



# **Solving Tetris Using Reinforcement Learning**

by

Divan van der Bank  
23603526

Mechatronics Project 488

Department of Mechanical and Mechatronic Engineering  
Stellenbosch University

Supervisor: Prof. H. Kamper

November 2023

## Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.

*Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.*

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.

*I agree that plagiarism is a punishable offence because it constitutes theft.*

3. Ek verstaan ook dat direkte vertalings plagiaat is.


*I also understand that direct translations are plagiarism.*

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelike aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.

*Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism*

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.

*I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.*

|  |   |
|--|---|
| <b>23603526</b>                                  |  |
| Studentenommer / <i>Student number</i>           | Handtekening / <i>Signature</i>   |
| <b>D.K. van der Bank</b>                         | <b>27 October 2023</b>  |
| Voorletters en van / <i>Initials and surname</i> | Datum / <i>Date</i>   |

# Executive Summary

|  |
|--|
| <b>Title of Project</b>  |
| Solving Tetris using reinforcement learning.   |
| <b>Objectives</b>  |
| Developing a stable Tetris environment for the development and testing of the reinforcement learning algorithms. Custom reinforcement learning models are developed to solve the Tetris game.  |
| <b>What is current practice, and what are its limitations?</b>   |
| <p>Atari games such as <b>Ms Pacman</b> and published OpenAI Gym environments such as <b>Cartpole</b> are the main reinforcement learning testing platforms due to the availability and simplicity of the environments.</p> <p>As for reinforcement learning algorithms, Q-learning and policy gradient methods are the industry standard in 2023, with policy gradient methods outperforming Q-learning. The limitation is that using the same environments can lead to reinforcement learning algorithms overfitting to these cases, and a new environment, such as Tetris, could introduce new problems and improvements to the algorithms.</p> |
| <b>What is new in this project?</b>  |
| The reinforcement learning algorithms are designed from first principles and tested in a custom Tetris environment. The developed algorithms are tested using different environment setups to determine how the algorithms learn rather than just trying to optimize the algorithm.  |
| <b>If the project is successful, how will it make a difference?</b>  |
| The custom reinforcement learning models and the Tetris testing environment can be used as the building blocks for future projects that require reinforcement learning. This gives developers an understanding of which methods to use and the type of setups and environment that work best with the reinforcement learning method.   |
| <b>What are the risks to the project being a success? Why is it expected to be successful?</b>   |
| The processing power of the computer on which the reinforcement learning agent is trained can limit the training efficiency, and the time to train a model can be extensive. This is mitigated by using powerful faculty computers and planning for the training times of the agents around the load shedding schedule.  |
| <b>What contributions have/will other students made/make?</b>  |

A previous student has created a Sarsa reinforcement learning algorithm with linear approximation to play a simplified version of Tetris. This project does not use the implementations from the previous work that is done, but rather custom algorithms written from scratch.

**Which aspects of the project will carry on after completion and why?**

The Tetris environment and the selected reinforcement learning models can be carried over into a possible Master's investigation of the feasibility of using the reinforcement learning models to teach a human to play arcade games.

**What arrangements have been/will be made to expedite continuation?**

The reinforcement learning models are uploaded to a GitHub repository, where future developers can find and use the custom models and implementations. The code for this project can be found at [https://github.com/Divanvdb/RL\\_Tetris\\_V1.0](https://github.com/Divanvdb/RL_Tetris_V1.0).

# ECSA Outcomes

| <i>ECSA Outcomes Assessed in this Module</i>  |   |
|---|---|
| <i>ECSA Outcome</i>   | <i>Addressed in Sections</i>                  |
| <b>ELO 1.Problem solving:</b> Demonstrate competence to identify, assess, formulate and solve convergent and divergent engineering problems creatively and innovatively.  | 3.2, 3.3.2, 4.2, 4.3.7, 5.2                   |
| <b>ELO 2.Application of scientific and engineering knowledge:</b> Demonstrate competence to apply knowledge of mathematics, basic science and engineering sciences from first principles to solve engineering problems  | 2.2.2, 2.3, 4.2, 4.3.6, 5.2.1                 |
| <b>ELO 3.Engineering Design:</b> Demonstrate competence to perform creative, procedural and non-procedural design and synthesis of components, systems, engineering works, products or processes.                       | 3.2, 3.3, 4.2, 4.3.7                          |
| <b>ELO 5.Engineering methods, skills and tools, including Information Technology:</b> Demonstrate competence to use appropriate engineering methods, skills and tools, including those based on information technology. | 3.2, 3.3, 4.2.1 - 4.2.5, 4.3.1 - 4.3.5, 4.3.7 |
| <b>ELO 6.Professional and technical communication:</b> Demonstrate competence to communicate effectively, both orally and in writing, with engineering audiences and the community at large.                            | Oral presentation, 1, 2.2.1, 4.2, 4.3, 6.1    |
| <b>ELO 8.Individual, Team and Multidisciplinary Working:</b> Demonstrate competence to work effectively as an individual, in teams and in multi-disciplinary environments   | 3.3, 4.2.6, 4.3.8                             |
| <b>ELO 9.Independent Learning Ability:</b> Demonstrate competence to engage in independent learning through well-developed learning skills.   | 2, 6.2  |

# Acknowledgements

I want to thank my supervisor, Prof. Herman Kamper, for guiding me through this project and motivating me to deliver only the best quality of work.

Prof. Herman Engelbrecht and Prof. Hugo Touchette presented a course on reinforcement learning and helped me understand the algorithms' deeper mechanics. I also want to thank Tristan Legg, who didn't hesitate to help me when I got stuck with the implementations of the algorithms.

# Contents

|  |             |
|--|-------------|
| <b>Declaration</b>   | <b>i</b>    |
| <b>List of Figures</b>   | <b>ix</b>   |
| <b>List of Tables</b>  | <b>xi</b>   |
| <b>Nomenclature</b>  | <b>xiii</b> |
| <b>1. Introduction</b>   | <b>1</b>    |
| 1.1. Background . . . . .  | 1           |
| 1.2. Project Objectives . . . . .                                  | 2           |
| 1.3. Motivation . . . . .  | 2           |
| <b>2. Background: Reinforcement learning</b>                       | <b>3</b>    |
| 2.1. Main reinforcement learning objective . . . . .               | 3           |
| 2.2. Markov decision processes . . . . .                           | 4           |
| 2.2.1. Structure of a Markov decision process . . . . .            | 4           |
| 2.2.2. Value function and Bellman equation . . . . .               | 5           |
| 2.2.3. Actions and policies . . . . .                              | 6           |
| 2.3. Reinforcement learning setups and configurations . . . . .    | 6           |
| 2.3.1. Model-free and model-based reinforcement learning . . . . . | 7           |
| 2.3.2. Exploration vs. exploitation . . . . .                      | 7           |
| 2.3.3. Online vs. offline learning . . . . .                       | 8           |
| 2.3.4. On-policy vs. off-policy setup . . . . .                    | 9           |
| 2.4. Types of reinforcement learning algorithms . . . . .          | 9           |
| 2.4.1. Q-learning . . . . .  | 9           |
| 2.4.2. Policy gradient methods . . . . .                           | 10          |
| 2.5. Chapter summary . . . . .                                     | 10          |
| <b>3. Development of the Tetris environment</b>                    | <b>11</b>   |
| 3.1. OpenAI Gym environment . . . . .                              | 11          |
| 3.2. Tetris game environment . . . . .                             | 12          |
| 3.3. Applying reinforcement learning to Melax's Tetris . . . . .   | 14          |
| 3.3.1. Observation space . . . . .                                 | 15          |
| 3.3.2. Action space setup . . . . .                                | 16          |

|  |           |
|--|-----------|
| 3.3.3. Reward function . . . . .   | 17        |
| 3.4. Chapter summary . . . . .   | 17        |
| <b>4. Reinforcement learning methods development and testing in Melax's Tetris</b> | <b>18</b> |
| 4.1. Progress visualization with TensorBoard . . . . .                             | 18        |
| 4.2. Deep Q-learning . . . . .   | 19        |
| 4.2.1. Value function estimators . . . . .   | 20        |
| 4.2.2. Replay buffer . . . . .   | 20        |
| 4.2.3. $\epsilon$ -greedy policy . . . . .   | 21        |
| 4.2.4. Training loop . . . . .   | 22        |
| 4.2.5. Policy and target network optimization . . . . .                            | 22        |
| 4.2.6. Tetris configurations and results . . . . .                                 | 24        |
| 4.2.7. Final DQN results . . . . .   | 28        |
| 4.3. Proximal policy optimization . . . . .  | 30        |
| 4.3.1. Training loop . . . . .   | 31        |
| 4.3.2. PPO batched experience . . . . .  | 32        |
| 4.3.3. Actor-critic networks . . . . .   | 32        |
| 4.3.4. Action function . . . . .   | 33        |
| 4.3.5. Advantage estimate calculation . . . . .                                    | 33        |
| 4.3.6. Proximal policy optimization theory . . . . .                               | 33        |
| 4.3.7. Proximal policy optimization implementation . . . . .                       | 35        |
| 4.3.8. Melax's Tetris configurations and results . . . . .                         | 37        |
| 4.3.9. Final PPO model and conclusions . . . . .                                   | 38        |
| 4.4. Comparing DQN and PPO models . . . . .  | 39        |
| 4.5. Chapter summary . . . . .   | 41        |
| <b>5. Reinforcement learning for full Tetris</b>                                   | <b>42</b> |
| 5.1. Full Tetris challenges . . . . .  | 42        |
| 5.2. Different approaches for the DQN model for full Tetris . . . . .              | 43        |
| 5.2.1. Convolutional neural networks and frame stacking . . . . .                  | 43        |
| 5.2.2. Simplified Observation Space . . . . .                                      | 45        |
| 5.2.3. Action Space . . . . .  | 45        |
| 5.3. Final model structure and hyperparameters . . . . .                           | 46        |
| 5.3.1. Results of the final model . . . . .  | 47        |
| 5.4. Chapter summary . . . . .   | 48        |
| <b>6. Summary and conclusion</b>   | <b>49</b> |
| 6.1. Findings and discussion . . . . .   | 49        |
| 6.2. Final thoughts . . . . .  | 50        |



|   |           |
|---|-----------|
| <b>Bibliography</b>   | <b>51</b> |
| <b>A. Project planning schedule</b>                             | <b>53</b> |
| <b>B. Techno economic analysis</b>                              | <b>54</b> |
| B.1. Budget . . . . .   | 54        |
| B.2. Technical impact . . . . .                                 | 54        |
| B.3. Return on investment . . . . .                             | 55        |
| B.4. Potential for commercialization . . . . .                  | 55        |
| <b>C. Project risk assessment</b>                               | <b>56</b> |
| <b>D. Responsible use of resources and End-of-Life strategy</b> | <b>57</b> |
| D.1. Responsible use of resources . . . . .                     | 57        |
| D.2. End of life strategy report . . . . .                      | 57        |

# List of Figures

|   |    |
|---|----|
| 2.1. Implementation of the RL loop . . . . .                              | 4  |
| 2.2. Structure of an MDP . . . . .  | 5  |
| 2.3. Model-free vs. model-based RL . . . . .                              | 7  |
| 3.1. Full $20 \times 10$ Tetris field . . . . .                           | 13 |
| 3.2. Full $20 \times 10$ Tetris shapes . . . . .                          | 14 |
| 3.3. Melax's $10 \times 5$ Tetris environment . . . . .                   | 15 |
| 3.4. Melax's $10 \times 5$ Tetris shapes . . . . .                        | 15 |
| 4.1. TensorBoard log example . . . . .                                    | 19 |
| 4.2. DQN: Policy network structure . . . . .                              | 21 |
| 4.3. DQN: Exploration vs. exploitation with annealing threshold . . . . . | 22 |
| 4.4. DQN: Training loop . . . . .   | 23 |
| 4.5. DQN: Optimization function for the policy network . . . . .          | 23 |
| 4.6. DQN: Custom model vs. SB3 model . . . . .                            | 25 |
| 4.7. DQN: Sparse reward function vs. shaped rewards function . . . . .    | 26 |
| 4.8. DQN: Pure vs. rule-based imitation learning . . . . .                | 27 |
| 4.9. DQN: Use of $\epsilon$ -greedy vs. greedy policy . . . . .           | 27 |
| 4.10. DQN: Direct vs. arcade action space in Melax's Tetris . . . . .     | 28 |
| 4.11. DQN: Final arcade model training . . . . .                          | 29 |
| 4.12. DQN: Distribution of the arcade model scores . . . . .              | 29 |
| 4.13. DQN: Final direct model training . . . . .                          | 30 |
| 4.14. PPO: Training loop . . . . .  | 31 |
| 4.15. PPO: Visual representation of the main objective . . . . .          | 35 |
| 4.16. PPO: Sparse reward function vs. shaped reward function . . . . .    | 37 |
| 4.17. PPO: Effects of learning rate changes . . . . .                     | 38 |
| 4.18. PPO: Direct vs. arcade action space . . . . .                       | 38 |
| 4.19. PPO: Final training . . . . .                                       | 39 |
| 4.20. PPO vs. DQN models in Melax's Tetris . . . . .                      | 40 |
| 5.1. DQN: General structure of the CNN setup . . . . .                    | 44 |
| 5.2. DQN: Effect of frame stacking . . . . .                              | 44 |
| 5.3. DQN: CNN vs. linear structure . . . . .                              | 45 |
| 5.4. DQN: Simplified vs. original observation space . . . . .             | 46 |

|   |    |
|---|----|
| 5.5. DQN: Final model for full Tetris . . . . . | 48 |
|---|----|

# List of Tables

|  |    |
|--|----|
| 4.1. Hyperparameter values for the final DQN model to be used for Melax's Tetris | 28 |
| 4.2. Results produced by the arcade DQN model . . . . .                          | 29 |
| 4.3. Results produced by the direct DQN model . . . . .                          | 30 |
| 4.4. Hyperparameter values for the final PPO model to be used for Melax's Tetris | 39 |
| 4.5. Results produced by the PPO model . . . . .                                 | 39 |
| 4.6. Results for different algorithms in Melax's Tetris . . . . .                | 41 |
| 5.1. Hyperparameter values for the final DQN Model to be used for full Tetris .  | 47 |
| 5.2. Results produced by the final DQN model . . . . .                           | 48 |
| B.1. Budget for outlined project schedule . . . . .                              | 54 |

# List of Algorithms

|   |    |
|---|----|
| 2.1. $\epsilon$ -greedy policy . . . . .                                      | 8  |
| 3.2. The custom RL testing environment must follow the Gym structure. . . . . | 12 |
| 4.3. Custom PPO Algorithm . . . . .   | 32 |
| 4.4. Calculating the advantage estimate . . . . .                             | 36 |

# Nomenclature

## Symbols

### Markov decision processes

|                      |   |
|----------------------|---|
| $P(S_{t+1} S_t = s)$ | Probability of reaching state $S_{t+1}$ given current state $S_t$ |
| $\mathbb{E}[X]$      | Expectation for the variable $X$                                  |
| $S_t$                | The current state   |
| $A_t$                | The selected action for the input state $S_t$                     |
| $R_t$                | Reward for taking action $A_t$ and reaching state $S_{t+1}$       |
| $G_t$                | Discounted cumulative reward from state $S_t$                     |
| $v(s)$               | Value function for state $s$                                      |
| $q_*(s)$             | Bellman optimality equation                                       |
| $\pi(s)$             | Current policy evaluating state $s$                               |

### Superscripts and subscripts

|           |  |
|-----------|--|
| $\hat{X}$ | Approximation for the function $X$     |
| $X'$      | Next value of $X$                      |
| $X_\pi$   | Evaluate $X$ at current policy $\pi$   |
| $X_*$     | Evaluate $X$ at optimal policy $\pi_*$ |
| $X_t$     | Evaluate $X$ at time step $t$          |

### General reinforcement learning

|               |                        |
|---------------|------------------------|
| $\alpha$      | Learning rate          |
| $\gamma$      | Reward discount factor |
| $\mathcal{S}$ | Set of all states      |
| $\mathcal{A}$ | Set of all actions     |
| $\mathcal{R}$ | Set of all rewards     |

**Deep Q-learning**

|                              |   |
|------------------------------|---|
| $\epsilon$                   | Threshold value for $\epsilon$ -greedy policy               |
| $Q(S_t, A_t)$                | State-action Q-value  |
| $\text{TD}_{\text{targets}}$ | Temporal difference targets for Q-learning loss calculation |
| $C$                          | Target steps for updating DQN value network                 |

**Proximal policy optimization**

|                              |   |
|------------------------------|---|
| $\pi_{\theta}(A_t S_t)$      | Probability of choosing action $A_t$ given state $S_t$ for current weights $\theta$ |
| $\hat{A}$                    | Advantage estimate  |
| $L^{\text{clip}}(\theta)$    | Clipped surrogate loss  |
| $L^{\text{value}}(\theta)$   | Critic network loss   |
| $L^{\text{entropy}}(\theta)$ | Entropy loss  |

## Acronyms and abbreviations

|     |                                  |
|-----|----------------------------------|
| A2C | Asynchronous Actor-Critic Method |
| CNN | Convolutional Neural Network     |
| CSV | Comma-Separated Values           |
| DQN | Deep Q-Learning Network          |
| GAE | Generalized Advantage Estimate   |
| HPC | High Powered Computer            |
| MDP | Markov Decision Processes        |
| MSE | Mean Squared Error               |
| PG  | Policy Gradient                  |
| PPO | Proximal Policy Optimization     |
| RL  | Reinforcement Learning           |
| SB3 | Stable Baselines3                |
| TD  | Temporal Difference              |



# Chapter 1

## Introduction

Reinforcement learning (RL) is an effective subcategory of machine learning that is used in a wide range of applications. RL algorithms learn by placing an agent in an environment and allowing this agent to take actions and make decisions that affect this environment. The agent is then trained by rewarding desirable actions and punishing undesirable ones. Through this, the agent needs to learn a policy that helps it to achieve high rewards. There are many different types and implementations of RL, and it is often difficult to know which types of RL to choose for different use cases. The project involves the development (from first principles) and investigation of two RL algorithms – deep Q-learning (DQN) and proximal policy optimization (PPO) – to help new developers understand the implementation and choose the best algorithm for different use cases.

The algorithms are compared and evaluated in a test environment to determine the differences and to understand how different implementations of the environment will affect the performance of the models.

Two variants of Tetris – full Tetris and a simplified version, Melax’s Tetris – are developed and converted to OpenAI Gym testing environments. The algorithms are deployed in these environments and tested with different configurations of the environments. The RL agents are evaluated and compared to find the best algorithm for the environment. This evaluation is based on the training time, computational power, and the methods’ efficiency and complexity.

The project finds that both algorithms successfully learn to play Melax’s Tetris. However, the DQN model outperforms the PPO model in this simplified environment and is further developed to attempt to solve full Tetris. This is done successfully with the DQN model outperforming human capabilities in the game and reaching high scores of approximately 280 lines cleared.

### 1.1. Background

RL is found in various applications and industries, including finance, healthcare, and transportation. The field already impacts the world, and as more research and development is done, the capabilities of RL will continue to grow and evolve.

However, to develop an RL algorithm, the developers require a deep understanding of the algorithms. To develop this understanding, the developers must have a way of testing the agent in a safe and controlled environment. A popular and safe way for testing new models is by applying said models to retro games such as Tetris or Pacman. This approach allows the developer to evaluate and improve the performance of the models without the risk of financial or safety issues.

## 1.2. Project Objectives

This project aims to develop a RL agent to learn and play the arcade game Tetris. The objectives and criteria by which this project will be assessed are the following:

1. Develop a working Tetris environment that will serve as the testing base for the RL agents. This environment must be easily accessible and understandable for testing to commence efficiently.
2. Identify and research types of RL methods that will suit the problem, learning and playing Tetris. The methods must incorporate at least two different kinds of RL.
3. Develop and test the selected RL methods with the Tetris environment and compare the results to find the best RL method for playing Tetris.

## 1.3. Motivation

Tetris is an easy arcade game to learn. The rules and objectives of the game are simple and easily understandable for a first-time player. It is thus straightforward to explain to someone how to play Tetris. However, it is difficult to explain to someone how to improve the game quickly. Apart from gaining experience through hours of play, there is no simple way to quickly increase a player's ability.

This is where the RL agent is helpful. The agent will learn to play Tetris through a series of training algorithms and can serve as a model that the developer can use to analyze how the agent improves at Tetris. This can then be applied to teach humans how to improve faster and more efficiently.

Even though it is not the scope of this project, this method of using the RL agent can be applied to real-life situations, such as teaching a child to improve at reading or writing.

# Chapter 2

## Background: Reinforcement learning

This chapter will cover the basic concepts of RL that a new developer should understand before attempting to implement the methods. The different configurations of RL are also introduced and will be used in future chapters. The theory for two important categories of RL (Q-learning and policy gradient methods) are explained and will be the building blocks for implementing the algorithms.

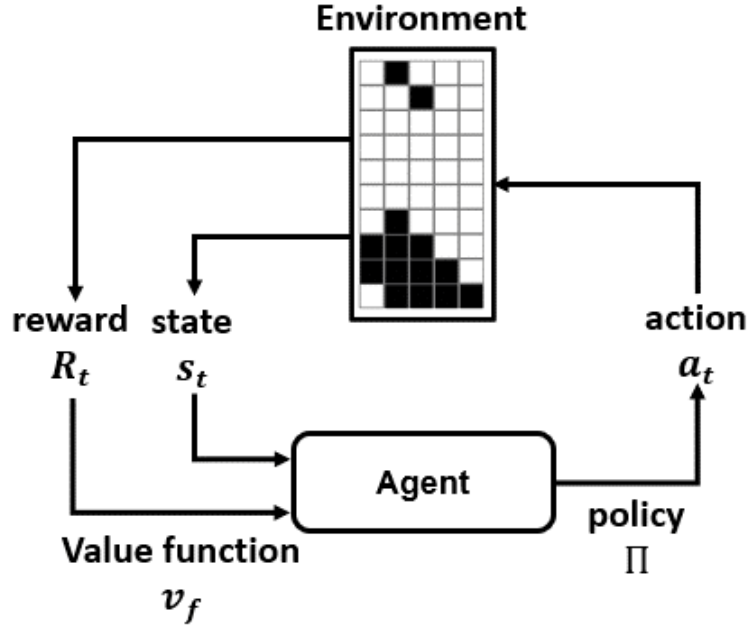
### 2.1. Main reinforcement learning objective

RL works based on a trial-and-error approach and aims to map state observations to actions. The goal is to maximize the reward signal in an environment, thereby learning the optimal control in an incompletely known Markov decision process (MDP).

RL differs from supervised and unsupervised learning as the process generates its own labeled training data through interaction with the environment. A reward signal is used to ‘label’ the data, and the agent uses this to optimize the actions selected for given situations. Thus, the developer has some input on what the agent should learn by changing the reward function, but the agent will only learn from the experience it has generated [1].

RL uses an agent to interact with an environment, in this case, the Tetris game. The agent is a combination of machine learning functions, such as neural network estimators that maximize the rewards received during interaction with the environment. This agent will observe the current state  $S_t$  of the environment and use a policy to determine the action  $A_t$  for this input at time step  $t$ . Once the action is chosen, the current state will transition to the next state – similar to a MDP. The agent will then receive a scalar reward value upon reaching the next state  $S_{t+1}$ . The essence of RL is to determine a policy that will select the actions that will maximize the cumulative reward  $G_t$  for all given states. This is done by updating the estimators used to select the actions by observing the difference between the actual reward received through interaction with the environment and the expected reward that the agent estimated.

The control loop in Figure 2.1 describes the RL implementation, and the structure is an example of a MDP.



**Figure 2.1:** In the implementation loop for RL, the agent will take an observation and use a policy to choose an action that triggers a state change. The agent then receives a reward and estimates the value of the next state with the value function [2].

## 2.2. Markov decision processes

An important property of RL is that it is Markovian by nature. This means that the current state information is only dependent on the information of the previous state, rather than the full history of the states up until the current state,  $\{S_0, S_1, S_2, \dots, S_{t-1}, S_t\}$  [3].

The MDP principle is used to explain the structure of the RL algorithm further and can be seen in Figure 2.2.

### 2.2.1. Structure of a Markov decision process

The Markov chain describes the transitions from a current state to the next. This transition can be described by the probability that the next state  $S_{t+1}$  will be reached given that the current state  $S_t = s$ . This is denoted by the transition probability:

$$P(S_{t+1}|S_t = s) \quad (2.1)$$

Next, a reward indicates how good or bad it is to reach a specific state. Each time a transition from the current state  $s$  to the next state  $s'$  occurs, a reward will be calculated for reaching this state  $s'$ . This reward signal is the backbone of the algorithm and will affect the agent's behavior directly.

The expected return is the expected cumulative reward that is received starting from a given state. This information is important as an agent will want to transition to states

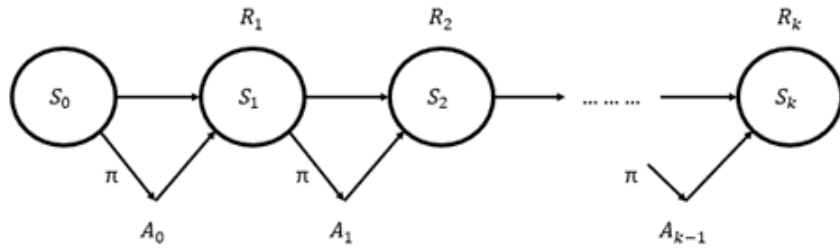
with a higher expected return than states with lower expectancies. The expected return is denoted by equation 2.2.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots \quad (2.2)$$

In most linear cases and especially in cyclic models, it is advantageous to add a discount factor  $\gamma \in [0, 1]$  to discount the effect of the future rewards and alter the equation 2.2 to the equation 2.3. This discount factor  $\gamma$  parameter will adjust the agent's focus to value rewards shortly over the same rewards later in the game.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \quad (2.3)$$

The structure of a MDP can be represented as a flow diagram denoted by Figure 2.2. From this, it is clear how the states  $S_t$  create the actions  $A_t$  under policy  $\pi$ , which in turn triggers state transformations to  $S_{t+1}$ , yielding a reward  $R_{t+1}$  for reaching this state.



**Figure 2.2:** For every state, the RL agent will observe a current state  $S_t$  will use a policy  $\pi$  to select an action  $A_t$  and receive a next state observation  $S_{t+1}$  and reward  $R_{t+1}$  for reaching this state.

### 2.2.2. Value function and Bellman equation

The value function is a metric that quantifies the value – how good or bad – of reaching a specific state. The value function estimates the expected return  $G_t$  for the given current state  $S_t = s$ . This estimation is often not the same as the actual return, and adjustments must be made to the value function estimator to improve the model's accuracy. The agent will use this estimation to transition to states with higher value functions to find the state with the optimal value. The value function is described by equation 2.4:

$$v(s) = E[G_t | S_t = s] \text{ for } \gamma \in (0, 1] \quad (2.4)$$

$$v(s) = E[R_{t+1} | S_t = s] \text{ for } \gamma = 0 \quad (2.5)$$

The value function is an important component of the Bellman equation, which provides a way to express the value of a state in the MDP in terms of the expected sum of rewards

that can be obtained from that state onward. This works because the information of the current state only relies on the information of the previous state. This is used to define the value of a state or, more importantly, the action-state pair, which will be used to choose actions later in the MDP. The Bellman equation is denoted by equation 2.6.

$$v(s) = E[R_{t+1} + \gamma V(S_{t+1}) | S_t = s] \quad (2.6)$$

The goal is to solve the Bellman equation for all the states, allowing the agent to follow the states-values in increasing order to solve for the environment and maximize the expected return for the actions.

### 2.2.3. Actions and policies

The actions are an important part of the MDP as this will determine the state transitions and affect the environment in a certain way. The action-state pairs can be estimated by the Bellman optimality equation and will result in the quantification of the expected reward  $R_{t+1}$  for taking a certain action  $a$  given the state  $s$ . This equation

$$q_*(s, a) = q(s, a) + \alpha(R_{t+1} + \gamma \max_{a'} q(s', a')) \quad (2.7)$$

estimates the value of being in the current state-action pair  $q_*(s, a)$  by incrementing the value with the discounted ( $\gamma \in [0, 1]$ ) information of the next state-action pair  $q(s', a')$ . The reward  $R_{t+1}$  will determine whether the value of the state-action pair increases or decreases and the learning rate  $\alpha$  determines how much the reward will change the current state-action pair value.

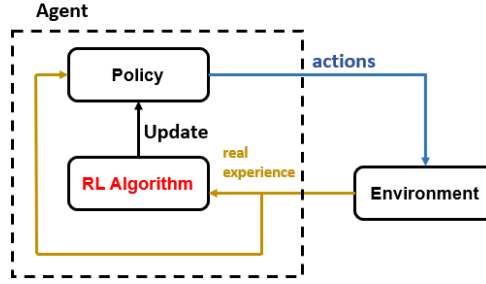
The actions are chosen based on the states that the agent observes in the environment through the concept of a policy. The policy is a mapping from states to probabilities of selecting each of the available actions. This can be through pure probabilities (as seen in policy gradient methods) or action-state value estimations (as seen in Q-learning methods).

The RL methods will specify how the agent's policy changes due to its experience. This will change the actions that an agent takes for the same observations and improve the agent's performance in the environment.

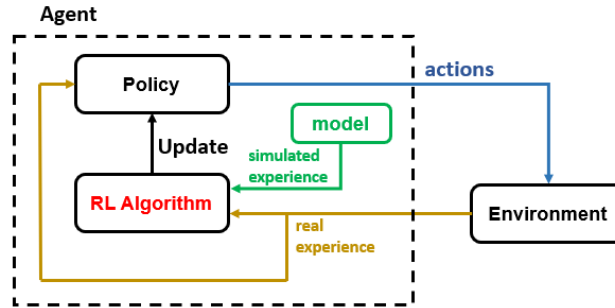
The MDP described the basic structure of the RL algorithms, but some different configurations and setups influence the performance of the models.

## 2.3. Reinforcement learning setups and configurations

There are different setups and important concepts of RL that affect key features of the model performances, such as training time, stability, and the accuracy of the trained model. These setups are different for different types of RL and have varying behaviors



(a) Model-free RL does not have a representation of the environment within the agent.



(b) Model-based RL requires a custom model of the environment and allows the agent to be more sample-efficient.

**Figure 2.3:** The structure of model-free (a) and model-based (b) RL and the differences between the two implementations. [3]

because of this. It is important to understand these setups before introducing the theory for Q-learning and policy gradient methods.

### 2.3.1. Model-free and model-based reinforcement learning

Model-free RL is a technique where the agent has no prior knowledge of the environment and can only learn from the experience gained by interacting with the environment. In this case, the model of the environment is external to the agent. Model-based RL contains a model of the environment in the agent and can be used to simulate experience without interacting with the environment. This allows the agent to plan and make smarter decisions as it is more sample-efficient than the model-free method. Figure 2.3 shows the structure of both cases and describes the control loop in more detail.

### 2.3.2. Exploration vs. exploitation

An important concept to understand is the idea of exploration vs. exploitation. During exploitation, the agent will use the information that it has gained through interaction with the environment and will select the action that is expected to yield the maximum reward for the given state.

At the start of the training, the agent will not know the environment. Thus, explorative actions must be taken to map the action-state values to the environment observations. The policy will be in an exploration mode, allowing the agent to take random actions within the environment to find the optimal actions and build experience for the RL algorithm to learn.

The  $\epsilon$ -greedy policy chooses the action by sampling a random value  $\text{Sample} \in [0, 1]$  and comparing this to the threshold value. This  $\epsilon$ -threshold value can be annealed throughout the training of the model to switch between exploration and exploitation. This can be seen in the algorithm 2.1 [4].

---

**Algorithm 2.1:**  $\epsilon$ -greedy policy

---

```

 $\epsilon_{\text{threshold}} \leftarrow \epsilon_{\text{stop}}(\epsilon_{\text{start}} - \epsilon_{\text{stop}}) \times e^{\frac{-\text{steps done}}{\text{decay rate}}}$ 
Sample  $\leftarrow$  Random in range  $[0, 1]$ 
if Sample  $> \epsilon_{\text{threshold}}$  then
    Return random action
else
    Return action with highest expected return
end if

```

---

### 2.3.3. Online vs. offline learning

The basis of RL is that the agent generates the experience it learns from. The performance of the different algorithms relies on the recency of the generated experience. This gives rise to the idea of online and offline methods:

1. Online methods can only train on the experience gained from the previous policy and are often used in policy gradient algorithms. This method prevents large changes to the current policy and increases stability. This also yields faster convergence and better online performance. However, the increase in stability also results in an increased chance of becoming trapped in local minima and is less likely to find the optimal policy.
2. Offline methods train on the experience gained by all the versions of the policy and are popular in Q-learning. The use of older experience means that larger changes are continuously made to the model and allows for greater exploration of the environment. The result is that the method is less likely to get stuck in local minima and more likely to find the optimal policy.

Offline learning is easier to implement and more generalized than its online counterpart. Policy crashes can be avoided by engineering the reward function and the hyperparameters of the environment and the model.



### 2.3.4. On-policy vs. off-policy setup

In RL, two different policy setups can be used during the training and playing process. These two setups are referred to as the on-policy and off-policy setups and differ based on the data that the agent uses and how the agent follows its policy.

On-policy methods are set up to allow the agent to select the actions directly. This is done by either outputting a probability distribution for the available actions or one hot encoding of the actions to be picked directly from the policy. In this case, the agent always follows its policy and uses methods such as PPO and advantageous actor-critic (A2C).

Off-policy methods work differently as the agent, in this case, can't select the actions. Instead, the agent estimates the value of each action, which will then be used in the policy to choose the action for the given state. This can be seen in the case of the  $\epsilon$ -greedy policy, where the agent has no control over the action that will be chosen. Off-policy methods include the Q-learning algorithms and will be used extensively during the project [5].

## 2.4. Types of reinforcement learning algorithms

Different methods of RL incorporate different combinations of the basic concepts and configurations that were described in the previous section. The RL models have different properties and advantages that make them suitable for applications and use cases. There are many different types of RL, but this report will focus only on methods in the Q-learning and policy gradient methods categories.

### 2.4.1. Q-learning

Q-learning is a model-free RL algorithm that attempts to learn the optimal policy in an MDP. It does this by incrementally updating the approximator function  $Q(s, a)$  until it has reached the optimal approximator function  $Q^*(s, a)$  for the action-state pairs in a given environment. The agent will interact with the environment and generate experience which is used to optimize the current value approximator function based on the Bellman optimality equation (equation 2.7) by using the current state  $S_t$ , current action  $A_t$ , next state  $S_{t+1}$  and reward for next state  $R_{t+1}$  in equation 2.8 [1].

$$Q_k(S_t, A_t) \leftarrow Q_k(S_t, A_t) + \alpha_k(R_{t+1} + \lambda Q_{k+1} - Q_k(S_t, A_t)) \quad (2.8)$$

The approximator function can be implemented in a tabular fashion (tabular Q-learning) or can be represented by a neural network structure (deep Q-learning) which caters to large observation and action spaces. In tabular and deep Q-learning, the state-action value function aims to approximate the expected cumulative return  $G_t$  for each available action given the current state  $S_t$ . From the expected cumulative return, the policy will select an

action that will yield the maximum expected reward.

The policy will also incorporate exploration and exploitation by using the  $\epsilon$ -greedy policy to guide the agent towards convergence. In tabular Q-learning, convergence can be guaranteed if the agent does more exploitation than exploration.

Q-learning is an off-policy method, meaning that experience from all policy versions can be used. Thus, the model can learn from human interaction, and rule-based methods can be used to train the model and guide the agent in the right direction to maximize the cumulative rewards in the environment [6].

### 2.4.2. Policy gradient methods

Policy gradient (PG) methods have the same goal as Q-learning in that the policy intends to choose the action that would yield the most rewards for the environment. However, PG methods differ from Q-learning in that the action estimator outputs a probability distribution of the set of available actions rather than the expected reward for each of these actions. The methods use an actor critic structure for this task, and the estimators for the rest of the PG methods are chosen to be neural networks. The actor network takes the current state as input and outputs the action selection probabilities. The weights of the network are the policy parameters. In PG methods, the action is selected by categorical sampling from this probability distribution output given by the actor network [7].

Policy-based methods also differ as this is an on-policy algorithm. This means that each update only uses data collected while acting according to the most recent version of the policy. Therefore the only experience that the current model  $\pi_\theta$  will be able to train on is the experience gained from the previous model  $\pi_{\theta_{old}}$  [8]. This will be the basis on which PPO is developed as seen in Section 4.3.6.

## 2.5. Chapter summary

The MDP describes how the agent will map the state observation to an action and how the action will be evaluated through a reward system. This will be the basis for developing the PPO and DQN model in chapter 4. The Bellman equation is also important since this will be the main feature of the DQN model.

The concepts and configurations such as online and offline learning, model-based and model-free RL, and on-policy and off-policy methods should also be considered and referred to during the development and implementation of the DQN and PPO models.

The basic Q-learning and PG theory will be the building blocks on which the DQN and PPO models are designed and will be referred to in upcoming chapters. The developed DQN and PPO models will be tested and validated in a Tetris environment.

# Chapter 3

## Development of the Tetris environment

In this chapter, the development of the RL testing environment, Tetris, will be discussed. The structure of a custom OpenAI Gym environment is introduced, together with the functions within this environment.

A basic overview and history of Tetris are given. It is important to understand the workings and the rules of the Tetris game, as this greatly affects the design of the RL models. The Tetris game was designed from an example by Bakibayev [9]. However, the OpenAI Gym class was created from scratch and all the configurations made were the student's own design decisions.

The different configurations of the environment are also explained to give new developers an idea of how the RL agents interact with the environment and which configurations work the best.

### 3.1. OpenAI Gym environment

To be able to design and test RL algorithms, a testing environment is required. Many open-source environments are available to test RL algorithms, and these are designed by OpenAI [10]. The environments are called Gym environments and follow a specific structure for the RL methods to interact with and learn. Example structures include the Cartpole or Gridworld environments and can be viewed on the OpenAI Gym website [here](#).

The structure of an OpenAI Gym environment has to comply with the following key factors:

#### **Reset function**

This will reset the environment to the start position and clear all the in-game data for a new game to start. This function is called when the environment reaches a terminal state.

#### **Step function**

This function will take an action that was given by the agent. This action will change the environment and return four pieces of information to the agent.

The *observation* of the current game state  $S_t$  will be formatted and returned by the step function for the agent to use. A *reward* is calculated based on the action taken and how the environment was changed. This reward function will reside in the step function and must be changed locally during this project. In the case of the environment reaching a terminal state, the step function will set and return a *terminal flag*. An *info* list is also returned and can contain any extra information, such as game scores or game lengths.

### Render function

This function is purely used for the human developer to see exactly what is happening between the agent and the environment. This function will visualize the environment and the relevant information that the developer would need. In this project, the render function is done using the PyGame libraries and was used to track the performance of the models visually.

### Control loop

The basic control loop of an RL model interacting with a Gym environment follows the same structure as given in algorithm 3.2. This OpenAI Gym structure is the building block on which the Tetris game will be developed.

---

**Algorithm 3.2:** The custom RL testing environment must follow the Gym structure.

---

```

Define environment
for  $e$  in episodes do
    Reset environment and gain first observation in episode
    while not terminal do
        Step within environment and receive next observation,
        reward, terminal flag and info
        Render the environment if required
    end while
end for

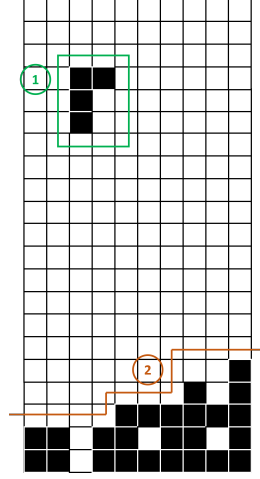
```

---

## 3.2. Tetris game environment

### Background

In this project, the test environment of choice will be the Tetris game. Initially developed in Russia by Alexey Pajitnov, this arcade game has grown popular in the 1980s and has since become a household name, selling millions of copies worldwide. The game works by dropping different shapes (or tetrominoes, but for the rest of this report they will be referred to as shapes) in a grid environment to fill an entire row, which results in the line being cleared and increasing the player's score. The game will continue until the placed



**Figure 3.1:** The falling shape (1) is controlled by the player and can move left, right, down, or rotate. The placed shapes (2) form the terrain of the field.

blocks exceed the playing field height limit, ending the game and giving the player the final score.

## Development

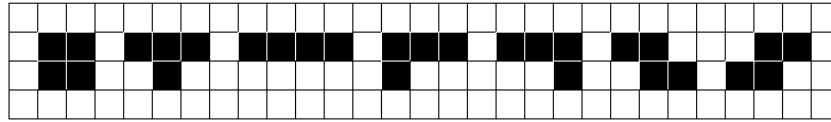
For the Tetris game to be helpful as an RL testing sandbox, the game must first be implemented as an OpenAI gym environment. This means that the game must be built in a Python environment and have a step, reset, and render function. The game functions are divided into 3 different classes combined to form the final game of Tetris.

The figure class represents the different shapes and the important information such as the  $x, y$  - positions and rotation of the shape. Next, the interaction class will include the downward and side-wise motion, the rotation, and the stacking of the shapes. This class will host the game rules and stop the player from making invalid moves. The control class will render the game environment and allow the player to choose and interact with the game. The OpenAI Gym class is used to communicate RL models. This class will follow the structure as in Section 3.1.

## Game representation

The game field is represented by a  $20 \times 10$  matrix of 1's and 0's, representing filled blocks and open spaces, respectively. The current falling shape can be represented by 2's in the matrix to give the player an indication of the current playing shape. The shape falls constantly, and a player can make multiple moves in a single downward step. Figures 3.1 and 3.2 describe the seven different shapes that exist in the Tetris game and will be given as input to the game in random orders and rotations. This makes it challenging to plan since the model does not know what the next shape will be.

A human player would observe the game as the  $20 \times 10$  matrix and make decisions



**Figure 3.2:** These are the seven different shapes available in the full  $20 \times 10$  Tetris environment.

based on this information. However, in some cases, it is beneficial for the RL model to convert the observation space to a flattened version of the matrix, which is used as the input layer to the neural network.

### Game goal and rules

The goal of Tetris is to clear as many lines as possible by moving the shapes to a desired location and dropping the shape into place, thereby increasing the player's score. Thus, a reward shall be given to the agent for actions that lead to a row of blocks being cleared. This is the main reward and can be used in the sparse reward setup. The reward function can guide the agent toward clearing lines by incorporating features such as the standard deviation of the game field, column heights, and holes created by the different actions.

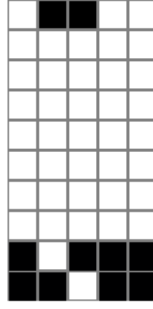
## 3.3. Applying reinforcement learning to Melax's Tetris

Melax's Tetris, like any Atari game, is an MDP. This means the game must have specific states that change based on an action signal and a reward signal to validate the selected action. The following subsections will define how the state observations, action space, and reward function are developed regarding Melax's Tetris. There are different setups for each category, and these will be compared to find the optimal configuration for RL testing. These topics are essential as this is how the RL agent will perceive, control, and learn the Tetris game.

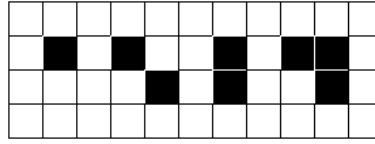
### Melax's Tetris setup for reinforcement learning development

The  $20 \times 10$  Tetris environment poses a problem for RL as the vector or matrix size of the game field is large, consisting of many different states, and struggles to learn due to sparse rewards. Therefore, for the development of the RL model, the environment is simplified into a  $10 \times 5$  matrix to test the different models and compare the results before moving to the  $20 \times 10$  Tetris. This smaller version of Tetris, as seen in Figure 3.3 can be referred to as Melax's Tetris and was developed for prototyping RL algorithms in the Tetris environment.

In this environment, the shapes are simpler and the drop area smaller, resulting in better performance in sparse reward environments. There are four shapes, as seen in



**Figure 3.3:** Melax's Tetris works the same as normal Tetris; the only differences are the simplified shapes and smaller playing field.



**Figure 3.4:** These are the four different shapes available in Melax's Tetris. Later in the project, a fifth shape can be added to challenge the RL model and test scalability.

Figure 3.4, that can be played in this environment, which represent the simplified versions of the  $20 \times 10$  Tetris pieces.

### 3.3.1. Observation space

The Melax's Tetris environment must be modified to serve as an input to the RL algorithms and can be configured to represent the game field or a combination of useful information. The game field is represented by a two-dimensional array of size  $10 \times 5$ , which is the height and width of the columns. The open spaces and the blocks are represented as 0's and 1's.

#### Original observation space

For the original setup, the only preprocessing done on the matrix is flattening the  $10 \times 5$  shape to a 50-element array and converting this to a tensor before it is given as an input to the RL algorithm. Given enough training time, this setup could allow the agent to generate a unique solution to the environment since this is the raw observation space.

#### Simplified observation space

In the simplified observation space, the size of the observation array is reduced by only giving the agent the height of each column in the playing field and the type and rotation of the current shape. This reduces the complexity of the observation space but also reduces the resolution of the observation that the agent will perceive. The configuration of the observation space is only used for full Tetris since the observation space for Melax's Tetris is small enough for the custom DQN and PPO models to successfully learn the environment.

### 3.3.2. Action space setup

The action space is the method that the RL agent uses to control the environment once a state observation has been perceived. Two different setups for the action space (the arcade and direct action space) will be considered.

#### Arcade action space

The first type of action space consists of a single discrete number between 0 and 3. This will represent the available actions in the environment and allow the agent to choose between rotating the shape, moving left or right, or dropping the shape into position.

This is similar to how a human would interact with the environment when playing on an arcade console or computer. However, a human only uses these four actions to move the shape to the desired location, which was already decided before taking one of the 4 actions. This is to say, the human player decides the shape's location rather than the series of actions needed to reach this location. This way of thinking can be engineered and called the direct action space.

#### Direct action space

The action space is changed to be an integer in the range of  $[0, 19]$  for Melax's Tetris. A single shape can be placed in 5 different  $x$ -positions and 4 different rotations, resulting in 20 different positions on the playing field (depending on the shape type). Thus, the model output will be converted from the integer value to the  $x$ -position and shape rotation.

The position and rotation of the shape are calculated using the game field width ( $\mathbb{W}$ ) and the number of rotations available for the current piece ( $\mathbb{R}$ ) to evaluate the current action ( $A_t \in [0, 19]$ ). Equation 3.1 describes this process:

$$\text{Shape position} = A_t \% \mathbb{W} \quad (3.1)$$

$$\text{Shape rotation} = \min(\mathbb{R}, \text{floor}(A_t / \mathbb{W})) \quad (3.2)$$

This change significantly reduces the complexity of the problem since the agent can focus on placing the shapes in an optimal position rather than having to manually move the shapes to the same position using the arcade controls as a human would. The effect of this simplification is that the model can learn significant moves quicker and train faster with better accuracy and efficiency.



### 3.3.3. Reward function

The reward system for Melax's Tetris must be engineered to allow the model to learn the environment effectively. Two reward setups were tested: a sparse reward function and a shaped reward function.

#### Sparse reward function

In the sparse reward scenario, the only rewards that the agent will receive +1 for a line cleared and  $-1$  for a game over. The agent must fully explore the environment before finding the rewards and building the network to guide the model toward the line clears. This allows the agent to generate a unique solution to the environment as there are no human-engineered qualities to the function, which could limit the agent's performance.

#### Shaped reward function

In the shaped rewards scenario, the standard deviation of the game field and the number of holes created were used as penalties to guide the agent toward the line clears and, thereby, a faster training time. Equation 3.3 shows the best weights for the reward function.

$$\begin{aligned} \text{Reward Function} = & 1 \times \text{Lines cleared} \\ & + 0.1 \times (\text{Previous Standard Deviation} - \text{Current Standard Deviation}) \\ & - 0.05 \times (\text{Holes Created}) \quad (3.3) \end{aligned}$$

The standard deviation of the column heights is calculated each time a piece is placed in the environment. Similarly, the number of holes created by the placement of a piece is calculated, and both values are used as input to the reward function.

## 3.4. Chapter summary

The OpenAI Gym structure is important when developing a custom RL testing environment as this allows the agent to observe, control, and learn the environment. This structure is followed to design Tetris and a simplified version (Melax's Tetris). Melax's Tetris will be used to prototype and develop the two chosen RL methods (DQN and PPO) and find the optimal configuration of the environment.

The next chapter finds that the simplified observation space, direct action space, and shaped reward function are the best configurations of the environment. It also finds DQN to outperform PPO in Melax's Tetris. The DQN model is further developed and successful in solving the full Tetris environment with the optimal environment configuration.

# Chapter 4

## Reinforcement learning methods development and testing in Melax's Tetris

In this chapter, the specific theory and development of the DQN and PPO models are discussed. This includes the algorithmic implementation and the calculations that must be understood before trying to develop the algorithms from scratch. The developed RL algorithms will be applied to Melax's Tetris to test the performance of the models and the ability to learn within the environment. The model that performs the best in Melax's Tetris (which is found to be DQN) will undergo further development and be applied to full Tetris.

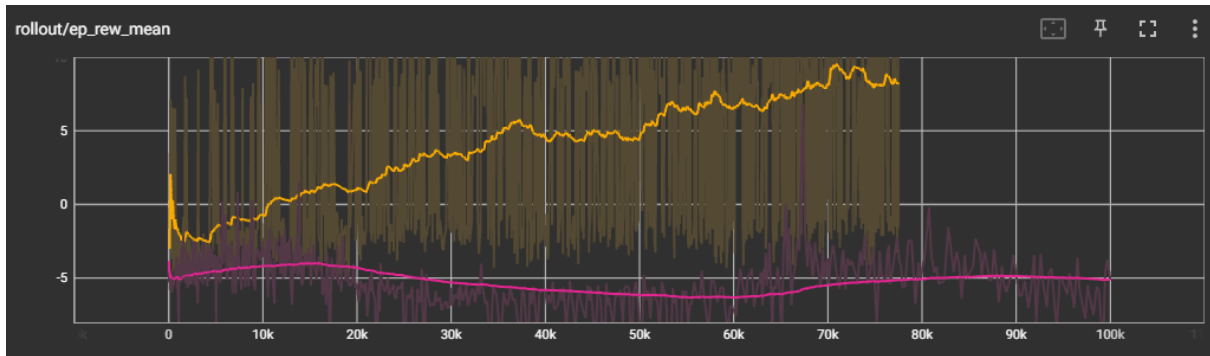
Before the development can begin, the developer needs a means of tracking the performance of the model. This is done through progress visualization within TensorBoard.

### 4.1. Progress visualization with TensorBoard

The verification for the training of the models is done by logging the agent's gameplay statistics to TensorBoard, where the progress of the models can be visualized and tracked to ensure that desirable performance is achieved.

The information that will be plotted has to reflect the direction and the performance of the model. This will include the total episodic reward, the length of a game any extra information such as the  $\epsilon$ -threshold for DQN or the value losses for PPO. The total reward will indicate whether the model is clearing lines and the frequency at which it does this. The game length indicates how long the agent survives in the environment and the effectiveness of the placements and action usage. For DQN the  $\epsilon$ -threshold value indicates what percentage of steps the agent will take random actions to explore the environment at a certain timestep. For PPO the value losses will indicate the size of the model changes that are made.

An example of a TensorBoard plot and a good trend can be seen in Figure 4.1. The yellow line yields higher episodic returns and a positive trend compared to the pink line,



**Figure 4.1:** The yellow line is more successful at learning the environment since the episodic reward is higher than the pink line.

which stays at a relatively constant episodic return. This shows us that the yellow model should continue training, whereas the pink model should be stopped and re-evaluated. This TensorBoard data can be extracted and saved as a comma-separated values (CSV) file, then used to plot custom graphs and representations of the training trajectory. These graphs will be used for the results and the validation explanations within the report.

To validate the RL models, it was also tested against standard RL libraries, such as the Stable Baselines3 (SB3) versions of the algorithms. This library contains various RL algorithms that can be used to test custom environments and learn about the algorithms. The SB3 models are generalized models that can work with many environments and must support various input types, from large input spaces to large action spaces and anything in between. This means that the SB3 algorithms will guarantee convergence; however, it might not yield the best results for the problem. This is because the algorithms are not as efficient as focused algorithms and can result in longer training time. The SB3 libraries are used as a starting point for most RL problems, and the algorithms are created from this starting point.

The custom DQN and PPO models are tested against the SB3 library to compare the performance and the efficiency of the custom model, explore possible improvements, and learn from the library's algorithms. The results of this comparison will be explained in the coming sections and the progress visualization will be used throughout the project to validate the DQN and PPO models.

## 4.2. Deep Q-learning

A custom DQN model is written and implemented with the help of an example notebook by Prof. Engelbrecht. This notebook was originally designed to solve the Gridworld problem but is converted and adapted to the Tetris environment. This is done by changing the input space and the action spaces to match the Tetris format. The PyTorch network and the optimization of the network were used as the original versions since they complied

with the theoretical and mathematical structure set out by the Mnih paper [5]. The training loop is designed based on the before-mentioned paper's structure, and the model validation is custom-designed.

This DQN model is first tested on the Cartpole and the Gridworld environment to test the functionality and tune the model's hyperparameters. Once the algorithm is confirmed, it is applied to Melax's Tetris environment. The algorithm works by finding the value function for each of the different environment states to determine the best action to take with the goal of maximizing the rewards and feedback received. To fully understand how this algorithm works, the three main components must be investigated. This includes the two value function estimators, the replay buffer, and the  $\epsilon$ -greedy policy.

### 4.2.1. Value function estimators

The backbone of a DQN lies in the estimators used to choose the best possible actions for the given input state. The estimators are chosen to be neural networks as this caters to large input and action spaces, as seen in the Tetris environment. Two networks are used together to accomplish this task; these will be called the policy network and the target network.

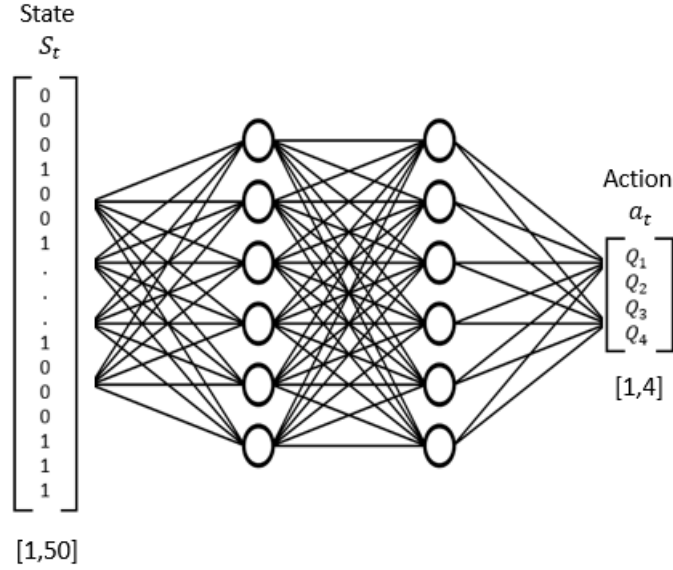
The policy network is used to estimate the expected cumulative return for each of the available actions for the given state. These action values can then be compared to select the action expected to yield the most reward. The input for this network will be the original observation space (refer to Section 3.3.1) for the Tetris game field, relating to a binary array of size  $[1, 50]$  with minimum value 0 and maximum value 1. This will then be fed to the hidden layers and result in an output layer with a size corresponding to the number of available actions, in the case of Tetris  $[1, 4]$ . The network structure can be seen in Figure 4.2.

The target network is used to validate the policy network. It estimates the value of reaching the next state based on the action taken. This network has the same structure as the policy network; however, the output layer of the target network is a single value, which can be referred to as the next-state value. This value is a key component in determining the temporal difference (TD) targets for the updates of the networks.

### 4.2.2. Replay buffer

The replay buffer is a memory filled with the experience gained through gameplay and interaction with the environment. The memory size is controlled as a hyperparameter and works on the first in, first out principle, meaning that once the memory is filled, the newer data will be pushed to the memory, and the oldest data will be popped to create space.

The experience stored contains information that will be used in the training of the value function estimators. This includes the current state, the action taken, the next state,



**Figure 4.2:** This is a representation of the policy network structure that will be used in the DQN model. The network takes the flattened game field matrix as input and outputs the Q-values for each of the available actions. Note that the hidden layer sizes are not to scale [11].

and the reward received. This information is pushed to the memory for each step in the environment and will store a default of  $50k$  entries for the estimator training.

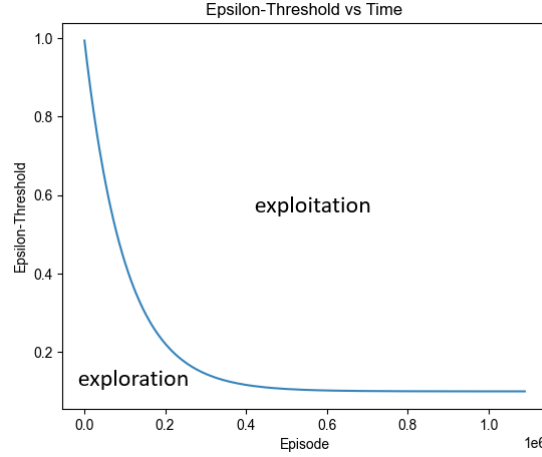
This experience is used to update the policy network during the model's training by randomly sampling from this experience. The sample size is set as a hyperparameter to adjust the amount of data the networks use to update.

Using a replay buffer with experience from all time frames of the gameplay means that the DQN model is an offline learning algorithm. This means that the model is less complex to develop and less computationally expensive than its online counterparts.

This also allows the agent to train on external experience and can be trained through imitation learning. However, this also means that the agent is not immune to policy crashes and could be affected by the large sample space.

### 4.2.3. $\epsilon$ -greedy policy

The DQN model will always choose the action that produces the maximum reward by exploiting the information it already knows about the environment. This means the algorithm will not efficiently explore different actions in the environment and tends always to take the same actions if the actions are not penalized. Thus, even though the agent will take actions that produce some reward, it will not be able to improve or find better actions and can get trapped in local minima. The goal of the  $\epsilon$ -greedy policy is to allow the agent to explore the environment by taking random actions for a percentage (usually 10%) of the time that the agent will interact with the environment.



**Figure 4.3:** The agent will explore the environment by taking random actions for a random value less than the annealing threshold curve. Otherwise, the agent will take the action with the highest Q-value [4].

This will force the agent to explore different actions for the same states and possibly find actions that maximize the reward. This can be seen as the exploration vs. exploitation terms of the model. The  $\epsilon$ -greedy policy strikes a fine balance by defining a parameter epsilon ( $\epsilon \ll 1$ ), which then plays a role in the action selection. The epsilon ( $\epsilon$ ) parameter can be a constant value, or it can be annealed from a high value to a lower value to give the agent more room to learn from. The effect of the  $\epsilon$ -greedy policy and the annealing on the choice between exploration and exploitation can be seen in Figure 4.3.

#### 4.2.4. Training loop

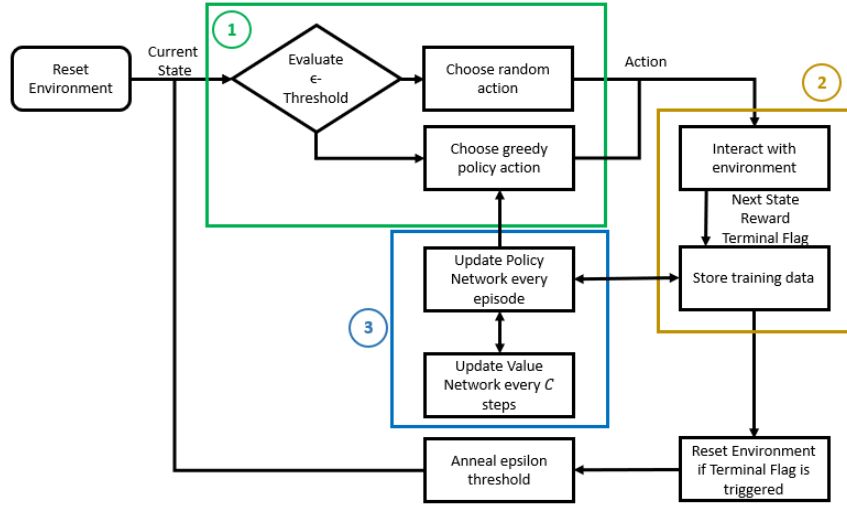
The training loop is the control structure and the implementation of the RL algorithm. The custom DQN model is developed to follow the guidelines and structure of the DQN Paper algorithm. The flow diagram in Figure 4.4 describes the integration of the three main components of the DQN model and the interaction with the environment, which is Tetris in the context of this project.

The policy network is used in step 1 to select an action using the greedy policy. For the agent to learn, this network is then updated and optimized in step 3.

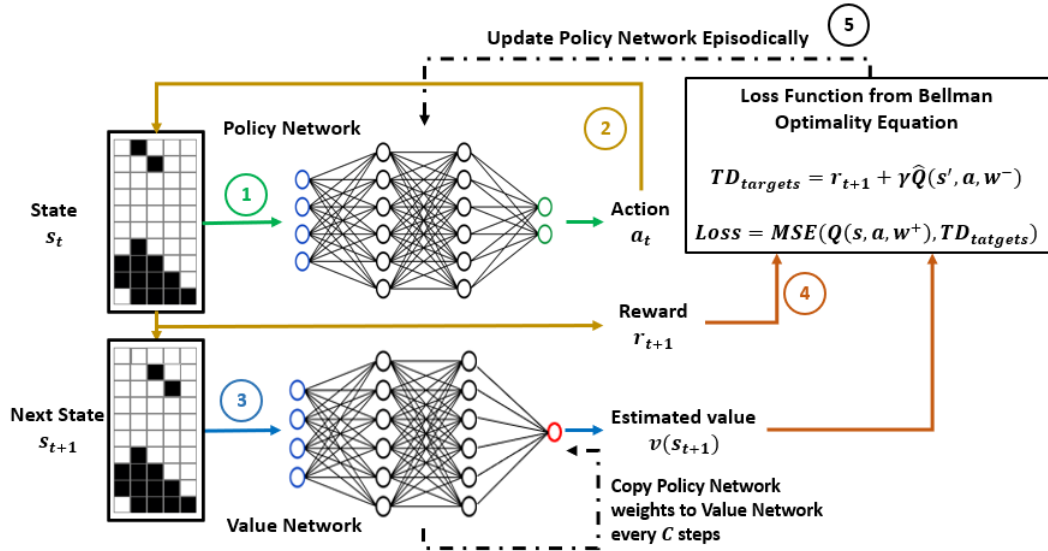
#### 4.2.5. Policy and target network optimization

The networks are updated using the loss generated between the policy network and the target network. This loss is generated by using a sampled batch from the experience in the replay buffer and the current versions of the two networks.

Figure 4.5 shows a simplified representation of the optimization function. Steps 1 and 2 are the training loop steps where the agent will interact with the environment to generate experience and explore the environment. The important values used in these steps are the



**Figure 4.4:** This is the training loop for the DQN algorithm. This includes the  $\epsilon$ -greedy policy for selecting and action (1), the storing and sampling of the replay memory (2), and the updating of the policy and target network at different intervals (3).



**Figure 4.5:** During the optimization of the policy network, both of the networks will be used to evaluate the expected values of the selected actions against the actual values (TD-targets). The networks are then updated according to the loss between the actual and expected values.

current action-state values, which are the expected returns for the actions selected for the current states at the time of the selection process. The rewards received for reaching the next state are also saved in the memory and used during optimization.

Step 3 occurs within the optimization function. This step involves the target networks and determines the expected return for the next state  $S_{t+1}$ . This is important and explained in the Q-learning Section 2.4.1 to how this will work.

Step 4 involves determining the loss of the policy network – in other words, how much the estimator differs from the real value. This is done by calculating the TD-targets, which are constructed using the actual rewards received from the actions taken and the discounted value approximation of the next state using the target network from step 3. This is done by exploiting Bellman’s optimality equation which is described in Section 2.2.2.

$$\text{TD}_{\text{targets}} = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \quad (4.1)$$

The loss is determined by Mean Squared Error (MSE) between the action-state values determined in step 1 by the policy network and the TD-Targets by the Rewards and the target network determined in the current step 4.

Step 5 refers to the update of the policy network weights after each optimization Loop. This is done using the loss calculated in step 4 and will use the Adam optimizer with a learning rate of  $\alpha$ .

It is important to note that only the policy network gets updated episodically, whereas the target network will only be updated every  $C$  steps. This feature improves the algorithm’s stability since the alternative leads to oscillations and divergence of the policy. This occurs when the update to the policy network also updates the target network, shifting the TD-targets to different locations with every update and making it difficult to converge to the desired solution.

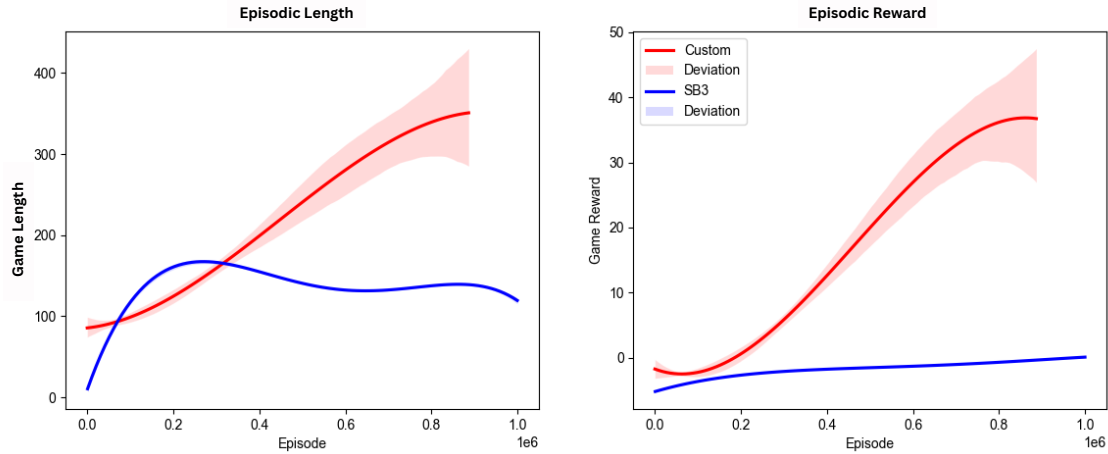
#### 4.2.6. Tetris configurations and results

For the model to interact with the environment and find a solution to the Tetris environment, there are specific changes that need to be made to both the environment and the DQN model.

The DQN algorithm is tested with various hyperparameters and setups to find the optimal strategy to apply the RL methods to Melax’s Tetris environment. These comparisons are made with the goal of selecting the best setup for the DQN model that will be applied to the Full  $20 \times 10$  Tetris environment.

When comparing two simulations of the DQN method, the tests must use the same version of both the algorithm and the environment to ensure that the tests are valid and any bias is eliminated. The comparison is made by training the models for 1 million timesteps





**Figure 4.6:** The custom model performed better than the SB3 model as it yielded higher episodic returns and lengths.

with the same hyperparameters and network sizes. Once the learning is completed, the plots are extracted and compared based on the episodic reward and length of the gameplay.

### Stable Baselines3 compared to the custom model

The SB3 model and the custom model are given the same version of Melax's Tetris game and the same hyperparameters to use during the training of the agents. Both agents are trained for the 1 million steps, and the results can be seen in Figure 4.6.

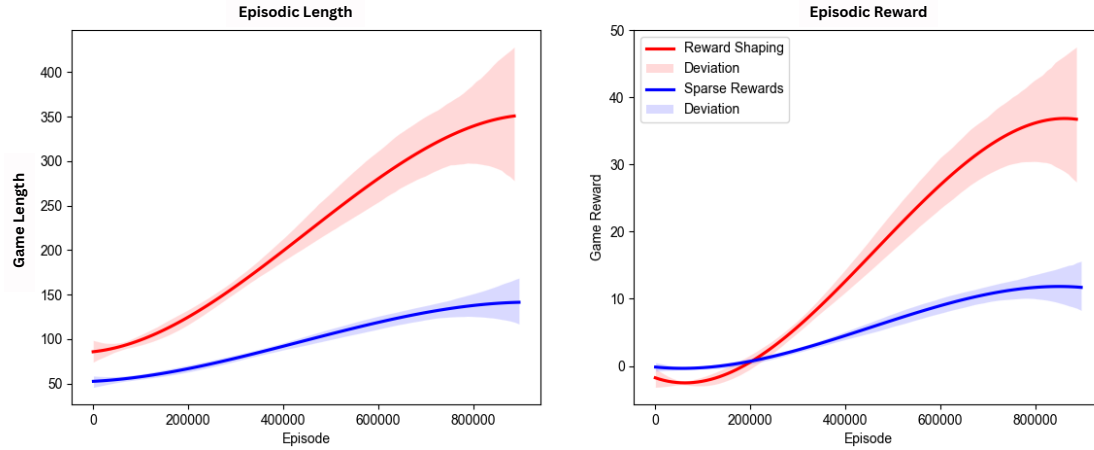
The custom model performed better than the generalized SB3 model since it has a network structure that caters to Melax's Tetris game. Although both models do improve at the game, the custom model proves to be sample efficient and trains quicker to the solution of the environment.

### Sparse reward function compared to reward shaping function

The first test is to compare the effect that reward shaping will have on the performance of the DQN model compared to the same model that is trained on sparse rewards. The reward system is described in Section 3.3.3 and will be the criterion by which this test is run.

The comparison can be seen in Figure 4.7 and clearly shows that reward shaping significantly improves the performance of the model. The model will converge quicker than the sparse reward counterpart since it has a better representation of the environment and is guided toward the line-cleared reward, rather than having to find the line-cleared reward by chance through exploration.

Given the fact that the reward shaping model did perform better, it is important to note that the process of engineering the reward function takes time and a deep understanding of the environment. Knowing this, the sparse reward model will be better for the initial



**Figure 4.7:** Both of the reward functions allow the agent to learn the environment. The shaped reward function gives the agent more information to learn from and thus it converges to the solution quicker.

test of an environment since it can adapt better and significantly helps with designing the reward function.

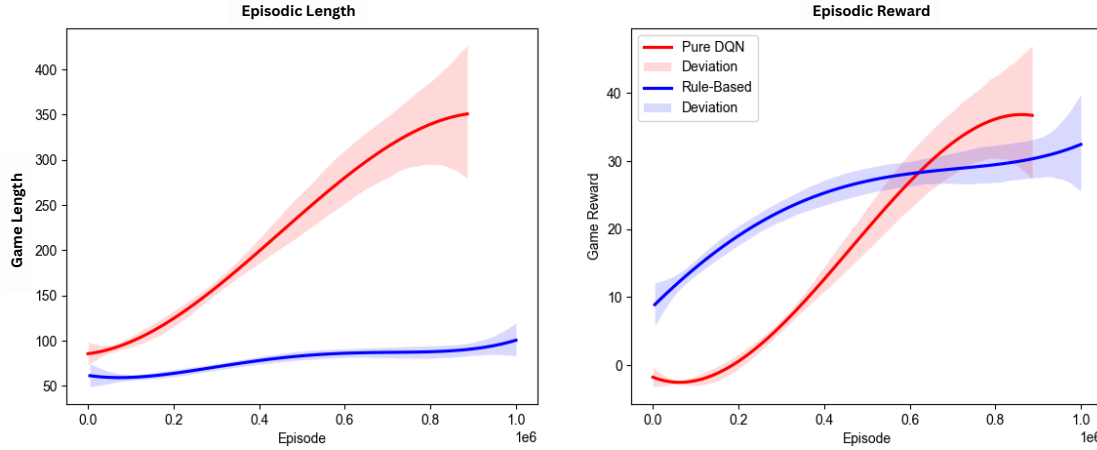
Another point is that if these models were trained for extensive periods of time, the sparse reward model might find better ways to obtain the sparse rewards. This could outperform the reward shaping function since the human qualities in the function could limit the performance of the model. However, this is out of the scope of this project and will not be investigated.

### Imitation learning through rule-based algorithm

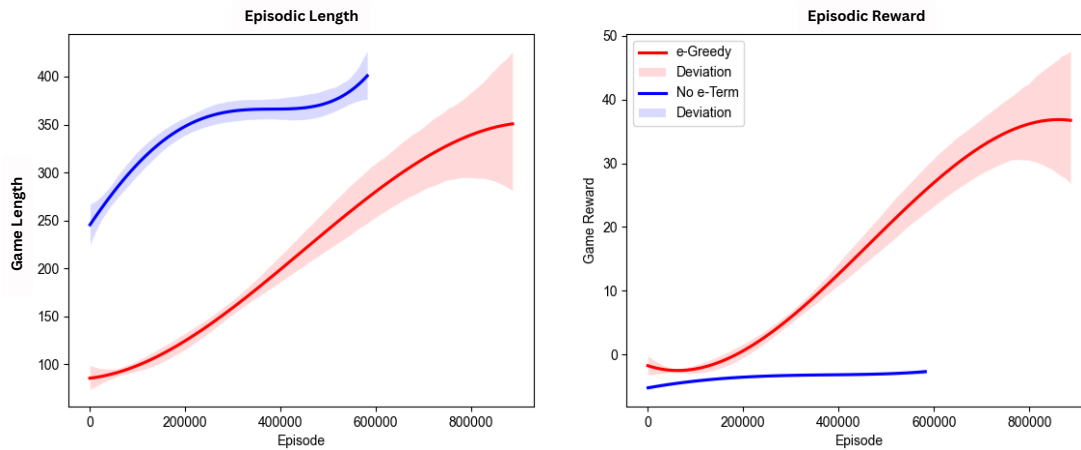
Another idea for improving the DQN algorithm is to use imitation learning with a rule-based algorithm to help guide the agent in the right direction. This is done by using the  $\epsilon$ -greedy policy to choose between the actions of the policy network and the rule-based method. However, in this case, the threshold value is filtered out and the policy network will take over from the rule-based method. The experience stored from this rule-based method is then used to train the policy and target network which will essentially transform part of the problem into a supervised learning scenario.

The rule-based method would work by trying each different position – x position and rotation of the shapes – that the current piece can move to and choosing the best possible case. The piece will then choose deterministic moves to move to the desired position and rotation and drop in place.

In this case, the rule-based method will start off with large rewards but will quickly decrease to a standard episode length and reward compared to the pure DQN model. The pure DQN model surpasses the rule-based method rewards at around 600k timesteps. This can be due to the model being limited by the rule-based method since it is designed by the developer's understanding of the environment, rather than finding the best strategy to



**Figure 4.8:** The rule-based method does start to learn faster in the beginning, but due to the human qualities in the rule-based formula, the method has limited capabilities and the pure DQN model quickly surpasses this.



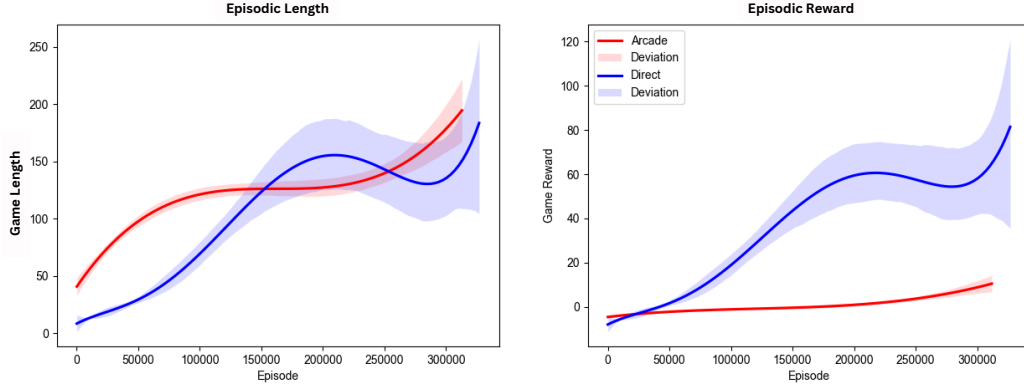
**Figure 4.9:** The model using the  $\epsilon$ -greedy (red) is less likely to get trapped in a local minima than the greedy policy (blue). This leads to higher episodic returns.

play the game.

### $\epsilon$ -greedy policy

Alternatively, to the rule-based algorithm, the  $\epsilon$ -greedy policy could be used to maximize the random actions for the starting period of the training, emphasizing the exploration of the model to find the best-suited actions for the given input states. This tends to have a longer training period, however, in some cases, the rewards for the given tests would yield better results than the vanilla DQN methods.

The introduction of an exploration term also allows the agent to find the best actions and prevents the agent from being trapped in local minima. In Figure ?? the comparison can be seen between the explorative model and the greedy model. The explorative model using  $\epsilon$ -greedy performs better by finding the best actions and the greedy policy makes little to no progress in increasing the total reward.



**Figure 4.10:** The direct action space simplifies the environment since the agent will learn from every step. The arcade action space has too many unimportant moves and is therefore less sample-efficient.

**Table 4.1:** Hyperparameter values for the final DQN model to be used for Melax’s Tetris

| Parameter                    | Min        | Value      | Max         |
|------------------------------|------------|------------|-------------|
| Learning Rate ( $\alpha$ )   | 0.0005     | 0.001      | 0.003       |
| Discount Factor ( $\gamma$ ) | 0.90       | 0.90       | 0.99        |
| Start $\epsilon$             | 0.7        | 1          | 1           |
| Stop $\epsilon$              | 0.01       | 0.1        | 0.2         |
| $\epsilon$ Discount Rate     | 10000      | 30000      | 100000      |
| Hidden Layer Size            | [256, 128] | [512, 256] | [1024, 512] |
| Batch Size                   | 128        | 256        | 512         |
| Target Update ( $C$ )        | 1000       | 5000       | 10000       |

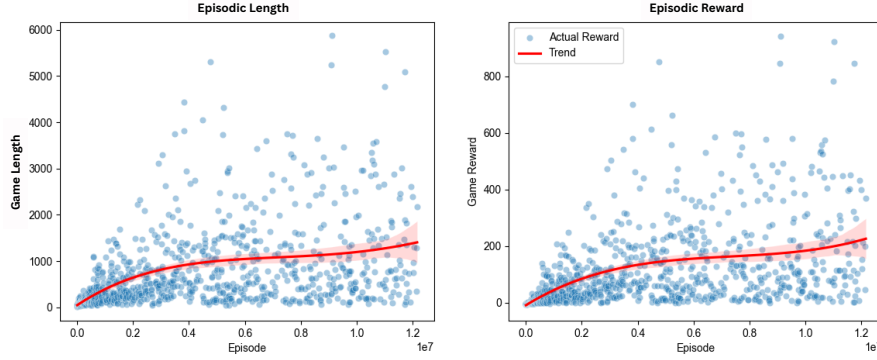
### Direct and arcade action space

Lastly, the difference between the direct and arcade action space is tested. These action spaces are covered in Section 3.3.2. The result is that the direct agent learns much quicker than the arcade counterpart, even though they have the same reward function. This can be seen in Figure 4.10.

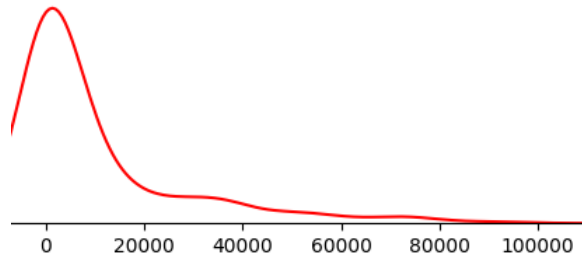
The episodic length of the two methods seems to be similar, however, it is important to remember that the direct agent places a piece with each action taken whereas the arcade agent can take up to 8 moves to complete this task. Therefore, the episodic lengths cannot be compared, only the rewards are comparable and from this, the direct agent outperforms the arcade agent substantially.

### 4.2.7. Final DQN results

The final model is trained using the reward shaping system with the  $\epsilon$ -greedy policy. This method proved the best suited for the environment, yielding efficient sampling and utilizing the training time better than other setups. For Melax’s Tetris environment, the final



**Figure 4.11:** The arcade DQN model consistently improved the gameplay of Melax’s Tetris as the training time increased. The model reached an average score of 200 lines cleared with an  $\epsilon$ -threshold of 0.1.



**Figure 4.12:** The distribution of the scores are centered around 20000 lines cleared, while reaching higher scores less frequently.

model is trained on the direct and arcade action space to simulate a human player playing the game by pure actions or by planning. The results for both of the Action spaces are given and compared for full analysis.

The hyperparameters for both cases of the model are finalized as seen in Table 4.1. The arcade model is trained for 12 million steps and yielded the results as seen in Figure 4.11 when applied to Melax’s Tetris. The model is then used to play the Tetris game and produce the results as seen in Table 4.2.

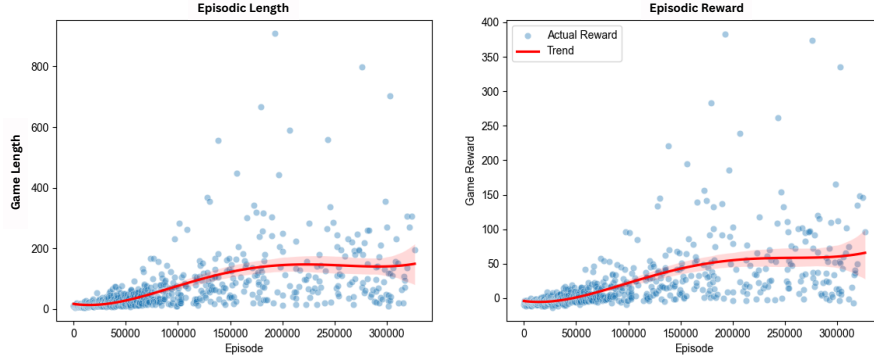
**Table 4.2:** Results produced by the arcade DQN model

|                        |       |
|------------------------|-------|
| Total Validation Games | 115   |
| High Score             | 92352 |
| Average Score          | 20627 |

The distribution of the game scores can be seen in Figure 4.12 and showcases that the model does have high scores, but regular lower scores are showcased. This emphasizes the off-policy method as usually, these methods do not perform as well online as on-policy methods.

The gameplay for this agent reveals the speed at which decisions are made and surpasses the capabilities of any human player in Melax’s Tetris environment.

The direct action space has a similar training trend but is only logged for 300000



**Figure 4.13:** The direct DQN model learned fast within the environment and provided higher scores than the arcade DQN model with a much lower training time.

**Table 4.3:** Results produced by the direct DQN model

|                        |        |
|------------------------|--------|
| Total Validation Games | 7      |
| High Score             | 404561 |
| Average Score          | 105670 |

training steps as seen in figure 4.13 and trained much faster than the arcade action space. However, the arcade version also managed to get high scores in Melax’s Tetris environment, proving that the agent can learn to play with the arcade action space given enough time.

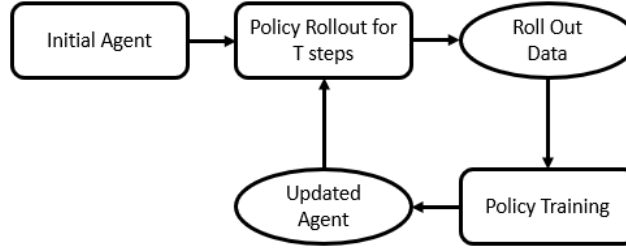
The direct action space is validated and produced the results from Table 4.3. The DQN model is finalized and the next task is to develop the PPO model.

### 4.3. Proximal policy optimization

To cover all bases, a PG method is tested on the Tetris environment to compare the capabilities of the PG methods to the DQN method. PPO is the chosen PG method which utilizes the ratio of the old policy log probabilities vs. the current policy log probabilities. Although difficult to implement and sensitive to changes and hyperparameters, this algorithm provides better convergence and stability when compared to other RL methods. The PPO method is popular amongst the RL community and is documented well with examples and explanations of the policy [8].

The custom implementation of the PPO algorithm includes work from different sources such as an implementation by Ilya Kostrikov [12] and Phil Tabor [13] This improves the efficiency and accuracy of the method by combining the strengths of each of the examples. This can be done since the PPO algorithm consists of a few key sections that work together to maximize the reward in an environment.

These sections include the experience memory, the actor and critic networks, the action function, the advantage function, and the optimization function. These functions and structures were chosen from different sources to increase the computational efficiency of



**Figure 4.14:** The PPO training loop involves the generation of the training data for the current policy by taking  $T$  steps within the environment. This training data is then used to update the current agent, which will be used to start the process again.

the model. However, before the implementation of the methods, these need to be fully understood and explained in the subsections below.

### 4.3.1. Training loop

The sections of the PPO algorithm are combined into the training loop to form the structure of the algorithm. The main objective is to create experience under the old policy – or the old version of the actor and critic networks – to serve as the input to the rest of the sections. This is done by running the agent in the environment for a set number of timesteps, with each step saved to the memory of the algorithm. This loop is known as the actor loop since this loop utilizes the actor network to generate the log probabilities for the action space and thereby the action choices that are used to make the state changes in the environment.

When the actor loop finishes, the advantage estimates are calculated by using the discounted return function together with the full history of the memory that is generated. These advantage estimates are then used to optimize the model weights and biases in the optimization stage of the algorithm. The advantage estimate calculation is explained in Section 4.3.5.

The optimizing loop will run for a specified number of epochs to update the networks based on the total loss that is calculated using a minibatch of the experience generated during the training loop. The PG loss is then combined with the value loss and the entropy loss to form the total loss which is used in the optimizer step function for the network update.

A simplified version of the training loop is shown in Figure 4.14

This flow diagram can be used to construct a pseudo-code representation of the model that will be used to develop the custom PPO implementation in *Python*. This pseudo-code representation can be seen in algorithm 4.3 and is the building block for the model.

**Algorithm 4.3:** Custom PPO Algorithm

---

```

Initialize memory class  $\mathcal{D}$ 
for ep in episodes do
  Initialize environment and first observation  $s_1$ 
  for  $t$  in timesteps do
    Generate rollout data with policy  $\pi_{\text{old}}$ 
    Compute the advantage estimate  $\mathbb{A}_1, \mathbb{A}_2, \dots, \mathbb{A}_t$ 
  end for
  for ep in epochs do
    Calculate the surrogate loss function with policy  $\pi_{\text{new}}$  vs.  $\pi_{\text{old}}$ 
    Update the network weights  $w$ 
  end for
   $\pi \leftarrow \pi_{\text{old}}$ 
end for

```

---

**4.3.2. PPO batched experience**

This class is used to store the experience that is gained from the interaction between the agent and the environment. This memory block will serve as the input to the optimization function where the experience will be sampled and used to calculate the PG loss for the actor network.

The relevant features saved in the memory are the current state, the log probabilities of the action that was chosen under the old model, the estimated values for the states by the critic network, the actions that were chosen under the old policy, the rewards received from the chosen actions and the terminal flags for the environment.

These features make up the basic structure of the main objective function. Since the PPO method is an Online method, this means that the memory will need to be cleared after each policy optimization to ensure that only the current policy and the old policy experience will be considered during the optimization phase of the algorithm.

**4.3.3. Actor-critic networks**

The network structure that is used is similar to the DQN model with two neural networks of the same shape, excluding the output layer of the networks. The difference lies not in the output shape of the networks, but rather the output type of these models.

The PPO actor networks determine the probability distribution for the action space rather than the expected reward for each action. The action is then chosen from this output by sampling from this distribution, which allows for exploration within the environment.

The critic network then estimates the value of being in the specific state as the expected return from the given input. This is then used to calculate the advantage function with the actual return from the gameplay.

This critic value can be used to compare against the actual value  $v^*(s, a)$  for the state



and action pair and allows the method to compare a loss which is used to update the weights of the agent and improve the performance in the environment. These networks are optimized to select the best action in the action function.

#### 4.3.4. Action function

When given an input state, the actor network will produce a probability distribution for all the actions that are available. This is used to choose the action for the state by sampling from this array of probabilities and using this to change the environment.

Together with this, the function will produce the logarithmic probability and the entropy parameter for the action that was chosen. In the same function, the value approximation for the state will be done and saved to the memory for use in the optimization phase of the algorithm.

The entropy for the selected action is also calculated in this function using the *Torch.entropy* function and is discussed in Section 4.3.6. All of these values generated, as well as the advantage estimates, are important for the calculations in Section 4.3.6.

#### 4.3.5. Advantage estimate calculation

The advantage estimates  $\hat{A}$  is the difference between the actual return and the expected return as calculated by the value function. This is done by using the action, rewards, and value histories that are sampled from the memory.

The actual values of the rewards can be used to calculate the real return for the given states, and this is then compared to the expected return from the critic network. This essentially transforms into a supervised learning situation and will give an indication of how good the actions actually were compared to the expected return.

The calculation of the advantage estimate is denoted by equation 4.2.

$$\hat{A} = \sum_{t=0}^T \gamma^t R_t - \mathbb{V}^{\pi}(s) \quad (4.2)$$

The effect of the advantage function on the updates of the network is further discussed in Section 4.3.6. The developer can now choose to normalize the advantage function or leave it in its original form. By normalizing the returns the speed and stability of the algorithm will be optimized. The advantage estimate is part of the concepts that are required to understand the PPO theory.

#### 4.3.6. Proximal policy optimization theory

Developing the PPO model requires a deep understanding of the mathematical concepts and the main surrogate objective. PPO aims to constrain the size of the policy updates for

a single time step. This is to ensure that the policy remains stable and is robust against policy crashes that can often occur in other methods such as DQN. PPO does this by using a clipping function in the main objective of the method. The result is that PPO is an elegant solution to the problem at hand.

The main objective for PPO is described in equation 4.3.

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}[\min(r_t(\theta)\hat{A}_t, \text{clip}(1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (4.3)$$

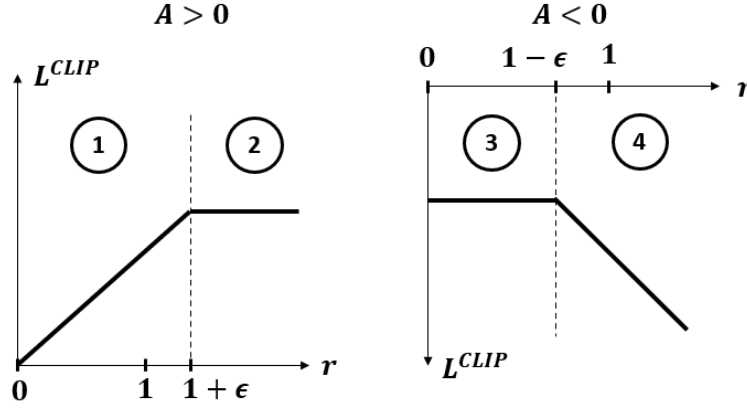
This formula compares the probability of the current action under the new policy to the old policy. In this equation, there are three parts at work to update the policy in the correct direction.

1. The probability ratio  $r_t$ . The idea is that for a case where the action is more likely in the new policy than the old policy, then the probability ratio  $r_t > 1$ . However, in the reverse case when the action is less likely, the result will be  $0 < r_t < 1$ . This gives an indication of the direction of the gradient of the policy.
2. The advantage function  $\hat{A}_t$ . This compares the actual rewards that were received during the interactions with the environment to the expected reward that the value function estimated. If an action received more reward than the expected reward, then the advantage function  $\hat{A}_t > 0$ , and the policy would want to increase the probability of choosing this action. However, if the actual reward was less than the expected reward then  $\hat{A}_t < 0$  and the policy would try to decrease the probability of choosing this action.
3. The clipping function. This clipping function limits the clip loss surrogate function, which in turn limits the size of the policy updates. This protects the policy against crashes during the updating processes.

These parts work together to guide the policy toward the solution in a unique manner. The result of this function is described in Figure 4.15.

If the action taken was less probable for the current policy than it was in the old policy, the loss would fall into region 1. In this case, the PG is in the wrong direction since it would want to increase the probability of choosing actions that yield higher rewards. This will update the policy proportional to the probability ratio to increase the probability of selecting the same action for the current state in the future.

Still considering the  $\hat{A}_t > 0$  graph, if the action was more probable for the current policy compared to the old policy, the loss would be in region 2. In this case, the loss is clipped to ensure that the size of the update is limited, and the policy would not update far from the original policy. The updated policy will have an increased probability of selecting this action since it yielded a positive advantage estimate.



**Figure 4.15:** The clipping functions ensure that the PPO model does not update by a large amount. This protects the policy against crashes and improves the stability of the model [8].

In the case where  $\hat{A}_t < 0$ , the selected action had a lower return than the expected return, and therefore the policy will want to decrease the probability of the same action being chosen for the given state. Given that the action was less probable for the policy compared to the older version of the policy, the loss value falls in region 3 of the graph. In this case, the PG is in the correct direction, however, the changes made are limited by the clipping function to protect against policy crashes and limit the update size.

However, when the action becomes more probable, the loss is in region 4 and the gradient is in the wrong direction. In this case, the policy update size is proportional to the error that the model is currently generating, and a larger update can be made since the model requires a re-evaluation.

The final loss formula incorporates the clipped surrogate loss, as well as the loss estimated from the critic network, and the entropy loss calculated from the action probabilities. The value loss merely compares the estimated value of the actions taken to the actual return from the rollout data. The entropy loss can be toggled to be incorporated or not and will introduce an exploration term into the loss function and allow the PPO model to explore more actions in the environment. This can be seen in equation 4.4

$$L_t^{\text{total}} = \mathbb{E}[L_t^{\text{clip}} - c_1 L_t^{\text{vf}} + c_2 L_t^{\text{entropy}}] \quad (4.4)$$

#### 4.3.7. Proximal policy optimization implementation

Once the theory is understood for the PPO method, the development process can begin. The implementation follows the theory explanation from Section 4.3.6 and uses classes to create each component of the algorithm.

### Storage class

This class is responsible for storing the rollout data from the experience generation phase of the algorithm. This class must store the information which relates the current states to the action that was selected. This must also then store the rewards received for this action and the logarithmic probability of selecting the action under the current policy. Finally, the value of reaching the next state – the critic network output – is also stored in this storage class.

All the features are stored as *NumPy* arrays to make calculations faster since the advantage estimate is calculated from these features. Once the policy optimization function starts and the calculations have been completed, the arrays are converted to *tensor* values to be used in the network updates.

### Advantage estimate calculation

The advantage estimate is the most computationally expensive calculation to be done in the algorithm and it is important that this is done efficiently and accurately to retain both the speed of the model and ensure that the model does not train on incorrect data.

This calculation is done using the rollout data for the rewards and the discount factor  $\gamma$  and follows the pseudo-code as written in algorithm 4.4.

---

**Algorithm 4.4:** Calculating the advantage estimate

---

```

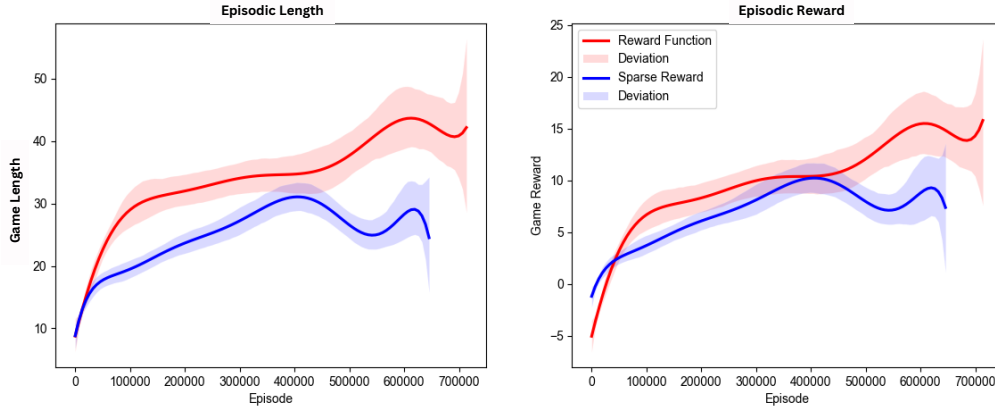
Initialize returns array with length equal to timesteps
for t in reversed timesteps do
    Check if the next state leads to a game over
    next_reward  $\leftarrow$  rewards[t]
    returns  $\leftarrow$  rewards[t]
end for
 $\mathbb{A}_t \leftarrow$  returns - values

```

---

### Loss calculation

The clipped loss is calculated using the advantage estimates and the probability ratio of the probability distribution for the actions under the old policy vs. the new policy. This probability ratio is calculated using the rollout data for the old policy and creating new probabilities in the optimization phase using the updated actor network with the same states as input. These new probabilities are used to see whether the policy is updating in the right direction and together with the advantage estimate will form the equation for the clipped loss (equation 4.3). The clipping function is completed by using the clip range value (usually  $\epsilon = 0.2$ ) and the *Torch.clamp* function. This creates the effect described by Figure 4.15.



**Figure 4.16:** From the figure it is clear that the shaped reward function has higher episodic returns. This is due to the extra information that the agent has when placing shapes and allows the model to converge faster.

#### 4.3.8. Melax's Tetris configurations and results

The PPO model is applied to different configurations of Melax's Tetris and trained for 1 million steps to determine the trend and the feasibility of the model and the optimal game setup. The initial tests in the environment use the arcade action space (Section 3.3.2) to interact with the environment and generate the necessary data for the network updates.

##### Sparse reward function compared to shaped reward function

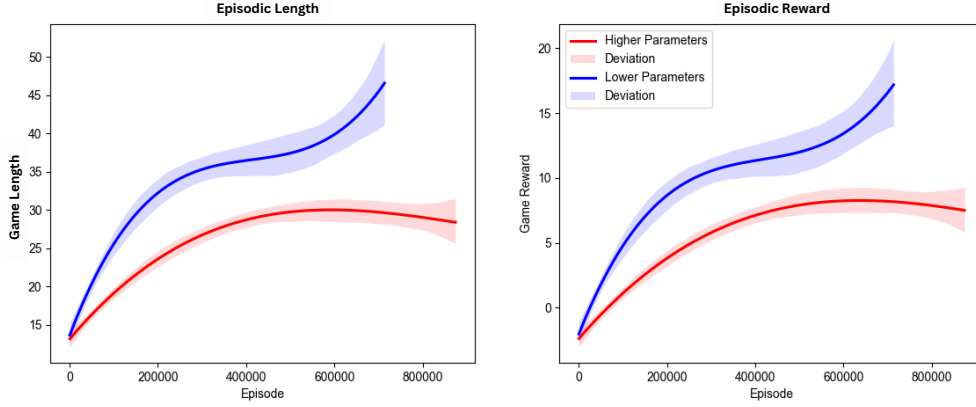
This test compares the sparse reward function with the shaped reward function and determines which setup would be optimal for the agent to learn from.

The results of the agents can be seen in Figure 4.16 which clearly shows that the shaped reward function setup improves the agent's performance and speed of learning. This is the same as the DQN model and shows the importance of having extra information when placing the shapes in desirable or undesirable locations and guides the model toward the line clears.

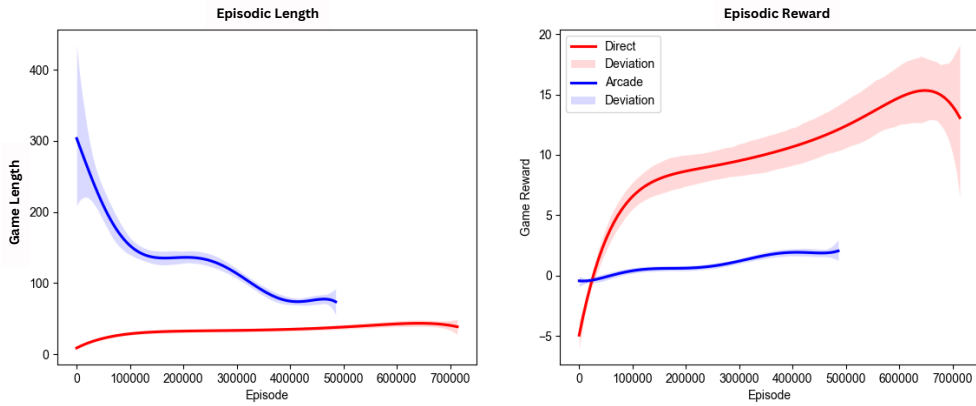
##### Hyperparameter ranges

A second test is done to test the effect of the learning rate and the total timesteps that the agent will interact with the environment to generate the training data. The learning rate is important as it will contribute to the stability of the PPO model and the amount of timesteps will allow the agent to update more or less often, which also contributes to the frequency of learning. The results are seen in Figure 4.17.

The effect is that a lower learning rate in the range of  $\approx 0.0001$  improves the stability of the agent and the decreased time between network updates allows the agent to be more sample efficient and increases the performance of the solution.



**Figure 4.17:** A lower learning rate ( $\alpha = 0.0001$  vs.  $\alpha = 0.001$ ) and frequency of learning (steps = 512 vs. steps = 1024) increased the performance and stability of the PPO model.



**Figure 4.18:** The direct action space learns to receive higher episodic rewards much faster than the arcade action space. Note that the action spaces differ and therefore the episodic length can not be compared, only the episodic reward.

### Arcade action space vs. the direct action space

The second test to be run is using the arcade action space vs. the direct action space to help the agent find the optimal placements without having to manually move the pieces to this location.

In Figure 4.18 this comparison is shown. The conclusion, similar to the DQN results, is that the direct method allows the agent to learn from more important experiences and can understand the environment faster. This is due to the simple efficiency of the direct model since every move is a significant move.

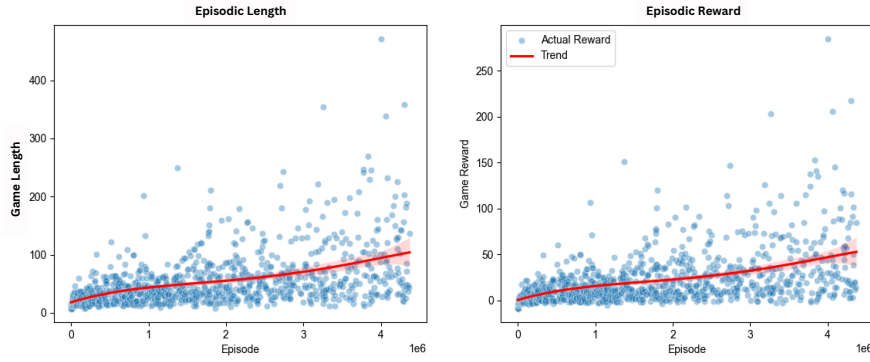
### 4.3.9. Final PPO model and conclusions

The final model used the direct action space to interact with the environment and a non-vectorized setup. The hyperparameters for this model were manually refined by comparing different tests and the final model parameters are given in Table 4.4.

The final model is trained for 4.5 million training steps and the final results of this

**Table 4.4:** Hyperparameter values for the final PPO model to be used for Melax’s Tetris

| Parameter                    | Min        | Value      | Max         |
|------------------------------|------------|------------|-------------|
| Learning rate ( $\alpha$ )   | 0.0005     | 0.001      | 0.003       |
| Discount factor ( $\gamma$ ) | 0.90       | 0.90       | 0.99        |
| GAE lambda                   | 0.90       | 0.95       | 0.99        |
| Hidden Layer size            | [256, 128] | [512, 256] | [1024, 512] |
| Batch size                   | 32         | 64         | 128         |
| Epochs trained               | 10         | 10         | 10          |
| Maximum gradient norm        | 0.2        | 0.5        | 0.7         |
| Value loss coefficient       | 0.3        | 0.5        | 0.7         |
| Entropy loss coefficient     | 0          | 0.001      | 0.1         |

**Figure 4.19:** The PPO agent was successful at learning to play Melax’s Tetris and improved its gameplay as the training time increased.

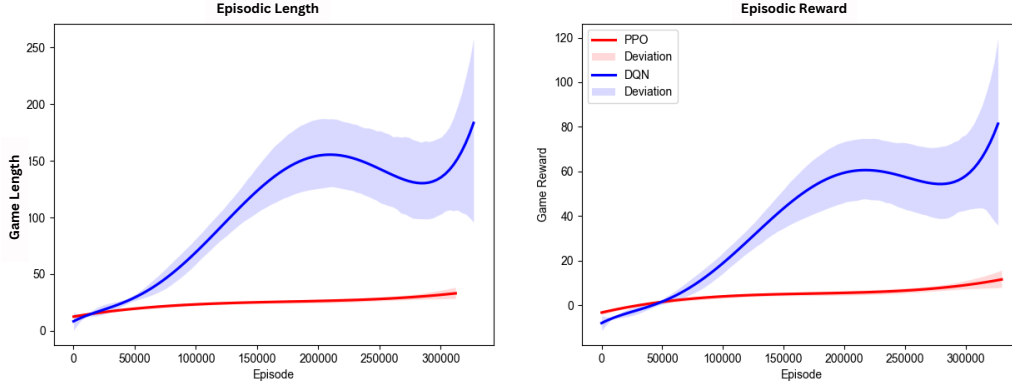
are shown in Figure 4.19 and Table 4.5. The agent showed improvement, but due to the complexity of the model and the time constraint on the project, the model could not be improved further. This will be set as future work to finetune the hyperparameters of the PPO model.

## 4.4. Comparing DQN and PPO models

Melax’s Tetris is only the testing phase to select a suitable algorithm to be used to attempt to solve the Full  $20 \times 10$  Tetris environment. In the previous sections, the two algorithms selected for this (PPO and DQN) were explained and implemented to solve Melax’s Tetris and both were successfully learned within the environment. It is important

**Table 4.5:** Results produced by the PPO model

|                        |     |
|------------------------|-----|
| Total Validation Games | 513 |
| High Score             | 182 |
| Average Score          | 42  |



**Figure 4.20:** The DQN and PPO models are compared using Melax’s Tetris as a testing environment. The result is that DQN outperforms PPO and will continue to undergo development to be applied to full Tetris.

to note that the DQN model did perform better than the PPO model and this is due to a few contributing factors. The models were both trained on the same network sizes to compare the functionality of the algorithms on an even base. The algorithms also used only one instance of Melax’s Tetris environment instead of a vectorized environment. This meant that PPO which is less sample efficient than DQN yielded larger training times and often got trapped in local minima.

The PPO model is much more difficult to implement than the DQN counterpart due to the complexity of the algorithm and the factors that influence this model. The testing revealed that the simplified setups often performed better. PPO is also very sensitive to hyperparameter changes and results in large deviations to the solution of the model if changed. The DQN model proved to be easy to implement and less sensitive to changes, helping in the development process to finetune the model and create a working model that successfully played Tetris past the capabilities of any human player.

The PPO model, although successfully improving at the game and learning basic mechanics, could not completely learn the environment, resulting in lower scores than the DQN model, and did not surpass the capabilities of the human player.

The final results of the models trained by the DQN and the PPO algorithm are compared by comparing the algorithms using the direct action space approach and can be seen in Figure 4.20. The clear choice for the completion of the project is to continue with the DQN model for the full  $20 \times 10$  Tetris since the performance of the custom DQN model is better than the PPO counterpart.

The results of the gameplay are displayed in table 4.6 to verify the choice of algorithm.



**Table 4.6:** Results for different algorithms in Melax’s Tetris

|                    | PPO Direct | DQN Direct | DQN Arcade |
|--------------------|------------|------------|------------|
| High Score         | 183        | 404561     | 92352      |
| Average Score      | 37         | 105670     | 20627      |
| Standard Deviation | 33         | 151898     | 20407      |

## 4.5. Chapter summary

Melax’s Tetris was used to develop and test custom DQN and PPO models. The models were first tested on known environments such as OpenAI Gym’s [Cartpole](#) environment to ensure the models are functional. These models were then used to find the optimal setups of Melax’s Tetris and the best hyperparameters for the agents, which will be essential for the final application of the model to full Tetris.

The tests confirmed that the direct action space and the shaped reward function were the best setup for the environment. The original observation space was used throughout as the complexity of the  $[1, 50]$  array was low enough for the models to successfully learn the environment.

Both models succeeded in learning Melax’s Tetris, with the DQN model performing better than the PPO model (discussed in Section 4.4). The DQN model is thus the best option for the application to full Tetris and will be further developed and optimized in the next chapter.

# Chapter 5

## Reinforcement learning for full Tetris

This chapter will cover the further development of the DQN model that succeeded in learning to play Melax’s Tetris. The model will be deployed within the original  $20 \times 10$  Tetris environment (Section 3.2). This version of Tetris will from now onwards be referred to as full Tetris.

There are different challenges related to the scalability and the performance of the DQN model when applied directly to full Tetris and will require changes to be made to the DQN model for optimal performance.

### 5.1. Full Tetris challenges

Once the method has been finalized, it can be used to attempt to solve the full Tetris environment. This environment is four times larger than Melax’s Tetris version and poses several problems for implementing and scaling the DQN model. The complexity of this environment is much higher than Melax’s Tetris, and some refinement will be needed to create a working model for the environment. The model’s scalability is tested and optimized for improved results and more efficient learning rates.

#### Large observation space

The observation space of the full Tetris environment is an array of size  $[1, 200]$ , meaning that the size of the network will also have to increase to cater to the complexity of the problem. There are also seven different pieces with more rotation options, leading to an increased number of possible states that the agent will have to learn. The use of convolutional neural networks (CNN) could help with this problem, introducing feature extraction and improving the scalability of the network.

Another method is to convert the observation space to a simplified version (Section 3.3.1) by preprocessing the game field before passing it as an input to the network structure.

## Sparse rewards

The full Tetris environment is more complex than Melax's Tetris environment as it has seven shapes – compared to five shapes – of larger size and more rotation options (Section 3.2). This, combined with the size of the game field, means that the chances of clearing lines by chance through the  $\epsilon$ -greedy policy are very small compared to Melax's Tetris.

A shaped reward function for the environment will be engineered and follow the structure of the Melax Tetris environment; however, it will have to be updated to improve the mapping of state to action in the algorithm.

## 5.2. Different approaches for the DQN model for full Tetris

Due to the complexity and challenges of full Tetris vs. Melax's Tetris, a few critical changes needed to be made to improve the performance and accuracy of the model. The goal of the changes is to reduce the complexity of full Tetris for the DQN agent to be able to learn faster and more efficiently. This builds on the comment in Section 4.4, 'simplicity is often better'.

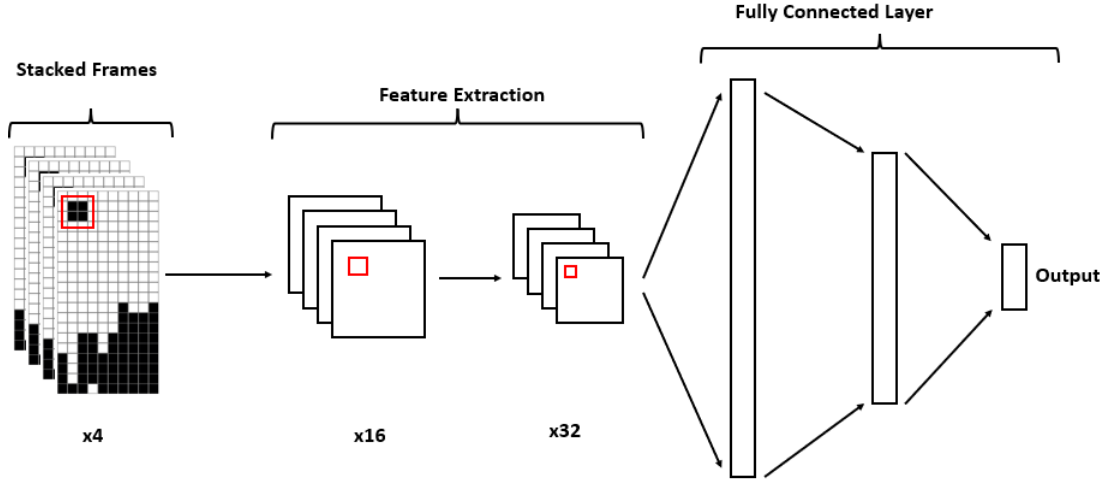
### 5.2.1. Convolutional neural networks and frame stacking

#### Explanation and Setup

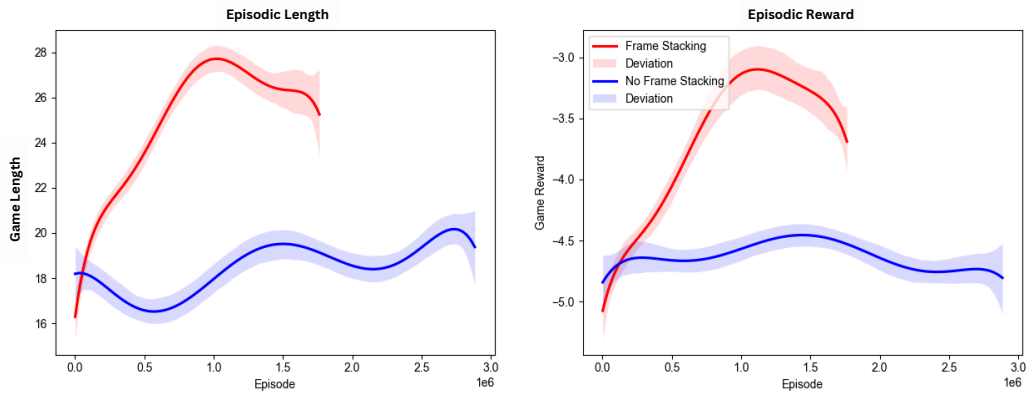
Since the complexity of the game field is higher, it is necessary to increase the size of the networks that will learn the environment. The first test is to add CNN layers to the network and update the size of the hidden layers. As stated in Section 5.1, this will introduce the concept of feature extraction to the matrix environment and generalize the input space. An important note is that CNN layers output multiple channels based on the weights of the kernels. This means that more than one channel can be used as an input to the agent's network, and the CNN output will be multiple channels, each extracting different features from the model.

This introduces the concept of frame stacking, where multiple successive observations are stacked on each other. This will include the current and the previous  $N$  states observed. By doing this and inputting these stacked frames, the agent will be able to get a sense of direction and velocity in an environment.

This extra information adds to the complexity of the problem. Still, when used with the CNN, the agent can learn more complex environments and perform better than the pure linear network structure. This was proven to be the case as frame stacking helped to increase the efficiency of the CNN model and allowed the agent to learn more complex situations.



**Figure 5.1:** The input to the CNN will be a frame stacked version of the current and previous three states. This will then be fed through the CNN to produce an output of 32 frames, which is flattened and passed to a fully connected linear layer. The output will be the estimated  $Q$ -values for each of the available actions. [14]



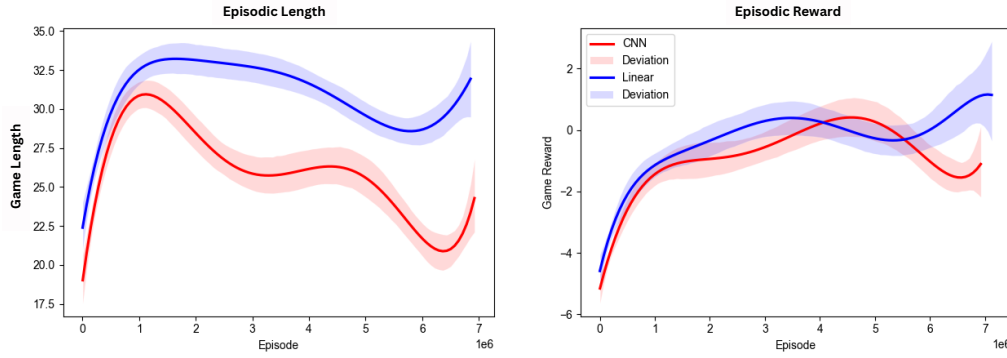
**Figure 5.2:** The effect of frame stacking is that the model was more successful in learning within the environment.

Using frame stacking, the structure of the CNN model can be described in Figure 5.1 and shows how the input states are processed to produce the output actions.

## Results

The CNN structure is applied to the Tetris environment with and without frame stacking. This was to test the importance of having previous state knowledge and its effect on the agent. The results clearly show that using frame stacking with the CNN model improved the agent's performance and stability. However, the model did take longer to train in real time due to the extra computational power required.

The CNN model is then compared to the linear model. Figure 5.3 shows that the linear model is more stable than the CNN counterpart and learns faster. This could be because the CNN overfits the environment and creates a complex and large input space as input



**Figure 5.3:** The linear model performed better than the custom CNN model. The training time of the CNN was also significantly longer than the linear counterpart due to the complexity of the model.

to the fully connected layers.

This is a significant factor, with the CNN taking nearly twice as long as the linear model to reach the same number of steps. This, combined with the instability of the CNN model, proved the linear model to be superior for the final design of the RL algorithm.

### 5.2.2. Simplified Observation Space

Noticing that the linear model outperformed the CNN model, this again confirms that often simple is better. The complexity of the observation space creates a difficult task for the agent to learn and is less sample-efficient. Therefore, the original observation space for full Tetris is converted to the simplified observation space, as described in Section 3.3.1. The observation space for the complete Tetris is reduced from the  $[1, 200]$  space to a  $[1, 11]$  space, only containing the *column height* for each column and the currently available piece.

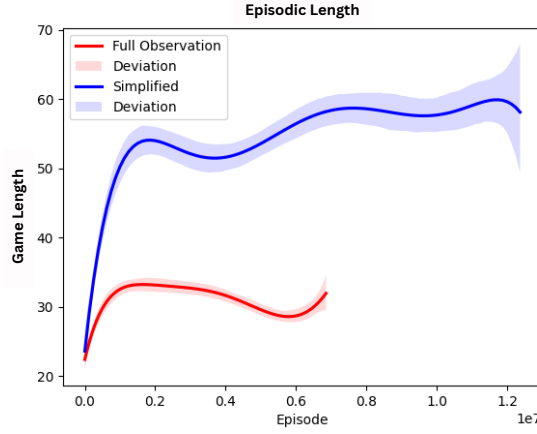
This change will reduce the complexity of the environment without removing too much information from the agent’s perspective. The result is that the agent trains faster and reaches higher scores with the simplified observation space. The network shape can also be reduced to improve the computational efficiency of the model.

In Figure 5.4 only the episodic length is compared. The reward function was the same for both runs, only the scale of the episodic reward plot differed for the models.

### 5.2.3. Action Space

Exploration is essential as the agent must navigate the environment to find the best actions to yield the most reward. After many attempts to create a working model with the current action space, the agent often moved the pieces around the playing field and seldomly dropped them into the correct locations.

This is resolved by converting from the arcade action space (with 4 different actions) to



**Figure 5.4:** The simplified observation space outperformed the original despite having a lower resolution and less information.

the direct action (with 40 different placements) space as described in Section 3.3.2. This allows the agent to learn directly from every placement rather than the trivial moves to position the shape into the correct location.

## 5.3. Final model structure and hyperparameters

### Network structure

The final model used to play the full Scale  $20 \times 10$  Tetris environment is derived from the previous sections' remarks and will pivot on the results collected. The final model will not be using a CNN as the simplified observation space performs better than the CNN model and is less computationally expensive. This is based on the results from Section 5.2.1.

### Observation space

The Observation space of the environment is chosen to be the simplified version as explained in Section 3.3.1. This setup significantly improved the learning ability and gameplay results and allowed the agent to score higher than the original observation space.

### Action space and reward function

The model will use the direct approach to make the learning more efficient and allow the agent to explore the essential actions, thereby improving the sample efficiency of the model. This choice is based on Melax's Tetris improvement, as seen in Figure 4.10.

With one change, the reward function will remain the same as the Melax Tetris function. The weight of the holes created and the standard deviation are increased to emphasize the effect of covering the entire playing field and avoiding making holes. The agent will find the line clear and play longer if this is done correctly. There will also be a penalty

**Table 5.1:** Hyperparameter values for the final DQN Model to be used for full Tetris

| Parameter                    | Min        | Value      | Max         |
|------------------------------|------------|------------|-------------|
| Learning rate ( $\alpha$ )   | 0.0005     | 0.001      | 0.003       |
| Discount factor ( $\gamma$ ) | 0.90       | 0.90       | 0.99        |
| Start $\epsilon$             | 0.7        | 1          | 1           |
| Stop $\epsilon$              | 0.01       | 0.15       | 0.2         |
| $\epsilon$ discount rate     | 10000      | 60000      | 100000      |
| Hidden layer size            | [256, 128] | [512, 256] | [1024, 512] |
| Batch size                   | 128        | 256        | 512         |
| Target update ( $C$ )        | 1000       | 5000       | 10000       |

for increasing the maximum height of the playing field. This can be modified to remove the lines cleared reward and incorporate a previous height variable. The final equation is represented by equation 5.1.

$$\begin{aligned}
 \text{Reward Function} = & (\text{Previous Height} - \text{Current Height}) \\
 & + 0.1 \times (\text{Previous Standard Deviation} - \text{Current Standard Deviation}) \\
 & - 0.05 \times (\text{Holes Created}) \quad (5.1)
 \end{aligned}$$

### Model hyperparameters

The final model is trained with the hyperparameters as shown in Table 5.1. The Melax's Tetris DQN hyperparameters were used as a guideline to choose the hyperparameters. These parameters were refined by comparing different setups in the same environment. The best-performing model was chosen from these groups and used in the final training.

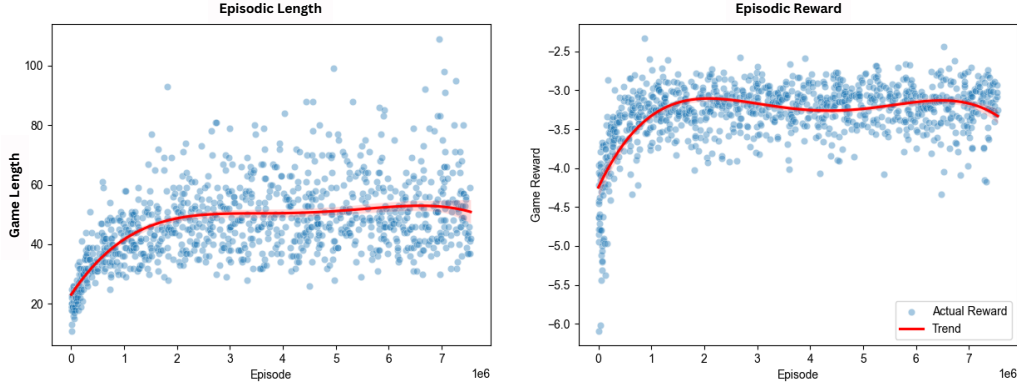
#### 5.3.1. Results of the final model

The model is trained for seven million steps; however, the model plateaus at around two million training steps. Figure 5.5 shows this trend.

The final DQN model was trained and produced the results from Table 5.2. To compare the model to average humans, 20 players were asked to play the game, and the highest score (line clears) of these players was recorded. The model performed well compared to average human players, who managed an average score similar to the model's performance.

In future work, the PPO model could be compared to the DQN model in full Tetris to investigate the effect of the complexity on the models and compare Online vs. Offline learning.

However, in this project's scope, the DQN model successfully solved the Tetris environ-



**Figure 5.5:** The final DQN model was successful in learning to play full Tetris and the agent was able to surpass human capabilities within the game.

**Table 5.2:** Results produced by the final DQN model

|                        | DQN model | Human players |
|------------------------|-----------|---------------|
| Total validation games | 500       | 20            |
| Line clears            | 280       | 106           |
| Average score          | 68        | 64            |
| Standard deviation     | 37        | 34            |

ment and proved that the model could reach and surpass human capabilities.

## 5.4. Chapter summary

The final DQN model was developed by analyzing the differences and the challenges associated with full Tetris compared to Melax’s Tetris. The CNN structure of the neural network and frame stacking was deployed within full Tetris but did not perform better than the linear model used in Melax’s Tetris.

The observation space was changed from the original to the simplified version to reduce the complexity of the problem. The action space and reward function were kept the same as the optimal setup for Melax’s Tetris, with a small change as in equation 5.1.

The DQN model was deployed in full Tetris and was able to learn to play the game at the average human level within two hours of real training time. This model was trained for longer and achieved higher scores and line clears than any human player who participated in the game.



# Chapter 6

## Summary and conclusion

This project was set out to create an environment for the testing, developing, and validating RL algorithms. The chosen development environment was Melax’s Tetris, a simplified version of the Tetris game. This environment was simple enough to develop basic RL algorithms yet complex enough to force the developer to use function approximators such as neural networks. The environment was designed and developed following an example structure [9] and converted to an OpenAI environment for integration with RL agents.

The different *observation* and *action* space options were identified and analyzed to find the optimal environment representation. The *reward* function for the environment was engineered throughout the project as different situations required more information.

A DQN and PPO model was developed using Melax’s Tetris environment. The development of the models compared different environment setups within the two algorithms to find the optimal solution for both models. The results were documented and discussed thoroughly to ensure an understanding of the problem, and both models could learn Melax’s Tetris. The PPO model and the results thereof were discussed since the model underperformed compared to industry standard PPO models. Once the optimal solution was found, the models were compared, and the best-performing model was chosen to solve the Full  $20 \times 10$  Tetris environment, in this case, the DQN model.

Melax’s Tetris environment was modified to the full Tetris environment. There were challenges identified with the scalability of the model and the complexity of the environment, and possible solutions were investigated to improve the model’s performance.

The final model was trained and succeeded at learning the full Tetris environment to a point where the DQN agent could play better than the average human player.

### 6.1. Findings and discussion

This project forces the developer to think about the inner workings of the two RL algorithms rather than just the implementation of DQN and PPO.

## Deep Q-learning

The DQN algorithm is easy to implement yet powerful and effective enough to learn complex environments quickly. The DQN agents were found to learn Melax's Tetris without worrying about extensive hyperparameter tuning or environment optimization. The model would learn irrespective of the observation space, action space, or reward function (sparse or shaped). These components did affect the outcome of the model, but all of the situations would succeed in learning. This algorithm proved to be time-efficient and computationally effective, with minimal calculations and loops required to work.

The DQN algorithm was the easiest to understand and implement correctly. This means it could benefit developers with little knowledge about RL to start with Q-learning before moving on to more complicated algorithms, such as PPO.

## Proximal Policy Optimization

The PPO agent was expected to outperform the DQN model in Melax's Tetris; however, the custom model did not reach this performance.

Firstly, on a developer basis. This model is difficult to understand, having many components working together to complete the final model. This makes it difficult to implement and find logical errors within the algorithm. The algorithm is also computationally expensive due to the advantage estimate calculation (see Section 4.3.5).

In future work, a different implementation of PPO could be tested on the environment to find whether DQN is truly better than PPO in the Tetris environment.

## 6.2. Final thoughts

Applying the custom RL algorithms to different complex environments proved to be possible but with a few key struggles along the way. Firstly, using a simplified version of the environments that the algorithms will be tested in is crucial in understanding the problem and finding the best setup of the RL models for the specific environment.

Building on the first comment, the RL models work better in less complex environments. Thus, any changes that can be made to the environment observation and action space to reduce the complexity of the problem are highly recommended, providing that the changes do not significantly reduce the information that the agent would normally receive.

Lastly, the RL models should be generalized. This makes it possible to apply the models to different environments and helps with the debugging process to find errors within the model and the environment.

From this project, the testing environment and the models that were developed can be used in future projects to test and develop new methods within RL and can serve as a basis on which future work is done.

# Bibliography

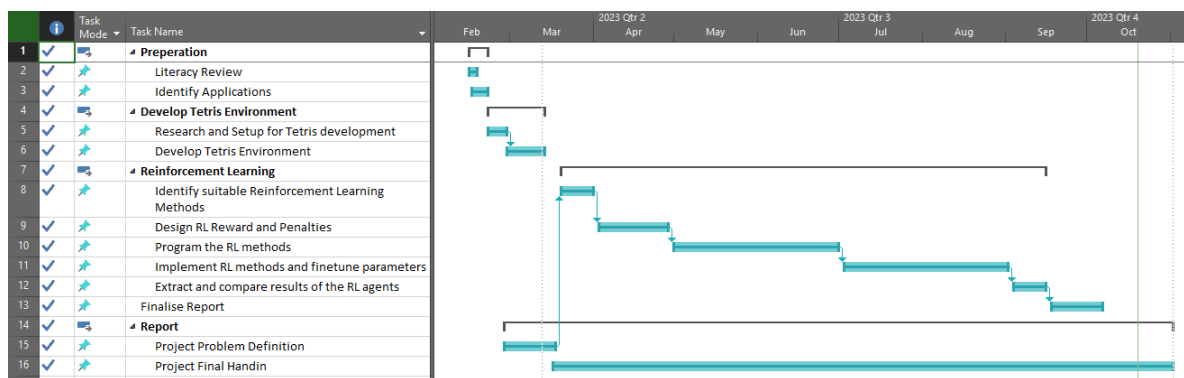
- [1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “A brief survey of deep reinforcement learning,” *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: <http://arxiv.org/abs/1708.05866>
- [2] Y. Nasir, J. He, C. Hu, S. Tanaka, K. Wang, and X. Wen, “Deep reinforcement learning for constrained field development optimization in subsurface two-phase flow,” *Frontiers in Applied Mathematics and Statistics*, vol. 7, 08 2021.
- [3] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” *The MIT Press*, 2017.
- [4] I. Sajedian, H. Lee, and J. Rho, “Double-deep q-learning to increase the efficiency of metasurface holograms,” *Scientific Reports*, vol. 9, 07 2019.
- [5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, Joel 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,” *Nature*, vol. 10.1038/nature14236, 2015. [Online]. Available: <https://doi.org/10.1038/nature14236>
- [6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: <http://arxiv.org/abs/1312.5602>
- [7] Y. Wang and S. Zou, “Policy gradient method for robust reinforcement learning,” 2022.
- [8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: <http://arxiv.org/abs/1707.06347>
- [9] T. Bakibayev, “How to write tetris in python,” *Medium*, May 2020.
- [10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.
- [11] V. Zhou, “Neural networks from scratch,” 2019. [Online]. Available: <https://victorzhou.com/series/neural-networks-from-scratch/>

- [12] I. Kostrikov, “pytorch-a2c-ppo-acktr-gail,” 2021. [Online]. Available: <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail/tree/master>
- [13] P. Tabor, “Reinforcement learning ppo,” 2020. [Online]. Available: <https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PPO/torch>
- [14] Phung and Rhee, “A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets,” *Applied Sciences*, vol. 9, 10 2019.

# Appendix A

## Project planning schedule

The planned events for this project are given in the following figure and will be followed to ensure the completion and testing of the RL models within the time frame set out.



# Appendix B

## Techno economic analysis

### B.1. Budget

**Table B.1:** Budget for outlined project schedule

| Activity                     | Estimated total |       | Actual time |       | Facility use | Final total |
|------------------------------|-----------------|-------|-------------|-------|--------------|-------------|
|                              | Hr              | R     | R           | Hr    | R            | R           |
| Literacy review              | 10              | 4500  | 8           | 3600  |              | 3600        |
| Identify applications        | 5               | 2250  | 5           | 2250  |              | 2250        |
| Research and setup           | 25              | 11200 | 22          | 9900  |              | 9900        |
| Develop tetris               | 25              | 11200 | 19          | 8550  | 500          | 9050        |
| Identify RL methods          | 10              | 4500  | 12          | 5400  |              | 5400        |
| Design rewards and penalties | 15              | 6750  | 13          | 5850  |              | 5850        |
| Program RL methods           | 25              | 11200 | 37          | 16650 | 500          | 17150       |
| Implement RL methods         | 25              | 11200 | 36          | 16200 | 500          | 16700       |
| Extract results              | 10              | 4500  | 10          | 4500  |              | 4500        |
| Finalise report              | 20              | 9000  | 22          | 9900  | 9900         |             |
| TOTAL                        | 180             | 81000 | 184         | 82800 | 1500         | 84300       |

The project exceeded budget by R3300 since the time it took to develop and implement the RL models was significantly under budgeted, and a total of four hours were spent over the estimated time.

The use of faculty resources such as the Firga computers is also taken into account. The developer did not require the use of the faculty HPC, so the cost is not considered.

### B.2. Technical impact

RL is difficult to understand and implement without prior knowledge of the field. This project attempts to give some context and explain how the different RL algorithms work and how these models perform in different situations and against each other.

This project can be the building block to simplify future RL projects. The models are designed to be easily interpretable, allowing future developers to understand the logic and make modifications.

### **B.3. Return on investment**

The developed RL models have the potential to be used in commercial environments, and the software could be sold with a software license for industry use. The models will optimize specific processes and tasks and potentially save companies money based on these optimization services.

### **B.4. Potential for commercialization**

RL has the potential for commercialization in a variety of applications. One example is robotics, where RL can be used to train robots to perform complex tasks with greater efficiency and accuracy. Another example is finance, where RL can be applied to optimize investment strategies and portfolio management. RL can also be used to develop autonomous vehicles, which can help improve self-driving cars' safety and efficiency.

Companies can develop and sell software solutions that incorporate RL algorithms for specific industries or applications to commercialize RL. Additionally, companies can offer consulting services to help clients develop and implement RL solutions tailored to their specific needs. As the field of RL continues to grow and evolve, there is significant potential for commercialization in a wide range of industries.

# Appendix C

## Project risk assessment

The main risk is that the computational power needed to train the RL models can exceed the limitation of the laptop that the program and models are being developed. This will prevent the models from completing training and result in non-optimized results. A sub-risk for the completion of the project is the ongoing load shedding, where the power goes off daily for 2 hours at a time. This is a risk as the project is entirely based on a computer, and simulations and programming require high computational power, which consumes more battery than a laptop.

These risks can be avoided by working in the Stellenbosch University Engineering Faculty on the Firga computers provided to the students. This will ensure that the program has access to sufficient processing power and that power will always be available during load-shedding times. Furthermore, the progress of the code will also be saved on the Stellenbosch University drive to prevent data loss. Data loss is another risk to be considered and can be prevented by uploading code commits to a service such as GitHub.



# **Appendix D**

## **Responsible use of resources and End-of-Life strategy**

### **D.1. Responsible use of resources**

This project is a pure simulation study and thus is developed and evaluated thoroughly in a computer environment. However, addressing the minor resources in the final product's design is still essential. This project will require a computer with high processing power capabilities. Training the RL algorithms can become computationally expensive and need the computer to run at total capacity for several hours. The computer will, therefore, require a reliable power connection and use the grid power and the generator power during load shedding. This means that costs for the power and the fuel required to run the generators must be considered.

Suppose the HPC must be used, which is a significant, shared university resource. In that case, the student will ensure that the code that will be run on the machine is thoroughly tested on a local computer to prevent the misuse of the HPC and the strict schedule.

A stable network connection is also imperative to the project's success, as developing the Tetris and the RL components requires a stable internet connection to run on web-powered compilers.

### **D.2. End of life strategy report**

The project's primary focus will be to evaluate and identify the best RL algorithms for learning and playing Tetris in a simulated environment. The project will thus end when the final RL results are compared and the best algorithms are selected.

However, if time permits and the student has the capabilities and knowledge to do so, the project could be extended to investigate further the RL algorithms and improve the testing setup for use in a possible master's.

The work done in this report forms part of a more significant study of the feasibility of using reinforcement learning to teach humans how to improve at Tetris. Once completed,

the code for the project will be released on GitHub with an MIT License and serve as a base for the next student to open and install to extend the project's life.