

Generating Levels and Playing Super Mario Bros. with Deep Reinforcement Learning

Using various techniques for level generation
and Deep Q-Networks for playing

RUBEN NYGÅRD ENGELSVOLL
ANDERS GAMMELSRØD
BJØRN-INGE STØTVIG THORESEN

SUPERVISORS

Ph.D. candidate Per-Arne Andersen
Dr. Morten Goodwin

University of Agder, 2020

Faculty of Engineering and Science
Department of Engineering Sciences

Abstract

This thesis aims to explore the behavior of two competing reinforcement learning agents in Super Mario Bros. In video games, PCG can be used to assist human game designers by generating a particular aspect of the game. A human game designer can use generated game content as inspiration to build further upon, which saves time and resources. Much research has been conducted on AI in video games, including AI for playing Super Mario Bros. Additionally, there exists a research field focused on PCG for video games, which includes generation of Super Mario Bros. levels. In this thesis, the two fields of research are combined to form a GAN-inspired system of two competing AI agents. One agent is controlling Mario, and this agent represents the discriminator. The other agent generates the level Mario is playing, and represents the generator. In an ordinary GAN system, the generator is attempting to mimic a database containing real data, while the discriminator attempts to distinguish real data samples from the generated data samples. The Mario agent utilizes a DQN algorithm for learning to navigate levels, while the level generator uses a DQN-based algorithm with different types of neural networks. The DQN algorithm utilizes neural networks to predict the expected future reward for each possible action. The expected future rewards are denoted as Q-values. The results show that the generator is capable of generating content better than random when the generator model takes a sequence of tiles as input and produces a sequence of predictions of Q-values as output.

Acknowledgments

We want to thank our supervisors Ph.D. candidate Per-Arne Andersen and Dr. Morten Goodwin, for help and support during the span of writing this thesis. We received much helpful information, and anytime we asked questions, we would always receive useful feedback. We also want to thank Mr. Haoi Nhien Vu for allowing us to use his Overleaf template for our master's thesis.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goal	2
1.2.1	Field of research	2
1.3	Problem statement	2
1.4	Contributions	3
2	Background	5
2.1	Reinforcement Learning	5
2.1.1	Q-learning	6
2.1.2	Deep Q-Networks	7
2.1.3	Experience replay	8
2.1.4	Recurrent Neural Networks	8
2.1.5	Generative Adversarial Networks	10
2.1.6	Procedural Content Generation	12
2.2	Related work	12
3	Methods	27

3.1	Generative Playing Networks	27
3.2	Experiments	28
3.3	Mario agent	30
3.4	Level generator	33
3.4.1	Generator network model	33
3.4.2	Map generation	35
3.4.3	Experience gathering	39
3.4.4	Reward function	41
3.4.5	Generator training	44
3.5	Hyperparameters	48
4	Experimental results	50
4.1	Seq2Tile generation	50
4.2	Seq2Seq generation	55
5	Discussion	60
5.1	Learning outcomes	61
5.2	Future development	62
6	Conclusion	65
References		69
A Mario observation space		A-1
B Generator action space		A-3

C Code listings **A-4**

D Experiment configurations **A-10**

List of Figures

2.1	Example model of an RL algorithm. In this model, the agent derives actions through a neural network [1].	5
2.2	Q-table featuring 11 environment states and four possible actions for each state.	7
2.3	Diagram showing connected LSTM cells [2]	9
2.4	Generative Adversarial Networks architecture [3].	11
3.4	Visualization of the LSTM network input and the new chunk of tiles on the map.	37
3.5	The generator memory contains indexes representing tiles in the order they were generated. The red field is the sliding window, while the blue field is the new tile inserted into the memory.	38
3.6	The connection between the replay memory and the transitions.	40
3.7	Generator reward function with $v' = 0.3$	43
3.8	Generator reward function with $v' = 1$	43
3.9	Generator reward function with $v' = 3$	43

4.1	Result for the Mario agent competing against the Seq2Tile generator. The x-axis shows the number of steps the Mario agent has performed and the y-axis shows the episode reward. The graph is split up as a result of the program restarting multiple times during training.	52
4.2	Unfiltered episode reward for the Seq2Tile generator.	52
4.3	Episode reward for the Seq2Tile generator with the Savitzky-Golay filter applied with window size 1,001 and polynomial degree 4.	53
4.4	Seq2Tile results with each tile labeled with its Q-value.	54
4.5	Example of random generation.	55
4.6	Result for the Mario agent competing against the Seq2Seq generator. The x-axis shows the number of steps the Mario agent has performed and the y-axis shows the episode reward. The graph is split up as a result of the program restarting multiple times during training.	56
4.7	Unfiltered episode reward for the Seq2Seq generator.	56
4.8	Episode reward for the Seq2Seq generator with the Savitzky-Golay filter applied with window size 1,001 and polynomial degree 4.	57
4.9	Seq2Seq results with each tile labeled with its Q-value.	59

List of Equations

2.1	Q-function.	6
2.2	Agent reward.	24
3.1	Epsilon decay.	39
3.2	Generator reward function.	41
3.3	Optimal Mario velocity.	42

List of Listings

3.1	Pseudocode for Mario reward function. The full method is listed in Listing C.1.	30
3.2	Action space list.	32
3.3	Seq2Tile <i>create_generator</i> method.	33
3.4	Seq2Seq <i>create_generator</i> method.	35
3.5	Pseudocode for the <i>generate</i> method in <i>level_gen</i> . The full method is listed in Listing C.2.	35
3.6	Pseudocode of the initialization of training and target lists in the <i>train</i> method. The full method is listed in Listing C.3.	44
3.7	Pseudocode for updating Q-values in the <i>train</i> method. The full method is listed in Listing C.3.	45
3.8	Pseudocode of the Seq2Tile generation model fitting. The full code is listed in Listing C.4.	46
3.9	Pseudocode of the Seq2Seq generation model fitting. The full code is listed in Listing C.5.	47
C.1	Reward function for the Mario agent.	A-4
C.2	<i>generate</i> method in <i>level_gen.py</i> .	A-5
C.3	<i>train</i> method in <i>generation.py</i> .	A-7
C.4	Fitting the Seq2Tile generation model.	A-8

C.5 Fitting the Seq2Seq generation model. A-9

D.1 Configuration file *constants.py* A-10

List of Tables

2.1	Overview of related work.	26
3.1	The most relevant hyperparameters.	49
A.1	Tiles and entities in the Mario agent’s observation space.	A-2
B.1	Tiles and entities in the generator’s observation space.	A-3

Abbreviations

A2C Advantage Actor-Critic	23, 25
AI Artificial Intelligence	i, 3, 14, 16, 25, 26, 50, 55
API Application Programming Interface	33
ASP Answer Set Programming.....	17
CIS Computational Intelligence Society	14
CMA-ES Covariance Matrix Adaption Evolution Strategy	20, 21
DNN Deep Neural Network	6, 7
DQN Deep Q-Networks	i, 7, 16, 25, 30, 63–65
DRL Deep Reinforcement Learning	3, 6, 12
FFNN Feed-Forward Neural Network.....	8, 16
GAN Generative Adversarial Networks	i, viii, 2, 3, 10–12, 20, 21, 25, 26
GPN Generative Playing Networks	25–27
GVGAI General Video Game AI	16, 17, 24, 26
IEEE Institute of Electrical and Electronics Engineers.....	14

IMPALA	Importance Weighted Actor-Learner Architectures	63
LSTM	Long Short-Term Memory	viii, 2, 8–10, 21, 22, 25, 26, 28, 33, 34, 37, 38, 44, 60, 65
MDP	Markov Decision Process	18, 20
PCG	Procedural Content Generation	i, 5, 12, 17–19, 23–25, 62
PPCG	Progressive Procedural Content Generation	17, 18
PPO	Proximal Policy Optimization	63
RL	Reinforcement Learning	viii, 2, 3, 5, 6, 18, 19, 23, 25, 61, 62
RNN	Recurrent Neural Network	8, 29
Seq2Label	Sequence-to-Label	33
Seq2Seq	Sequence-to-Sequence..	ix, xi, xii, 14, 16, 17, 25, 26, 28, 29, 34, 35, 38, 47, 55–60, A-9
Seq2Tile	Sequence-to-Tile	ix, xi, 14, 16, 17, 25, 26, 28, 29, 33, 36, 46, 50, 52–54, 56, 58, 60, A-8

1 | Introduction

Super Mario Bros. is a side-scrolling game consisting of two-dimensional levels where the player takes control over Mario and attempts to navigate towards the flagpole located at the far-right side of each level, marking the goal of the level. Each level consists of various tiles, which are their basic building blocks. In this thesis, an attempt is made to create and play levels using two deep reinforcement learning agents; one for creating the levels and one for moving Mario through the levels. This thesis attempts to lay the groundwork for further research within the topic of multi-agent reinforcement learning in Super Mario Bros. by creating a framework suited to incorporate artificial intelligence agents into an emulated version of the game. This framework is used to test how the level generator agent and the Mario agent coincide and evolve in a competition. The Mario agent attempts to traverse a generated level by moving towards the flagpole to the right. Meanwhile, the level generator attempts to create challenging, but completable levels for the Mario agent. The results show that the generator can create levels containing obstacles given enough training, although the levels are not always completable. Most of the focus in this thesis is directed towards the level generator, as reinforcement learning algorithms for controlling Mario have already obtained optimal solutions for many existing levels [4].

1.1 Motivation

Procedural content generation in video games has a lot of useful applications in assisting game designers. Some examples include level generation, sound generation, and game architecture. In *Super Mario Bros.*, the level generation aspect is especially interesting, and some research has already been conducted within this specific topic as can be seen in subsection 2.2. The existing approaches for this problem focus on generating entire levels before they can be played. In this thesis, a different approach will be explored, where the level is generated while Mario is navigating through it. The level generator should

generate new level content slightly in front of the sliding window which is visible to the player, and receive feedback derived from player actions when the player reaches the newly generated content. The motivation for this thesis is to investigate if the generator will be able to create coherent levels and how often, if at all, the generator creates impossible level segments.

1.2 Goal

The goal is to use a system consisting of two reinforcement learning agents to automatically create as challenging levels as possible, which Mario can complete without any human interference.

1.2.1 Field of research

To be able to reach this goal, research within Reinforcement Learning (RL), Long Short-Term Memory (LSTM) networks, and Generative Adversarial Networks (GAN) is essential. The RL agent generating levels will have similarities to a GAN generator in the sense that it learns from the feedback the RL agent controlling Mario provides. The RL agent controlling Mario essentially replaces the discriminator of a traditional GAN. The RL agent generating levels will also utilize LSTM to improve its ability to create meaningful sequences of tiles.

1.3 Problem statement

This thesis presents a system of two RL agents. The first agent is a level generator which will generate levels for Super Mario Bros., column by column. The second agent will be controlling Mario and try to move to the right as quickly as possible. The level generator is supposed to generate levels possible to complete and adhere to a particular difficulty. The levels should not be impossible, meaning a wall of bricks or an impossible gap in the

ground should never appear. The generator will start without any prior knowledge about the game or its goal; It will only receive rewards as feedback for its actions.

The problem statement for this thesis can be summarized in the following research question:

Is it possible to design and implement a system where a Super Mario Bros. level is generated continuously, while the level is being played by an RL agent controlling Mario?

Following the problem statement comes the hypotheses that:

1. It is possible to generate Super Mario Bros. levels using a system of two AI-algorithms to generate and test levels.
2. It is possible to train the level generator to create levels fitting the player's skill level.

1.4 Contributions

The contribution this thesis aims to provide to the state-of-the-art within Artificial Intelligence (AI) is to test if a GAN-inspired Deep Reinforcement Learning (DRL) system can both generate and evaluate levels in *Super Mario Bros.* simultaneously. This thesis can contribute to further development of the GAN method and DRL. Combining GAN with DRL systems has been done before, but the field is still relatively new, and this thesis will attempt to provide a new perspective to the research. The thesis could eventually be used to gradually develop more challenging environments for other AI systems. Dynamically changing environments can be a potential workaround to the problem of overfitting. Overfitting means that the AI memorize the training data rather than finding a general predictive rule [5]. Autonomous driving is a developing field of research within RL. If this thesis can provide an environment where the scenarios are increasingly difficult, it could provide a useful learning environment for this field. The research could potentially

be used for learning purposes for schools, more precisely a personalized learning program for each student. A personalized learning program could be especially useful for learning impaired students.

2 | Background

This chapter aims to explain some of the existing technologies and techniques within the field of RL. The most fundamental technologies used in this thesis will be explained first, followed by related work and the state-of-the-art within Procedural Content Generation.

2.1 Reinforcement Learning

Reinforcement Learning is a branch of machine learning where an agent interacts with an environment and receives a reward defined by a reward function. The agent will attempt to maximize the cumulative reward, which in most cases means it will try to predict expected future reward following a particular action and act accordingly. RL applications consist of a continuous cycle where an agent observes the state of the environment, an action is chosen and performed, and a reward is given to the agent. Based on the received reward, the agent will adjust the probabilities of performing the possible actions for the corresponding environment state¹.

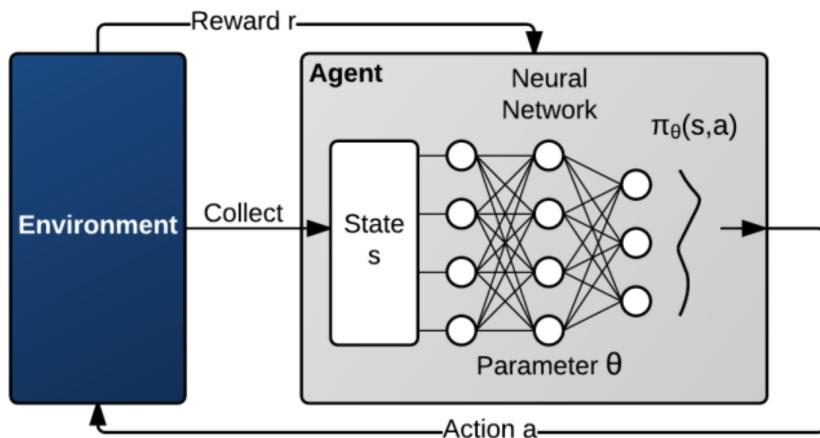


Figure 2.1: Example model of an RL algorithm. In this model, the agent derives actions through a neural network [1].

¹Depending on the implementation, the agent may also adjust action probabilities for environment states that are similar to the current state.

The RL model shown in Figure 2.1 is an example of a *Deep Reinforcement Learning* model. The difference between RL and DRL is simply that DRL applications make use of a Deep Neural Network (DNN) in the action decision process. Any RL model is essentially just a function mapping an environment state to an action. Consequently, an RL model aims to construct a list of state-action pairs, which is used to select an optimal action a given a state s .

2.1.1 Q-learning

Q-learning is a popular approach for RL applications with relatively simple environments. It equips agents with the ability to learn to act optimally in Markovian domains by experiencing the consequences of actions, without needing them to build maps of the domains [6].

Q-learning agents pick actions based on probabilistic reward. The probabilistic reward is based on the current environment state and action and calculated using a Q-function, seen in Equation 2.1.

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{R_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{max future Q}} \right) \quad (2.1)$$

Equation 2.1: Q-function.

The equation describes how Q-values are updated for state-action pairs. By observing the state s_t an agent finds itself in at timestep t , it can evaluate the *quality* of each action a it can perform in that state. This gives the agent access to an exhaustive list of *future Q-values* which it can use to determine the best action to take. The action the agent takes at timestep t is denoted a_t . This will result in the agent receiving a reward R_t . The agent observes the new environment state, and can now perform the same evaluation as earlier, where it checks the quality of each possible action. The action with the maximum

Q-value in this new state is denoted $\max_a Q(s_{t+1}, a)$. The learning rate and discount factor are hyperparameters. At this point, the agent has all the needed information to update the Q-value $Q(s_t, a_t)$, as seen in Equation 2.1. When the agent finds itself in state s_t at a later point, it will have a better idea of which actions are good and which are bad.

The basic principle of Q-learning is to use a Q-table to map each possible environment state to Q-values for each action the agent can take in the given environment state. The Q-values express the *quality* of an action, where a high Q-value indicates a favorable action, and a low Q-value indicates an unfavorable action. Given the state of the environment, a greedy agent will always choose the action with the highest Q-value. If the agent is non-greedy, it will sometimes choose a random action instead of the action with the highest Q-value. Figure 2.2 shows an example of a Q-table.

	action				
	0	1	2	3	
state	0	-14.79248	-1.49871	-2.07021	-4.58099
	1	-10.83543	-8.19600	9.68810	-9.20992
	2	-10.46631	5.16310	-1.92974	14.41184
	3	6.37114	7.03697	-13.57335	2.54439
	4	-2.51698	-1.29315	-3.76682	-10.11419
	5	14.77269	-5.01495	-5.28100	11.48026
	6	12.79048	-5.36579	5.63247	-2.03548
	7	-10.28700	11.89286	-14.71812	-11.27121
	8	2.40367	-3.46079	13.74898	-12.91274
	9	1.42715	1.80348	5.86456	-12.03239
	10	0.25854	-6.11244	-5.72707	-5.29016

Figure 2.2: Q-table featuring 11 environment states and four possible actions for each state.

2.1.2 Deep Q-Networks

One of the issues with Q-learning is that it is not scalable. However, it is possible to combine Q-learning with a DNN, which makes it scalable. Combining these two methods results in deep Q-learning, and DNNs that approximate the Q-function are called Deep Q-Networks (DQN). The DQN takes a state as input and approximates the Q-values for

each action based on the input state [7].

2.1.3 Experience replay

Experience replay is a method of storing and retrieving the experiences of an agent, without having to perform any calculations on the experiences when retrieving them. Experience replay maintains a buffer of information from transitions containing the current state, the selected action in that state, the reward received for the selected action, and the next state as a result of the selected action. The experience replay is used to train the network on the stored information in random order as opposed to training on the information in the chronological order it was experienced, which makes the learning phase separate from gaining experience. It also allows the collected experience to update the agent's network more than once, as a single transition can be selected as part of a training session multiple times.

2.1.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are often used to lay the foundation for algorithms used for sequential data and are used by Apple's Siri and Google's Assistant. It remembers its input in an internal memory, which makes it ideally suited for machine learning problems that involve sequential data.

Long Short-Term Memory

Long Short-Term Memory networks are an extension for RNNs, extending the memory such that relevant information persists over a long period of time. In ordinary Feed-Forward Neural Networks, each layer consists of a number of artificial neurons. LSTM networks replace these neurons with *LSTM cells*, shown in Figure 2.3.

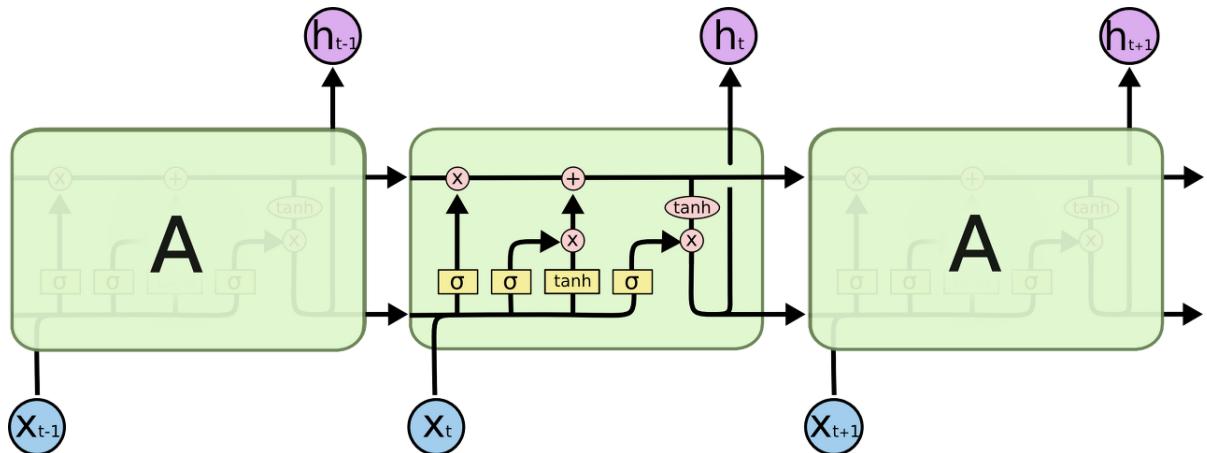


Figure 2.3: Diagram showing connected LSTM cells [2]

The figure displays a diagram of LSTM cells, where x_t denotes the input at timestep t , and h_t denotes the output at timestep t . Each cell also has a *cell state*. The diagram shows the cell state entering each cell at the top left input arrow, and leaving the cell at the top right output arrow. The cell state holds information about previous data, and each cell can manipulate its cell state by utilizing *gates*. The gates in an LSTM cell are structures used to open or close the flow of information. Each cell has three gates: the forget, input, and output gate.

The forget gate is used to decide if information should be forgotten. The forget gate takes the previous cell's output concatenated with the current cell's input and puts it through a sigmoid layer, which outputs a number between 0 and 1 for each number the cell state carries. If the forget gate outputs a 0, no information passes through it. On the other hand, if the forget gate outputs a 1, all the information is allowed to pass. The cell state is updated by performing element-wise multiplication of each value in the cell state and the forget gate output.

The input gate controls what new information is stored in the cell state. A sigmoid function selects which information should be updated. A tanh function creates new values that could be inserted into the cell state. Both functions use the previous cell's output concatenated with the current cell's input data as input. The output from the sigmoid

and tanh functions are multiplied element-wise and added to the cell state.

The output gate determines what information should be passed as output from the cell. The cell state is not updated at this gate and will be passed directly to the next cell. However, the cell state is also used to determine the output of the cell. The cell state is forked and passed through a tanh function, and the previous cell's output is passed through a sigmoid function. These two values are multiplied element by element and result in the output from the cell, h_t at timestep t [2].

The LSTM layer calculates a number of output values, dependent on how many cells the layer contains. These output values can later be connected to other neural network layers, such as dense layers, to produce predictions.

2.1.5 Generative Adversarial Networks

Generative Adversarial Networks were first introduced by Goodfellow et al. [8]. The system confines an adversarial process where two neural networks are trained simultaneously; a *generator* and a *discriminator*. The generator is a neural network taking random noise as input, and producing output that resembles the data in the dataset containing real data after training. The goal of the generator is to output data that is indistinguishable from the real data to trick the discriminator into classifying the generated data as real data. The discriminator is a neural network trained to classify input data as either real or fake. The discriminator receives a sample as input from either the real data or the generator's output. The discriminator then attempts to determine whether the input comes from the dataset or if it comes from the generator and outputs 1 for real or 0 for fake, respectively. Figure 2.4 shows the architecture of a GAN.

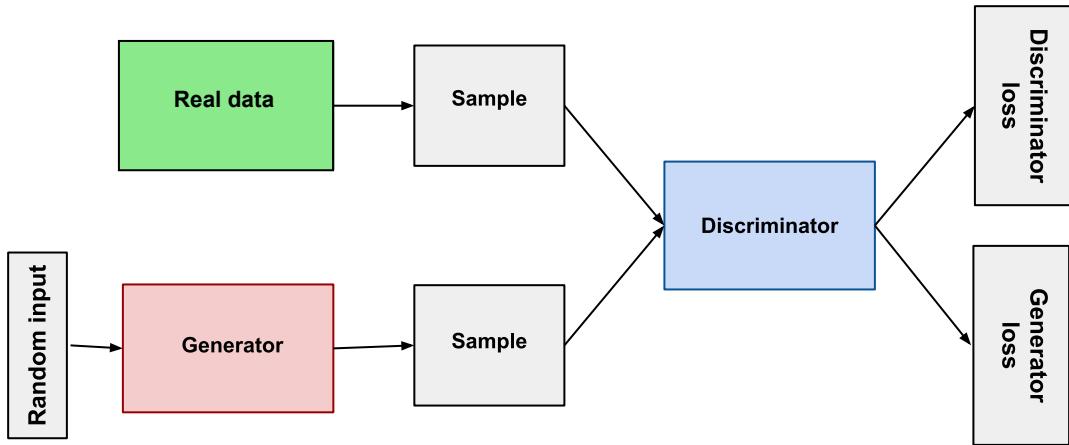


Figure 2.4: Generative Adversarial Networks architecture [3].

The generator and discriminator together constitute the GAN model. The generator receives random noise as input, and the output is a measure of how realistic the generator output is, ranging from 0 to 1. Training a GAN is synonymous with training the generator. During the generator training, the expected output label is set to 1 since the generator should produce realistic data that makes the discriminator output values close to 1. Since the generator initially produces random data, the loss is high. Therefore, the back-propagation will adjust the weights of the generator model to produce more realistic data. During the generator's training, the discriminator is set to non-trainable. The discriminator is only used as a classifier and should not change during the learning process of the generator. The system implemented for this thesis is relatively comprehensive, and many experiments were run with faulty implementations that were uncovered after the experiments' results were ready. The faulty implementations resulted in much time being cut from the available training time for the final experiments. The discriminator and generator are trained successively in a loop consisting of four main steps:

1. Set the discriminator to trainable.
2. Train the discriminator by feeding samples from the real data in addition to samples from the generator, and have the discriminator classify them.

3. Set the discriminator to non-trainable.
4. Train the generator by feeding random noise into the GAN model, and have the discriminator classify the output from the generator.

The loop runs until both the generator and the discriminator are unable to improve any further [9].

2.1.6 Procedural Content Generation

Procedural Content Generation (PCG) is a name for various methods that generate content using computer procedures, or algorithms. The content is generated using a random or pseudo-random process, resulting in an unpredictable range of possible content. In terms of games, the content can be levels, maps, game rules, textures, and stories [10].

In 2016 a book that would accompany all research fields within this definition along with the keyword "games" was released. In this book, there is detailed research, including level generation in *Super Mario Bros.*, or commonly an open-source alternative *Infinite Mario Bros.* [11].

2.2 Related work

In this section, related work will be discussed initially, before gradually moving towards state-of-the-art within PCG. PCG is a somewhat broad subject, primarily due to the popularity of video games taking advantage of PCG. Many techniques exist for PCG, and it can be adopted in many forms. However, this thesis focuses on PCG through DRL. As of writing this thesis, PCG, combined with DRL, is a relatively unexplored field of research with much potential. A central figure in the PCG for games community is Dr. Togelius at New York University [12].

Procedural Level Design for Platform Games

This paper describes a procedural level design algorithm identifying and grouping pre-existing content based on the effect the content has on the player agent. The level design algorithm will then, based on the grouping of the content, be able to generate a level with some complexity and difficulty associated with it. The content is represented in a four-layer hierarchy and focuses on the repetition, rhythm, and connectivity of the four layers. The game's physics engine allows the level generator to understand how the game works and possibilities the player has, which can then be used to calculate the difficulty of the level.

Pattern recognition is an essential aspect of the level generator. It starts with a list of all the components available, as well as the start and endpoint of a cell. A cell is an area of the level where the size is determined by the physical space in addition to some degree of randomness. The cell is generated from the components, and their set parameters are used in a hill-climbing algorithm. Hill-climbing is a technique where the generator is always attempting to reach the highest peak, meaning the most optimal solution. In order to do this, it uses the components' parameters to find the best match and sub-dividing components between the start and finish points. The next cell or cells are then selected based on predetermined values for the level.

The paper suggests personalized content generation based on player action, using this kind of level generation. However, it has not been implemented by the time of writing the paper. The authors did not find any other paper depicting the development of personalized content, so they had nothing to compare with the personalized content generation.

The results of the research show that the algorithm works, and it is capable of generating patterns successfully. However, an algorithm for building cell structures had not been implemented at the time the paper was written. [13].

The work shown in [13] by Compton et al. is based on patterns and rules. The level

generator does not learn to generate levels. It instead uses predetermined parameters to generate levels. In this thesis, Sequence-to-Sequence (Seq2Seq) and Sequence-to-Tile (Seq2Tile) is used to generate levels. The generation is dynamic and not based on predetermined parameters.

The 2010 Mario AI Championship

Super Mario Bros. has been a popular research field for Artificial Intelligence, and a Super Mario Bros. AI community was established in 2009.

In 2010 the Institute of Electrical and Electronics Engineers (IEEE) Computational Intelligence Society (CIS) held The 2010 Mario AI championship, where a competition was held to create a level generator, a gameplay track, and a learning track in *Infinite Mario Bros.* The winner in the level generator category used a Probabilistic Multi-pass generator [14].

Launchpad: A Rhythm-Based Level Generator for 2-D Platformers

Launchpad uses rhythm groups to generate levels, where rhythm is a way to identify the design of a level. This paper defines rhythm as a group with three main properties: type, length, and density. They are non-overlapping sets of components that encapsulate an area of challenge. The level geometry is based on the rhythm of the player actions within each rhythm group.

Launchpad is based on user input, meaning that the user decides the parameters for the rhythm. Launchpad also has a physics model that maintains information about the playing agent's capabilities in order to avoid generating obstacles the playing agent is unable to overcome. Based on the input from the rhythm decided by the player, Launchpad can generate a variety of different geometries for the level. Critics decide the final geometry.

One critic is decided by the general path the designer wants the agent to follow, and the other is a component frequency critic which analyses the components in the level compared to the probability of the component to occur. The component frequency critic uses chi-square goodness-of-fit [15] to find the level with the best component distribution to the desired style.

Launchpad works for level generation through input and rules, but the design is, as the authors admit, limited to a particular type of game and a specific playing style [16].

Automatic level generation for platform video games using Genetic Algorithms

In this paper, the goal was to generate levels for *Prince of Persia* using Genetic Algorithms and to make the algorithm general enough for other platform games.

The algorithm is based on working with levels that are divided into cells. The genetic algorithm will group generated levels with specific genotypes. If these genotypes work well, they will be more likely to appear in the following generation. The generated levels are referred to as individuals, and a fitness function evaluates these individuals.

The fitness function evaluates several aspects of each individual. The path structure uses the possible movements to evaluate whether all cells in the individual are reachable and that the path is not linear. Additionally, each cell is evaluated individually to ensure they are sensible and valid. Aesthetics and the amount of space used is also included in this evaluation. A high fitness score will make the individual more likely to stay in the gene pool. The individuals can be subject to mutation and crossover, which causes the levels to change and introduces new genotypes to the gene pool. The genetic algorithm generates the layout of the levels. Components and enemies are added in post-processing using rule-based algorithms.

The results show that the genetic algorithm is capable of generating a level that is not

straight forward and achieving a good fitness score. This paper describes a different method of generating levels, and does not generate levels based on a playing agent, but instead uses a fitness function based on rules and algorithms. Hence, the generation of levels is comparable. However, while the paper describes learning from a rule-based fitness function, the level generator introduced in this thesis is based on an AI-controlled agent. Furthermore, the level generation in the paper is genetic, while the level generators in the Seq2Tile and Seq2Seq algorithms in this thesis are based on Deep Q-Networks [17].

General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms

General Video Game AI (GVGAI) is a framework meant to design General Game Artificial Intelligence. The framework has been used for video game AI competitions.

The framework allows level generation with settings for the level generation and rule generation. FFNN Models govern both settings so that agents can evaluate the generated content.

The framework supplies the level generator with necessary information about the game for generating a level. The levels are generated in a two-dimensional matrix of characters where the characters represent game sprites. The most relevant part of this paper is the level generation, and therefore the paper will be isolated to this subject. The paper explains three methods for the level generation: constructive, search-based, and constraint-based.

The constructive method works in a rule-based manner. The framework identifies the role of sprites and uses the solid sprites to construct the outline for the level. The method also includes cellular automata, N-gram, and labeled patterns.

Search-based methods rely on simulations to test levels and verify that they are playable.

The constraint-based method uses Answer Set Programming (ASP) rules to generate levels. ASP rules are split into three specific rule types. The first type is basic rules that will keep the level's content simple; for example, one sprite per tile. The second will have game-specific rules; for example, only one flag per *Super Mario Bros.* level. The third type is additional rules to minimize the search space; for example, a maximum or a minimum number of each sprite.

The paper describes the level generators as different in efficiency and accuracy, and the level generators can generate completable levels. The constructive method is efficient but also unreliable. The search-based and the constraint-based methods take more time but use the same principle as the Seq2Tile and Seq2Seq algorithms, where the fitness function is based on the performance of an automated agent [18].

Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation

Using GVGAI, this paper attempts to utilize Progressive Procedural Content Generation (PPCG) in order to generate levels. PPCG introduces the idea of generating new levels where the learning algorithm controls the level's difficulty. The idea is that the learning algorithm will increase the difficulty of levels as the agent learns.

If the agent successfully completes a level, the algorithm will introduce a level it considers more difficult than the completed level. On the other hand, if the agent loses a level, an easier level will be presented subsequently. The difficulty changes incrementally such that all previous episodes influence the difficulty, hence the *progressive* part of PPCG.

The paper is mostly comparing the performance of the agent in four different games using four different training approaches. The four games tested are *Zelda*, *Solarfox*, *Frogs*, and *Boulderdash*. The training approaches use either a human-generated level, a random human-generated level, levels generated using PPCG, or levels generated using PCG X.

In PCG X, X ranges between 0 and 1 and denotes the difficulty of the generated level. PCG 1 means the difficulty of the level is relatively high from the beginning and stays high during the entire training process. The agents' performances in the various environments are then compared, as well as the performance of each method of level generation in each game.

PPCG is classified as a search-based method for level generation. Through its learning algorithm, it attempts to classify the difficulty of each level by looking at the distribution of elements.

The results and conclusion show that PPCG results with improved agents. The dynamic difficulty allows agents to become more capable of solving complex levels. However, in *Solarfox* and *Boulderdash*, PPCG did not achieve the maximum training difficulty. As a result, PCG performed better than PPCG. The paper also proves that it is imperative to subject RL agents to environments of varying difficulty, such that agents adapt to new environments more easily [19].

PCGRL: Procedural Content Generation via Reinforcement Learning

According to the best of the writers' knowledge, using an RL algorithm in PCG is a completely new science. Togelius et al. mention that the difference between using RL in PCG and the previous approaches is that RL in PCG searches the policy space to generate content, while other methods search the space of game content. Togelius et al. also mention that generating content through RL would also be much faster than the search-based methods.

The content is not generated as a whole level at once, but instead as an iterative task. The content can be seen as a Markov Decision Process (MDP), where the agent gets input and responds with an action for each step.

The level generation starts with a level populated by random tiles. At each step, the generator is allowed to make a small change. The generation is then judged by the system based on the level’s target goal, and the agent is assigned a reward.

Togelius et al. attempt to create a PCGRL framework, where the algorithm can be implemented for any game. In order to make this easy, the framework is broken down into three parts: Problem module, the Representation module, and the Change Percentage.

The Problem module provides all the information about the current generation tasks. An example of the information provided is the size of the level and the type of objects that can occur for *Super Mario Bros*. The module assesses the change in the quality of the generated content and determines when the goal is reached.

The representation model is responsible for defining the state space, action space, and transition function. Its role is to initialize the problem, maintaining the current state, and modify the state based on the agent’s action.

Change percentage defines how many tiles the generator is allowed to change. The amount of change the agent is allowed to do affects the training of the agent. If the change percentage is low, the generator will become more reactive.

For the experiments, the framework is implemented as an OpenAI Gym [20] interface, which makes it easy to incorporate existing RL algorithms. The *Stable Baselines* library is used to train the RL agents. This library is an improved implementation of OpenAI baselines [21]. The experiment uses Proximal Policy Optimization to train the agents.

The paper shows that the generator searches in the content generator space rather than the content space to make content generation an iterative improvement problem.

The paper concludes that the generator performed well in the Binary problem. The algorithm was also tested with *Zelda* and *Sokoban*, where it struggled to design challenging

levels, but generated a significant amount of playable levels [22].

The work of Togelius et al. shows that they generate a random map and then allow the generator to change the map according to certain values. This random map generation can be compared to the latent vector representation used in Generative Adversarial Networks. The paper also describes generating the content iteratively and compares it to a Markov Decision Process.

Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network

Volz et al. attempt to emulate the creation of game levels using Generative Adversarial Networks on the game *Super Mario Bros.* in this paper. The GAN uses Covariance Matrix Adaption Evolution Strategy (CMA-ES) to find ideal inputs to the GAN from its latent vector space.

The GAN is trained in phase 1, where the network is fed with real level samples encoded as a multi-dimensional array to train. To differentiate the tiles in the multi-dimensional array, each tile is represented as a distinct integer. The generator operates on a Gaussian noise vector and attempts to output levels using the same representation as the sample levels use. The discriminator will attempt to distinguish between real and generated levels.

Phase 2 trains the generator It takes a latent vector representing the level as input and generates a level consisting of tiles from the vector representation. However, the level is not represented in the same format as in the latent vector. The reason is that the CMA-ES introduces exploration for the generator, meaning that the vector space is searched for desirable properties and distribution of tiles.

The paper’s experiments are split into two distinct sections, one being representation-

based testing and the other being agent-based training. Representation-based training uses CMA-ES to optimize a certain distribution of tiles. This representation wants to investigate if the approach can generate a certain amount of floor tiles. It also wants to investigate if the ratio between ground tiles and enemies can be used to generate levels of multiple subsections that gradually increase in difficulty. Agent-based testing includes an agent that provides the generator with playthrough data to test the playability of the generated maps. The agent's fitness function is based on the number of jumps the agent performs and the distance it traverses in the level. The generator is discouraged from creating levels that can not be completed by using the fitness function.

The results show that the generator can generate playable maps and that the GAN based level generation is controllable. In the agent-based testing, the generator attempts to design levels optimized for a certain number of jumps. The paper points out that the fitness function was a difficult aspect of the experiments, and could use some improvements. It is also pointed out that LSTM could be used to fix broken structures [23].

Volz et al. use GAN to improve the generation of levels. The agent in this paper uses a simple fitness function to improve the map generation, whilst this thesis uses an agent that trains as the maps are generated, and the generator will, depending on the performance of the agent, adjust the generated maps.

Super Mario as a String: Platformer Level Generation Via LSTMs

In this paper, Summerville and Mateas attempt to use Long Short-Term Memory to generate levels from a corpus of *Super Mario Bros.* levels.

LSTM blocks can consist of multiple layers with multiple LSTM blocks per layer. Summerville and Mateas use three internal layers, with 512 LSTM blocks per layer. The network's input layer is one-hot encoded. The final LSTM layer goes to a *softmax* layer, which acts as a categorical probability distribution for the one-hot encoding.

The network is trained using Torch7 [24], and 200 timesteps are back-propagated at a time. The network uses dropout to avoid overfitting [25].

To allow the LSTM to have a probability distribution over possible next items and predict the most likely item, Summerville and Mateas arranged the data in sequences. However, they identified multiple drawbacks with representing data this way: the LSTM was unable to identify previously unseen slices in the data, and the size of input was drastically increased. Instead, Summerville and Mateas chose to represent the data as tiles in a sequence. The tiles are grouped, which allows the generator to separate their effects.

Summerville and Mateas identify three methods of generating the map, *Bottom-To-Top* vs. *Snaking*, *Path information*, and *Column depth*. *Bottom-To-Top* means that the level is generated from the bottom of the level to the top. *Snaking* means that the level alternates between being generated from Bottom-To-Top and Top-To-Bottom. *Snaking* proves to improve locality in the sequence. *Path information* introduces tiles representing the path of an A* agent playing the level. *Column depth* denotes how far the generator has progressed in generating a level.

These three methods can be switched on and off, and eight separate networks were run with different combinations of these methods. Only Snaking was used during testing. The network with all three methods active proved to have the lowest error. Path information proved to be very important for the network. Without it, all other methods proved worse with them active than with them inactive.

The results Summerville and Mateas present show that the playing agent is capable of completing 97% of the levels when they are generated with *Snaking*, *Path information*, and *Column depth* all active. These results come from a trained network with 4000 generated levels, 2000 from above ground seed, and 2000 from below ground seed. The generated levels were also compared to human-generated levels. In all measured aspects, Snaking-Path-Depth matched the standard deviation except in the percentage of the level taken up by the optimal path through the level, where it has a much higher value.

This result indicates that the generator dedicates more tiles to the optimal path than human-generated levels. Still, 97% is higher than previous PCG approaches, according to Summerville and Mateas. It also seems that the generated levels are similar to a human-generated levels [26].

Fully Differentiable Procedural Content Generation through Generative Playing Networks

On February 18th, 2020 this paper concerning PCG was released, written by Philip Bontrager and Julian Togelius. The paper depicts research where a level generator and a playing agent work in tandem using an Advantage Actor-Critic (A2C) RL algorithm.

The paper describes a cooperative adversary relationship between the level generator and the playing agent. The generator wants to make the level challenging for the playing agent, without exceeding the agent’s ability to solve the generated levels, as this leads to negative reward for both adversaries.

The cooperation between the generator and the playing agent will be further explained after a brief explanation of how they work individually. The agents are based on the RL algorithm know as A2C.

The playing agent will be rewarded based on whether or not the level is completed. The time it takes the agent to either complete the level or lose, is added as a positive or a negative reward, respectively. The agent’s reward ranges from 0 to 1. Equation 2.2 shows the algorithm.

The generator loop will update the generator to create environments. It is updated by sampling a minibatch of latent variables and mapping them to environments that are evaluated by the agent. The procedure will then update the weights of the generator. The network will be updated a few times solely to increase diversity as the generator can

$$R(S_n) = \begin{cases} 1 - \frac{n}{N}, & \text{if agent wins} \\ 1 + \frac{n}{N}, & \text{if agent loses} \\ -1, & \text{if environment does not compile} \\ 0, & \text{else} \end{cases} \quad (2.2)$$

Equation 2.2: Agent reward.

create very similar levels.

The playing agent works without any input from the generator; all it is meant to do is to play the levels it is given and learn to maneuver them to the best of its ability. The generator is, however, very dependant on the playing agent's performance, and will attempt to create an environment that is best suited for the playing agent's current abilities.

The entire experiment was conducted on the GVGAI framework, which has previously been described.

The research showed inconsistent results, where the level generator would occasionally create maps that could not be completed. The research does, however, suggest that the level generator can learn and that the interaction between the agent and the level generator does lead to progressively more complex environments [27].

Summary

Procedural Content Generation in games has been around for a very long time, where the first instances were in roguelike games such as *Beneath Apple Manor* (1978) and *Rogue* (1980). Different approaches to PCG has been explored and described in this section, and this summary will attempt to summarize and compare the approaches explored.

Most commonly, rule-based approaches are used for PCG, as seen in [13] and [16]. Rule-

based approaches use rules and algorithms in order to generate playable levels. The incorporation of Artificial Intelligence for PCG has become increasingly popular.

Frameworks for AI and PCG have been used for several pieces of research. These frameworks include the framework by Perez-Liabana [18].

Julian Togelius at New York University has become a central figure in PCG and was involved in [14], [19], [22], and [18], all referenced in this section. He also wrote the book describing the progress PCG and playing agents have made within games in [11]. There may also be several other papers Togelius was involved in that could be relevant but are not referenced in this thesis.

Mourato et al. [17] and Volz et al. [23] use fitness functions instead. Summerville and Mateas [26] use an A* agent to generate the level from the path the agent takes. Togelius et al. introduce the problem with overfitting for RL agents without dynamic environments and highlight this in their paper [19]. Similar to this thesis, Togelius et al. [27] use a playing agent controlled by Advantage Actor-Critic to train the generator. In this thesis, however, Deep Q-Networks are used to train the generator.

This thesis proposes a GAN-inspired system to make the agent and the generator interact. The design is very similar to Generative Playing Networks (GPN) seen in the paper by Bontrager and Togelius [27], with some key differences found in subsection 3.1. GAN is also used in work by Volz et al. [23], although rather than using it for interaction between the generator and agent, it is only used to train the generator.

In this thesis, Long Short-Term Memory is used in the generator model. Two variations were explored; Sequence-to-Sequence, and Sequence-to-Tile. The generation is done as an iterative task, where only a part of the level is generated at each step. The iterative generation is also described in PCGRL by Togelius et al. [22]. LSTM generation is also seen in Summerville and Mateas [26].

Table 2.1 shows an overview of related work by categorizing the works in terms of *Playing Agent*, *Rule-based generator*, *year of publication*, and *source*. *Playing agent* indicates whether the levels are tested using a playing agent controlled by AI, where the generator generates levels influenced by the playing agent’s performance. *Rule-based* means that the level generator is controlled by rules and not any form of AI.

Name	Playing agent	Rule-based	Year	Source
Platform Games	No	Yes	2006	[13]
Launchpad	No	Yes	2011	[16]
Genetic Algorithm	No	Partially	2011	[17]
GVGAI	Yes	No	2019	[18]
Illuminating	Yes	Yes	2018	[19]
PCGRL	Yes	Yes	2020	[22]
GAN	Yes	No	2018	[23]
LSTM	Yes	No	2016	[26]
GPN	Yes	No	2020	[27]
Seq2Tile & Seq2Seq	Yes	No	2020	-

Table 2.1: Overview of related work.

3 | Methods

This chapter aims to elaborate on the methods used, and the experiments implemented for this thesis.

The implementation of this Generative Playing System in *Super Mario Bros.* builds upon the implementation from the *PythonSuperMario* GitHub repository created by *Marblexu* [28].

3.1 Generative Playing Networks

The relationship between the agents in the implemented system is similar to that of Generative Playing Networks, introduced by Philip Bontrager and Julian Togelius [27]. A key difference is that the environment agent policy π and the environment value estimator Q have been replaced with the level generator. The implemented system consists of two symbiotic agents; one controlling Mario, and the other generating the level. The objective for the agent controlling Mario is to move towards the flagpole marking the goal of the level as quickly as possible without dying. The objective for the level generator is to create challenging levels suited to the player’s skill level.

The relationship between the two agents is symbiotic because the actions of one agent directly affect the other agent. More precisely, the average velocity of Mario over any given generated chunk in a level determines the generator’s reward for generating that particular chunk. Hence, the Mario agent affects the reward given to the generator. On the other hand, the generator is creating the map the Mario agent is navigating. Consequently, the generator is affecting the observation space of the Mario agent. Figure 3.1 shows the architecture of the implemented system.

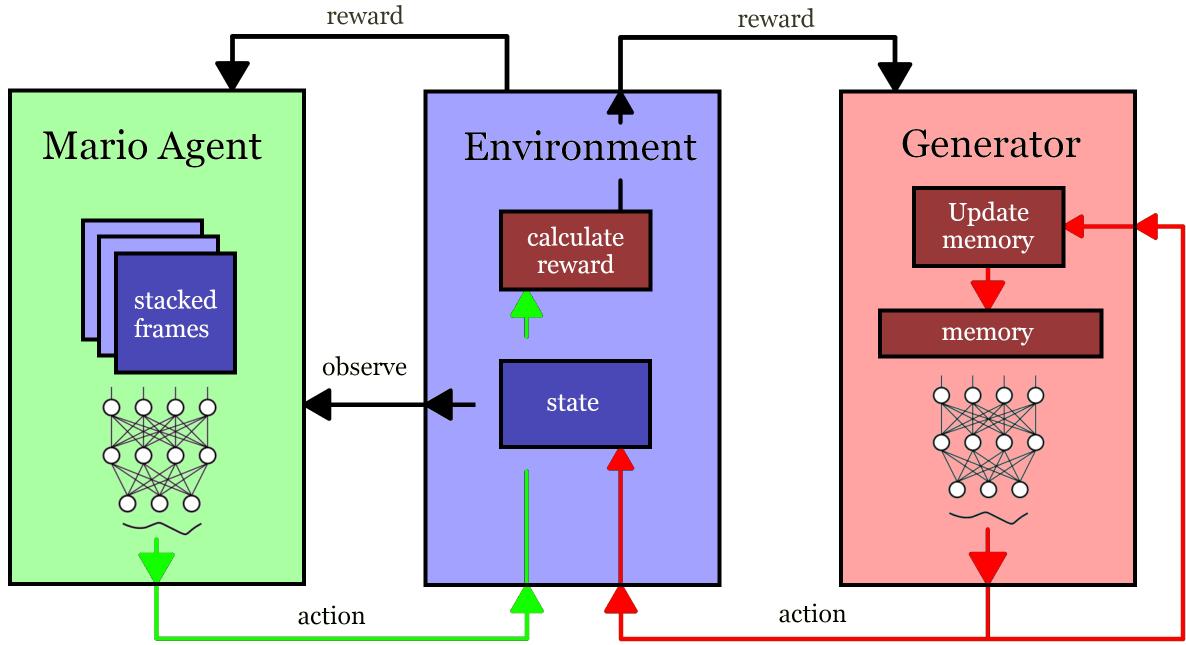


Figure 3.1: The architecture of the implemented system.

3.2 Experiments

In this thesis, two experiments are conducted. The two experiments differ by the model the generator utilizes in order to generate new level content. Both generator models include LSTM layers. However, the output shapes of the networks are different.

The first experiment implements an LSTM network, which accepts a sequence of previously generated tiles as input. It predicts Q-values of a single new tile following the last tile in the input. The predicted Q-values are used to pick new tiles to be inserted into the game. This model has similarities to sequence labeling models, and the experiment is labeled Seq2Tile generation.

The second experiment also implements an LSTM network, which accepts a sequence of previously generated tiles as input. However, instead of predicting Q-values of a single new tile, it predicts Q-values for a sequence of new tiles. The Q-values are, like in the

first experiment, used to pick new tiles to be inserted into the game. The model in this experiment is based on Seq2Seq models, and the experiment is therefore labeled Seq2Seq generation.

The generator generates tiles in *chunks*. A chunk is simply a set number of columns of tiles in the game. The generator generates chunks slightly ahead of the visible sliding window the Mario agent can observe. When the Mario agent has successfully traversed a generated chunk, the generator is given a reward according to how quickly Mario was able to cross the chunk. The reward is saved in a transition that is inserted into a replay memory. This replay memory is later used to update the generator Q-values, through an *experience replay*. The Q-values are updated in a Q-Learning fashion, using a Bellman-based equation.

Figure 3.2 visualizes the difference in the generator network between the two experiments. The Seq2Tile generator generates a single tile at a time while the Seq2Seq generator generates a sequence of tiles in one prediction. Thus, the Seq2Seq generator should have a better basis for learning the inherent connection between different tiles and recognize structures that fit well together. Furthermore, the Seq2Seq generator is exceptionally faster in regards to level generation and training. The most prominent reason is that the Seq2Seq generator only produces one prediction per generated chunk while the Seq2Tile generator produces one prediction per tile.

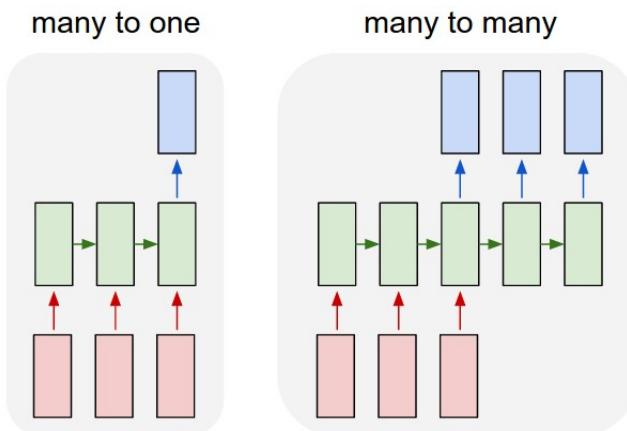


Figure 3.2: Many-to-one and many-to-many RNN topologies [29].

The system implemented for this thesis can be found on the master branch of the GitHub repository: <https://github.com/bjortho/SMBgen>. The conducted experiments are stored in separate branches.

3.3 Mario agent

The Mario agent utilizes a DQN to learn a state-action policy, using the RLlib library [30]. Since the library allows the learning process to be distributed, multiple instances of the environment are created, and each Mario agent uses a central policy. For each timestep, the Mario agent observes the environment, selects an action to perform using the central policy, and receives a reward from the environment. The reward function for the Mario agent is defined in the *MarioEnv* class in *mario_env.py* and pseudocode for the method can be seen in Listing 3.1 below.

Listing 3.1: Pseudocode for Mario reward function. The full method is listed in Listing C.1.

```

1 def mario_reward:
2     """Pseudocode for the Mario reward function"""
3
4     # Set reward to difference in current x-position and last x-position
5     reward = difference in mario x-position
6
7     # Add difference in remaining time to reward
8     reward += difference in game clock
9
10    If mario is dead:
11        reward = -15
12
13    return reward

```

The Mario agent is rewarded for moving right, and punished for moving left. The agent also receives a small punishment each time the game clock counts down one second. Additionally, the Mario agent receives a harsh punishment when Mario dies or the time runs out.

The observation space of the Mario agent consists of a square of tiles surrounding Mario in a specified radius, where Mario marks the center of the square, see Figure 3.3.



Figure 3.3: A visualization of an observation made by the Mario agent with the observation radius set to 5. 0.42 represents an empty tile, 0.5 represents a box, and 0.64 represents a solid tile.¹

¹A complete overview of the observation space of the Mario agent is listed in appendix A.

The action space of the Mario agent is defined by one of three possible lists defined in *actions.py*. These lists are:

- RIGHT_ONLY
- SIMPLE_MOVEMENT
- COMPLEX_MOVEMENT

Throughout this thesis, the only action space list used was COMPLEX_MOVEMENT, which consists of the following button combinations seen in Figure 3.2:

Listing 3.2: Action space list.

```

1 COMPLEX_MOVEMENT = [
2     ['NOOP'],
3     ['right'],
4     ['right', 'A'],
5     ['right', 'B'],
6     ['right', 'A', 'B'],
7     ['A'],
8     ['left'],
9     ['left', 'A'],
10    ['left', 'B'],
11    ['left', 'A', 'B'],
12    ['down'],
13    ['up']
14]
```

NOOP (no operation) means the agent does not provide any input to Mario at the given timestep. Right, left, up and down refers to pressing the respective direction on the directional-pad, where right and left moves Mario in the corresponding direction, up makes Mario climb up climbable surfaces, and down allows Mario to descend into vertical pipes, or crouch if Mario is big. Pressing the A button makes Mario jump, while the B button can be held down in order to make Mario sprint, or it can be pressed once to

shoot a fireball if Mario has obtained the fire flower upgrade. When multiple buttons are supplied in a list, all of these buttons are pressed simultaneously.

3.4 Level generator

The level generator is a separate agent from the Mario agent, and the *Generator* class is defined in the *generation.py* file.

3.4.1 Generator network model

Throughout the different experiments, a variety of neural network models have been tested with the generator, and different methods of generating new map content were tested.

Seq2Tile LSTM generation

A Seq2Label model is a model that predicts a label from a sequence of items. In this variant of the generator network, a Seq2Tile model is used, which is similar to a Seq2Label model. The input is a sequence of one-hot encoded tiles, and the output is a list of Q-values used to derive a single new tile. The network utilizes an LSTM layer to handle the sequence of previously generated tiles. The network is created with Tensorflow's Keras API, defined in the *create_generator* method shown in Listing 3.3.

Listing 3.3: Seq2Tile *create_generator* method.

```
1 model = Sequential()
2 model.add(Input(shape=(c.MEMORY_LENGTH, len(c.GENERATOR_TILES))))
3 model.add(LSTM(c.LSTM_CELLS))
4 model.add(Dense(len(c.GENERATOR_TILES), activation='linear'))
5 model.compile(loss='mse', optimizer=Adam(lr=c.LEARNING_RATE), metrics=['accuracy'])
```

c.MEMORY_LENGTH is the number of preceding tiles the generator model will receive

as input when predicting a new tile, and `len(c.GENERATOR_TILES)` is the number of different tiles the generator can predict. `c.LSTM_CELLS` denotes the number of LSTM cells used in the LSTM layer, meaning how many timesteps the model can look back from the newest tile. The Dense layer condenses the number of outputs down to the number of different tile types, such that each output node corresponds to the Q-value of one particular tile. The activation in the final dense layer is set to linear, meaning the output from the nodes in that layer, remains unchanged. The reason is that the generator is predicting Q-values for each tile instead of the label for a new tile. The Q-values are continuous variables; hence the problem the generator solves is a regression problem as opposed to a classification problem where the output is discrete. Mean squared error is used as the loss. By using mean squared error as the loss function, the optimizer will be able to measure the distance from a predicted Q-value to a target Q-value and adjust the network’s weights accordingly.

Seq2Seq LSTM generation

A Seq2Seq model is a model that takes a sequence of items as input and outputs another sequence of items. In this variant of the generator network, the items are one-hot encoded tiles. The network used for the Seq2Seq LSTM generation also uses LSTM layers, although this variant has one LSTM layer for handling the sequence of previously generated tiles, as well as one LSTM layer for predicting Q-values of future tiles. The final LSTM layer provides a two-dimensional list containing Q-values, where the number of lists corresponds to the number of tiles in a chunk. The number of tiles in a chunk is defined by `self.gen_size`, where the generation size is dependent on the number of tiles per column, and the number of columns per chunk. The network is defined in the `create_generator` method shown in Listing 3.4.

Listing 3.4: Seq2Seq *create_generator* method.

```
1 model = Sequential()
2 model.add(LSTM(units=self.gen_size, input_shape=(c.MEMORY_LENGTH, len(c.GENERATOR_TILES))
    ↪ ))
3 model.add(RepeatVector(self.gen_size))
4 model.add(LSTM(units=self.gen_size, return_sequences=True))
5 model.add(TimeDistributed(Dense(len(c.GENERATOR_TILES), activation='linear'))))
6 model.compile(loss='mse', optimizer=Adam(lr=c.LEARNING_RATE), metrics=['accuracy'])
```

3.4.2 Map generation

The *level_gen.py* file is the python file used to generate the level in the *Super Mario Bros.* game. This file contains a *generate* method. The pseudocode for this method is shown in Listing 3.5. When called, the method creates a dictionary of different tile and entity types with associated coordinates and Q-values. This *tiles* dictionary is built in the *build_tiles_dict* method. Coordinates and Q-values for each tile are inserted into a list and appended to the corresponding list in the *tiles* dictionary. The dictionary is then used to provide arguments for appropriate methods for inserting the new content into the map.

Listing 3.5: Pseudocode for the *generate* method in *level_gen*. The full method is listed in Listing C.2.

```
1 def generate:
2     """Pseudocode for the generate method"""
3
4     # Create empty dictionary
5     tiles = {}
6
7     # Fill dictionary keys with strings representing tile types
8     # Map each tile type to an empty list
9     tiles.keys = tile types
10    tiles[type] = empty list
```

```
11
12     # Call generator to generate new level content
13     new_tile_columns = generator.generate()
14
15     # Fill tiles dictionary using build_tiles_dict method
16     iterate over each column in new_tile_columns:
17         tiles = build_tiles_dict(column)
18
19     Add new tiles and entities to respective sprite groups
```

When the *read* variable specified in *Level* class is False, the *generate* method in the *Generator* class is used to create a list of encoded values, which in turn is used to build the dictionary of tile and entity types. The encoded values are stored in a two-dimensional list. The encoding corresponds to the **ID** field in the table in Appendix B, and this list is returned to the *generate* method in *level_gen.py*. The *tiles* dictionary is built in the *build_tiles_dict* method, and the new map content is inserted into the level.

The *generate* method defined in *generation.py* is responsible for generating a list of strings representing columns of tiles that will be inserted into the generated level. Each character in the strings is an encoded value for a tile or enemy, with the same encoding as mentioned above. The method also returns a two-dimensional list of Q-values corresponding to the generated tiles.

Seq2Tile LSTM generation

When *c.RANDOM_GEN* is set to *False*, the generator model is used to derive new tiles using the internal memory of previously generated tiles. Figure 3.4 shows an example of input data to the generator network, and where the new tiles appear. In the Figure, the two solid tiles at the bottom of each column are inserted automatically. They are not included in the network input or output. The length of the input to the generator network is determined by *c.MEMORY_LENGTH*. The number of generated columns depends on *c.GEN_LENGTH*. The generator input is retrieved from the *self.memory* list in the *Generator* class, which is a one-dimensional list of previously generated tiles in

order from oldest to newest. The new map is generated tile-by-tile, and the generator input works like a sliding window where each newly generated tile is inserted into the generator memory, and the window slides one tile forward, see Figure 3.5. If the memory list is shorter than `c.MEMORY_LENGTH`, it is padded with empty tiles at the beginning. It is then provided as input to the LSTM network.

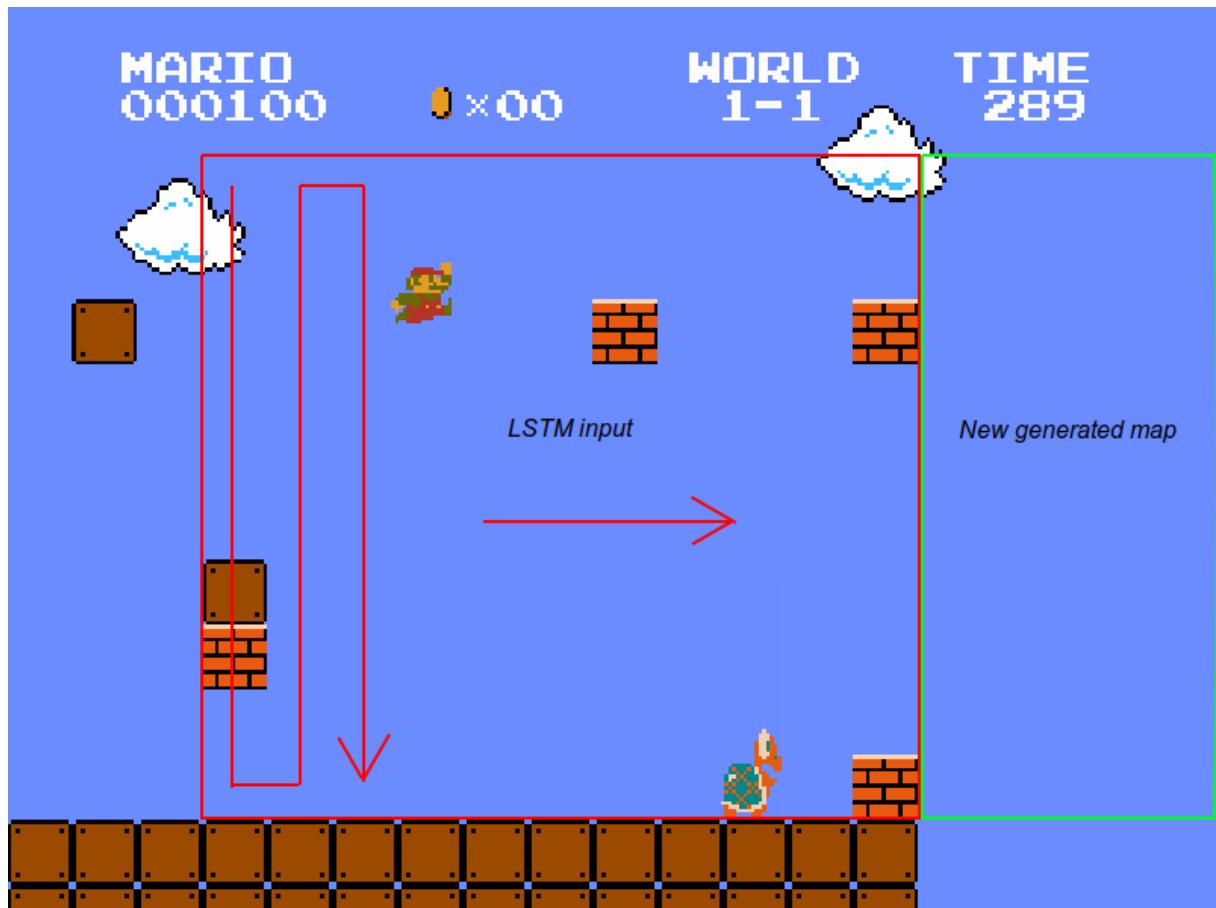


Figure 3.4: Visualization of the LSTM network input and the new chunk of tiles on the map.

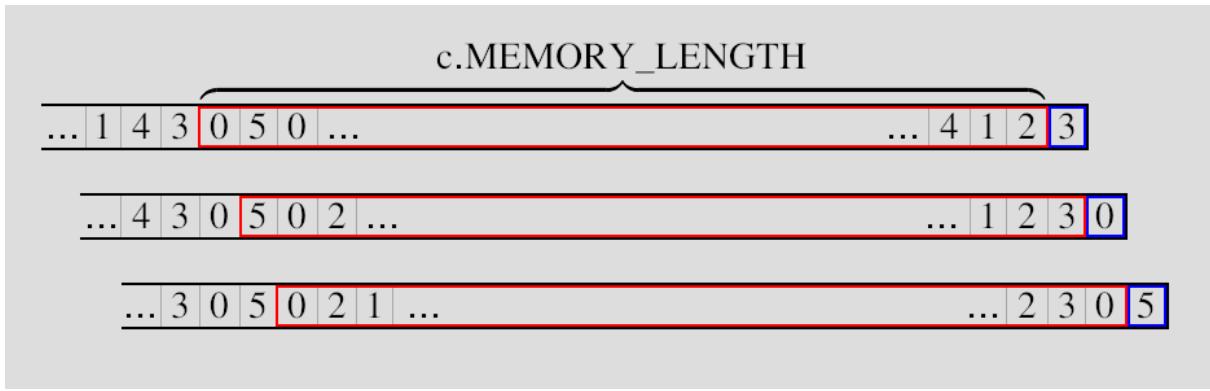


Figure 3.5: The generator memory contains indexes representing tiles in the order they were generated. The red field is the sliding window, while the blue field is the new tile inserted into the memory.

Seq2Seq LSTM generation

The `generate` method in the `Generator` class works slightly differently in the Seq2Seq LSTM generation implementation. The difference is that this implementation only calls `predict` on the generator network once to create a chunk of tiles, as opposed to once per tile in a chunk.

Epsilon

Epsilon is the factor of exploration, meaning how often the generator will pick the tile with the highest Q-value (greedy) versus how often the generator will pick a random tile (non-greedy). When the generator is greedy and chooses the tile with the highest Q-value, the generator prioritizes *exploitation*, as opposed to when the generator chooses a random tile and prioritizes *exploration*. The epsilon variable is simply a measure of how likely the generator is to perform a greedy action versus a non-greedy action. The start epsilon value is set when the generator is initialized, and the default value is 1.0. Epsilon will gradually decay throughout training and will end at a specified minimum value denoted by `c.MIN_EPSILON`. Epsilon is updated each time the environment is done, meaning when Mario dies, runs out of time, or completes the level. Epsilon is updated according to the formula seen in Equation 3.1.

$$\epsilon = \begin{cases} \max(\epsilon_{min}, \epsilon \cdot \epsilon_{decay}), & \text{if } \epsilon > \epsilon_{min} \\ \epsilon_{min}, & \text{otherwise} \end{cases} \quad (3.1)$$

Equation 3.1: Epsilon decay.

For this thesis, two different approaches for greedy selection were tested. One approach is to determine greedy selection for each tile, such that the generator will check if it should perform a greedy tile selection or a random tile selection for each tile. The other approach is to determine greedy selection for an entire chunk of tiles, which means the generator will generate c.GEN_LENGTH columns of tiles, where all of these tiles will be generated with either greedy tile selection or non-greedy tile selection. This chunk-based greedy selection can be toggled with the CHUNK_BASED_GREEDY constant.

3.4.3 Experience gathering

The generator utilizes an experience replay architecture for training. The experience is stored in a *replay memory*, which is a double-ended queue containing transitions ordered from oldest to newest. c.REPLAY_MEMORY_SIZE adjusts the maximum size of the replay memory. Since it is a double-ended queue, the oldest transitions in the queue are removed first when the queue is full.

Transitions are added to the replay memory when Mario has successfully traversed a chunk of generated tiles. Each transition is a tuple of five elements, as seen in Figure 3.6

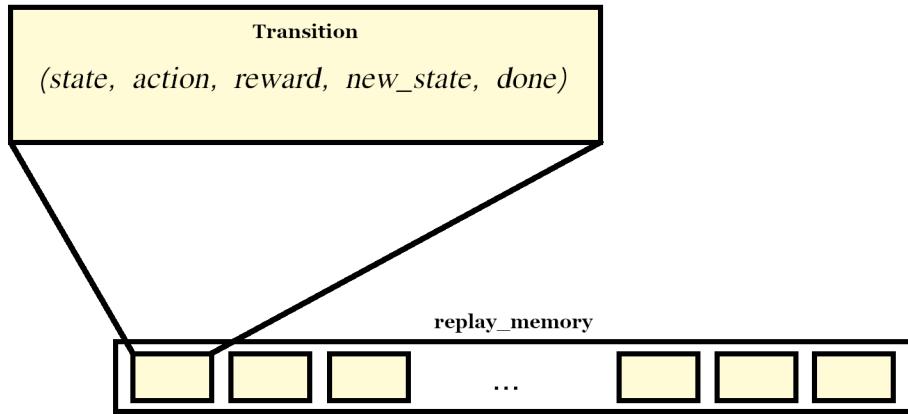


Figure 3.6: The connection between the replay memory and the transitions.

- **state**

The state is a list of tiles leading up to the first tile of the generated chunk in the transition, where the generated chunk is the *action* the generator performed. The state list is used to create the input for the generator network, and the length of the list can be adjusted with `c.MEMORY_LENGTH`.

- **action**

The action is a list of the tiles that were generated, and Mario was able to navigate across. The size of this list is determined by how many tiles are generated for each generation. The generation size can be adjusted by `c.GEN_LENGTH`, and `c.INSERT_GROUND`. The constant `c.COL_HEIGHT` sets the number of tiles per column. If `c.INSERT_GROUND` is *True*, two solid tiles will be placed at the bottom of each column, and not included in the action list. `c.GEN_LENGTH` refers to the number of generated columns per generation.

- **reward**

The reward is simply a float corresponding to the generator's reward as a consequence of the action, given the state. The reward is calculated when a transition is to be inserted into the replay memory, as the reward is dependent on the average velocity of Mario across the generated chunk.

- **new_state**

The new_state is a list of tiles representing the state for the next transition. The length of the new_state list is the same as the state list, and the new_state list includes tiles from the action list, as those are the most recently generated tiles.

- **done**

The done flag is used to denote if a state in a transition is a terminal state to avoid using future state Q-values during training for a terminal state.

For each timestep, the *check_gen_reward* method is called in *level_gen.py* after the game logic has been updated. This method checks if Mario has traversed a generated chunk and compiles relevant data for inserting a new transition into the replay memory. It then calls *update_replay_memory* in the *Generator* class, which uses the provided data to insert the transition into the replay memory.

3.4.4 Reward function

The generator reward is calculated in *calc_gen_reward* and is defined in Equation 3.2.

$$\text{Reward} = \begin{cases} \sin(\frac{\pi}{2v'}v), & \text{if } v < v' \\ e^{-\frac{1}{2}(v-v')^2}, & \text{otherwise} \end{cases} \quad (3.2)$$

Equation 3.2: Generator reward function.

Where v is the average velocity of Mario across the generated chunk and v' is the optimal Mario velocity. These velocities are measured in pixels per frame. The optimal Mario velocity is always greater than zero, and therefore the range of the generator reward is between 0 and 1. The reward is close to 1 when the average Mario velocity is close to the optimal Mario velocity.

Since the objective of the level generator is to create difficult but completable maps, the

generator is rewarded according to how quickly Mario can move across a chunk. The optimal Mario velocity is defined in Equation 3.3.

$$v' = \frac{x_{flag} - (x_{mario} + w_{mario})}{t \cdot base_fps} \quad (3.3)$$

Equation 3.3: Optimal Mario velocity.

Where x_{flag} is the x-position of the flagpole marking the goal of the level, x_{mario} is the current x-position of the leftmost edge of the Mario sprite, and w_{mario} is the width of the Mario sprite. All these distances are measured in pixels. Therefore, the fraction numerator corresponds to the pixel distance between Mario and the flagpole. In the denominator, t refers to the time left on the game clock, and $base_fps$ is the number of frames rendered (and game updates) before the game clock counts down one second.

Therefore, the optimal Mario velocity is the average velocity Mario needs to maintain to reach the goal flag exactly when the time runs out. When Mario moves towards the flag, the nominator of the optimal Mario velocity fraction will decrease. However, the time left on the clock is constantly decreasing, making the denominator of the fraction decrease simultaneously. If Mario moves towards the goal faster than the optimal velocity, the generator should create more difficult maps. On the other hand, if Mario moves too slowly across the map, the generator should create easier maps. In both cases, the generator will gain sub-optimal rewards. Therefore, the generator should attempt to find a difficulty suited to Mario’s skill level. Figures 3.7, 3.8 and 3.9 show the generator reward function with three different cases of optimal Mario velocity.

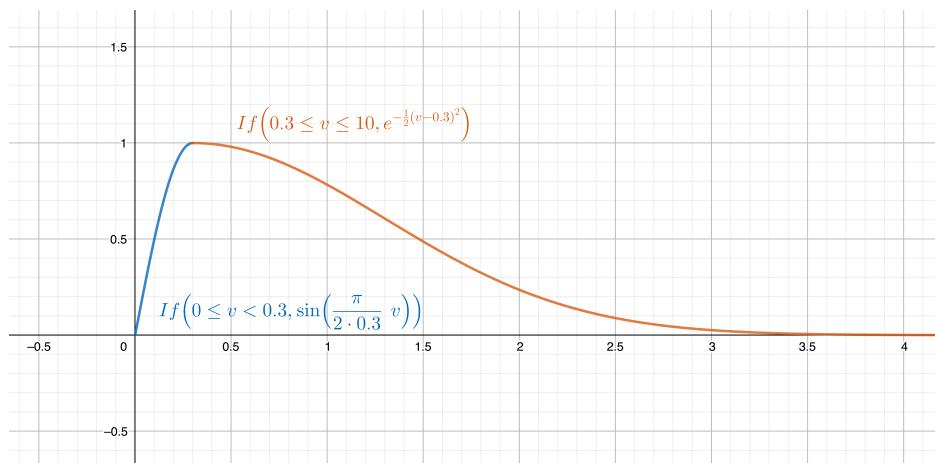


Figure 3.7: Generator reward function with $v' = 0.3$.

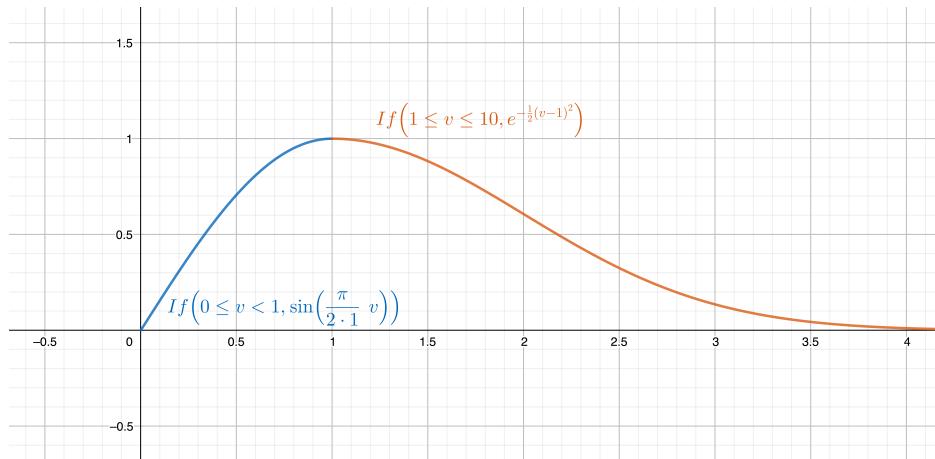


Figure 3.8: Generator reward function with $v' = 1$.

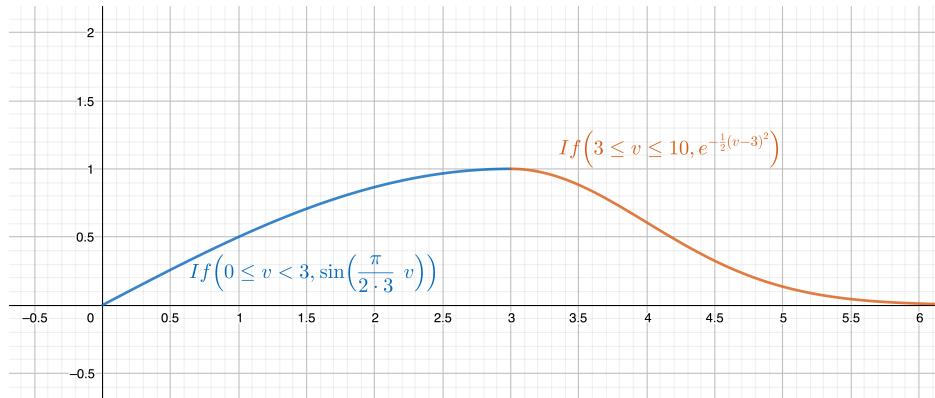


Figure 3.9: Generator reward function with $v' = 3$.

3.4.5 Generator training

The generator is trained in the *train* method in the *Generator* class. The implementation of the *train* method differs slightly between the various generator network architectures, where the only difference is the shape of the input X and target y lists used for fitting the generator model to accommodate the different shapes accepted by the networks.

When `c.TRAIN_GEN` is set to True, the *train* method in the *Generator* class is invoked from the *generate* method in *level_gen.py*, which in turn means the generator will be trained at most once per generated chunk. The training starts only if the replay memory contains a sufficient amount of transitions, specified by `c.MIN_REPLAY_MEMORY_SIZE`.

When a training session begins, a minibatch is created by selecting random transitions from the replay memory. The number of transitions in the minibatch is specified by `c.MINIBATCH_SIZE`.

The LSTM network is then used to create predictions of new sequences following each original state in the minibatch. It is also used to create predictions of sequences following all the new states in the minibatch.

Listing 3.6 shows pseudocode for the initialization of the training process.

Listing 3.6: Pseudocode of the initialization of training and target lists in the *train* method.

The full method is listed in Listing C.3.

```
1 # Only start training if we have enough transitions in replay memory
2 if number of transitions in replay memory < minimum number of transitions:
3     return 0
4
5 # Get a minibatch of random samples from replay memory
6 minibatch = random sample from replay_memory
7
```

```
8 # Get current states from minibatch and create list of predicted sequences
9 current_states = list of states from minibatch
10 current_predicted_sequences = list of predictions given current_states
11
12 # Get future states from minibatch, and create new list of sequence predictions
13 new_states = list of new states from minibatch
14 future_predicted_sequences = list of predictions given new_states
```

Training and target lists are initialized, and the minibatch is enumerated.

For each non-terminal state, a list of max future Q-values is created, along with a new list of Q-values where the reward is added to each Q-value in the *max_future_Qs* list. If a state is terminal, all the values in the *new_Qs* list are set to *reward*.

The Q-values predicted for the current state are updated, but only the Q-values for the tiles that were generated on the map. The Q-values for the other tiles remains unchanged.

Listing 3.7 shows pseudocode for the described procedure.

Listing 3.7: Pseudocode for updating Q-values in the *train* method. The full method is listed in Listing C.3.

```
1 # Create empty input and target lists, X and y
2 X = empty list
3 y = empty list
4
5 # Enumerate transitions
6 enumerate index and transition in minibatch:
7
8     # If not a terminal state, get new qs from future states, otherwise set it to reward
9     if not transition.done:
10         max_future_Qs = max Q-values for Qs in the current index of
11             ↪ future_predicted_sequences
12         new_Qs = list of transition.reward added to the current max_future_Qs with a
13             ↪ slight discount
14     else:
15         new_Qs = list of transition.reward
```

```
15     Get the Q-values of the tiles generated in transition.action.  
16     Update the corresponding Q-values in current_predicted_sequences to new_Qs
```

Seq2Tile generation

The generator only predicts one new tile for each prediction. The generator input is formatted as a list containing a "sliding window" of tiles, which moves one tile forward for each prediction, as shown in lines 1 to 6 in Listing 3.8 below. The generator input list will contain lists of tiles, where each tile list moves one tile forward. The tile lists are one-hot encoded to match the input shape of the generator network.

The generator input and target Q-values are appended to the input X and target y lists, as shown on lines 8 to 10 in Listing 3.8.

After the transitions have been processed, the generator model is fitted with the training data, as shown on lines 12 to 18 in Listing 3.8.

Listing 3.8: Pseudocode of the Seq2Tile generation model fitting. The full code is listed in Listing C.4.

```
1  # Insert sliding window states into X  
2  generator_input = empty list  
3  Iterate i from 0 to the number of generated tiles:  
4      tmp_state = list of tiles skewed i tiles from the start of transition.  
5      ↪ current_state  
6      One-hot encode tmp_state and append it to generator_input  
7  
7  # Append to training data  
8  Append generator_input to X  
9  Append updated current_predicted_sequences to y  
10  
11 # Fit on all transitions in minibatch  
12 iterate i from 0 to the number of training samples in X:  
13     generator.fit(X[i], y[i])  
14  
15 return 1
```

Seq2Seq generation

The current state in the transition is all the input the generator needs to produce the predictions. The only remaining formatting for the generator input is to one-hot encode the current state.

The generator input and target Q-values are appended to the input X and target y lists.

After the transitions have been processed, the generator model is fitted with the training data.

Listing 3.9 shows pseudocode describing the finalization of the training data and the fitting process.

Listing 3.9: Pseudocode of the Seq2Seq generation model fitting. The full code is listed in Listing C.5.

```
1 # One-hot encode current state from transition
2 generator_input = transition.current_state one-hot encoded
3
4 # Append to training data
5 Append generator_input to X
6 Append updated current_predicted_sequences to y
7
8 # Fit on all transitions in minibatch
9 generator.fit(X, y)
10
11 return 1
```

3.5 Hyperparameters

The run configurations rely on a significant number of hyperparameters. These can be tuned to achieve better results. The configurations used for the conducted experiments are outlined in Table 3.1. Only the most relevant parameters are listed. The full configurations can be found in Appendix D.

Parameter	Value	Effect
INSERT_GROUND	True	Determines if two solid tiles should be inserted at the bottom of each generated column.
READ	True	Determines if the generator reads and generates tiles from a text file of previously generated content.
WRITE	False	When WRITE is True, the generator will write the generated content to a text file.
SNAKING	True	When SNAKING is True, previously generated tiles will alternate between being read from top to bottom and from bottom to top. Otherwise, tiles will only be read from bottom to top.
CHUNK_BASED_GREEDY	False	Sets greedy or non-greedy actions for all tiles in a chunk if True. Otherwise, the generator determines greedy or non-greedy action on each tile.
GREEDY	False	Generator only performs greedy actions when True.
LEARNING_RATE	0.001	Learning rate for the generator.
MEMORY_LENGTH	64	Number of previously generated tiles used as input for the LSTM networks, also denotes the number of LSTM units in the Seq2Seq model.

EPSILON_DECAY	0.995	Factor of decay for epsilon. Epsilon decays after each episode.
MIN_EPSILON	0.01	Minimum epsilon value.
REPLAY_MEMORY_SIZE	100	Maximum number of transitions in the replay memory.
MIN_REPLAY_MEMORY_SIZE	50	The minimum amount of transitions required in the replay memory before the training starts for the generator.
MINIBATCH_SIZE	10	Number of transitions sampled from the replay memory for each generator training session.
DISCOUNT	0.99	Factor of discount for future rewards.
OSB_RADIUS	10	The observation radius of the Mario agent.
OBS_FRAMES	21	Number of stacked frames in the Mario agent's observation.

Table 3.1: The most relevant hyperparameters.

4 | Experimental results

In this section, the results obtained from the experiments are discussed. The findings are then compared to the hypotheses defined in section 1.3.

The first hypothesis claims that it is possible to generate Super Mario Bros. levels by using a system of two AI-algorithms to generate and test levels. The experiments described in this chapter show that this is indeed possible.

The second hypothesis claims that it is possible to train the generator to create levels fitting to the player’s skill level. Some of the results below indicate that the difficulty of the generated levels are adapted to the player’s skill level. However, more training is required to verify the hypothesis.

4.1 Seq2Tile generation

The Seq2Tile generator was used to generate level content tile-by-tile, which the Mario agent evaluated by playing the generated level. Together, these two AI-algorithms confirm the first hypothesis defined in section 1.3. The Seq2Tile generator was trained for approximately 4,300 episodes, and Figure 4.2 shows the unfiltered generator reward for each episode. The reward does not appear to increase nor decrease during the training and does not converge at any point. The reward could indicate that the generator did not finish training or that the Mario agent was able to converge faster than the generator. Figure 4.1 shows the results of the training for the Mario agent. The graph is split up due to the program restarting multiple times during training, which can make it quite hard to read. However, the graph seems to indicate that the mean episode reward has somewhat stabilized at approximately 120, which may indicate that the generator and the Mario agent are improving at the same rate. However, it can also mean that both agents are

at a stand-still. The step number on the x-axis starts at approximately 6 million because the same checkpoint was used to train earlier generator models.

The Mario agent's rewards and the generator's rewards are connected since the more transitions Mario can traverse, the more reward both agents receive. This is naturally dependent on how quickly the Mario agent is able to traverse the generated level. Since the generator will never receive a negative reward, the more transitions Mario can traverse, the better for the generator.

Figure 4.3 shows the episode reward for the generator with the Savitzky-Golay filter applied. Drops in the reward can be clearly seen around episode 500-1250 and episode 2500-3000. This could correlate to when the program had to restart. Upon a restart, all the variables are reset to their initial values, except the agent models and the replay memory, which are loaded from disk. However, variables such as *epsilon* will be reset to their initial values, which could drastically change the generator's behavior.

Figure 4.1 shows indents in the generator reward which may match up with the restarting of the program. However, it is difficult to be sure, as many of the recorded statistics have been skewed and overlap due to the restarts and model saving interval. Since the generator models are saved for every 100 training sessions by default, some models from an earlier episode could be loaded after a restart, and this may cause the lines in the graph to overlap.

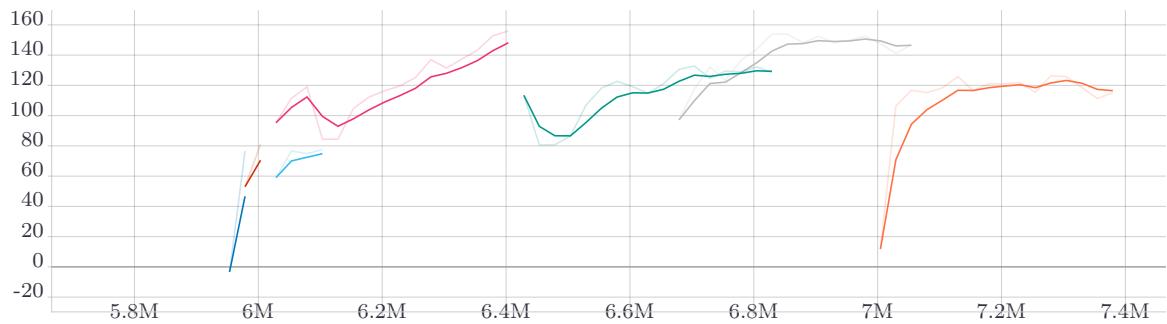


Figure 4.1: Result for the Mario agent competing against the Seq2Tile generator. The x-axis shows the number of steps the Mario agent has performed and the y-axis shows the episode reward. The graph is split up as a result of the program restarting multiple times during training.

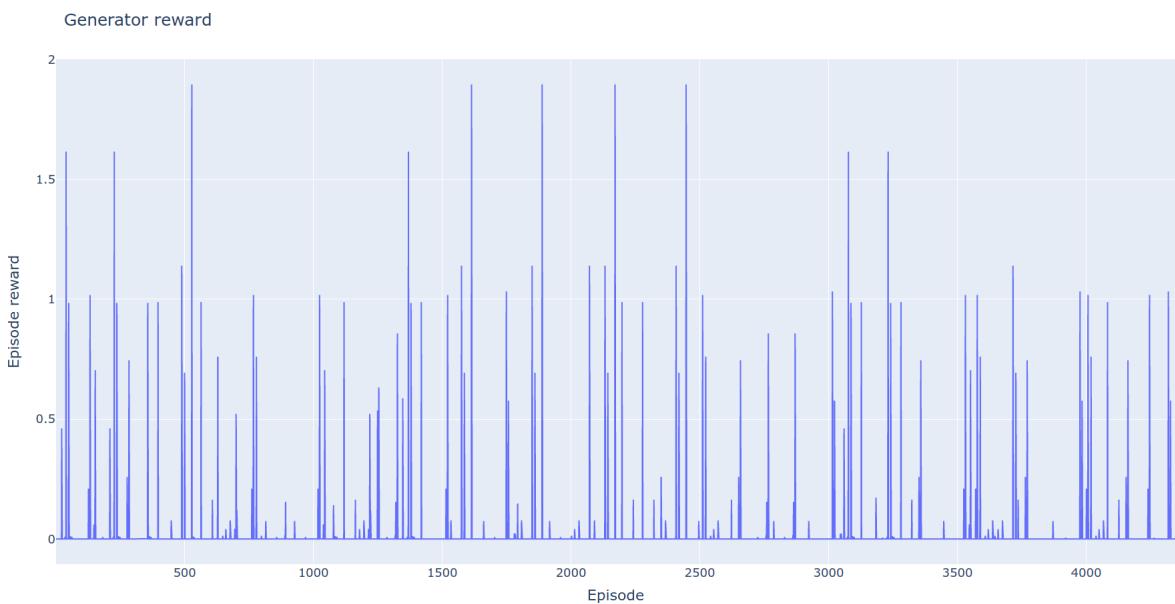


Figure 4.2: Unfiltered episode reward for the Seq2Tile generator.



Figure 4.3: Episode reward for the Seq2Tile generator with the Savitzky-Golay filter applied with window size 1,001 and polynomial degree 4.

Figure 4.4 shows an example of a level generated by the Seq2Tile generator along with Q-values associated with each generated tile. During training, the generator starts by performing random actions, and as the epsilon value decays, it progressively performs more greedy actions. Therefore, the generator was configured to only perform greedy actions during the evaluation. The generator was also set to be non-trainable as it was being evaluated.

The Q-values on the tiles are quite low, meaning it expects a low reward for generating the tiles. The reason why the Q-values are low is probably that the generated level appears to be impossible for the Mario agent to complete. The generated level indicates that the generator is not able to create levels fitting to the player's skill level, which deems the second hypothesis defined in section 1.3 false. Comparing the results in Figure 4.4 with a generator generating random tiles, as seen in Figure 4.5, the random generator appears to give Mario a better chance at moving forward than the Seq2Tile generator.

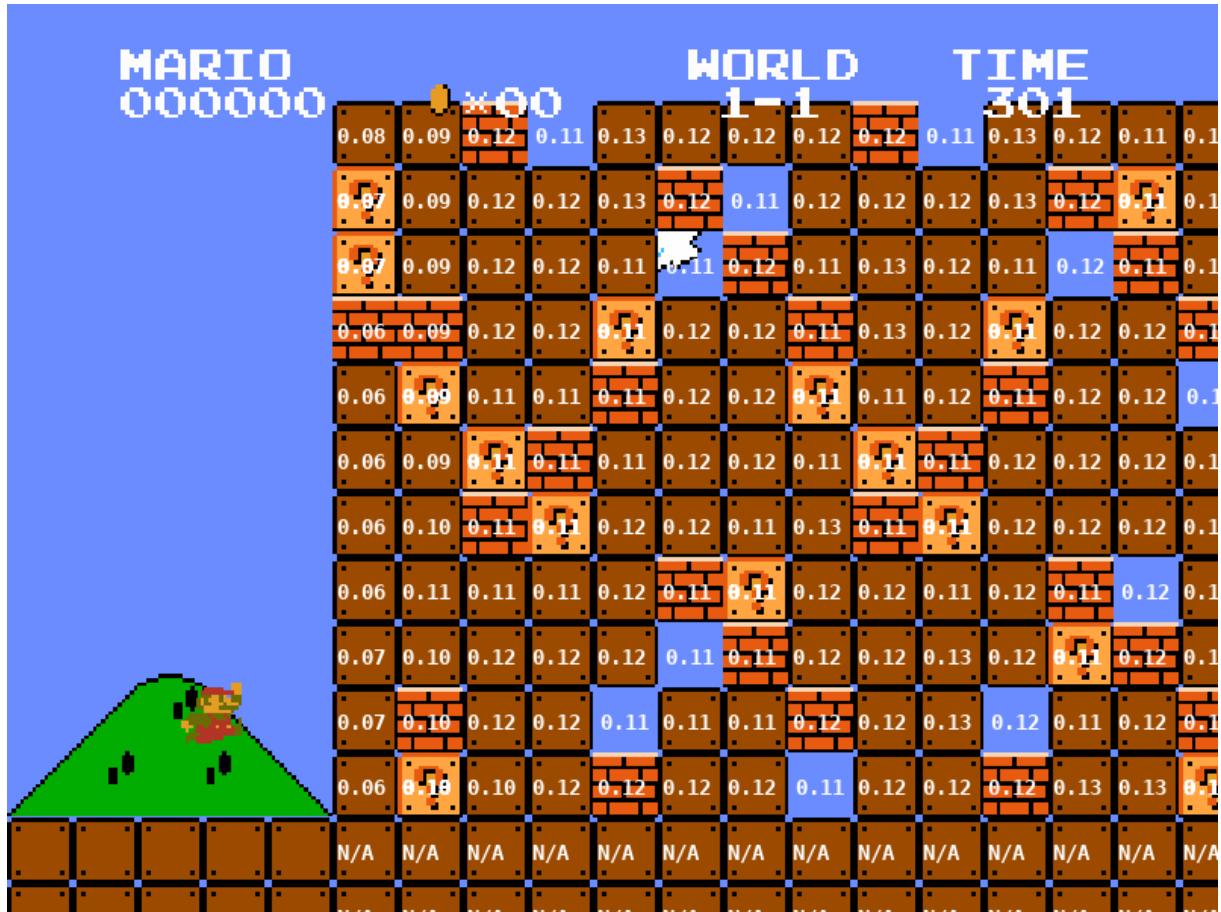


Figure 4.4: Seq2Tile results with each tile labeled with its Q-value.



Figure 4.5: Example of random generation.

4.2 Seq2Seq generation

In this experiment, the generator takes a sequence of tiles as input and predicts a sequence of tiles as output. The Mario agent evaluates the level's content. Working together, the two AI-algorithms confirm the first hypothesis defined in section 1.3. The Seq2Seq generator was trained for approximately 12,200 episodes, and Figure 4.7 shows the unfiltered generator reward for each episode. Figure 4.6 shows the results of the training for the Mario agent. Again, the graphs are split up due to the program restarting multiple times during training. In this case, the filtered reward graph in Figure 4.8 shows more divots and slightly lower reward than the Seq2Tile generator on average. The reason for both

could be that the program restarted more during this training than the Seq2Tile training.

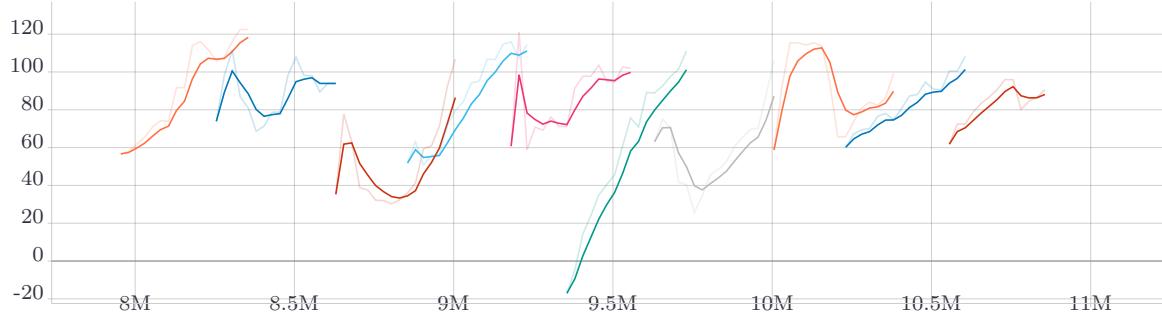


Figure 4.6: Result for the Mario agent competing against the Seq2Seq generator. The x-axis shows the number of steps the Mario agent has performed and the y-axis shows the episode reward. The graph is split up as a result of the program restarting multiple times during training.

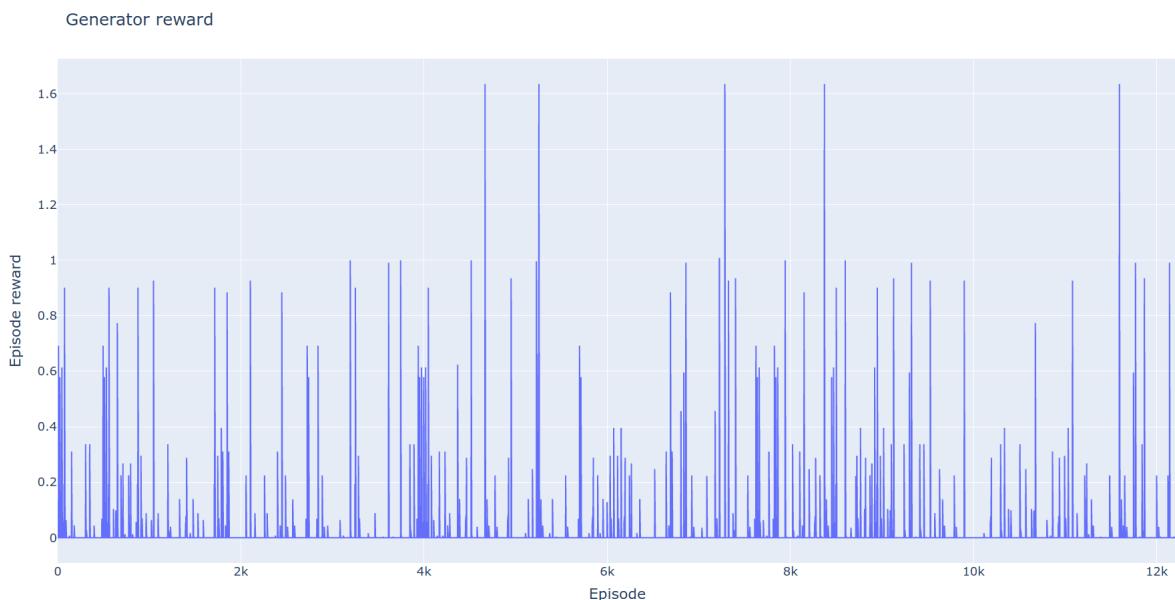


Figure 4.7: Unfiltered episode reward for the Seq2Seq generator.

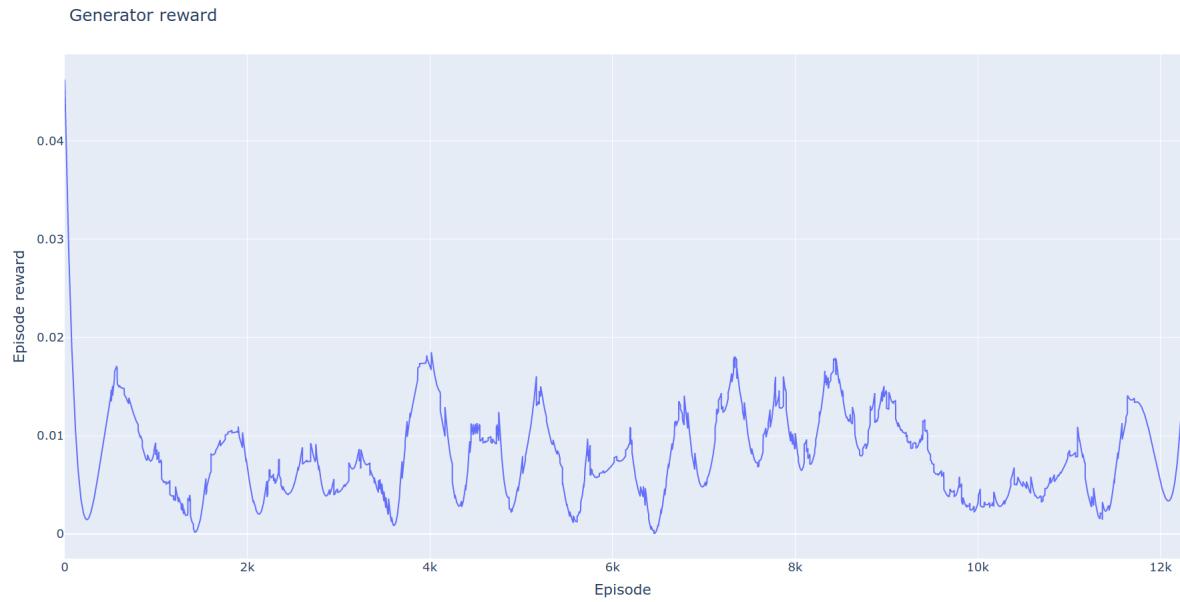


Figure 4.8: Episode reward for the Seq2Seq generator with the Savitzky-Golay filter applied with window size 1,001 and polynomial degree 4.

The generated levels have a maximum length of 8,505 pixels or almost 200 tiles. Mario's start position is at x-position 110, meaning the distance from Mario to the flag is 8,395 pixels. Since the game clock starts at 301 seconds, it can count down 300 seconds before Mario loses. Furthermore, since the Mario agent receives -1.0 reward each time the game clock ticks down one second, the minimum reward the Mario agent can receive when completing a level is 8,095.

None of the graphs showing the Mario agent's episode rewards are remotely near the minimum possible reward Mario can obtain when completing a level. Hence, the generated levels are too difficult for the Mario agent, and the generator has not been able to adhere to a suitable difficulty, as hypothesized in section 1.3. However, the generator and Mario agent may adapt after more training.

It should be noted that even though the generator rewards appear to be very low compared to the mean episode reward of the Mario agent, the Mario agent has a significantly larger pool of rewards it can collect compared to the generator. The Mario agent receives a reward of 1.0 for each pixel Mario moves to the right. The generator can receive a

maximum reward of 1.0 for each transition Mario traverses. When one transition consists of five columns of tiles, where each tile is 43 pixels wide, the reward ratio between the Mario agent and the generator is 215:1 in the optimal scenario for the generator.

Figure 4.9 shows the results of the Seq2Seq generator along with Q-values associated with each generated tile. As with the Seq2Tile generator, the generator was configured to be non-trainable and only perform greedy actions during the evaluation. The reward the generator can receive for generating any given tile ranges between 0 and 1. However, the Q-values printed on each tile in Figure 4.9 range between 0.001 and 0.005, which is very low, especially considering the Q-values also account for future rewards. We believe the reason for this is the behavior of the Mario agent.

When the generator is set to greedy and non-trainable, it will always generate the same level. When evaluating with these settings, we observed that the Mario agent always jumped directly into the column of enemies and died. Judging by the layout of the beginning of the level, it seems possible for Mario to advance at least to the brick wall by waiting for the enemies to fall to the ground before jumping on them. When attempting to evaluate the generated level with a human player controlling Mario, it was indeed possible to advance to the brick wall. However, Mario can only jump over four tiles vertically, and the high brick wall is five tiles in height, which makes it impossible to pass. It is even impossible to overcome by bouncing on an enemy, as this does not provide sufficient altitude for Mario. Judging by the Q-values, however, it is not far-fetched to believe that the generator attempted to create an obstacle for Mario. The generator put the highest Q-value of 0.003 on the bottom brick, 0.002 on the next, and 0.001 on the following three bricks. After these, the Q-value of an air tile overtook the Q-value of the brick tile.

This behavior could indicate that the generator knows it should generate a certain number of bricks to create an obstacle, but it does not know precisely how many. The behavior is also an indication that the second hypothesis defined in section 1.3 can be deemed valid, but more training is needed to conclude. Regardless, the results observed in Figure 4.9 show that the generator can generate levels with inherent structure. By comparing

Figure 4.9 to Figure 4.5, which showcases a generator randomly generating tiles, it can be argued that the Seq2Seq generator has learned some structure for generating levels in Super Mario Bros.



Figure 4.9: Seq2Seq results with each tile labeled with its Q-value.

5 | Discussion

In this chapter, we discuss the thesis in its entirety. We will look at each of the conducted experiments and compare them against each other, and connect our findings to the relevant theory. Next, we discuss the learning outcomes of this thesis. Finally, using this learning outcome, we will examine the thesis' future development.

The system implemented for this thesis is relatively comprehensive, and many experiments were run with faulty implementations that were uncovered after the experiments' results were ready. The faulty implementations resulted in much time being cut from the available training time for the final experiments.

When it comes to the Seq2Tile LSTM generation model versus the Seq2Seq LSTM generation model, they had some advantages and disadvantages. One advantage of the Seq2Tile LSTM model is that the output shape of the network allows trained models to work with different generation sizes without having to train the model from scratch. The generator predicts tiles one-by-one, and the input works like a sliding window. The number of generated columns and tiles per column can be changed and tested with the same model. The major disadvantage with the Seq2Tile LSTM model is that it is very slow, both when it comes to generating new tiles and training the generator. The reason is that the model has to produce one prediction per generated tile.

While working on the experiments with the LSTM generation, we considered the size of the replay memory. The size was initially set to 10,000. However, this may lead to the generator training on outdated experience as the Mario agent improves over time. A large replay memory allows early experience from when the Mario agent was not very good at traversing the level to be included in training for a long time. To avoid this, we set the replay memory size to 100. The smaller size will make the generator train on more recent

and relevant experience.

Because the implemented system consists of a multi-agent environment with distributed workers updating the Mario agent’s policy, the training process requires a substantial amount of processing power. Additionally, the game runs extra slowly when there are a significant amount of enemies present within the update radius of Mario. Since we, unfortunately, did not have access to the university’s best hardware, the training process was far more time-consuming than we had anticipated. We trained the seq2seq generator model for two days straight on the best hardware we had access to, and it only completed 12,200 episodes. Considering that the graph for the generator reward never converged, we would deem the training amount insufficient.

5.1 Learning outcomes

As the Generative Playing System described in this thesis is built upon the *Super Mario Bros.* Python implementation by Marblexu, a significant amount of time was spent to get familiar with the implementation. The implementation was made using the pygame library, and throughout this work, we became more familiar with and obtained a better understanding of the library.

The implemented framework is incorporated into an OpenAI gym environment. As such, we learned how to design and implement a gym environment from scratch using the gym library.

The Mario agent was implemented through the Ray RLlib library, which is a library focusing on scalability and distributed workflow in RL applications. We learned a great deal about RLlib, including how distributed workers are used to update a central policy, and how to use RLlib with a custom gym environment.

Q-Learning was used extensively for the implementation of the generator, and the process of implementing the generator was highly educational. We learned how an experience

replay is used to store experience and how Q-values are updated through a Q-function. Furthermore, we learned how to design and tune a reward function for an RL agent to suit our needs.

Throughout the writing of this thesis, we have learned about current state-of-the-art techniques within PCG, and we have gained a more profound understanding of the RL paradigm in general.

5.2 Future development

This thesis has created a framework for conducting experiments within reinforcement learning in Super Mario Bros. This framework can be used and developed further for similar projects revolving around multi-agent competition in Super Mario Bros.

One major issue we encountered while working with the RILib library was that it would occasionally cause the program to crash, either due to a shortage of memory, or some other unknown cause. In order to mitigate this problem, we let our program simply restart and continue from where it left off. This solution did cause some unwanted side-effects, such as the fragmented graphs showing Mario’s performance. Additionally, each time the program restarted, it would reset all the variables other than the agent models and the replay memory, which were saved to disk. As a result, variables such as epsilon were reset multiple times during training, which would drastically affect the generator’s behavior. The program would also have to be restarted manually from time to time, which further restricted how much the agents were trained. Therefore, finding the cause of the program crashing and fixing it should be prioritized in the future.

The tested generator models were relatively similar. However, they do not need to be. Different network models could be tested in the future, such as convolutional models, which may be more adept at spotting patterns in input data and correlate this to the output. Additionally, the experiments rely on a significant number of hyperparameters,

which could be optimized further to achieve better results.

The Mario agent utilized the Ape-X DQN algorithm for training, which is a reasonable choice considering the algorithm has achieved excellent results in 57 Atari games compared to other reinforcement learning algorithms [31]. However, other algorithms can also be tested for training the Mario agent, such as Importance Weighted Actor-Learner Architectures (IMPALA) or Proximal Policy Optimization (PPO).

The Mario agent’s observation space includes several tiles that the generator is unable to create, which means the observation space of the Mario agent can be decreased. Another change that can be made is to convert from a normalized observation space where each tile is represented as a number in the range $[0, 1]$, to a one-hot encoded representation. These changes could improve the Mario agent’s performance since the various tiles become more distinct from each other.

An attempt was made to implement a curiosity-based reward for the Mario agent, as this will improve the Mario agent’s performance on chaotic maps. However, the existing implementations of curiosity are challenging to incorporate into our implementation. The problem mainly lies in making the curiosity implementation cooperate with RLLib. However, it would be beneficial to include in the future, if possible.

RLLib allows the Mario agent to use distributed workers in order to explore the environment. With distributed workers, each worker has a separate python process, and each worker is interacting with its own instantiated environment. Since the generator is instantiated inside the environment, each worker is competing with a separate generator, and the generators have no way to communicate. A drawback of this design is that only one of the generators’ models are saved to disk as a result of the generator with the least amount of training progress overwriting the previously saved models with its models. The issue can be fixed by swapping the environment type RLLib instantiates for the Mario agent. Instead of using a single-agent environment, a multi-agent environment can be used, which is described in the Ray RLLib documentation [32].

As previously mentioned, the generator was trained with an experience replay of size 100 due to the problem of training on outdated experience. Implementing a *prioritized experience replay* is an alternative method for mitigating the problem of training on outdated or irrelevant experience. This type of experience replay is utilized in the Ape-X DQN algorithm used by the Mario agent and could be implemented in the generator in the future.

Optimizing the implementation of the game could make it run faster and allow quicker training. The generator sometimes filled the whole map with enemies or animated blocks, which made the game run slowly. We changed the game to only update objects within a certain radius from Mario, which made the game run significantly faster. However, we believe the implementation could be further optimized, allowing even faster training.

6 | Conclusion

In this thesis, a system consisting of two symbiotic reinforcement learning agents was implemented into a Super Mario Bros. framework. The goal was to have the agents engage in a competition, where the agent controlling Mario would attempt to navigate through the generator's challenging but completable levels. The generator was trained without any prior experience of how the game works or what its goal was. The Mario agent utilized the Ape-X DQN algorithm while the generator implemented a DQN-based algorithm with LSTM. It is possible to create new level content that the Mario agent can interact with by giving the generator a sequence of previously generated tiles as input to predict Q-values of new tiles. The results show that the amount of training is insufficient to draw any immediate conclusions regarding the difficulty of the generated content. However, observing the generated level content reveals that the generator has attempted to create structured obstacles for the Mario agent. If the generator can be trained more, we believe it will learn to create levels suited to the player's skill level.

References

- [1] L. Chen, J. Lingys, K. Chen, and F. Liu, “Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization.” <https://dl.acm.org/citation.cfm?doid=3230543.3230551>, Aug 2018. Accessed: Dec. 17, 2019.
- [2] Colah, “Understanding lstm networks.” <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Accessed: May. 27, 2020.
- [3] G. Developers, “Overview of gan structure.” https://developers.google.com/machine-learning/gan/gan_structure?hl=th. Accessed: May. 12, 2020.
- [4] roclark, “Super mario bros. dqn.” <https://github.com/roclark/super-mario-bros-dqn#Progress>. Accessed: May. 19, 2020.
- [5] T. Dietterich, “Overfitting and undercomputing in machine learning,” *ACM computing surveys (CSUR)*, vol. 27, no. 3, pp. 326–327, 1995. Accessed: May. 26, 2020.
- [6] C. Watkins and P. Dayan, “Technical note: Q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 05 1992.
- [7] V. Mnih, K. Kavakcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning.” <https://www.nature.com/articles/nature14236>, 2015. Accessed: Dec. 17, 2019.
- [8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets.” arxiv, <https://arxiv.org/pdf/1406.2661v1.pdf>, Jun 2014. Accessed: Feb. 18, 2020.
- [9] N. Shibuya, “Understanding generative adversarial networks.” <https://medium.com/activating-robotic-minds/>

- understanding-generative-adversarial-networks-4dafc963f2ef. Accessed: May. 19, 2020.
- [10] “What pcg is.” <http://pcg.wikidot.com/what-pcg-is>. Accessed: May. 27, 2020.
- [11] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Springer Publishing Company, Incorporated, 1st ed., 2016. Accessed: May. 05, 2020.
- [12] “Julian togelius.” <https://engineering.nyu.edu/faculty/julian-togelius>. Accessed: May. 21, 2020.
- [13] K. Compton and M. Mateas, “Procedural level design for platform games,” p. 109–111, 2006. Accessed: May. 12, 2020.
- [14] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, “The 2010 mario ai championship: Level generation track,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, pp. 332–347, Dec 2011. Accessed: Feb. 18, 2020.
- [15] W. J. Ridgman, “Statistical methods, ames: Iowa state university press (1989),” *The Journal of Agricultural Science*, vol. 115, no. 1, p. 76–79, 1990.
- [16] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha, “Launchpad: A rhythm-based level generator for 2-d platformers,” p. 16, 03 2011. Accessed: May. 13, 2020.
- [17] F. Mourato, M. Santos, and F. Birra, “Automatic level generation for platform videogames using genetic algorithms,” p. 8, 11 2011. Accessed: May. 13, 2020.
- [18] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms,” *IEEE Transactions on Games*, vol. 11, no. 3, pp. 195–214, 2019. Accessed: May. 07, 2020.

- [19] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, “Procedural level generation improves generality of deep reinforcement learning,” *CoRR*, vol. abs/1806.10729, 2018. Accessed: May. 20, 2020.
- [20] “Openai gym.” <https://gym.openai.com/>. Accessed: May. 25, 2020.
- [21] “Openai baselines.” <https://openai.com/blog/openai-baselines-dqn/>. Accessed: May. 25, 2020.
- [22] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, “Pcgrl: Procedural content generation via reinforcement learning.” arxiv, <https://arxiv.org/pdf/2001.09212v1.pdf>, Jan 2020. Accessed: Jan. 31, 2020.
- [23] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, “Evolving mario levels in the latent space of a deep convolutional generative adversarial network,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO ’18, (New York, NY, USA), p. 221–228, Association for Computing Machinery, 2018. Accessed: Jan. 13, 2020.
- [24] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” 2011. Accessed: May. 26, 2020.
- [25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. Accessed: May. 26, 2020.
- [26] A. Summerville and M. Mateas, “Super mario as a string: Platformer level generation via lstms,” 2016. Accessed: Feb. 25, 2020.
- [27] P. Bontrager and J. Togelius, “Fully differentiable procedural content generation through generative playing networks,” 2020. Accessed: May. 07, 2020.
- [28] Marblexu, “Pythonsupermario.” Github, <https://github.com/marblexu/PythonSuperMario>, 2019. Accessed: Jan. 23, 2020.

- [29] S. Kaushik, “Engineering intelligent nlp applications using deep learning - part 2.” <https://www.slideshare.net/saurabhkaushikin/engineering-intelligent-nlp-applications-using-deep-learning-part-2>. Accessed: May. 26, 2020.
- [30] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, “Ray rllib: A composable and scalable reinforcement learning library,” *CoRR*, vol. abs/1712.09381, 2017.
- [31] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, “Distributed prioritized experience replay.” <https://arxiv.org/pdf/1803.00933.pdf>. Accessed: May. 15, 2020.
- [32] “Multi-agent and hierarchical.” The Ray Team Revision, <https://docs.ray.io/en/master/rllib-env.html#multi-agent-and-hierarchical>. Accessed: May. 21, 2020.

A | Mario observation space

Table A.1 shows the exhaustive list of tiles and entities included in the Mario agent's observation space.

Image	Name	ID	Value
	star	x	0.0
	1-up	u	0.07142857142857142
	mushroom	m	0.14285714285714285
	fireflower	i	0.21428571428571427
	fireball	o	0.2857142857142857
	flagpole	f	0.3571428571428571
	air	—	0.42857142857142855
	box	q	0.5
	brick	b	0.5714285714285714
	solid tile	s	0.6428571428571428

	goomba	<i>o</i>	0.7142857142857142
	koopa	<i>1</i>	0.7857142857142857
	fly koopa	<i>2</i>	0.8571428571428571
	fire koopa (bowser)	<i>5</i>	0.9285714285714285
	fire	<i>w</i>	1.0

Table A.1: Tiles and entities in the Mario agent’s observation space.

B | Generator action space

Table B.1 shows the exhaustive list of tiles and entities included in the generator's action space.

Image	Name	ID
	air	—
	box	q
	brick	b
	solid tile	s
	goomba	o
	koopa	1

Table B.1: Tiles and entities in the generator's observation space.

C | Code listings

Listing C.1 shows the reward function for the Mario agent.

Listing C.1: Reward function for the Mario agent.

```
1 def _reward(self):
2     """Mario reward function"""
3
4     # Current x-value of mario
5     current_x = self.game.state_dict[c.LEVEL].player.rect.x
6
7     # Difference in current x-value and last x-value
8     reward = current_x - self.mario_x_last
9
10    # Update last x-value
11    self.mario_x_last = current_x
12
13    # Time left on game clock
14    clock_now = self.game.state_dict[c.LEVEL].overhead_info.time
15
16    # Difference in remaining time. Clock counts down from 300,
17    # hence no clock tick: reward += 0, clock tick: reward += -1
18    reward += self.clock_last - clock_now
19
20    # Update last clock value
21    self.clock_last = clock_now
22
23    # If mario is dead, set reward to -15
24    if self.game.state_dict[c.LEVEL].player.dead:
25        reward = -15
26
27    return reward
```

Listing C.2 shows the *generate* method in *level_gen.py*.

Listing C.2: *generate* method in *level_gen.py*.

```
1 def generate(self):
2     tiles = {}
3     for item in self.GEN_DICT.items():
4         item = item[-1]
5         tiles[item] = []
6
7     if self.read:
8         line_num = 0
9         limit = self.gen_line + c.GEN_LENGTH
10    with open(self.map_gen_file) as file:
11        for line in file:
12            if line_num >= limit:
13                break
14            if line_num >= self.gen_line:
15                tiles = self.build_tiles_dict(tiles, line)
16            line_num += 1
17
18            if self.gen_line >= self.gen_file_length:
19                self.read = False
20
21    else:
22        new_terrain = []
23        q_values = []
24        if self.map_data[c.GEN_BORDER] >= self.map_data[c.MAP_FLAGPOLE][0]['x'] - c.
25        ↪ GEN_PX_LEN or c.ONLY_GROUND:
26            for _ in range(c.GEN_LENGTH):
27                new_terrain.append(str(c.SOLID_ID * 2))
28        else:
29            new_terrain, q_values = self.generator.generate()
30
31        flag_x = self.map_data[c.MAP_FLAGPOLE][0]['x']
32
33        # Check if the generator has reached the flag
34        if not self.generator_done:
35            gen_border = self.player.rect.x + self.player.rect.w + c.GEN_DISTANCE + c.
36            ↪ GEN_PX_LEN
37            self.generator_done = gen_border >= flag_x
38            self.gen_list.append({c.GEN_LINE: self.gen_line,
39                                c.DONE: self.generator_done})
40
41    # Check if self.gen_list entries still should be added to generator replay memory
```

```

39         elif not self.gen_list_done:
40             offset = 4 * c.GEN_PX_LEN + int(c.MEMORY_LENGTH * c.TILE_SIZE / self.
41             ↪ generator.tiles_per_col)
42             gen_border = self.player.rect.x + self.player.rect.w + offset
43             self.gen_list_done = gen_border >= flag_x
44
45             # Check if mario has reached the flag
46             elif not self.mario_done:
47                 gen_border = self.player.rect.x + self.player.rect.w
48                 self.mario_done = gen_border >= flag_x
49
50             # Build tiles dict using encoded tiles in new_terrain
51             for n, line in enumerate(new_terrain):
52                 try:
53                     tiles = self.build_tiles_dict(tiles, line, q_values[n])
54                 except IndexError:
55                     tiles = self.build_tiles_dict(tiles, line)
56
57             # Add new tiles and entities to respective sprite groups
58             self.setup_brick_and_box(tiles['bricks'], tiles['boxes'])
59             self.setup_solid_tile(tiles['steps'], self.step_group, 0, 16)
60             self.setup_solid_tile(tiles['ground'], self.ground_group, 0, 0)
61             self.setup_solid_tile(tiles['solid'], self.solid_group, 432, 0)
62             self.setup_enemies(tiles['enemies'])
63
64             if self.gen_line > c.PLATFORM_LENGTH and c.PRINT_Q_VALUES:
65                 for q_data in tiles['air']:
66                     textsurface = self.q_font.render(q_data[2], True, (255, 255, 255))
67                     self.background.blit(textsurface, (q_data[0] + 5, q_data[1] + 15))
68
69             if c.TRAIN_GEN and not self.mario_done and self.insert_zero_index:
70                 # Update weights in generator network and increment training_sessions if model is
71                 ↪ trained
72                 self.training_sessions += self.generator.train()
73
74                 if not (self.training_sessions % c.GEN_MODEL_SAVE_INTERVAL) \
75                     and self.training_sessions > self.generator.start_checkpoint:
76                     self.generator.save_model(num=self.training_sessions)
77                     self.generator.save_replay_memory(num=self.training_sessions)

```

Listing C.3 shows the beginning of the *train* method in *generation.py*.

Listing C.3: *train* method in *generation.py*.

```
1 def train(self):
2     """Train the generator model if replay memory is large enough.
3     Return 1 if the generator model is trained, 0 otherwise"""
4
5     # Only start training if we have enough transitions in replay memory
6     if len(self.replay_memory) < c.MIN_REPLAY_MEMORY_SIZE:
7         return 0
8
9     print("Training on", c.MINIBATCH_SIZE, "transitions")
10
11    # Get a minibatch of random samples from replay memory
12    minibatch = random.sample(self.replay_memory, c.MINIBATCH_SIZE)
13
14    # Get current states from minibatch and create list of predicted sequences
15    current_states = np.array([transition[0] for transition in minibatch])
16    current_predicted_sequences = np.array(self.predict_new_states(current_states,
17                                          ↪ return_qs=True))
18
19    # Get future states from minibatch, and create new list of sequence predictions
20    new_current_states = np.array([transition[3] for transition in minibatch])
21    future_predicted_sequences = np.array(self.predict_new_states(new_current_states,
22                                          ↪ return_qs=True))
23
24    X = []
25    y = []
26
27    # Enumerate transitions
28    for index, (current_state, action, reward, new_current_state, done) in enumerate(
29                                          ↪ minibatch):
30
31        # If not a terminal state, get new qs from future states, otherwise set it to
32        ↪ reward
33        if not done:
34            max_future_Qs = [np.max(qs) for qs in future_predicted_sequences[index]]
35            new_Qs = np.array([reward + c.DISCOUNT * fqs for fqs in max_future_Qs])
36        else:
37            new_Qs = np.array([reward for _ in range(len(future_predicted_sequences[index
38                                          ↪ ]))])
39
40        # Update Q-values for given state
```

```

36     current_qs = current_predicted_sequences[index]
37     for n, action_n in enumerate(action):
38         current_qs[n][action_n] = new_Qs[n]

```

Listing C.4 shows the fitting part of the *train* method in *generation.py* for the Seq2Tile generation.

Listing C.4: Fitting the Seq2Tile generation model.

```

1   # Insert sliding window states into X
2   generator_input = []
3   for i in range(self.gen_size):
4       state_slice = max(0, i - c.MEMORY_LENGTH)
5       tmp_state = np.concatenate((current_state[i:], action[state_slice:i]))
6       generator_input.append(self.one_hot_encode(tmp_state))
7
8   # Append to training data
9   X.append(generator_input)
10  y.append(current_qs)
11
12  # Fit on all transitions in minibatch
13  X = np.array(X)
14  y = np.array(y)
15  for i in range(len(X)):
16      self.generator.fit(X[i], y[i], batch_size=self.gen_size, verbose=0, shuffle=False
17      ↪ , callbacks=[self.tensorboard])
18
19  return 1

```

Listing C.5 shows the fitting part of the *train* method in *generation.py* for the Seq2Seq generation.

Listing C.5: Fitting the Seq2Seq generation model.

```
1  # One-hot encode current state from transition
2  generator_input = self.one_hot_encode(current_state)
3
4  # Append to training data
5  X.append(generator_input)
6  y.append(current_qs)
7
8  # Fit on all transitions in minibatch
9  X = np.array(X)
10 y = np.array(y)
11 self.generator.fit(X, y, batch_size=c.MINIBATCH_SIZE, verbose=0, shuffle=False,
12 ↪ callbacks=[self.tensorboard])
13 return 1
```

D | Experiment configurations

Listing D.1 shows *constants.py* which is the configuration file for the experiments.

Listing D.1: Configuration file *constants.py*

```
1 # Settings
2
3 __author__ = 'marble_xu'
4
5 DEBUG = False
6 DEBUG_START_X = 110
7 DEBUG_START_Y = 534
8
9 SCREEN_HEIGHT = 600
10 SCREEN_WIDTH = 800
11 SCREEN_SIZE = (SCREEN_WIDTH, SCREEN_HEIGHT)
12
13 ORIGINAL_CAPTION = 'Super Mario Bros.'
14
15 # COLORS
16 #          R      G      B
17 GRAY      = (100, 100, 100)
18 NAVYBLUE  = ( 60,  60, 100)
19 WHITE     = (255, 255, 255)
20 RED       = (255,   0,   0)
21 GREEN     = (  0, 255,   0)
22 FOREST_GREEN = ( 31, 162,  35)
23 BLUE      = (  0,   0, 255)
24 SKY_BLUE  = ( 39, 145, 251)
25 YELLOW    = (255, 255,   0)
26 ORANGE    = (255, 128,   0)
27 PURPLE   = (255,   0, 255)
28 CYAN      = (  0, 255, 255)
29 BLACK     = (  0,   0,   0)
30 NEAR_BLACK = ( 19,  15,  48)
31 COMBLUE   = (233, 232, 255)
32 GOLD      = (255, 215,   0)
33
```

```

34 BGCOLOR = WHITE
35
36
37 SIZE_MULTIPLIER = 2.5
38 BRICK_SIZE_MULTIPLIER = 2.69
39 SOLID_SIZE_MULTIPLIER = 2.69
40 BACKGROUND_MULTIPLIER = 2.679
41 GROUND_HEIGHT = SCREEN_HEIGHT - 62
42
43 GAME_TIME_OUT = 301
44
45 # STATES FOR ENTIRE GAME
46 MAIN_MENU = 'main menu'
47 LOAD_SCREEN = 'load screen'
48 TIME_OUT = 'time out'
49 GAME_OVER = 'game over'
50 LEVEL = 'level'
51
52 # MAIN MENU CURSOR STATES
53 PLAYER1 = '1 PLAYER GAME'
54 PLAYER2 = '2 PLAYER GAME'
55
56 # GAME INFO DICTIONARY KEYS
57 BASE_FPS = 'base fps'
58 COIN_TOTAL = 'coin total'
59 SCORE = 'score'
60 TOP_SCORE = 'top score'
61 LIVES = 'lives'
62 CURRENT_TIME = 'current time'
63 LEVEL_NUM = 'level num'
64 PLAYER_NAME = 'player name'
65 PLAYER_MARIO = 'mario'
66 PLAYER_LUIGI = 'luigi'
67
68 # MAP GENERATION
69 GENERATE_MAP = True
70 RANDOM_GEN = False
71 TRAIN_GEN = True
72 LOAD_GEN_MODEL = True
73 INSERT_GROUND = True
74 ONLY_GROUND = False
75 READ = True
76 WRITE = False
77 SAVE_LEVEL = False
78 PRINT_GEN_REWARD = False
79 PRINT_Q_VALUES = False

```

```

80 SNAKING = True
81 CHUNK_BASED_GREEDY = False # If False -> Tile based greedy (greedy determined for each
     ↪ individual tile)
82 GREEDY = False # Only perform greedy actions
83 UPDATE_RADIUS = 1 * SCREEN_WIDTH
84 GEN_DISTANCE = 2 * SCREEN_WIDTH
85 GEN_HEIGHT = 580
86 GEN_LENGTH = 5
87 PLATFORM_LENGTH = 5
88 Y_OFFSET = 64
89 COL_HEIGHT = 13
90 TILE_SIZE = 43
91 GEN_PX_LEN = GEN_LENGTH * TILE_SIZE
92 GEN_MODEL_SAVE_INTERVAL = 100 # Number of training sessions
93 REP_MEM = 'rep_mem'
94 GEN_BORDER = 'gen_border'
95 GEN_LINE = 'gen_line'
96 Timestep = 'timestep'
97 PLAYER_X = 'player_x'
98 REWARD = 'reward'
99 OPTIMAL_V = 'optimal_v'
100 DONE = 'done'
101
102 # GENERATION IDENTIFIERS
103 AIR_ID = '_'
104 GROUND_ID = 'g'
105 BRICK_ID = 'b'
106 BOX_ID = 'q'
107 STEP_ID = 't'
108 SOLID_ID = 's'
109 PIPE_ID = 'p'
110 FLAG_ID = 'f'
111 COIN_ID = 'c'
112 MUSHROOM_ID = 'm'
113 FIREFLOWER_ID = 'i'
114 FIREBALL_ID = 'o'
115 STAR_ID = 'x'
116 LIFE_ID = 'u'
117 GOOMBA_ID = '0'
118 KOOPA_ID = '1'
119 FLY_KOOPA_ID = '2'
120 PIRANHA_ID = '3'
121 FIRESTICK_ID = '4'
122 FIRE_KOOPA_ID = '5'
123 FIRE_ID = 'w'
124 ENEMY_IDS = [GOOMBA_ID, KOOPA_ID, FLY_KOOPA_ID, PIRANHA_ID, FIRESTICK_ID, FIRE_KOOPA_ID,

```

```

    ↪ FIRE_ID]
125 SOLID_IDS = [GROUND_ID, BRICK_ID, BOX_ID, STEP_ID, SOLID_ID, PIPE_ID]
126
127 TILES = [
128     STAR_ID,
129     LIFE_ID,
130     MUSHROOM_ID,
131     FIREFLOWER_ID,
132     FIREBALL_ID,
133     FLAG_ID,
134     AIR_ID,
135     BOX_ID,
136     BRICK_ID,
137     SOLID_ID,
138     GOOMBA_ID,
139     KOOPA_ID,
140     FLY_KOOPA_ID,
141     FIRE_KOOPA_ID,
142     FIRE_ID
143 ]
144
145 GENERATOR_TILES = [
146     AIR_ID,
147     BOX_ID,
148     BRICK_ID,
149     SOLID_ID,
150     GOOMBA_ID,
151     KOOPA_ID
152 ]
153
154 # GENERATION NETWORK PARAMETERS
155 LEARNING_RATE = 0.001
156 MEMORY_LENGTH = 64
157 EPSILON_DECAY = 0.995
158 MIN_EPSILON = 0.01
159 REPLAY_MEMORY_SIZE = 100
160 MIN_REPLAY_MEMORY_SIZE = 50
161 MINIBATCH_SIZE = 10
162 DISCOUNT = 0.99
163
164 # GYM COMPONENTS
165 ENV_NAME = 'MarioEnv'
166 ACTION_KEYS = 323
167 EVALUATE = False
168 SKIP_MENU = True
169 HUMAN_PLAYER = False

```

```

170 LOAD_CHECKPOINT = True
171 PRINT_OBSERVATION = False
172 PRINT_LEVEL = False
173 OBS_RADIUS = 10
174 OBS_SIZE = 2 * OBS_RADIUS + 1
175 OBS_FRAMES = OBS_SIZE
176
177 # MAP COMPONENTS
178 MAP_IMAGE = 'image_name'
179 MAP_MAPS = 'maps'
180 SUB_MAP = 'sub_map'
181 MAP_GROUND = 'ground'
182 MAP_PIPE = 'pipe'
183 PIPE_TYPE_NONE = 0
184 PIPE_TYPE_IN = 1           # can go down in the pipe
185 PIPE_TYPE_HORIZONTAL = 2   # can go right in the pipe
186 MAP_STEP = 'step'
187 MAP_BRICK = 'brick'
188 STEP_NUM = 'step_num'
189 BRICK_NUM = 'brick_num'
190 TYPE_NONE = 0
191 TYPE_COIN = 1
192 TYPE_STAR = 2
193 MAP_BOX = 'box'
194 TYPE_MUSHROOM = 3
195 TYPE_FIREFLOWER = 4
196 TYPE_FIREBALL = 5
197 TYPE_LIFEMUSHROOM = 6
198 MAP_ENEMY = 'enemy'
199 ENEMY_TYPE_GOOMBA = 0
200 ENEMY_TYPE_KOOPA = 1
201 ENEMY_TYPE_FLY_KOOPA = 2
202 ENEMY_TYPE_PIRANHA = 3
203 ENEMY_TYPE_FIRESTICK = 4
204 ENEMY_TYPE_FIRE_KOOPA = 5
205 ENEMY_RANGE = 'range'
206 MAP_CHECKPOINT = 'checkpoint'
207 ENEMY_GROUPID = 'enemy_groupid'
208 MAP_INDEX = 'map_index'
209 CHECKPOINT_TYPE_ENEMY = 0
210 CHECKPOINT_TYPE_FLAG = 1
211 CHECKPOINT_TYPE_CASTLE = 2
212 CHECKPOINT_TYPE_MUSHROOM = 3
213 CHECKPOINT_TYPE_PIPE = 4     # trigger player to go right in a pipe
214 CHECKPOINT_TYPE_PIPE_UP = 5   # trigger player to another map and go up out of a pipe
215 CHECKPOINT_TYPE_MAP = 6      # trigger player to go to another map

```

```

216 CHECKPOINT_TYPE_BOSS = 7           # defeat the boss
217 MAP_FLAGPOLE = 'flagpole'
218 FLAGPOLE_TYPE_FLAG = 0
219 FLAGPOLE_TYPE_POLE = 1
220 FLAGPOLE_TYPE_TOP = 2
221 MAP_SLIDER = 'slider'
222 HORIZONTAL = 0
223 VERTICAL = 1
224 VELOCITY = 'velocity'
225 MAP_COIN = 'coin'
226
227 # COMPONENT COLOR
228 COLOR = 'color'
229 COLOR_TYPE_ORANGE = 0
230 COLOR_TYPE_GREEN = 1
231 COLOR_TYPE_RED = 2
232
233 # BRICK STATES
234 RESTING = 'resting'
235 BUMPED = 'bumped'
236 OPENED = 'opened'
237
238 # MUSHROOM STATES
239 REVEAL = 'reveal'
240 SLIDE = 'slide'
241
242 # Player FRAMES
243 PLAYER_FRAMES = 'image_frames'
244 RIGHT_SMALL_NORMAL = 'right_small_normal'
245 RIGHT_BIG_NORMAL = 'right_big_normal'
246 RIGHT_BIG_FIRE = 'right_big_fire'
247
248 # PLAYER States
249 STAND = 'standing'
250 WALK = 'walk'
251 JUMP = 'jump'
252 FALL = 'fall'
253 FLY = 'fly'
254 SMALL_TO_BIG = 'small to big'
255 BIG_TO_FIRE = 'big to fire'
256 BIG_TO_SMALL = 'big to small'
257 FLAGPOLE = 'flag pole'
258 WALK_AUTO = 'walk auto'      # ignoring key input in this state
259 END_OF_LEVEL_FALL = 'end of level fall'
260 IN_CASTLE = 'in castle'
261 DOWN_TO_PIPE = 'down to pipe'

```

```

262 UP_OUT_PIPE = 'up out of pipe'
263
264 # PLAYER FORCES
265 PLAYER_SPEED = 'speed'
266 WALK_ACCEL = 'walk_accel'
267 RUN_ACCEL = 'run_accel'
268 JUMP_VEL = 'jump_velocity'
269 MAX_Y_VEL = 'max_y_velocity'
270 MAX_RUN_SPEED = 'max_run_speed'
271 MAX_WALK_SPEED = 'max_walk_speed'
272 SMALL_TURNAROUND = .35
273 JUMP_GRAVITY = .31
274 GRAVITY = 1.01
275
276 # LIST of ENEMIES
277 GOOMBA = 'goomba'
278 KOOPA = 'koopa'
279 FLY_KOOPA = 'fly koopa'
280 FIRE_KOOPA = 'fire koopa'
281 FIRE = 'fire'
282 PIRANHA = 'piranha'
283 FIRESTICK = 'firestick'
284
285 # GOOMBA Stuff
286 LEFT = 'left'
287 RIGHT = 'right'
288 JUMPED_ON = 'jumped on'
289 DEATH_JUMP = 'death jump'
290
291 # KOOPA STUFF
292 SHELL_SLIDE = 'shell slide'
293
294 # FLAG STATE
295 TOP_OF_POLE = 'top of pole'
296 SLIDE_DOWN = 'slide down'
297 BOTTOM_OF_POLE = 'bottom of pole'
298
299 # FIREBALL STATE
300 FLYING = 'flying'
301 BOUNCING = 'bouncing'
302 EXPLODING = 'exploding'
303
304 # IMAGE SHEET
305 ENEMY_SHEET = 'smb_enemies_sheet'
306 ITEM_SHEET = 'item_objects'
```