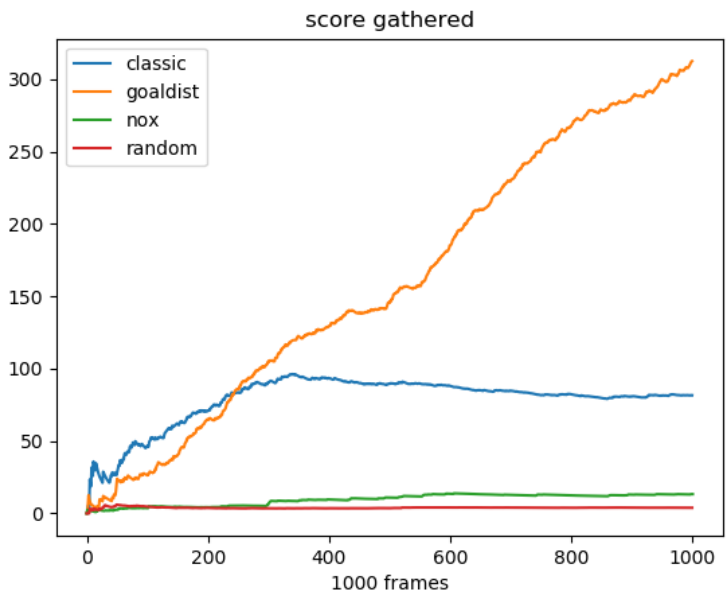

FIGURE 3.24: The averaged score for classic, goaldist, nox, and random.

Chapter 4

Experiments: Randomized Graphics

This chapter describes the second set of experiments that have been conducted. The goal was to test how good the agent can abstract from the game's graphics. While most of the parameters were identical to those used for the experiments described in Chapter 3, each level used a randomly chosen tileset for the blocks. This results in a game that is conceptually the same, but with changing graphics every level.

To distinguish these experiments from the *basic* experiments, these will be referred to as *rndgfx*.

4.1 Training Data

The generation of training data is done in the same way as for the basic experiments, with two differences. 1200 worlds total were created, 200 with "standard" parameters and 1000 with parameters leading to easier worlds.

The first difference to the basic experiments' training data is the score position. While the score was displayed in the upper left corner in the basic experiments, its position is randomly chosen (from upper left, upper center, and upper right) for all 1200 worlds this time. This was done for the same reason as the changed graphics: to test the adaptability of deep reinforcement learning. The agent could learn to read the current score from the game screen, gaining additional information in this process. This is way more difficult when the score is not displayed at the same position all the time. Even if the network doesn't learn to read the score and just ignores the black pixels, it has to adapt to these pixels moving around. Refer to Table 3.1 for the other world generator parameters.

The second difference is the randomized tileset. For all blocks except for *air*, *spawn*, and *goal*, several graphic files were provided and for each block one

TABLE 4.1: The amount of different graphic files per block type.

Block type	Amount of different graphics
air	1
ground	21
spawn	1
goal	1
enemy	6
coin	3
player	4

of these was chosen randomly during world generation.[1] The graphic file for the player itself was also randomized in this way. The graphics that should be used are saved in the world file and only one graphic file per block is used, so the same world will look the same each time it is loaded in the game. *Spawn* and *goal* weren't changed to allow the AI to at least recognize these reliably. *Air* wasn't changed in order to keep the game somewhat like a "real" game: while it is relatively easy to replace a flat brown *ground* block by a brick wall and still have a realistic looking ground, this is way more difficult for an *air* block. The game currently doesn't support different graphics for the same block within one level, so e.g. having clouds in the sky would only be possible if every *air* block had a cloud in it, which would make the level look "overloaded". *Air* is also special in being a background block without any effect, so not changing its graphics will make learning the game simpler for the AI, as it is easier to detect other blocks when the *air* block doesn't have any texture.

The different graphic files for the blocks were chosen to be somewhat similar regarding color and/or structure (although the AI only receives greyscale images in this experiment). Coins are always round and yellowish, enemies are reddish, and ground is always some kind of brown. Table 4.1 shows the amount of different block graphics for each block.

4.2 Network and Learning Algorithm

The network as well as the learning algorithm are unchanged from the *basic* experiments. Refer to Sections 3.2 and 3.3.

[1] The additional graphic tiles were taken from
https://opengameart.org/content/dungeon-crawl-32x32-tiles
https://opengameart.org/content/coins-asset
https://opengameart.org/content/items-and-elements
https://opengameart.org/content/superpowers-assets-characters
and are all licensed under CC0 (public domain).

4.3 The Reward Functions

The same reward functions as for the basic experiments were used. As the observations don't differ too much compared to the *basic* experiments, there is no detailed description for each reward function here. The next section compares the results to the *basic* experiments, refer to Appendix A.1 for more detailed graphs of the three *rndgfx* variants.

4.4 Comparison

The loss already hints that this task was harder than the basic experiments. Most of the time, the loss is higher in the *rndgfx* version than in the *basic* counterpart. For *classic*, the loss functions look roughly the same, only that the *basic* version is lower in general. This is also true for *goaldist*, but here the *rndgfx* variant sinks lower than the *basic* loss towards the end. However, the other graphs indicate that the *basic goaldist* still by far outperforms the *rndgfx* version. For *nox*, the *rndgfx* version looks as strange as the *basic* one: it strongly rises and then slowly falls again, overall increasing. It shows a strong jump towards the end, exceeding the other variant's losses by a great amount. Although it is not sure what caused this extreme loss increase, the bad loss fits the poor performance of this variant, that performs the worst for nearly all aspects of the game.

Figures 4.2 shows that *rndgfx* performs surprisingly similar to *basic* regarding the beaten levels. The curves for both experiments look really similar for all three variants and only *goaldist* shows a larger difference in values. While it can't keep up with its counterpart from the *basic* experiments, *rndgfx goaldist* manages to surpass even the basic versions of *classic* and *nox* by a large margin. *rndgfx nox* beats even fewer levels than its *basic* counterpart, making it the worst performing agent regarding beaten levels so far (*random* not included).

As can be seen from Figure 4.3, all *rndgfx* agents die more often than their *basic* counterparts, but behave quite similar to them when compared to each other: the least dying *rndgfx* agent is nox, even better than *basic classic* for most of the time. It shows a deterioration towards the end, similar to its loss function, although this one starts somewhere around 800,000 actions while the loss went up after about 660,000 actions already. *classic* roughly dies as much as *basic goaldist*, which was the most dying agent from the *basic* experiments and *rndgfx goaldist* dies the most. The reason for the increasing death rate of *nox* towards the end is unclear, but apart from that all three variants perform pretty much like their *basic* counterparts, only generally worse due to the changing graphics.

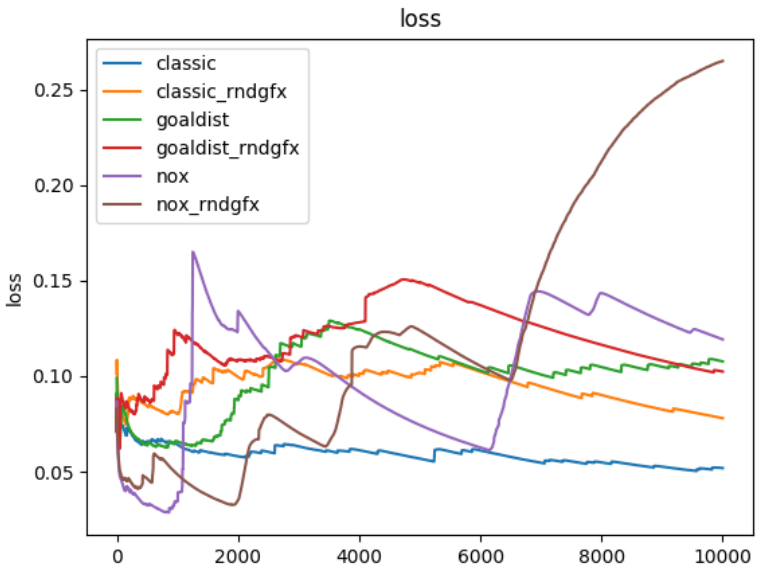

FIGURE 4.1: Averaged loss for classic, goaldist, and nox for basic and rndgfx.

Likewise to the beaten levels, *classic*'s and *nox*'s results are slightly worse but generally really close to their *basic* variant regarding collected coins (Figure 4.4) and achieved score (Figure 4.5), while *goaldist* indeed shows a similar behavior but a clearly greater deterioration. Nonetheless it outperforms not only *classic* and *nox*, but also their correspondents from the basic experiments.

As the *rndgfx* experiments have shown, deep reinforcement learning is able to abstract from the graphics and train an agent capable of playing the game, although all three agents show a degradation in performance when compared to the *basic* experiments. This is due to the increased task difficulty and not unexpected. Interestingly, *goaldist* showed a significantly larger performance loss than the other two agents. This could be explained with the agent being more dependent on the graphics than its variants, but since the goal graphic didn't change, there is no obvious reason why this should be the case. Despite the comparatively bad performance, it still beat *classic* and *nox* in every aspect except deaths.

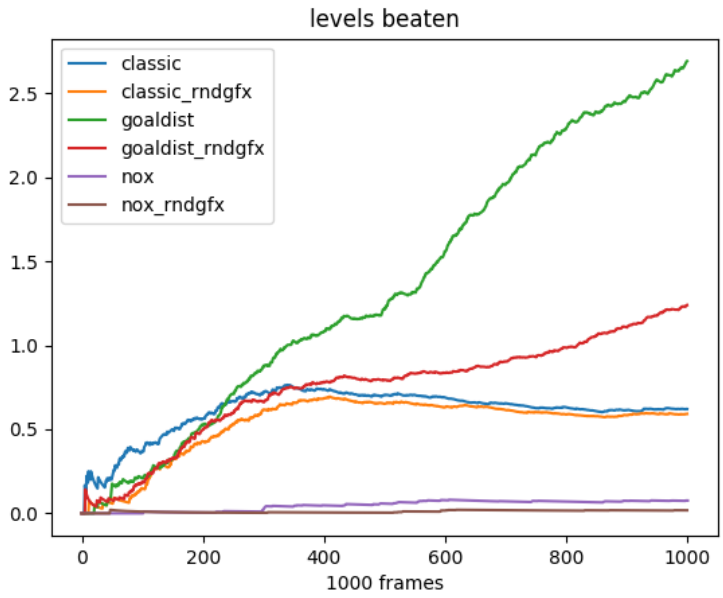

FIGURE 4.2: The averaged level count for classic, goaldist, and nox for basic and rndgfx.

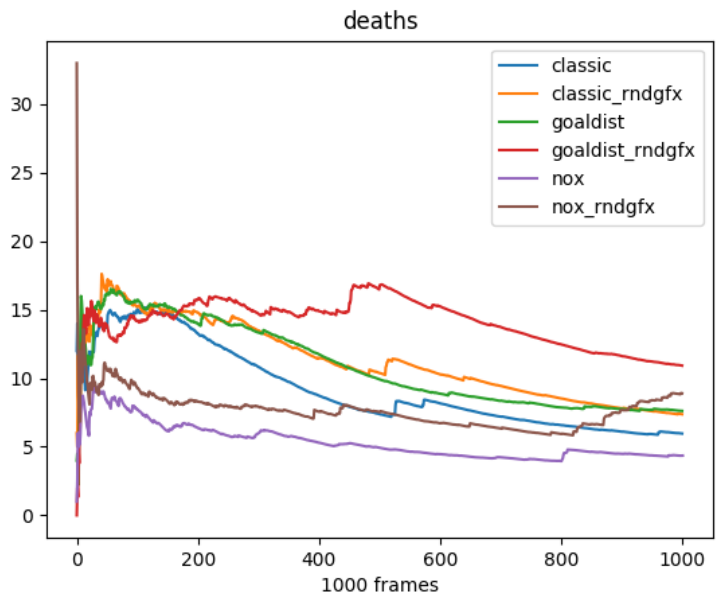

FIGURE 4.3: The averaged death count for classic, goaldist, and nox for basic and rndgfx.

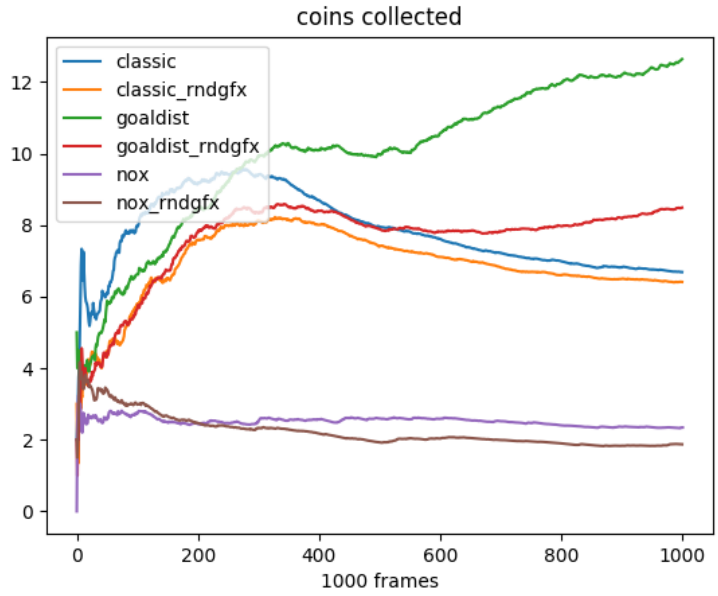

FIGURE 4.4: The averaged coin count for classic, goaldist, and nox for basic and rndgfx.

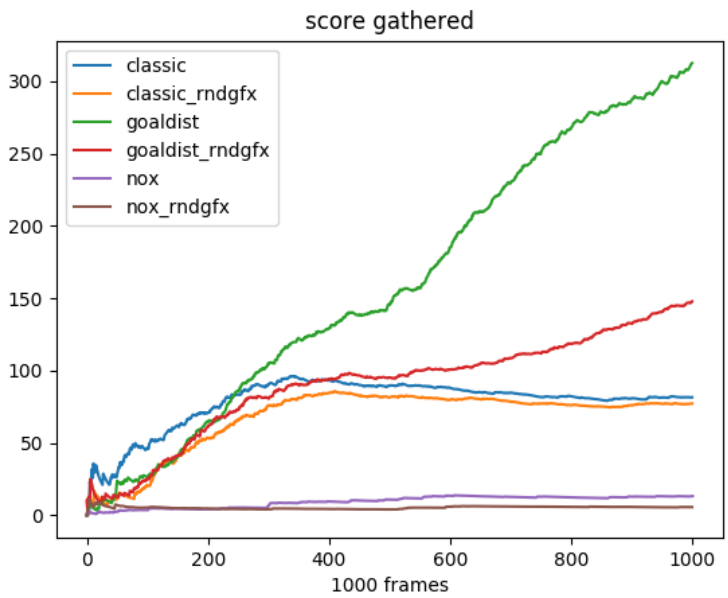

FIGURE 4.5: The averaged score for classic, goaldist, and nox for basic and rndgfx.

Chapter 5

Experiments: Atari

Although the focus of this thesis is not on the neural network used for deep reinforcement learning, it is interesting to know how the network affects the learning. Therefore this experiment set was conducted using the neural network from [12].

These experiments will be referred to as *atari* experiments.

5.1 Training Data

The exact same training data as generated for the *basic* experiments was used. Refer to Section 3.1 for a description.

5.2 Network and Learning Algorithm

The neural network for this experiment set is the one used to play the Atari games in [12]. It consists of two convolutional layers - convolving from 84x84x4 to 20x20x16 to 9x9x32 - and one fully-connected layer with a size of 256, using a rectifier nonlinearity after each layer. Refer to [12] for a more detailed description of the network. The input to the network is built from the last four screenshots of the game, greyscale and scaled to 84x84 pixels, and the network was set to four output neurons.

The learning algorithm wasn't changed, as [7] already uses the deep Q-learning as described in [12]. The parameters for the learning were unchanged from the basic experiments, which use the parameters from [7]. They are probably not optimal for the changed network, but this applies to the parameters from [12] too for the changed problem. For the sake of comparability to the other experiments, the same parameters have been chosen.

5.3 The Reward Functions

This time, only two of the reward functions from the *basic* experiments were used: *classic* and *goaldist*. Learning with the *nox* reward function is clearly the most complex task (compared to the other two reward functions) and it doesn't really work well. The *atari* network is significantly simpler in structure than the network from [7], which is used in the other experiments and it is expected to perform generally worse compared to the *basic* experiments due to this. As the more complex network already led to a quite bad performing *nox* agent, this experiment is assumed to be not fitted for the *nox* reward function, which is why this variant was skipped in this experiment set. Detailed figures showing the results can be found in Appendix A.2.

5.4 Comparison

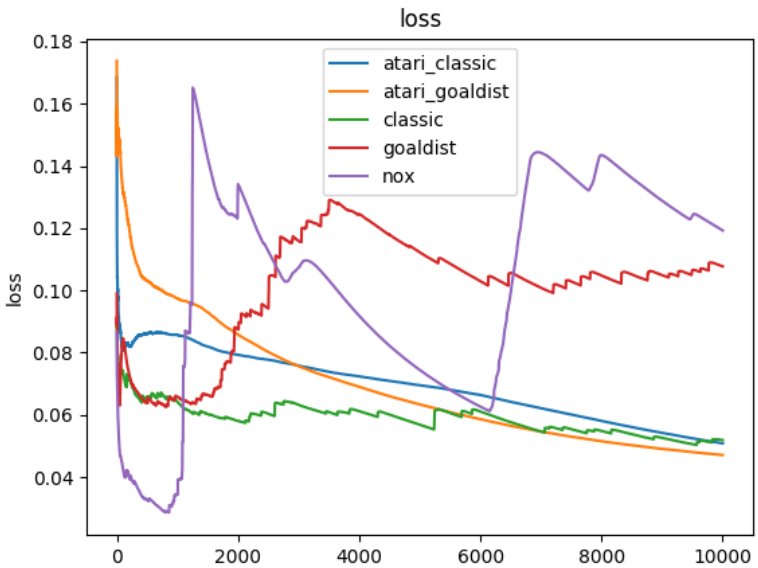

FIGURE 5.1: Averaged loss comparison for basic and atari experiments.

The loss for both *atari* experiments looks way more like a typical loss is expected to look like. This implies that the learning worked well for these experiments - better than for the other ones. Apparently the *atari* network learns better than the one from the Super Mario project. This rises the question whether it learns better or just faster: maybe the 1,000,000 simulated actions were enough to train this network, but not the more complex one used in the other experiments. Due to technical constraints it is unfortunately not possible to further continue the experiments within the scope of this thesis, but future research could investigate whether the loss values for the other experiments look more reasonable if trained for more than 1,000,000 actions.

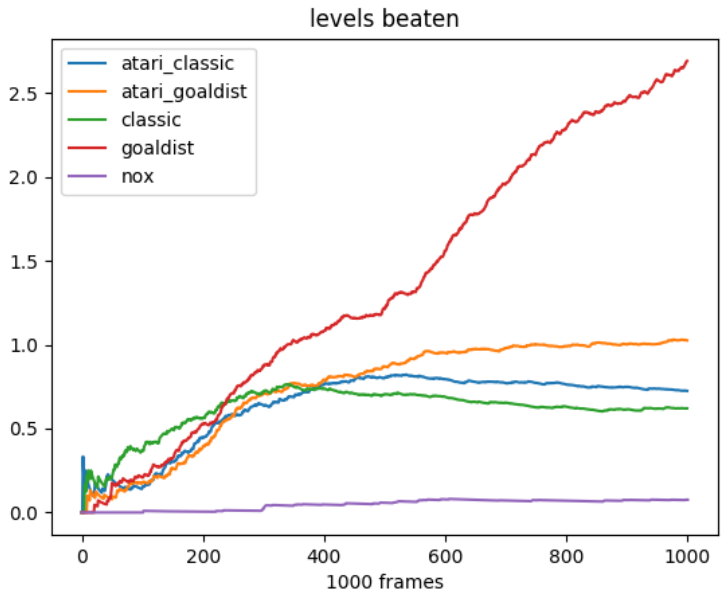

FIGURE 5.2: The averaged level count comparison for basic and atari experiments.

Both *atari* versions seem to perform quite well. This is somewhat surprising, as they were expected to perform worse due to the simpler network structure. Instead the *atari classic* variant not only surpassed *basic nox*, but also *basic classic* by a small margin. *goaldist* performed even better, but couldn't compete against the great results of the *basic* counterpart. It is possible that the simpler *atari* network is sufficient to play the game - at least for the *classic* variant - and performs similarly well while being easier to train. Another possible explanation is the one mentioned in the above paragraph: the amount of simulated actions is not enough to fully train the Super Mario network, but it could perform significantly better than the *atari* network if trained for a longer time.

Similar to the *basic* experiments, the agents learn to avoid dying. However, both *atari* variants consistently die more than all *basic* agents. This is interesting, as this is the only aspect of the game where the *atari* agents perform significantly worse. Apparently the *atari* variants have more trouble dodging enemies. This hints at a structural advantage of the Super Mario network regarding this aspect. Unfortunately, it isn't clear whether the problem is identifying the enemies, actually jumping over them, or something else. Also interesting is that *classic* died more often than *goaldist* - this was the other way around in the *basic* experiments and could be explained with the *goaldist* agent running back and forth towards the end of the level. They are quite close to each other though.

Figure 5.4 and Figure 5.5 both show results similar to Figure 5.2: the *atari* agents beat the *basic* variants *nox* by a large and *classic* by a small margin but can't keep up with *goaldist*.

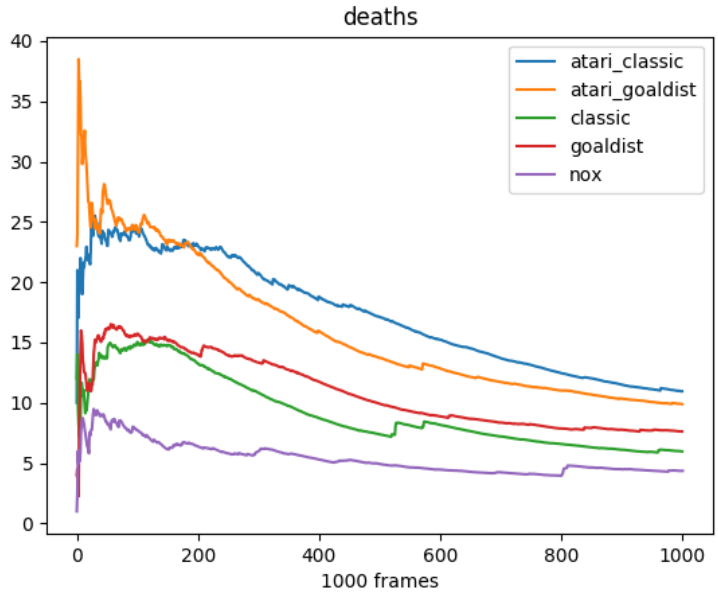

FIGURE 5.3: The averaged death count comparison for basic and atari experiments.

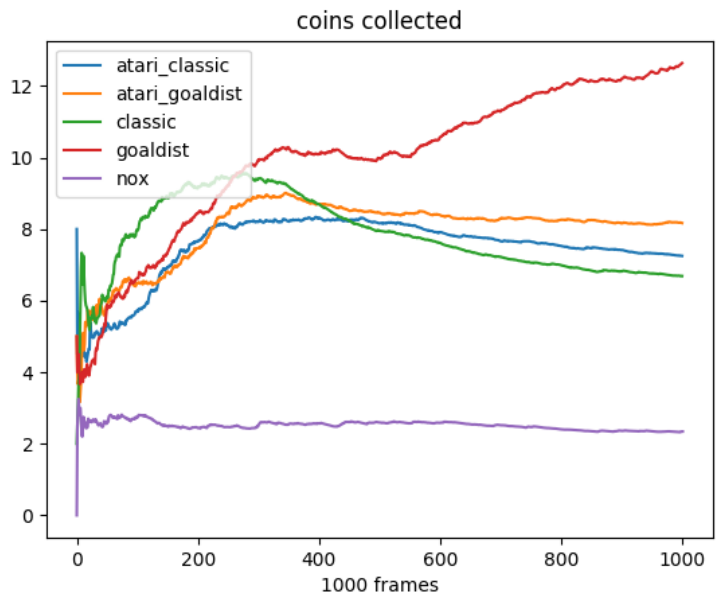

FIGURE 5.4: The averaged coin count comparison for basic and atari experiments.

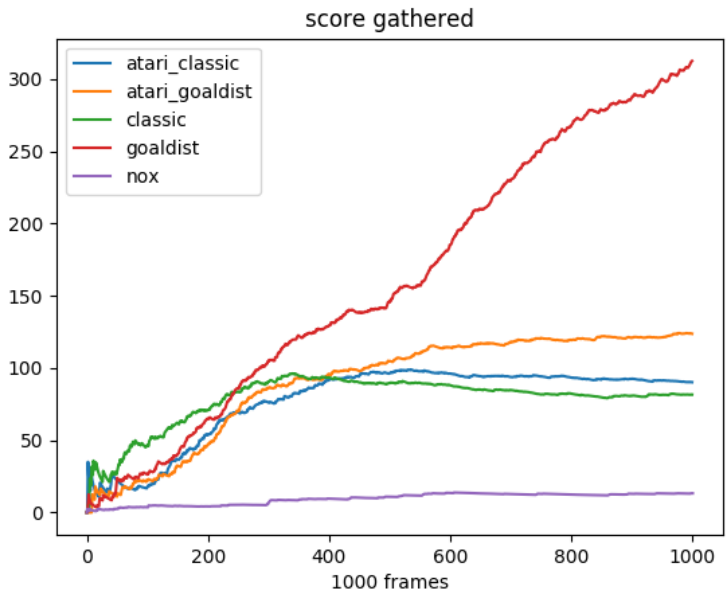

FIGURE 5.5: The averaged score comparison for basic and atari experiments.

Chapter 6

Experiments: Bidirectional

In the *bidirectional* experiment set, roughly half of the generated worlds have the spawn on the left and the goal on the right while it is the other way around for the rest. This is thought to be extremely challenging for the AI, as it needs to figure out in which direction the goal is for each level. Since spawn and goal are placed in the first and last 20% of the world, the player usually spawns close to one end and the direction towards the goal can be determined from the first screenshot. This is due to the fact that the game keeps the player in the center of the screen, unless this would show areas "outside" the world. Therefore if the player is shown left of the center when spawned, it has to be close to the left end of the world and thus needs to move right. This works accordingly if the player is offset to the right. However, the generated worlds are up to 75 blocks long while only 24 blocks in width are shown on one screen, so it is possible for the spawn to be far enough from the end of the world so that the player will be shown in the center when spawned. In this case, the AI needs to find the correct direction by trial and error. In both cases, it needs to remember this direction, which is probably hard, as the network doesn't have recurrent elements or LSTM modules, apart from the action history that gets the last few screenshots and chosen actions for an input.

6.1 Training Data

This experiment uses the same world generator parameters as shown in Table 3.1, except that *spawn_left_probability* is set to 0.5 for all generated worlds. World generation is otherwise similar to the *basic* experiments.

6.2 Network and Learning Algorithm

The network and the learning algorithm are unchanged from the *basic* experiments. Refer to Sections 3.2 and 3.3.

6.3 The Reward Functions

Having the goal on changing sides of the worlds contradicts *classic*'s idea of rewarding movement in one fixed direction and if adapted to dynamically change this direction based on the current world, it would closely resemble *goaldist*. Thus only the *goaldist* and *nox* reward functions were used in this experiment set. The results will be compared to the *basic* experiments in the next section, detailed graphs for this experiment set can be found in Appendix A.3.

6.4 Comparison

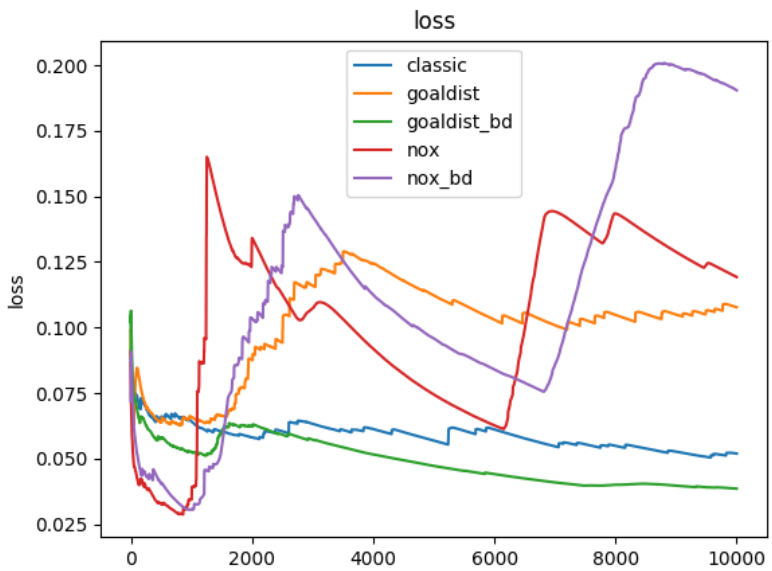

FIGURE 6.1: Averaged loss comparison for basic and bidirectional experiments.

Figure 6.1 shows again a loss that doesn't really fit the performance: while the loss of the *bidirectional nox* agent looks quite similar - and the agent also performs quite similar - to the *basic* correspondent, this isn't true for *goaldist*. Here the *bidirectional* variant has an even lower loss than all of the agents from the *basic* experiments, but it performs significantly worse than most of them.

Regarding beaten levels (see Figure 6.2), it becomes obvious that changing the direction of the goal makes the game a lot more difficult. The *nox* agent didn't deteriorate much compared to its *basic* counterpart, but there was not much potential for deterioration, as it already performed really bad at the *basic* experiments. However, *goaldist* shows a huge performance loss, going from the best performing agent by far to a performance only slightly better

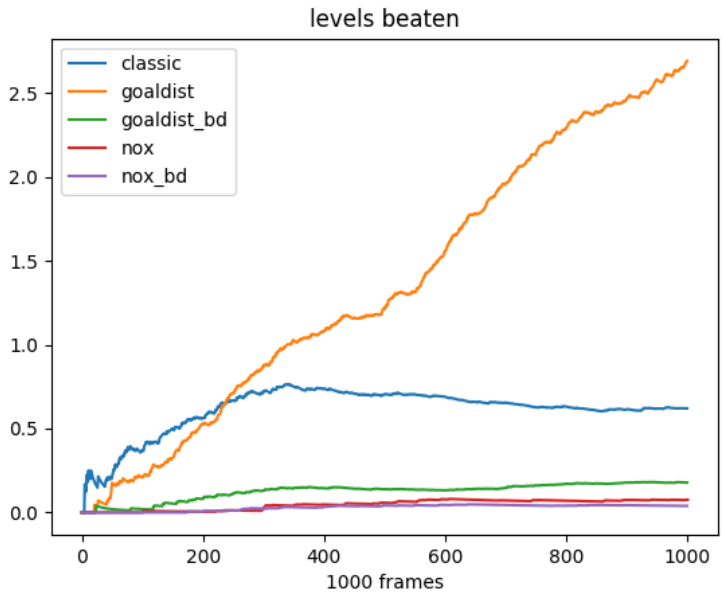

FIGURE 6.2: The averaged level count comparison for basic and bidirectional experiments.

than *basic nox*. Observing the agents while playing shows that both of them indeed have problems remembering the correct direction and often get stuck somewhere in the middle of the world, continuously changing their direction back and forth.

Comparing the deaths (Figure 6.3) is quite interesting. In the basic experiments, *nox* had the fewest deaths while *goaldist* had the most and *classic* was in the middle of both. In comparison to that, both *bidirectional* agents converged towards *basic classic* - *nox* died more often than before while *goaldist* died less. It was assumed that the *basic goaldist* variant often died while walking back and forth towards the end of the level, hoping to reach the goal. This happens less often for the *bidirectional* version, as it gets stuck in the middle of the world sometimes. When observed, the *bidirectional nox* agent seems to actually move more through the world instead of just moving back and forth all the time like the *basic* version does. This could explain the higher death rate. It is, however, unclear why it moves more. Also, the *nox* agent displays a strong increase in deaths after around 800,000 actions, which happened in a very similar fashion in the *rndgfx* experiment set, so this seems to be some property of the *nox* agent that appears sometimes. The reason for this is not clear either.

The graphs for collected coins (Figure 6.4) and score (Figure 6.5) look similar to the one for beaten levels: *nox* is pretty close to its *basic* counterpart while *goaldist* suffered a huge performance loss and now is only slightly better than the *nox* agents.

The observations made while training the agents suggest that they have trouble deciding for the correct direction, or at least remembering that direction.

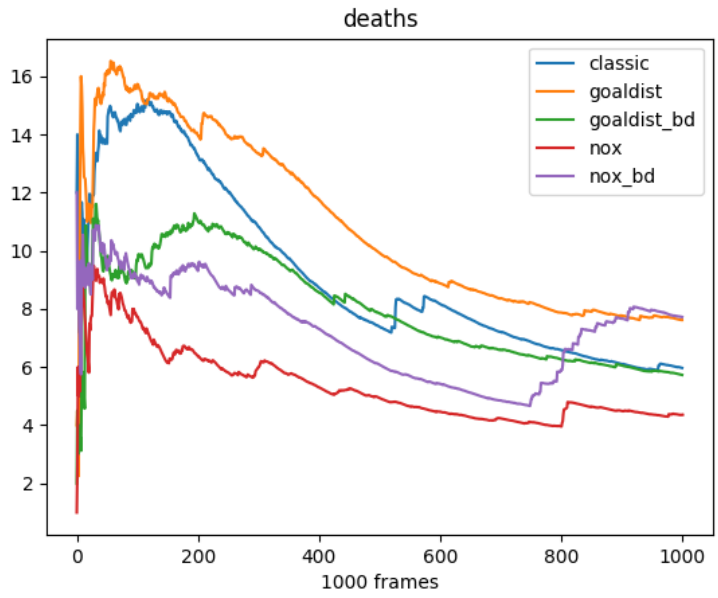

FIGURE 6.3: The averaged death count comparison for basic and bidirectional experiments.

The network gets the last few actions for an input, so it could theoretically use them to remember a chosen direction. However, these are very few actions and if the agent doesn't move sideways for about four frames - e.g. because it is jumping and waiting to ascend high enough to jump over an enemy - this information is lost and it is probably impossible for the agent to recover it while in the middle of a world. It is likely that adding a recurrent or LSTM module to the network could provide the missing possibility of saving a direction and greatly improve the performance of the *bidirectional* agents.

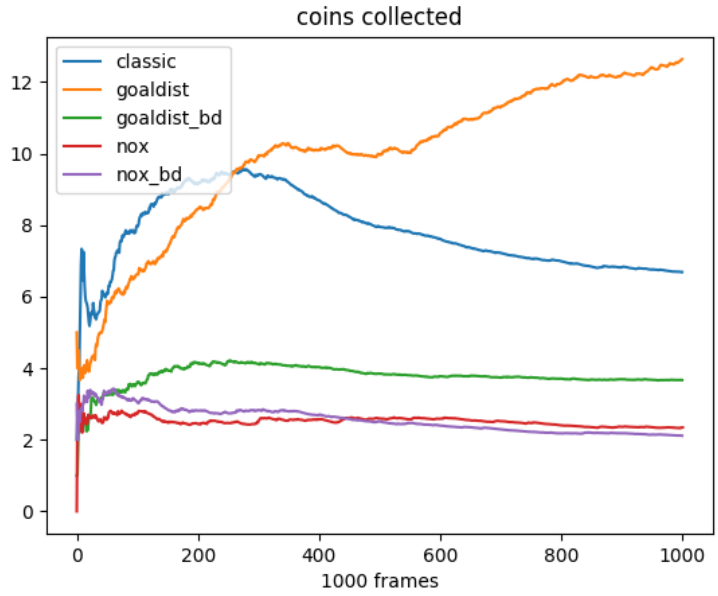

FIGURE 6.4: The averaged coin count comparison for basic and bidirectional experiments.

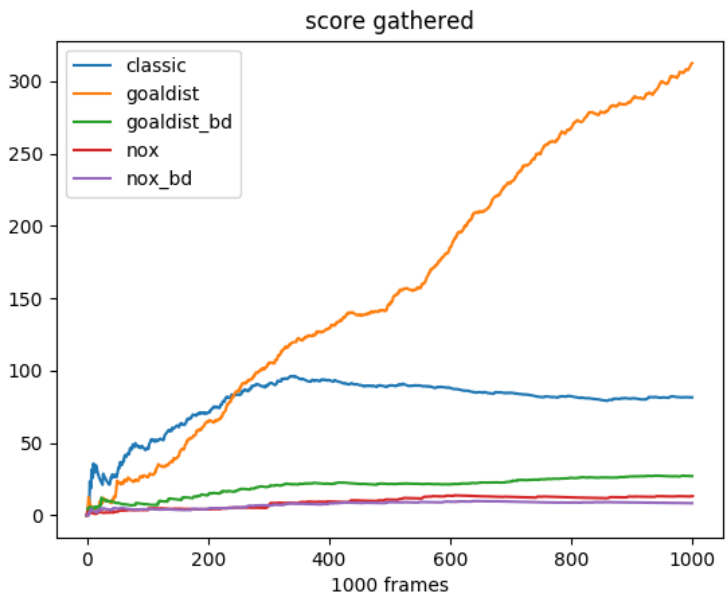

FIGURE 6.5: The averaged score comparison for basic and bidirectional experiments.

Chapter 7

Experiments: Double Jump

In this experiment set, the agent is able to "double jump". This works as follows: the agent can jump as usual. Jump width and height are set to 3, as for the *basic* experiments. However, at any time during a jump, the agent can initiate another jump. This can only be done once during a jump and to reset its jump ability, the agent has to touch ground again. The second jump will disrupt the first one immediately and start executing its ascending phase as described in Section 2.2.2.

This experiment set will be referred to as *doublejump*.

7.1 Training Data

New training data was generated for these experiments in the usual fashion. Table 7.1 shows the world generator parameters. The main changes compared to *basic* are the ability to double jump and increased values for maximum enemy height and maximum terrain height change.

The ability to double jump makes the worlds easier for the agent, as it is able to travel many blocks while in the air, allowing it to pass multiple smaller enemies at once. This is somehow balanced by the increased enemy and terrain height, that forces the agent to actually use the double jump. Terrain height changes (with or without enemy on top) can create obstacles of a total height of 6 blocks. Reaching the maximum possible height that is required to jump over such an obstacle only works with the correct timing: if the second jump isn't issued at the highest point of the first one - meaning after the last ascending step and before the descending phase - the agent won't reach the necessary height.

It should also be mentioned that the world generator creates coins at a certain percentage of positions eligible for a coin and this percentage is unchanged from the other experiments. Simply speaking, a position is eligible for a coin if the player can reach it (this is computed via the *max_ height_ diff* parameter). All other experiments use only single jump with a jump height of 3, so the agent can reach four blocks in height above the ground level: the block directly above the ground - it is walking on this level - and three more when

TABLE 7.1: World generator parameters for doublejump training data. Intervals in the *Enhanced value* column denote the range the value is randomly chosen from, empty entries mean no change from the standard value.

Parameter	**Standard value (200 worlds)**	**Enhanced value (1000 worlds)**
jump	2	
jump_ height	3	
jump_ width	3	
score_ position	-1	
world_ size_ x	75	[50, 75]
world_ size_ y	20	
max_ height_ diff	6	
max_ height_ terrain	10	
height_ change_ probability	0.2	(0.1, 0.2]
height_ up_ probability	0.5	
height_ down_ factor	0.01	[0.01, 0.1)
enemy_ spawn_ probability	0.15	[0.1, 0.15)
enemy_ max_ width	1	
enemy_ max_ height	4	
enemy_ grow_ probability	0.25	[0.1, 0.25)
coin_ spawn_ probability	0.05	
spawn_ area_ percentage	0.2	
spawn_ left_ probability	1.0	

jumping. In the *doublejump* experiments, however, the agent can reach seven blocks above the ground (walking level + 3 from first jump + 3 from second jump). This means that the worlds generated for this experiment set have roughly 75% more positions eligible for coins and accordingly more coins than the worlds from other experiments.

7.2 Network and Learning Algorithm

This experiment set uses the same network and learning algorithm as the *basic* experiments.

7.3 The Reward Functions

classic, *goaldist*, and *nox* reward functions were used for the experiments. Detailed graphs can be found in the Appendix (A.4).

7.4 Comparison

As for the other experiment sets too, the losses (see Figure 7.1) are somewhat interesting. For *classic*, the *doublejump* loss is higher than the *basic* one, although getting closer towards the end. For *goaldist*, it's exactly the other way around. And *nox* shows a huge increase at around 500,000 actions. This looks similar to the *rndgfx nox* loss (see Figure 4.1), but the increase is significantly bigger here.

Regarding the amount of beaten levels, the results for *doublejump* look very similar to the ones for *basic*: an incredibly good *goaldist* agent that never stops improving, a really bad *nox* agent that only rarely beats a level and a mediocre classic agent that shows a performance peak at around 300,000 actions before it slightly deteriorates. This is not unexpected, as the three reward functions mostly created similarly behaving agents for the other experiment sets too and the *doublejump* experiments are probably the ones closest to *basic*. Over all three reward functions, the *doublejump* agents beat marginally fewer levels. When observing the agents while training, it became obvious that the task of finding the correct timing to reach the maximum possible height with a double jump is indeed not trivial to the AI and the agent often gets stuck at high obstacles for some time before it manages to get to the necessary height. This is probably the reason for the slightly worse performance compared to *basic* here.

The *doublejump classic* and *goaldist* agents die less than their *basic* equivalents. As mentioned before, the double jump allows the agent to travel long distances while in the air. This is especially useful for several smaller enemies

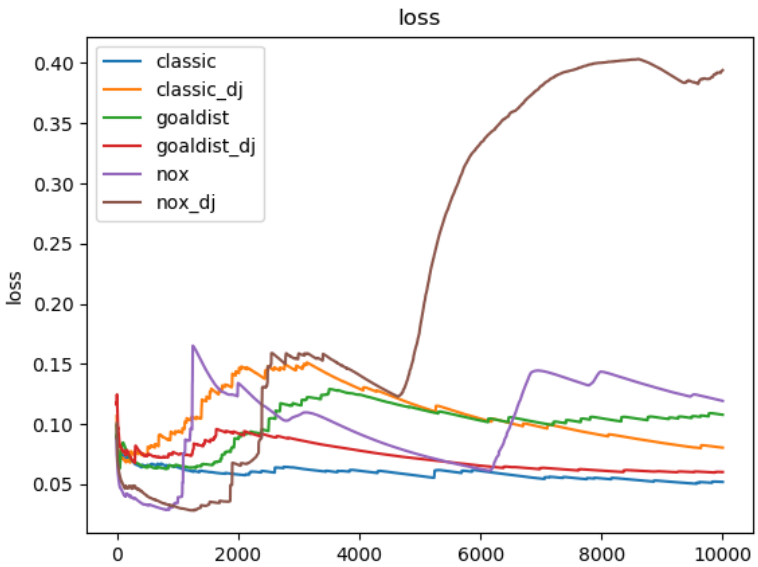

FIGURE 7.1: Averaged loss for classic, goaldist, and nox for basic and doublejump.

close to each other: the agent can jump over all of them with one double jump, while a *basic* agent would need to land somewhere between the enemies and is more prone to dying while doing so. Thus the improvement regarding deaths (see Figure 7.3) is not surprising.

Also not surprising is the improvement regarding coins. As Figure 7.4 shows, all three *doublejump* agents collected more coins than their respective *basic* counterparts. This is most probably due to the aforementioned fact that the worlds used for training have roughly 75% more coins than the ones from the *basic* training set. However, the agents didn't improve by that amount. Many coins are at positions that are so high that the agent would need to ascend to the maximum possible height to get them, which is difficult. Furthermore it will miss some coins on the ground between enemies when jumping over multiple enemies with a long double jump. It would be interesting to know how the *doublejump* agents perform without the increase in the amount of coins.

For the score (see Figure 7.5), all *doublejump* agents perform incredibly similar to their *basic* counterparts. The score can be increased by reaching the goal or by taking coins, while the increases from the latter one can be reduced or negated by dying. Both *classic* and *goaldist* have beaten slightly fewer levels than their *basic* versions, but they collected more coins and died less, which seems to have compensated for that. The *doublejump nox* agent took more coins and has beaten roughly the same amount of levels as its *basic* correspondent, but it died significantly more often which probably negated the score from coins and lead to a result very similar to *basic*.

Altogether, the agents trained in this experiment set behaved pretty much as

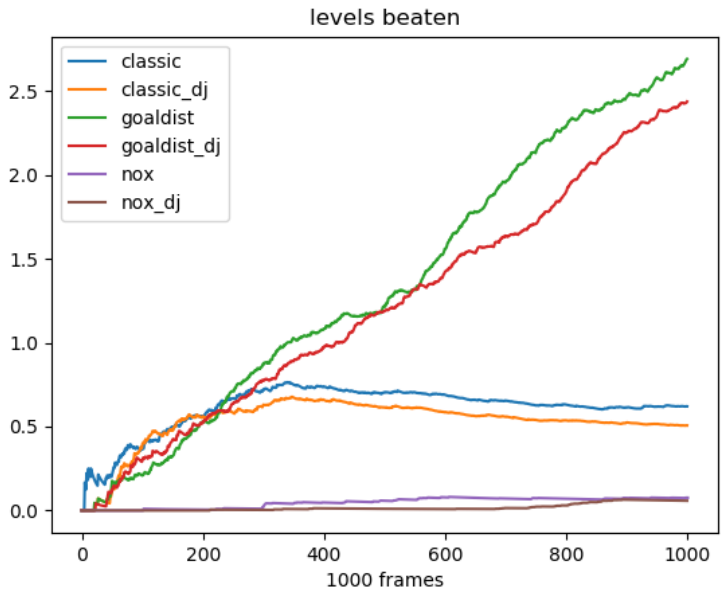

FIGURE 7.2: The averaged level count for classic, goaldist, and nox for basic and doublejump.

expected compared to *basic*: more coins, because there were more coins in the worlds, fewer deaths, because the double jump ability makes it easy to cross multiple small enemies at once, and slightly fewer beaten levels, because reaching the maximum possible height necessary for some obstacles is more difficult with double jump than it is with single jump. Similar to most of the other experiments, *classic* and *goaldist* reward functions produced somewhat skilled agents while *nox* had trouble learning and performed bad.

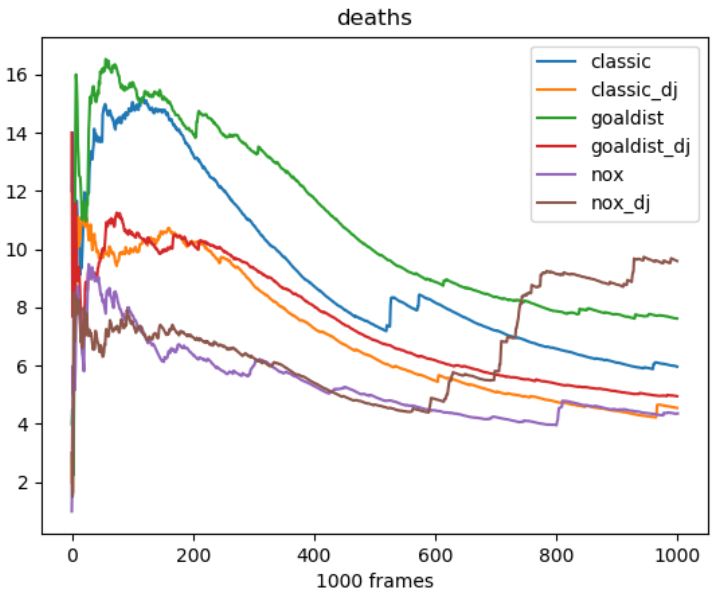

FIGURE 7.3: The averaged death count for classic, goaldist, and nox for basic and doublejump.

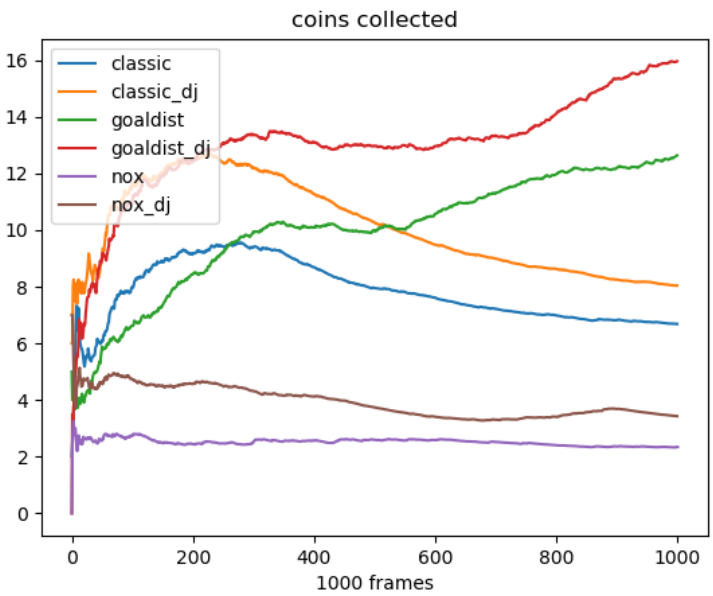

FIGURE 7.4: The averaged coin count for classic, goaldist, and nox for basic and doublejump.

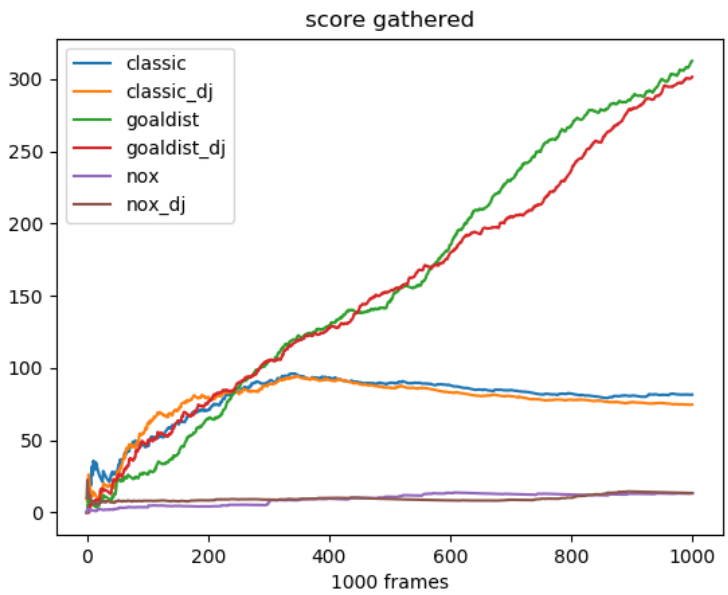

FIGURE 7.5: The averaged score for classic, goaldist, and nox for basic and doublejump.

Chapter 8

Testing

After the training, all agents were tested on new levels. Different test sets were created: basic, rndgfx (has randomized graphics and score position), bidirectional (goal can be on either side), doublejump (higher enemies and terrain changes), and long (double the length of normal levels). For each test set, 3000 worlds were created using the parameters shown in Table 8.1.

The agents were tested using the testing algorithm from [7], with a few changes to adapt it to this project. For each test, 50,000 actions were simulated. In [7], an ϵ of 0.25 is used for testing, as this improves the agent's performance. To verify this is also true for this project, three tests were run using the classic agent and an ϵ of 0.0, 0.1, and 0.25 respectively. It quickly became clear that ϵ=0.0 is bad: the agent uses only right and jump actions, and it only jumps when necessary. However, it doesn't always jump when it should. While it usually jumps when approaching enemies, it sometimes doesn't for terrain height increases, leaving it running against the wall until a new world is loaded. Also, the agent behaves completely deterministic and will always chose the same action in the same situation - meaning if it dies one time in a level, it will die in exactly the same way on all subsequent tries. Increasing the ϵ will add some randomness and therefore prevent the agent from getting stuck and also break up the deterministic behavior. The tests showed that an ϵ of 0.25 resulted in the best performance, thus this was kept from the Super Mario project.

8.1 Results

All agents were tested on two different test sets: the *basic* test set and the one specific for their experiment. Since the agents from the *basic* and the *atari* experiments don't have specific world generator parameters, they were tested on the *long* test set for their second test. Observations have shown that the agents trained with the same reward function behave similar for all test sets, also some test sets have shown commonalities between all agents tested on them. This section will shortly summarize these similarities.

The *classic* agent generally shows a strong tendency towards right actions. Actually, it always uses right actions unless it approaches an enemy or terrain

TABLE 8.1: World generator parameters for the different test sets. Values for the columns apart from basic are only specified where they differ from the basic value. Only the rndgfx test set features randomized graphics (the same ones as used for training).

Parameter	**basic**	**rndgfx**	**bd.**	**dj.**	**long**
jump	1			2	
jump_height	3				
jump_width	3				
score_position	-1	{-1, 0, 1}			
world_ size_ x	75				150
world_ size_ y	20				
max_ height_ diff	3			6	
max_ height_ terrain	10			12	
height_ change_ probability	0.2				
height_ up_ probability	0.5				
height_ down_ factor	0.01				
enemy_ spawn_ probability	0.15				
enemy_ max_ width	1				
enemy_ max_ height	3			6	
enemy_ grow_ probability	0.25				
coin_ spawn_ probability	0.05				
spawn_ area_ percentage	0.2				
spawn_ left_ probability	1.0		0.5		

height increase - then it will jump - or the ϵ-greedy policy makes it randomly take another action. It will nearly always reach the end of the world, however it often jumps over the goal and then runs against the end of the level and doesn't beat it. This happens especially often when the goal is one or two blocks right of something that makes the agent jump, as the taken right actions while in the air will carry it over the goal. Similarly, multiple higher enemies next to each other are problematic as they will have the agent jump over the first one and then crash into the next one while still in the air.

goaldist is the better *classic* agent. It also moves right most of the time, but not always. It too sometimes jumps over the goal and keeps running right against the world's end. However, if it lands close to the goal, it will start jumping a lot and moving back and forth until it eventually hits either the goal or an enemy. When jumping over the goal, it will sometimes walk back a short distance, but in other cases it won't, moving back and forth somewhere a few blocks to the right of the goal. All in all, *goaldist* looks similar to *classic*, but is way better at hitting the goal.

Agents trained with the *nox* reward functions are the worst performing agents through all the experiments. Maybe when trained for a longer time or with more importance put on the goal this would be possible, but with the parameters chosen for these experiments, the *nox* reward function is not suitable for training an AI capable of playing the game. Some *nox* agents always move back and forth close to the spawn, trying to avoid the small punishment for not moving, others ignore this punishment and take mostly *jump* and *no action* actions. They are only rarely moving through the world and thus don't beat many levels. The *nox* agents are also similarly bad across all test sets and are generally less affected by changes to the training or test sets.

Interesting are especially the results from testing on the *basic* test set for the agents not trained on the *basic* training set, but on a specific one (see Figure 8.1). The *doublejump goaldist* agent beats all other agents by a huge margin - and as the jumping behavior is written in the world files, it wasn't able to do a double jump in the *basic* test. It also has a higher death rate than the *basic goaldist* agent. Seemingly some aspect of training with the *doublejump* parameters made the agent learn better than training with the *basic* parameters. As the *doublejump* agents perform quite similar to the *basic* ones during training, it is however not clear which aspect this is.

Contrary to the training, the *rndgfx* agents actually performed similarly or even slightly better during training compared to the *basic* ones. This hints that it learns slower, but the AI is actually able to abstract from the graphics pretty well. This is especially interesting for the *goaldist* agents, as the *rndgfx* variant performed significantly worse during training but surprisingly similar in the tests, see Figure 8.2.

The *bidirectional* training set is probably the most difficult one. The agents trained on it seem to have a preferred direction and will sometimes reach the goal if it is located in this direction, but they have trouble moving in the other direction. Also, they often get stuck somewhere in the middle of the world

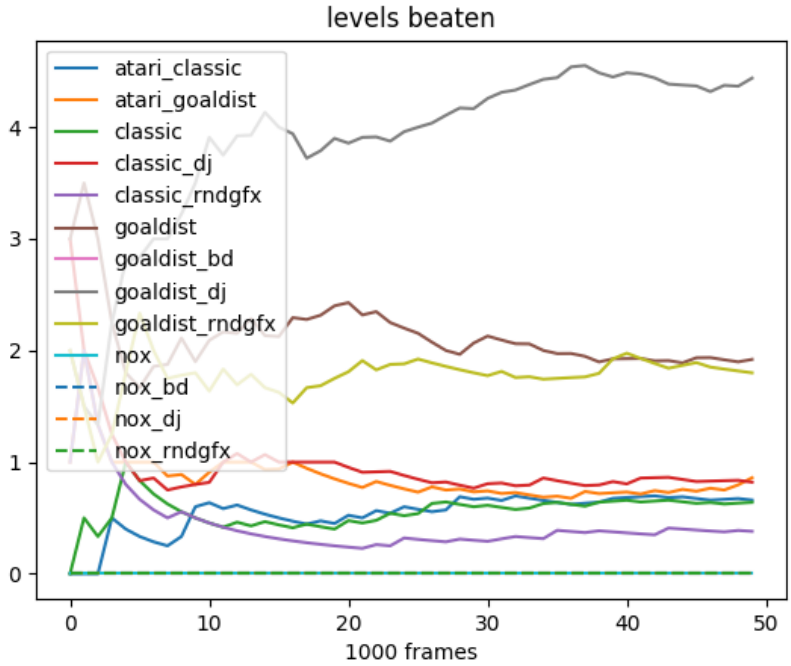

FIGURE 8.1: Averaged amount of beaten levels during testing on the basic test set.

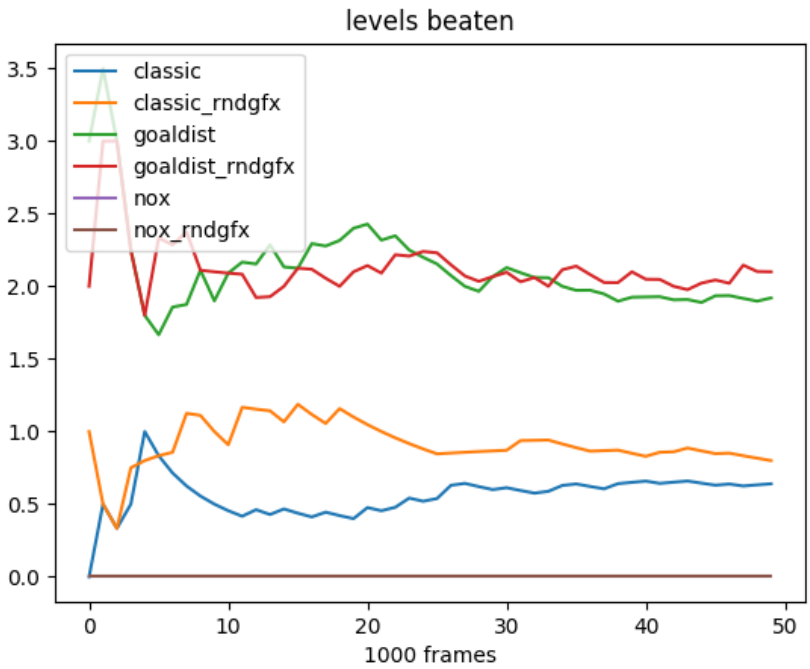

FIGURE 8.2: Averaged amount of beaten levels for the basic and rndgfx agents on their respective test sets.

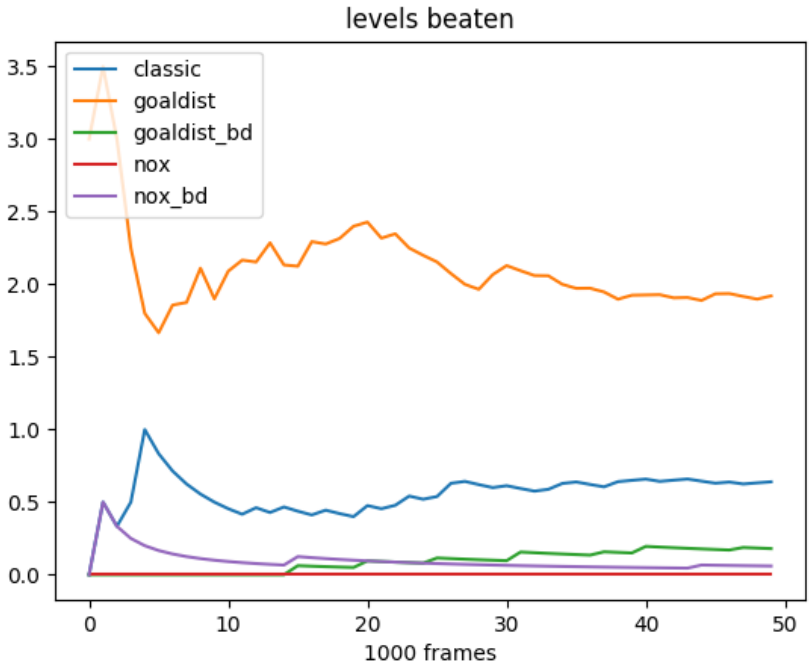

FIGURE 8.3: Averaged amount of beaten levels for the basic and bidirectional agents on their respective test sets.

moving back and forth. This results in a significantly worse performance for the tests. As Figure 8.3 shows, the *nox* agent actually improved regarding beaten levels, but this is likely to be based on sheer luck and not generally better behavior.

When observing the agents being tested on the *doublejump* test set, it became clear that it is indeed more difficult to reach the maximum possible height with a double jump than with a single jump. The agents were often stuck in front of high obstacles for some time before getting the timing right. The tests further show the same interesting behavior already noticed for the training: compared to the *basic* agents, *doublejump goaldist* improved while *doublejump classic* slightly deteriorated, see Figure 8.4.

There were no surprises for the *long* test set (see Figure 8.5): the agents trained with *nox* performed very similar compared to the *basic* test set, for the other two reward functions the agents took more coins, died more often, and beat fewer levels which also resulted in a lower score than in the *basic* test.

Further graphs depicting the test results can be found in Appendix B.

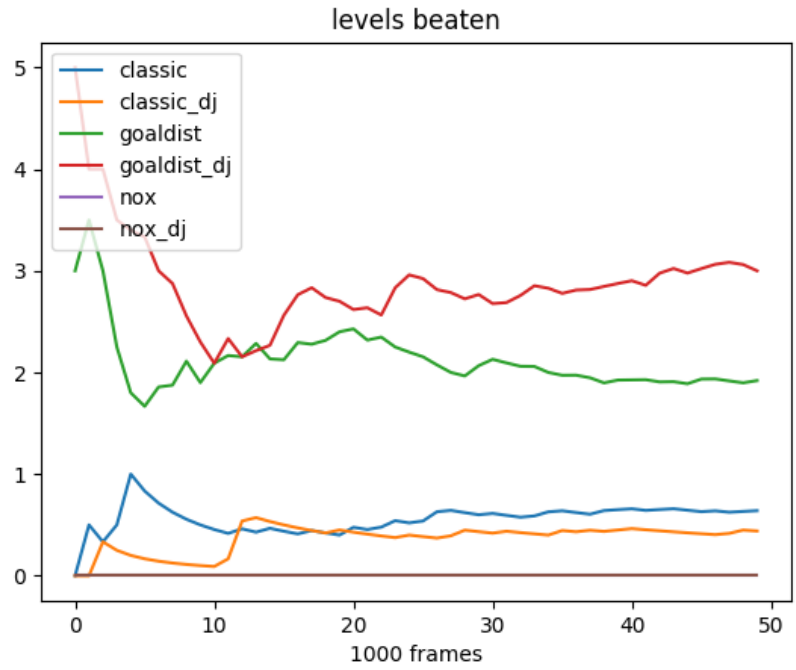

FIGURE 8.4: Averaged amount of beaten levels for the basic and doublejump agents on their respective test sets.

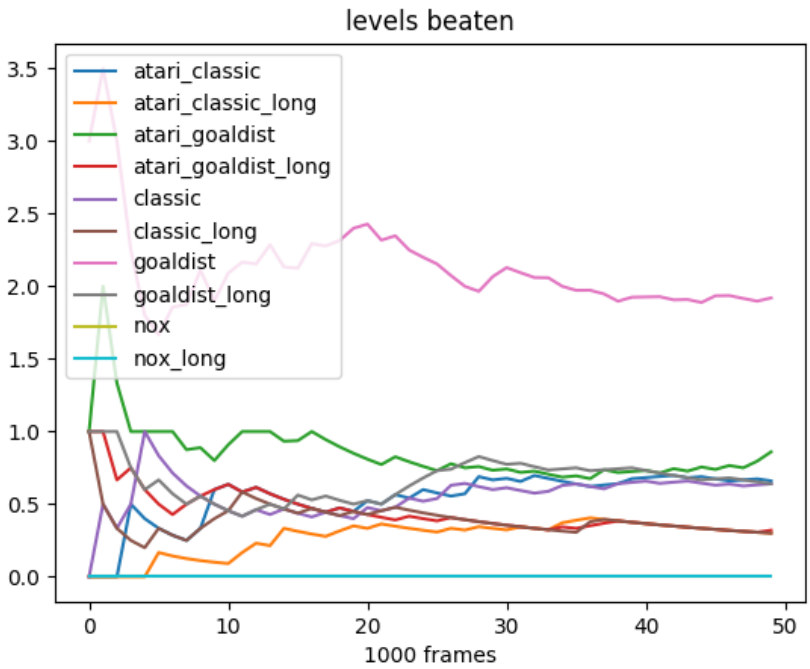

FIGURE 8.5: Averaged amount of beaten levels for the basic and atari agents on the basic and long test sets. All agents shown here were trained on the basic training set, the "_long" suffix indicates testing on the long test set.

Chapter 9

Future Research

Training an AI on a game is an area that offers a vast amount of research opportunities that are impossible to cover in one thesis. This work has shown some of the possibilities, but there are many more experiments that could be interesting to conduct. A few ideas on future research are mentioned in this chapter.

As already explained in Section 2.4.3, the world generator strongly limits the potential of the game. Hand-crafting levels or creating a more powerful world generator could greatly increase the variety and complexity of the worlds. While this doesn't need any code changes to the game, enhancing the game itself by e.g. adding new block types or alternative game modes again provides many possibilities for further experiments.

This thesis mainly focused on reward functions and slight alterations to the game parameters. There are, however, many more things that influence the agent's performance. Neither the learning algorithm, nor its hyperparameters were changed from the Super Mario project. It is unknown, whether these are the best or even good parameters or if other values would lead to way better agents. It could also be interesting to use kind of staggered training sets where the levels increase in difficulty as the agent gets better. Most of the experiments also use the same network - the one from the Super Mario project. The network most probably has a very strong influence on the agent's performance, but it is difficult to intuitively say which networks are good and which are bad for training an agent to play the game. Networks containing modules capable of remembering something could improve the performance for the *bidirectional* experiments while a network that gets a colored instead of a greyscale input could show better results for the *rndgfx* experiments. On the other hand, it would also be interesting to know what the simplest network is that still creates agents capable of playing the simple game version used in the *basic* experiment set.

The statistics file that is logged by the game while playing not only saves some numerical values like death count and collected coins, but also the names of the worlds belonging to these values. However, this was not used so far. Analyzing the similarities between worlds with similar noticeable values for one or more aspects could give more detailed information whether there are certain block formations that cause e.g. a high death count. If such

block combinations can be found, they could provide further hints on what the AI is peculiarly good or bad at and how it could be improved.

This thesis also mentioned two issues with the current implementation: the unknown effects of the selective MSE error function taken from [7] explained in Section 3.3 and the problem mentioned in 2.3.2 that the AI only gets called if it can actually do something and thus doesn't see some screens a human player would see. The first one wasn't investigated and it is unclear whether using another error function or improving the selection behavior would result in better agents. Regarding the latter issue, a *classic* agent was trained as for the *basic* experiments, but with a "dummy" AI call after the agent hit the goal before the new world was loaded. The idea was to inform the AI about the score increase and generally put more emphasis on reaching the goal. The chosen action was discarded. The agent trained with this dummy call managed to beat roughly twice the amount of levels on average as the *basic classic* agent without dummy call. This strongly suggests that putting more emphasis on the goal will significantly improve the agents. To confirm this, the experiments could be repeated with the dummy call or an improved version of the game. The importance of the goal for training and how it can be taught to the AI provide further research topics.

While the ϵ-greedy policy works in making the agent explore the worlds during training, it is a quite primitive approach. Using more advanced methods for artificial curiosity, for example as proposed in [3], could lead to better agent performance.

In [8], another reinforcement learning approach is chosen. An AI using graphs of teams of programs - called *tangled program graphs* - is trained to play games of the ALE and compared to the agents from [11]. The trained AI results in a performance similar or better than the deep Q-learning trained agents from [11] for single games and even when trained to be able to play three different games. It would be interesting to know how this approach would perform for the different experiments described in this thesis.

However, investigating deep reinforcement learning is not limited to the game proposed in this thesis. Since 2017, there exists an interface for *StarCraft 2* that was created to train AIs to play the game [19]. As a real-time strategy game, it is not only way more complex and difficult to play than the simple jump'n'run game of this project, but also significantly differs in how it is played, leading to completely different tasks for the AI to master. In the paper, they were able to train AIs to play mini-games (each mini-game representing one or more aspects of a full *StarCraft 2* game) on human level, but for the full game, the trained AI couldn't even beat the easiest AI that comes with the game. This shows the difficulty of training an AI to play a real-time strategy game and offers plenty of research potential.

Chapter 10

Conclusion

This thesis introduced a new machine learning environment in form of a highly customizable jump'n'run game. The game's task is to reach the goal while avoiding stationary obstacles and passing changes in terrain height. Coins can be collected to increase the score. The levels are stored in simple human-readable text files and new levels can be created easily either manually or using a world generator.

Subsequently, this game was used to train multiple variants of an AI with deep reinforcement learning. Three different reward functions - differing in their reward for movement - were used: *classic*, rewarding movements to the right and punishing movements to the left, *goaldist*, rewarding moving closer to the goal and punishing moving further from the goal, and *nox*, which only gives a small punishment for not moving at all. In general, *goaldist* performed best and *nox* worst, though this is partly inverted regarding deaths. In the conducted experiments, two or all three of these reward functions were used to train agents using two different neural networks and four training sets generated using the world generator. The different training sets - a primitive one (*basic*), one with different graphic tiles (*rndgfx*), one with double jump enabled (*doublejump*), and one where the goal could either be right or left of the spawn (*bidirectional*) - led to agents greatly differing in quality. Training on the *doublejump* training set produced the best agents, while the agents trained on the *bidirectional* set weren't able to reach the goal most of the time. The observations made during training were confirmed by testing all agents on two of five test sets each.

The experiments and their results not only prove the game's capability to serve as a machine learning environment able to provide challenging tasks, but also fulfill this thesis' second goal of investigating the capabilities of deep reinforcement learning. However, this topic is far too big to be exhaustively explored within one project. Therefore this thesis lastly proposed several ideas for improving the agents trained in this work as well as further investigation possibilities in the area of deep reinforcement learning.

Appendix A

Figures from Training

This appendix contains graphs for the experiments other than *basic* which were omitted in the experiments' chapters.

A.1 Experiments: rndgfx

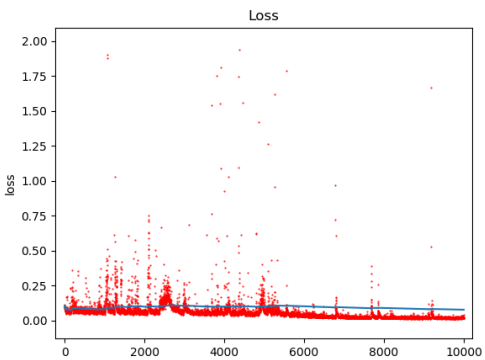

FIGURE A.1: The rndgfx classic loss. One data point per trained batch. This graph is cropped at 2.0, cutting off a few higher loss values.

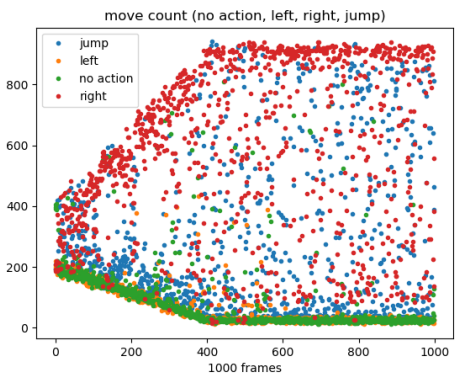

FIGURE A.2: The rndgfx classic move count. This includes randomly chosen actions from the ϵ-greedy policy.

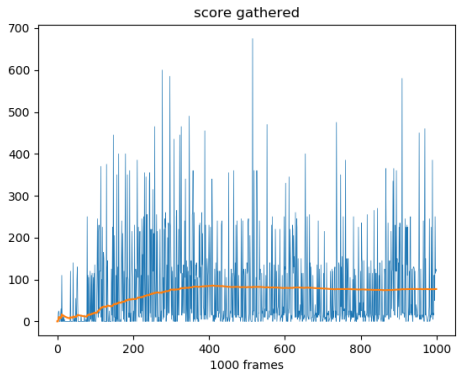

FIGURE A.3: The achieved score for rndgfx classic.

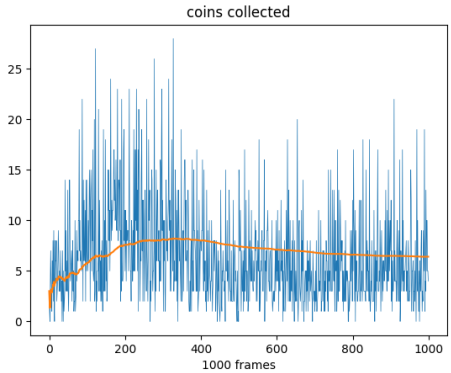

FIGURE A.4: The rndgfx classic coin count.

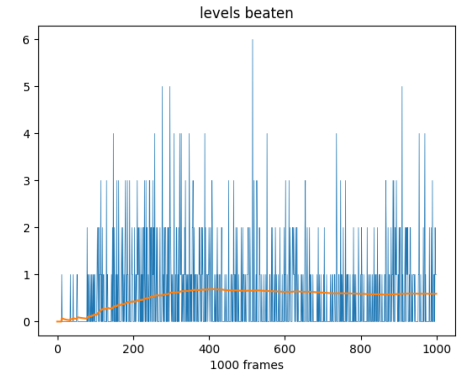

FIGURE A.5: The rndgfx classic level count.

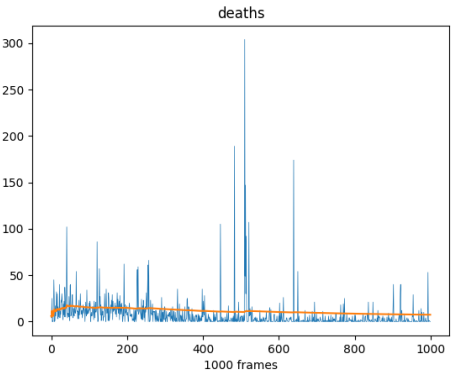

FIGURE A.6: The rndgfx classic death count.

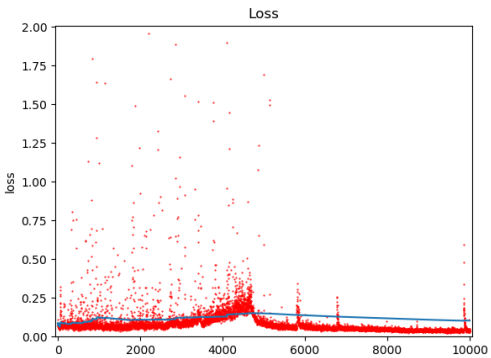

FIGURE A.7: The rndgfx goaldist loss. One data point per trained batch. This graph is cropped at 2.0, cutting off a few higher loss values.

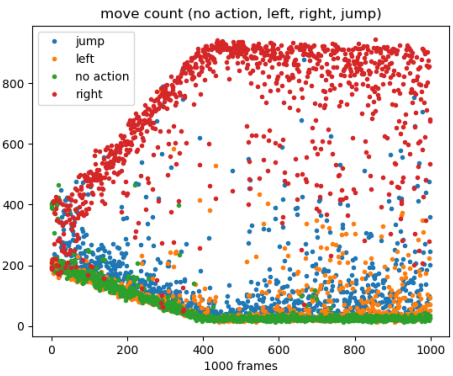

FIGURE A.8: The rndgfx goaldist move count. This includes randomly chosen actions from the ϵ-greedy policy.

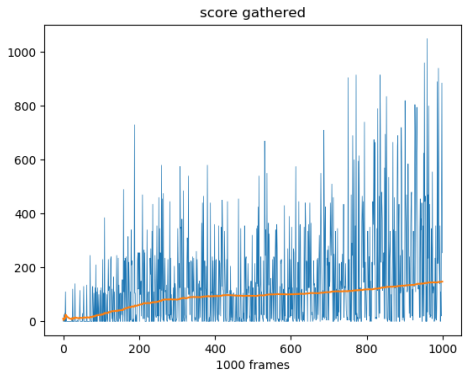

FIGURE A.9: The achieved score for rndgfx goaldist.

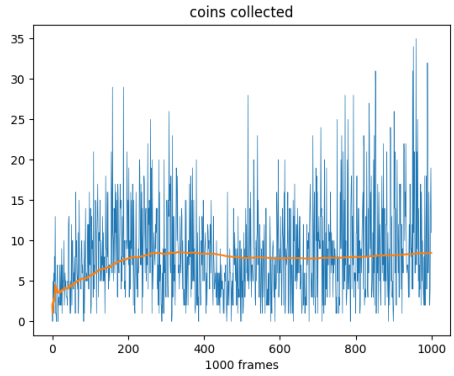

FIGURE A.10: The rndgfx goaldist coin count.

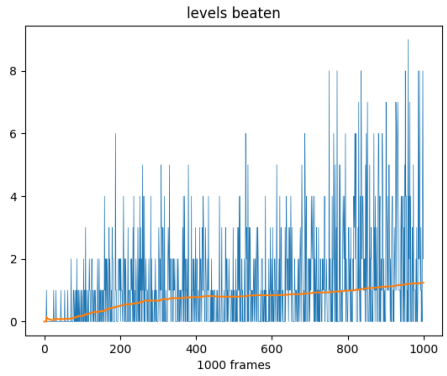

FIGURE A.11: The rndgfx goaldist level count.

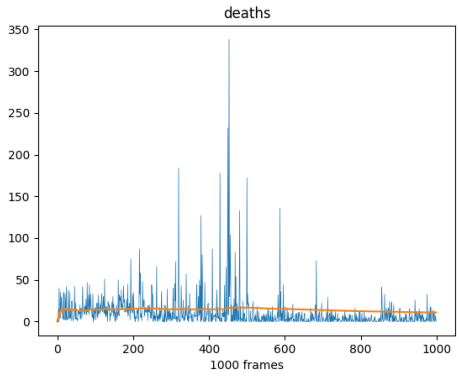

FIGURE A.12: The rndgfx goaldist death count.

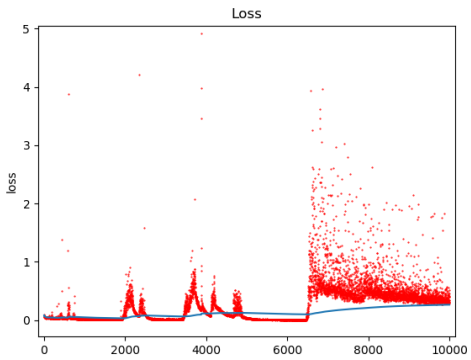

FIGURE A.13: The rndgfx nox loss. One data point per trained batch. This graph is cropped at 5.0, cutting off a few higher loss values.

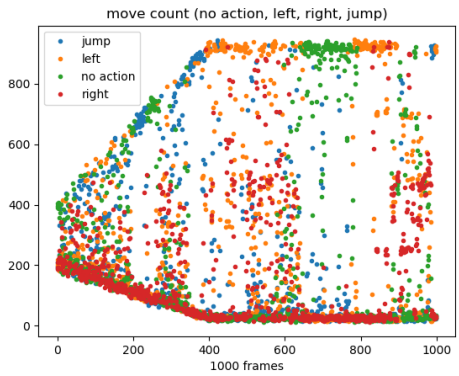

FIGURE A.14: The rndgfx nox move count. This includes randomly chosen actions from the ϵ-greedy policy.

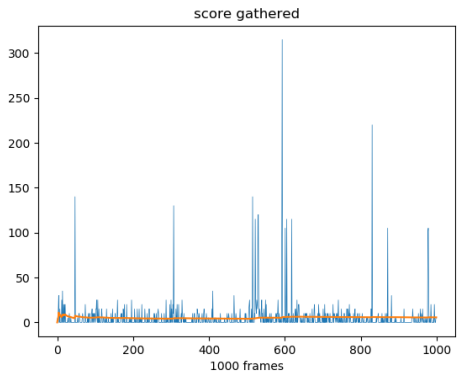

FIGURE A.15: The achieved score for rndgfx nox.

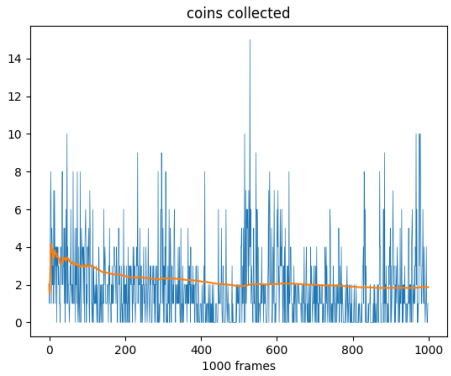

FIGURE A.16: The rndgfx nox coin count.

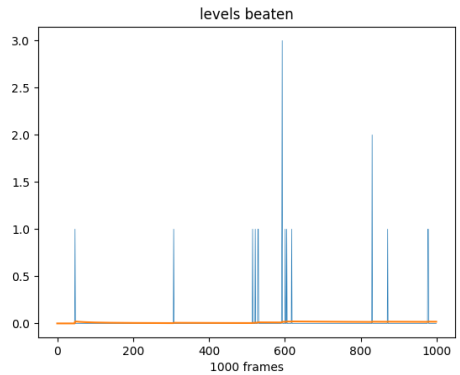

FIGURE A.17: The rndgfx nox level count.

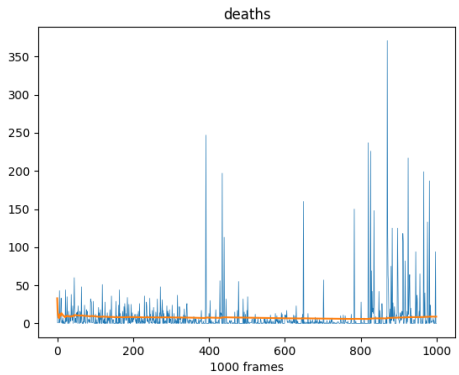

FIGURE A.18: The rndgfx nox death count.

A.2 Experiments: atari

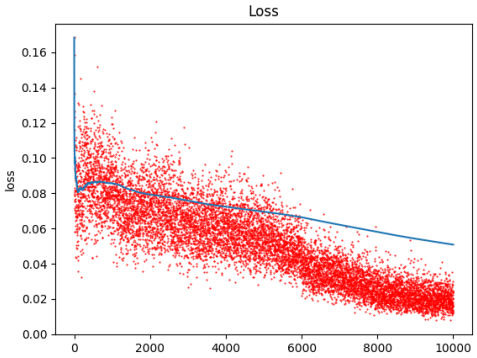

FIGURE A.19: The atari classic loss. One data point per trained batch.

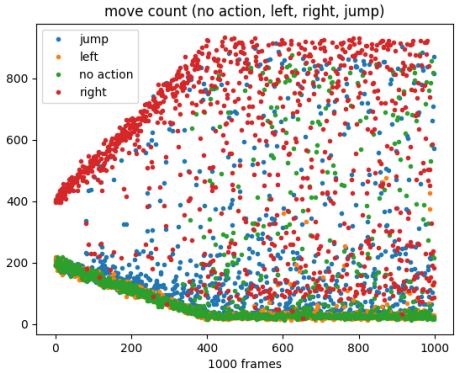

FIGURE A.20: The atari classic move count. This includes randomly chosen actions from the ϵ-greedy policy.

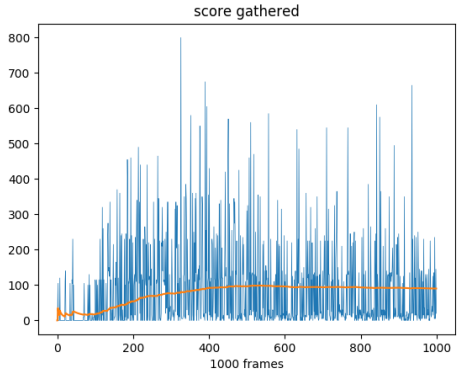

FIGURE A.21: The achieved score for atari classic.

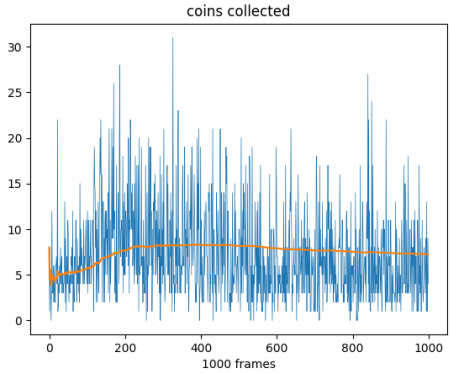

FIGURE A.22: The atari classic coin count.

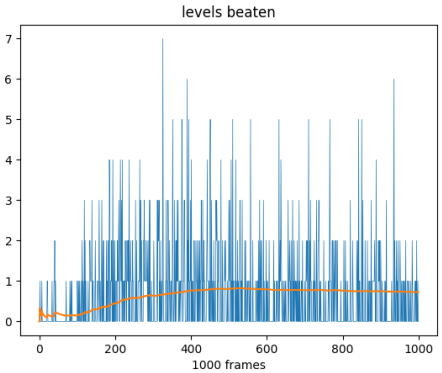

FIGURE A.23: The atari classic level count.

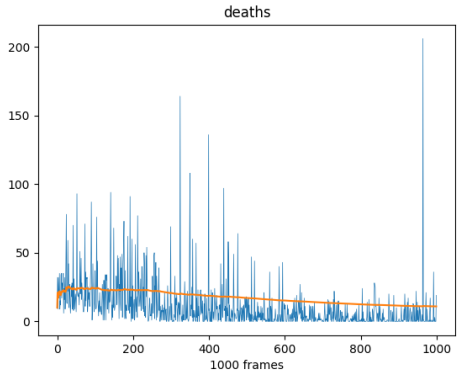

FIGURE A.24: The atari classic death count.

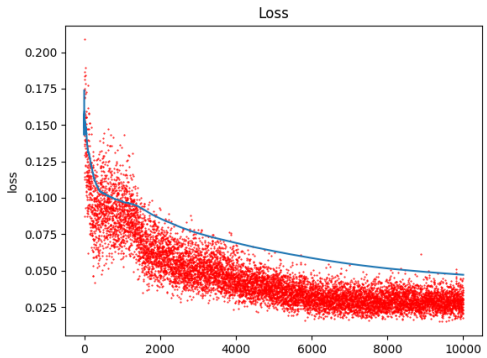

FIGURE A.25: The atari goaldist loss. One data point per trained batch.

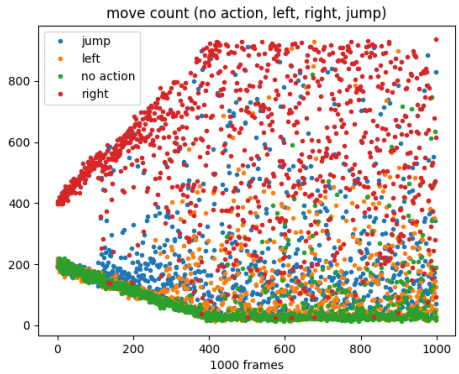

FIGURE A.26: The atari goaldist move count. This includes randomly chosen actions from the ϵ-greedy policy.

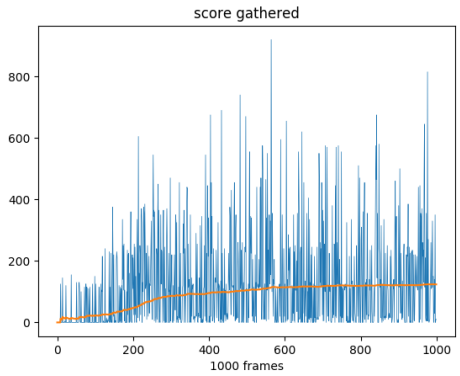

FIGURE A.27: The achieved score for atari goaldist.

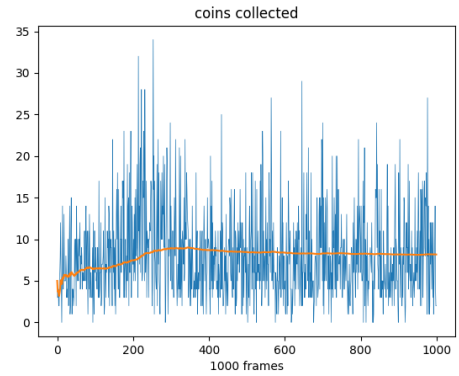

FIGURE A.28: The atari goaldist coin count.

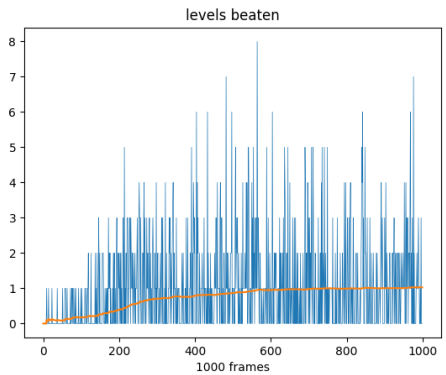

FIGURE A.29: The atari goaldist level count.

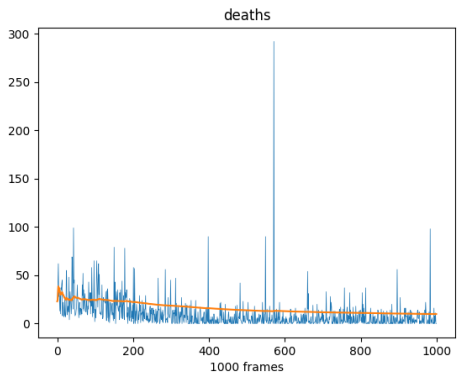

FIGURE A.30: The atari goaldist death count.

A.3 Experiments: bidirectional

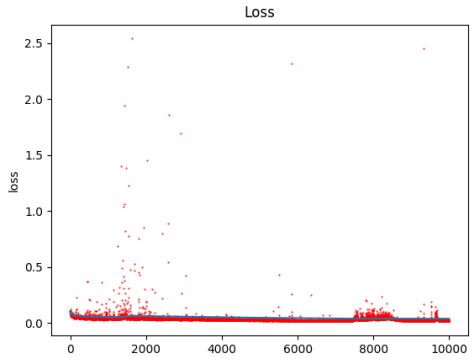

FIGURE A.31: The bidirectional goaldist loss. One data point per trained batch.

FIGURE A.32: The bidirectional goaldist move count. This includes randomly chosen actions from the ϵ-greedy policy.

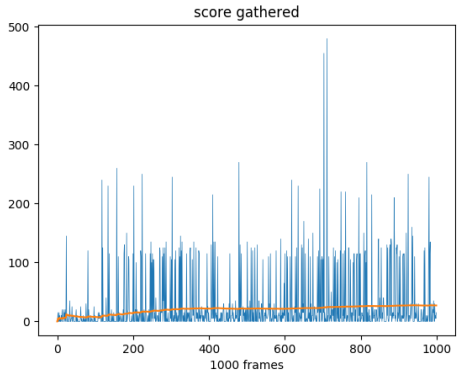

FIGURE A.33: The achieved score for bidirectional goaldist.

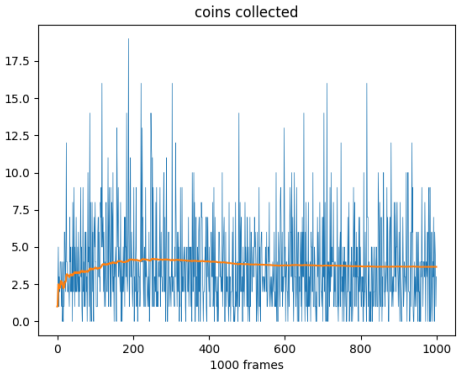

FIGURE A.34: The bidirectional goaldist coin count.

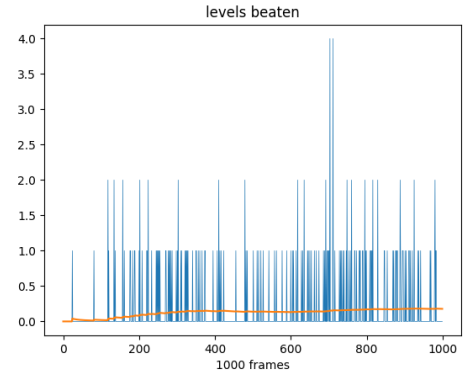

FIGURE A.35: The bidirectional goaldist level count.

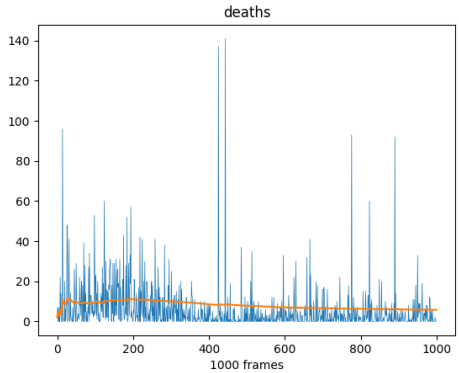

FIGURE A.36: The bidirectional goaldist death count.

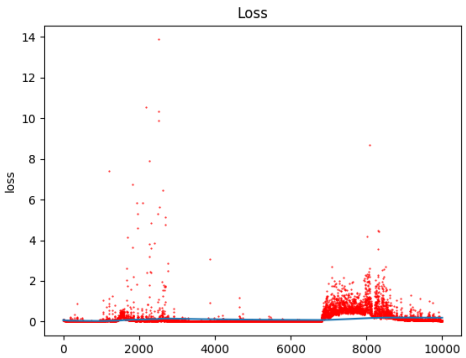

FIGURE A.37: The bidirectional nox loss. One data point per trained batch.

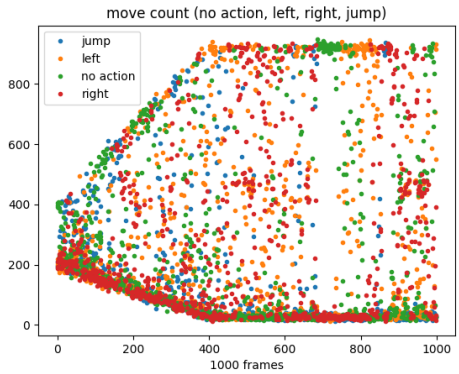

FIGURE A.38: The bidirectional nox move count. This includes randomly chosen actions from the ϵ-greedy policy.

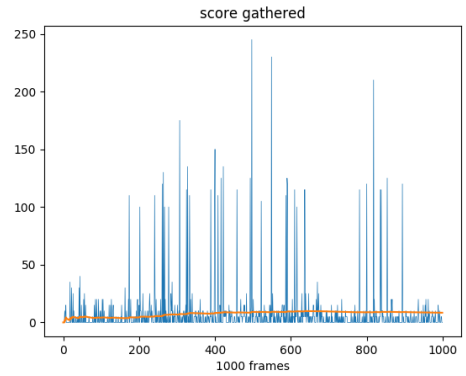

FIGURE A.39: The achieved score for bidirectional nox.

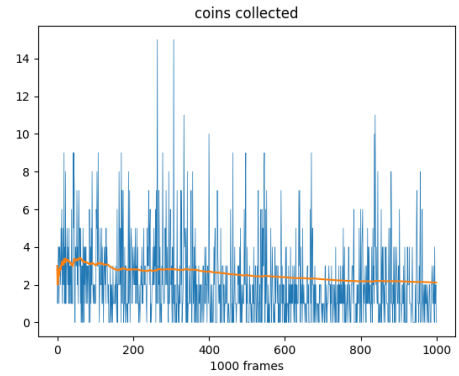

FIGURE A.40: The bidirectional nox coin count.

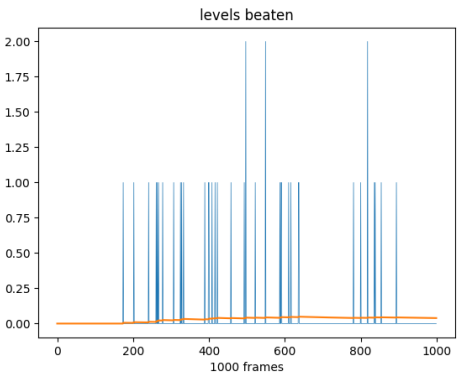

FIGURE A.41: The bidirectional nox level count.

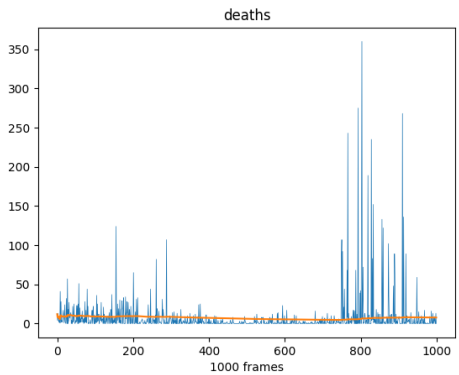

FIGURE A.42: The bidirectional nox death count.

A.4 Experiments: doublejump

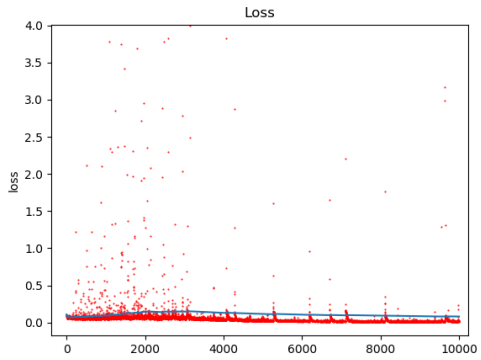

FIGURE A.43: The doublejump classic loss. One data point per trained batch. This graph is cropped at roughly 4.0, cutting off a few higher loss values.

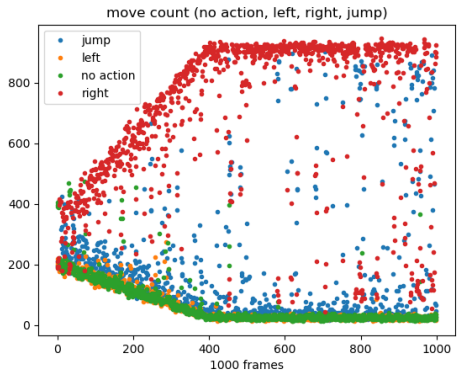

FIGURE A.44: The doublejump classic move count. This includes randomly chosen actions from the ϵ-greedy policy.

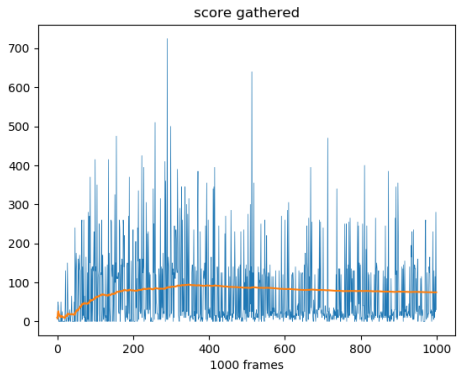

FIGURE A.45: The achieved score for doublejump classic.

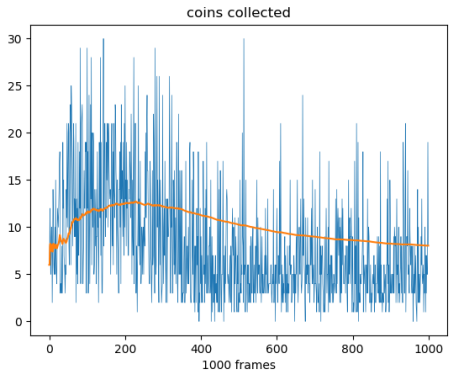

FIGURE A.46: The doublejump classic coin count.

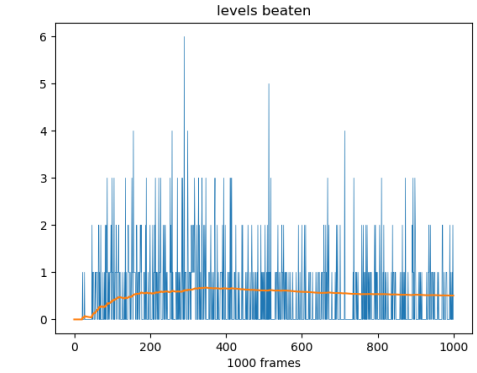

FIGURE A.47: The doublejump classic level count.

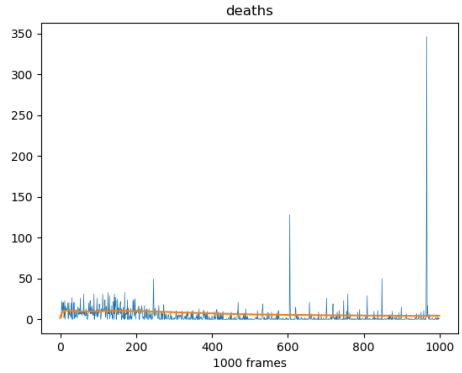

FIGURE A.48: The doublejump classic death count.

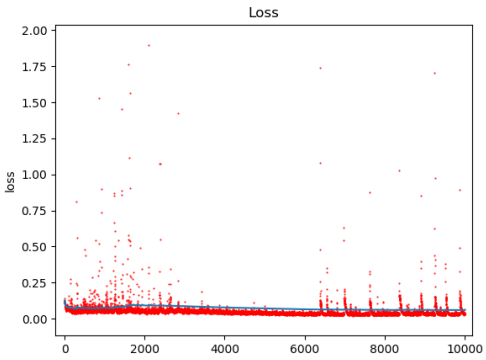

FIGURE A.49: The doublejump goaldist loss. One data point per trained batch. This graph is cropped at 2.0, cutting off a few higher loss values.

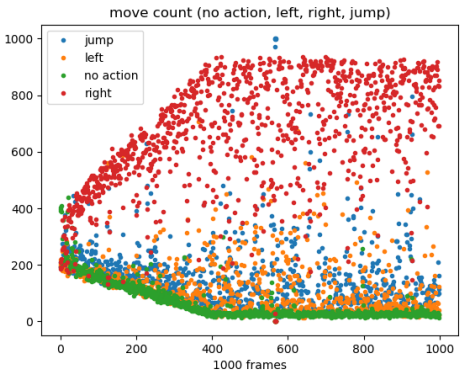

FIGURE A.50: The doublejump goaldist move count. This includes randomly chosen actions from the ϵ-greedy policy.

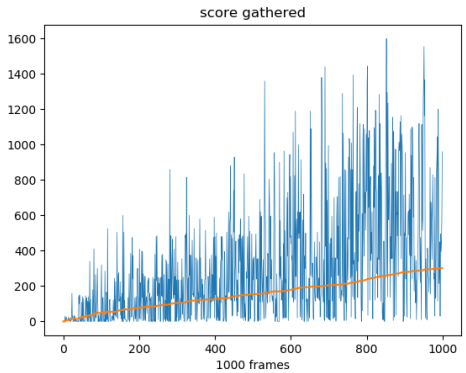

FIGURE A.51: The achieved score for doublejump goaldist.

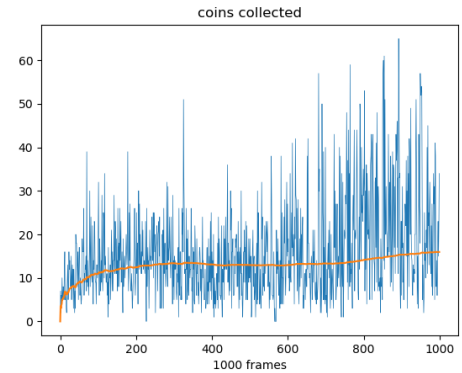

FIGURE A.52: The doublejump goaldist coin count.

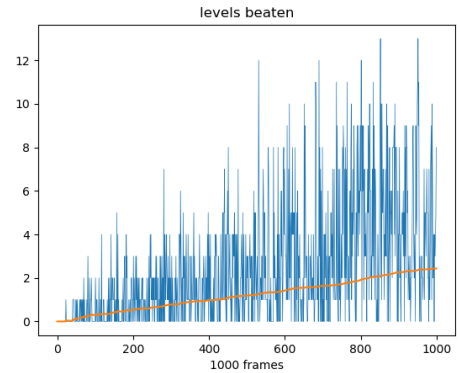

FIGURE A.53: The doublejump goaldist level count.

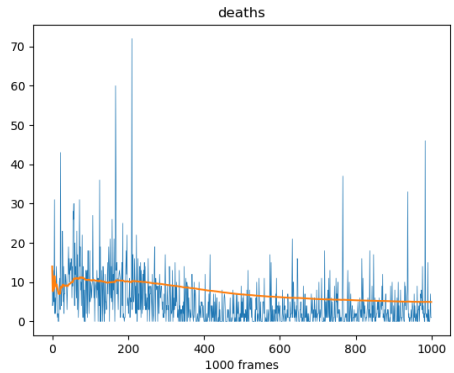

FIGURE A.54: The doublejump goaldist death count.

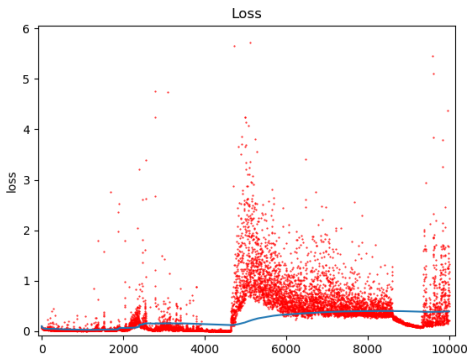

FIGURE A.55: The doublejump nox loss. One data point per trained batch. This graph is cropped at 6.0, cutting off a few higher loss values.

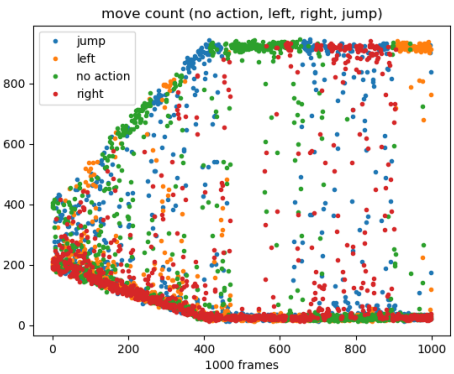

FIGURE A.56: The doublejump nox move count. This includes randomly chosen actions from the ϵ-greedy policy.

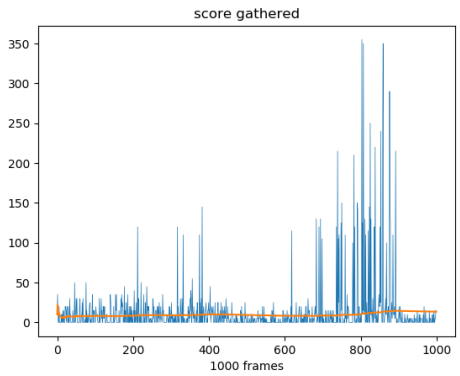

FIGURE A.57: The achieved score for doublejump nox.

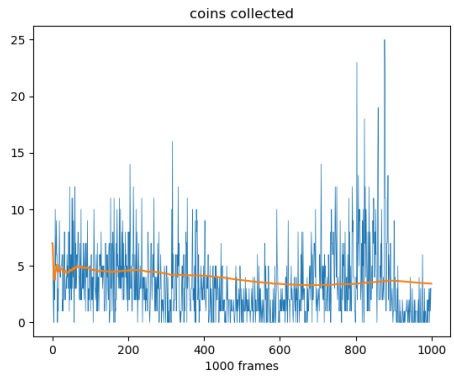

FIGURE A.58: The doublejump nox coin count.

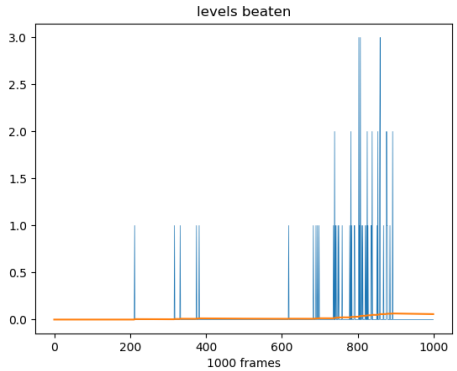

FIGURE A.59: The doublejump nox level count.

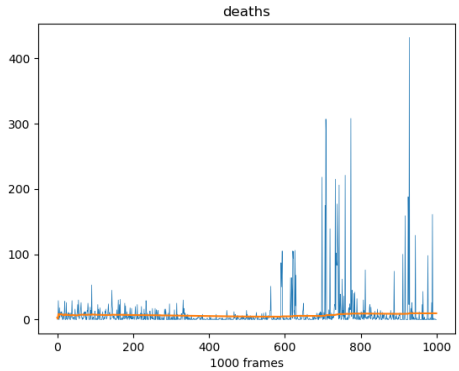

FIGURE A.60: The doublejump nox death count.

Appendix B

Figures from Testing

This appendix contains figures comparing the different agents that were omitted in Chapter 8. Figures B.1 to B.3 show the results for testing on the *basic* test set, the other figures compare the agents trained and tested on the *basic* sets to the ones trained and tested on experiment specific sets.

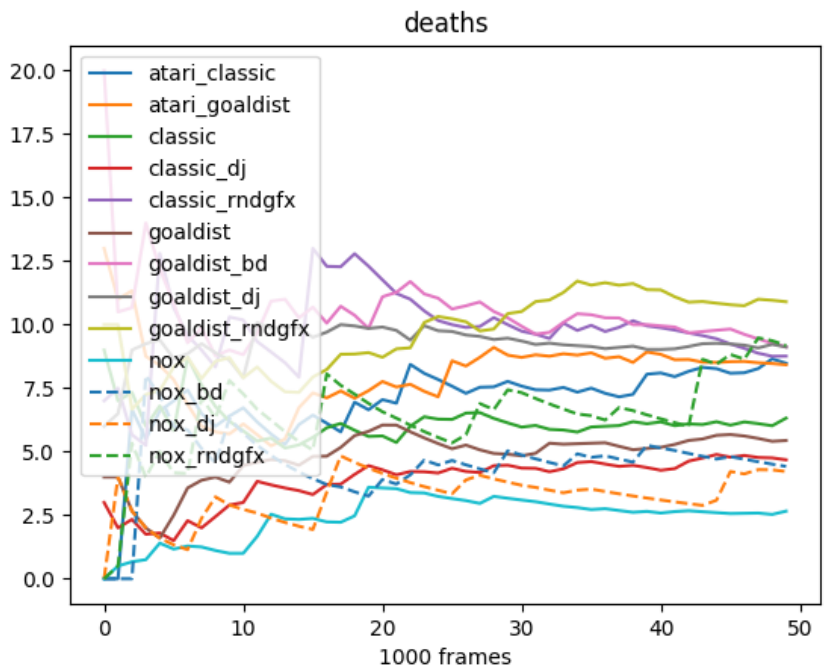

FIGURE B.1: Averaged death count on the basic test set.

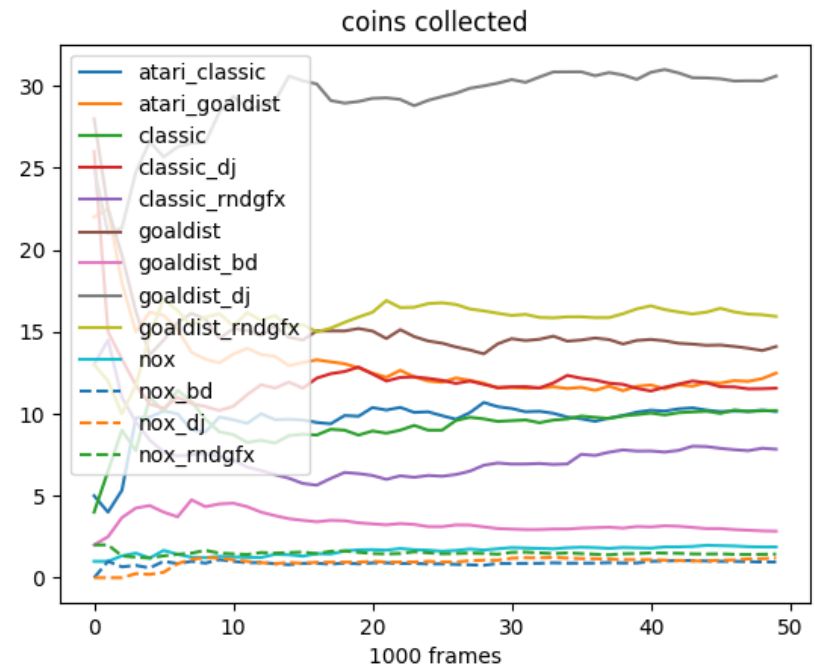

FIGURE B.2: Averaged amount of taken coins on the basic test set.

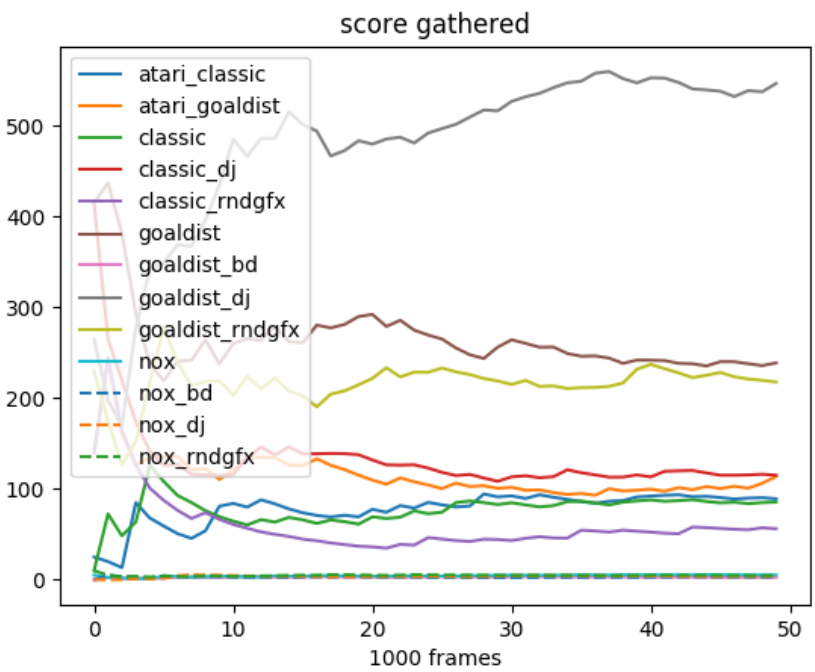

FIGURE B.3: Averaged score on the basic test set.

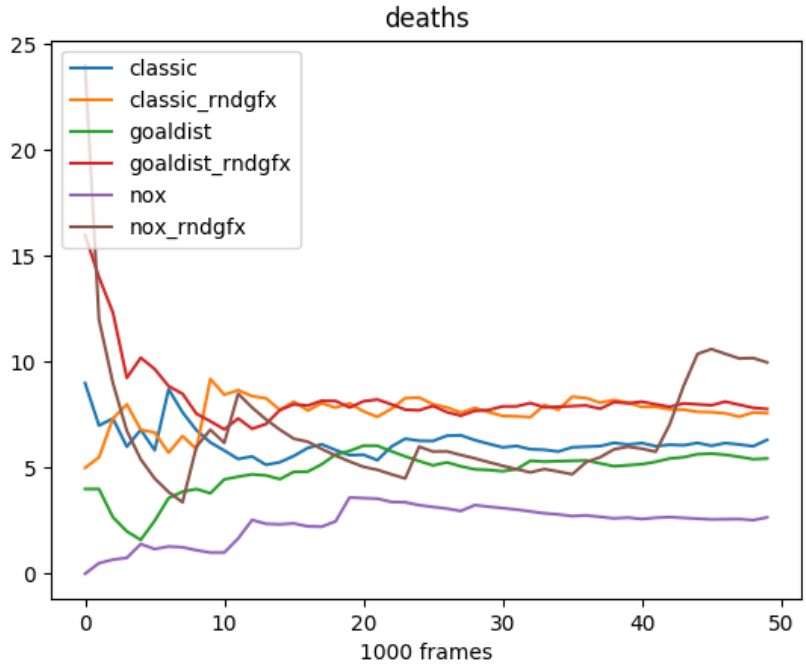

FIGURE B.4: Averaged death count for the basic and rndgfx agents on their respective test sets.

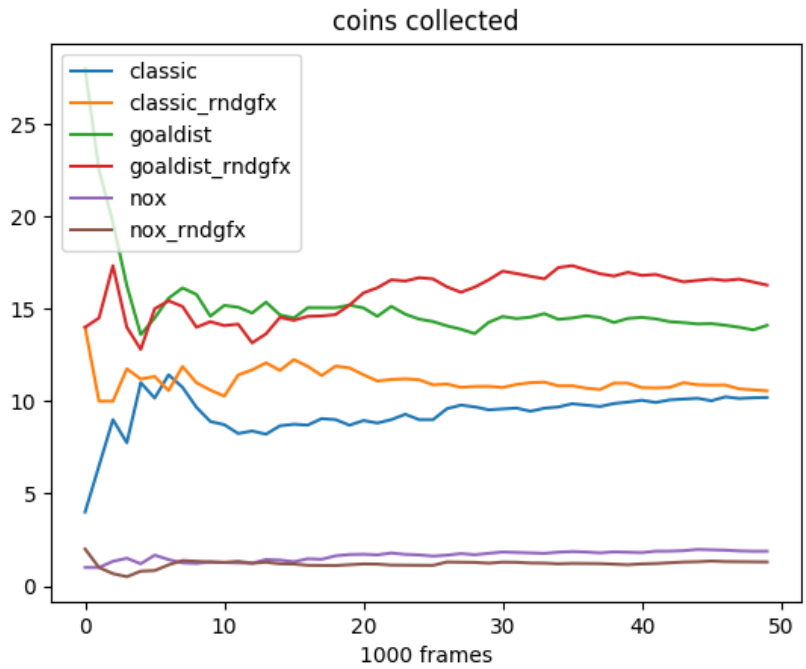

FIGURE B.5: Averaged amount of taken coins for the basic and rndgfx agents on their respective test sets.

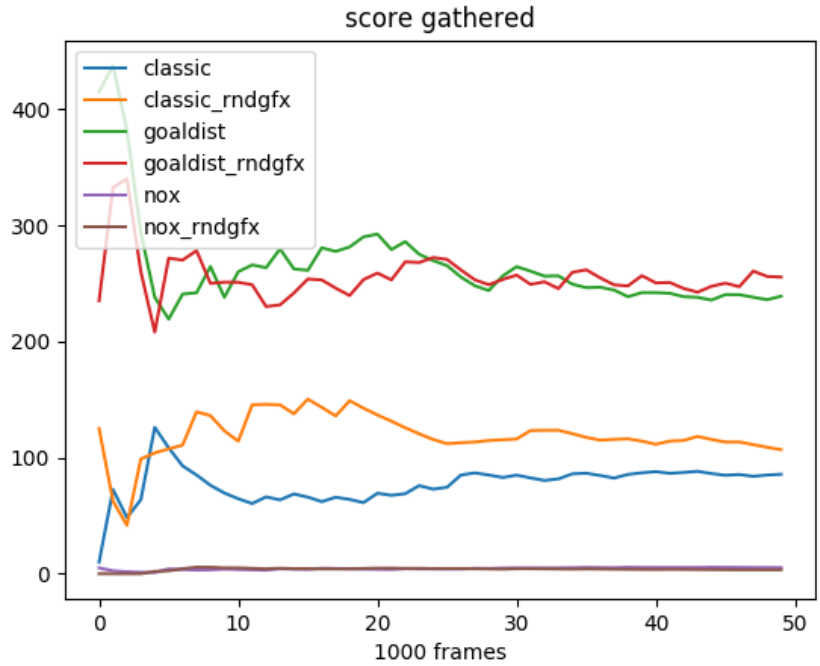

FIGURE B.6: Averaged score for the basic and rndgfx agents on their respective test sets.

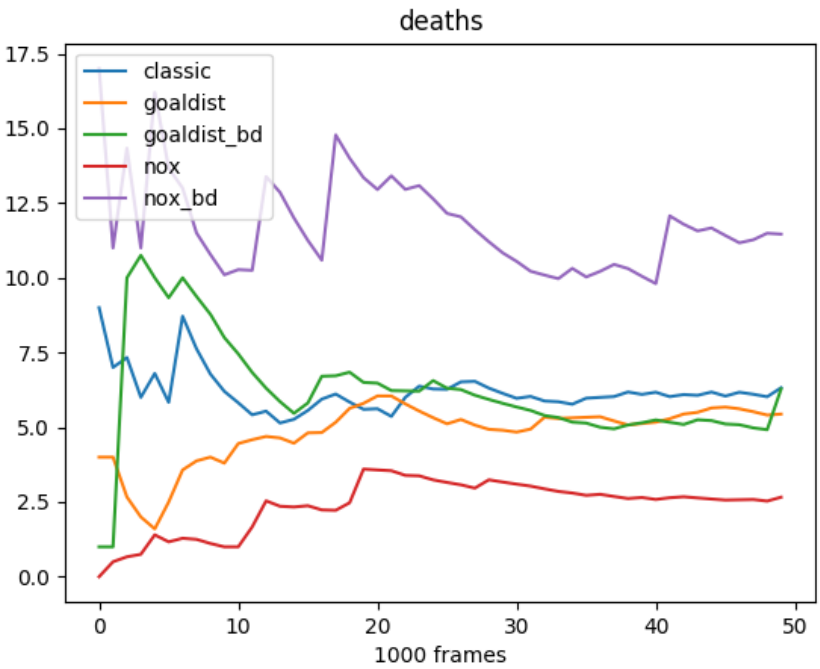

FIGURE B.7: Averaged death count for the basic and bidirectional agents on their respective test sets.

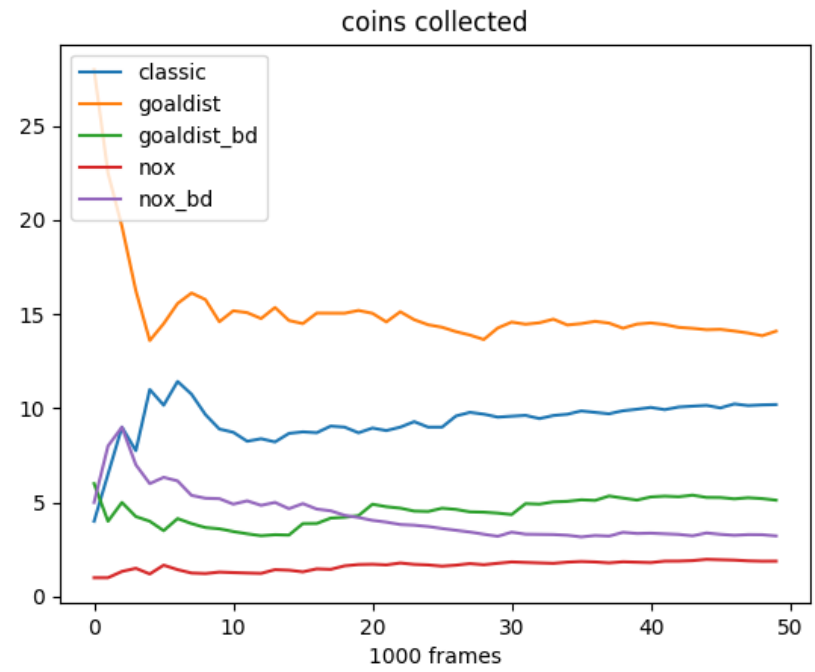

FIGURE B.8: Averaged amount of taken coins for the basic and bidirectional agents on their respective test sets.

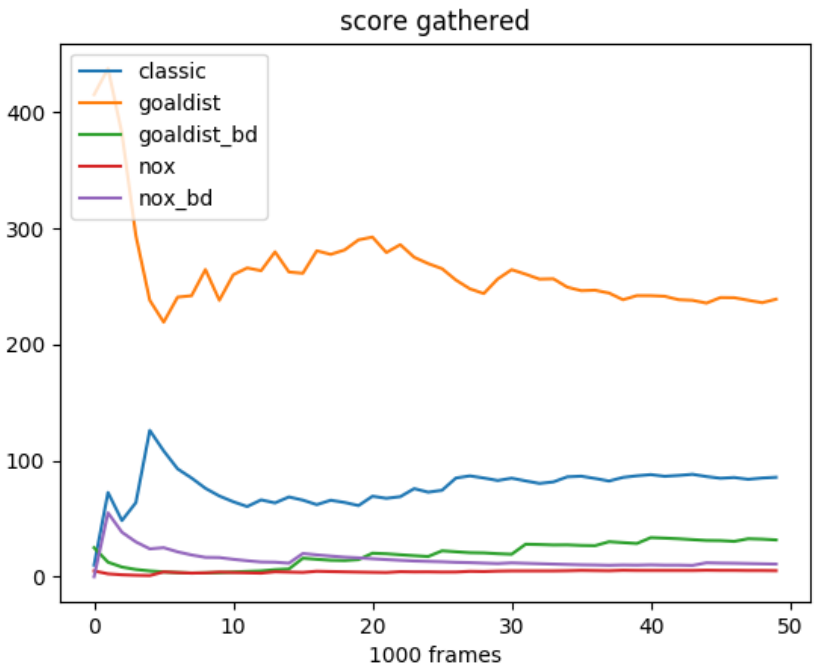

FIGURE B.9: Averaged score for the basic and bidirectional agents on their respective test sets.

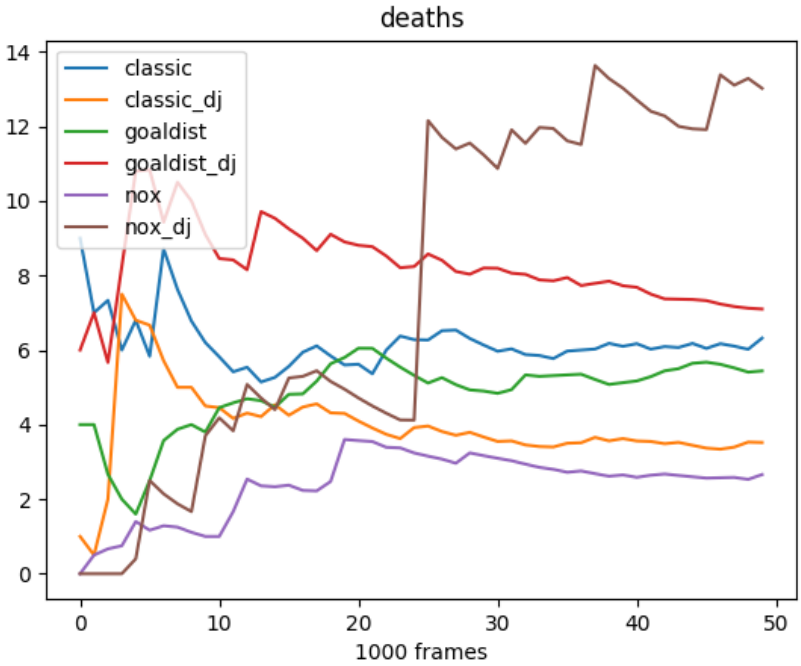

FIGURE B.10: Averaged death count for the basic and double-jump agents on their respective test sets.

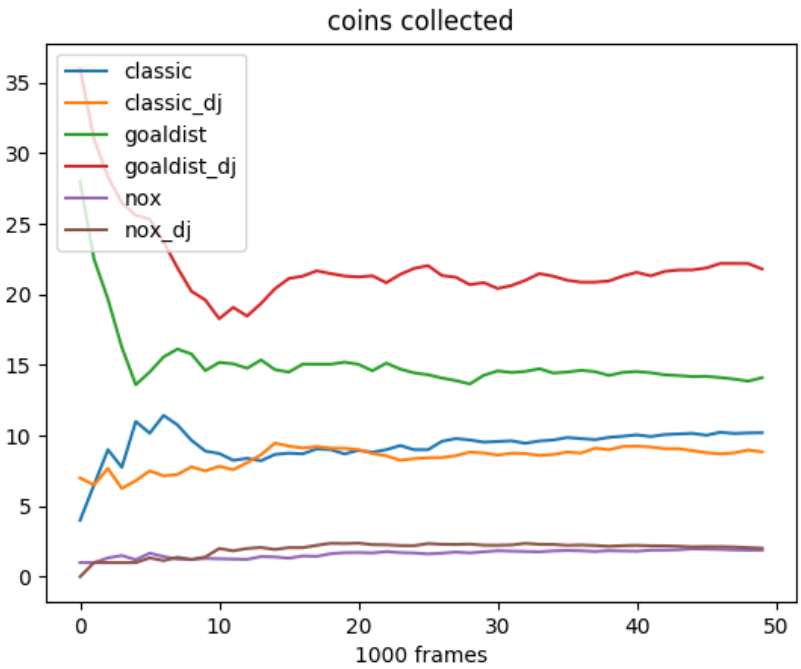

FIGURE B.11: Averaged amount of taken coins for the basic and doublejump agents on their respective test sets.

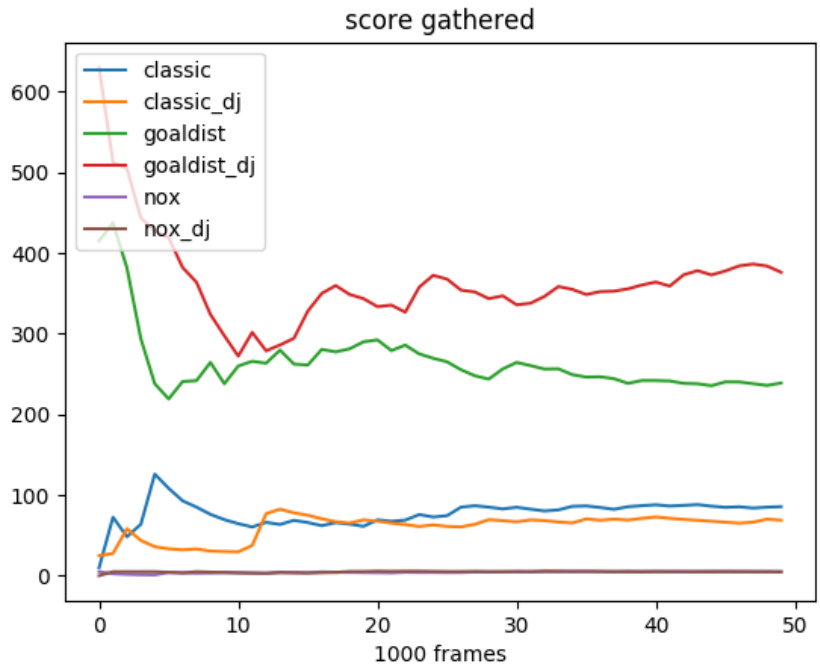

FIGURE B.12: Averaged score for the basic and doublejump agents on their respective test sets.

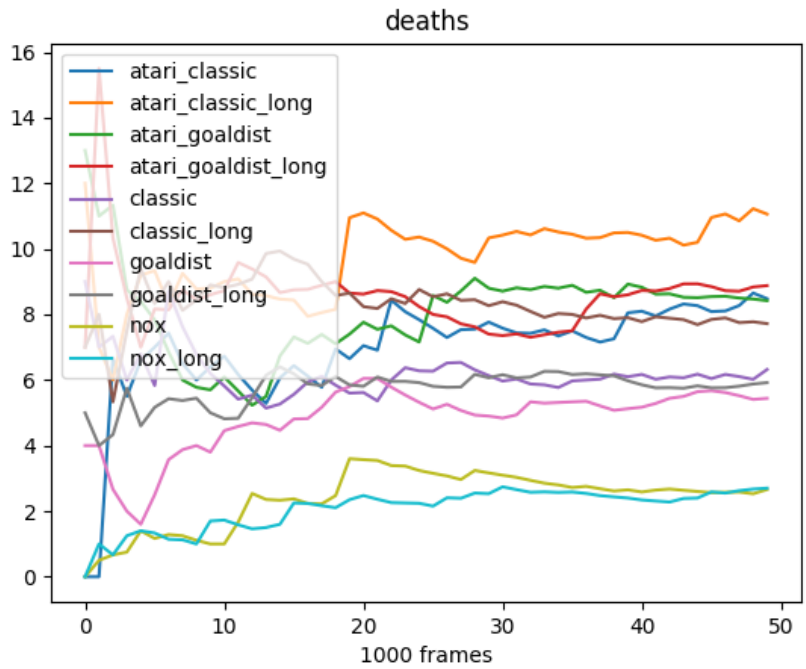

FIGURE B.13: Averaged death count for the basic and atari agents on the basic and long test sets. All agents shown here were trained on the basic training set, the "_long" suffix indicates testing on the long test set.

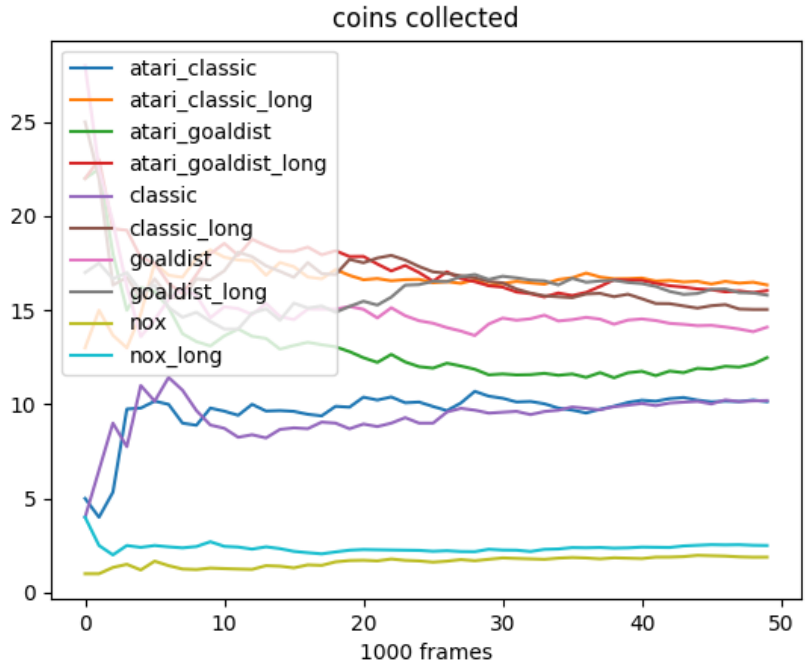

FIGURE B.14: Averaged amount of taken coins for the basic and atari agents on the basic and long test sets. All agents shown here were trained on the basic training set, the "_long" suffix indicates testing on the long test set.

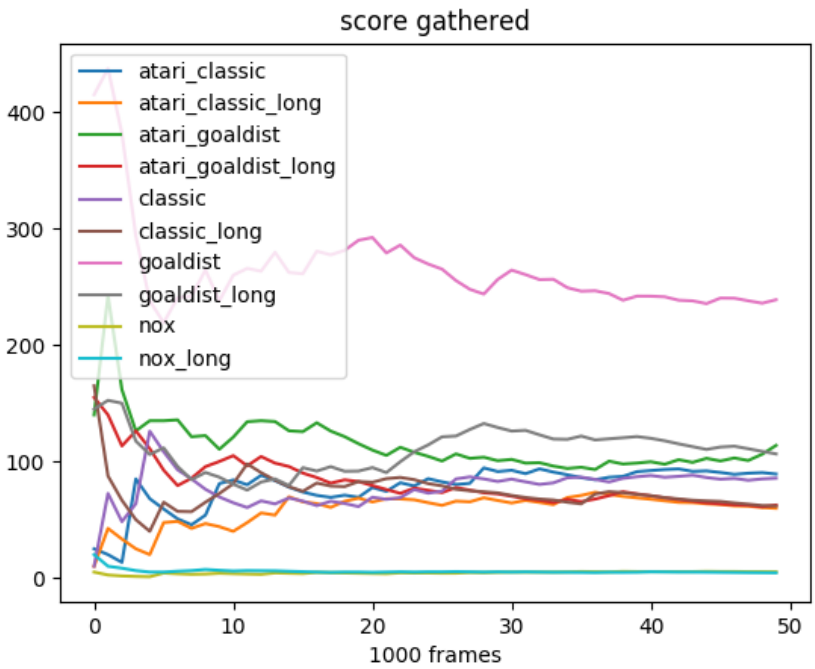

FIGURE B.15: Averaged score for the basic and atari agents on the basic and long test sets. All agents shown here were trained on the basic training set, the "_long" suffix indicates testing on the long test set.

Bibliography

[1] Bastian Bechtold. *Lunatic Python*. 2011. URL: https://github.com/bastibe/lunatic-python.

[2] Marc Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[3] Marc Bellemare et al. "Unifying count-based exploration and intrinsic motivation". In: *Advances in Neural Information Processing Systems*. 2016, pp. 1471–1479.

[4] DeepMind. *AlphaGo*. 2017. URL: https://deepmind.com/research/alphago/ (visited on 10/15/2017).

[5] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE. 2013, pp. 6645–6649.

[6] Max Jaderberg et al. "Spatial Transformer Networks". In: *CoRR* abs/1506.02025 (2015). URL: http://arxiv.org/abs/1506.02025.

[7] Alexander Jung. *Playing Mario with deep reinforcement learning*. 2016. URL: https://github.com/aleju/mario-ai.

[8] Stephen Kelly and Malcolm I. Heywood. "Multi-task Learning in Atari Video Games with Emergent Tangled Program Graphs". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 195–202. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071303. URL: http://doi.acm.org/10.1145/3071178.3071303.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[10] Steve Lawrence et al. "Face recognition: A convolutional neural-network approach". In: *IEEE transactions on neural networks* 8.1 (1997), pp. 98–113.

[11] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[12] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). URL: http://arxiv.org/abs/1312.5602.

[13] Gustavo Niemeyer. *Lunatic Python*. URL: http://labix.org/lunatic-python.

[14] A Rajavelu, Mohamad T Musavi, and Mukul Vassant Shirvaikar. "A neural network approach to character recognition". In: *Neural networks* 2.5 (1989), pp. 387–393.

[15] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[16] Andrei A Rusu et al. "Policy distillation". In: *arXiv preprint arXiv:1511.06295* (2015).

[17] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[18] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[19] Oriol Vinyals et al. *StarCraft II: A New Challenge for Reinforcement Learning*. 2017. eprint: arXiv:1708.04782.

[20] Jianwei Zhao et al. "A novel deep learning algorithm for incomplete face recognition: Low-rank-recovery network". In: *Neural Networks* 94 (2017), pp. 115–124.