Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Solving Tetris using Reinforcement Learning

by

Divan van der Bank

Thesis presented in fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University.

Supervisor: Prof. H. Kamper
Co-supervisor: Het ek een?

November 2023

Acknowledgements

I would like to thank my supervisor, Prof. Herman Kamper, for reviewing my first draft even though it is not completely done yet. I also would like to thank the inventor of coffee; without him/her, I would not be here.

Prof. Herman Engelbrecht
Prof. Hugo Touchette
Tristan Legg

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

forward together
sonke siya phambili
saam vorentoe

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 is 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 woordelikse 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.

23603526 Studentenommer / *Student number*	[illegible] Handtekening / *Signature*
D.K. van der Bank Voorletters en van / *Initials and surname*	**2 October 2023** Datum / *Date*

Executive Summary

Title of Project
The development of Tetris as a testing environment for the development and comparison of Reinforcement Learning algorithms.
Objectives
Developing a stable Tetris environment for the development and testing of the Reinforcement Learning algorithms. Custom Reinforcement Learning models will also be developed to solve the Tetris game.
What is current practice, and what are its limitations?
Atari games such as Ms Pacman and published OpenAI Gym environments such as Cartpole are the main Reinforcement Learning testing platforms due to the availability and simplicity of the environments. 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 RL algorithms overfitting to these cases, and a new environment, such as Tetris, could introduce new problems and improvements to the algorithms.
What is new in this project?
The Reinforcement Learning algorithms will be designed from first principles and tested in a custom Tetris environment. The developed algorithms will be tested using different environment setups to determine how the algorithms will learn rather than just trying to optimize the algorithm.
If the project is successful, how will it make a difference?
The custom Reinforcement Learning models and the Tetris testing environment can be used as the building blocks for future projects that require Reinforcement Learning and give the developers an understanding of which methods to use and the type of setups and environment that work best with the Reinforcement Learning method.
What are the risks to the project being a success? Why is it expected to be successful?
The processing power of the computer on which the Reinforcement Learning agent will train 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.

What contributions have/will other students made/make?
A previous student has created a Sarsa Reinforcement Learning algorithm with linear approximation to play a simplified version of Tetris.
Which aspects of the project will carry on after completion and why?
The Tetris environment and the selected Reinforcement Learning models will 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 chosen configuration will be defined and documented in sufficient detail so that the production demands and marketing potential can be determined.

ECSA Outcomes

Abstract

The English abstract.

Uittreksel

Die Afrikaanse uittreksel.

Contents

List of Figures

List of Tables

List of Algorithms

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_π	Evaluate X at current policy π
X_*	Evaluate X at optimal policy π_*
X_t	Evaluate X at time step t

General Reinforcement Learning

α	Learning Rate
γ	Reward Discount Factor
$\mathcal{S}$	Set of all states
$\mathcal{A}$	Set of all actions
$\mathcal{R}$	Set of all rewards

Deep Q-Learning

ϵ	Threshold value for ϵ-Greedy Policy
$Q(S_t, A_t)$	State-Action Q-value
$\mathrm{TD}_{\mathrm{targets}}$	Temporal Difference Targets for Q-Learning Loss Calculation
C	Target steps for updating DQN Value Network

Proximal Policy Optimization

$\pi_\theta(A_t \vert S_t)$	Probability of choosing action A_t given state S_t for current weights θ
$\hat{\mathbb{A}}$	Advantage Estimate
$L^{\mathrm{clip}}(\theta)$	Clipped Surrogate Loss
$L^{\mathrm{value}}(\theta)$	Critic Network Loss
$L^{\mathrm{entropy}}(\theta)$	Entropy Loss

Acronyms and abbreviations

A2C	Asynchronous Actor-Critic Method
CNN	Convolutional Neural Network
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
TD	Temporal Difference
TRPO	Trust Region Policy Optimization

Chapter 1

Introduction

Machine learning is found in various applications and industries, including finance, healthcare, transportation, retail, and entertainment. The field has an enormous impact on the world, and as more research and development is done, the capabilities of machine learning will continue to grow and evolve. However, to develop a Machine Learning model, there must be a way of testing the algorithm or 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.

1.1. Background

Reinforcement Learning is an efficient and effective Machine Learning algorithm that is used in a wide range of applications. The algorithms learn by placing an agent in an environment and allowing this agent to make choices. The agent is then trained to reward desirable decisions and punish undesired ones.

The project proposed in this report is to develop different types of reinforcement learning algorithms to learn and play the popular arcade game Tetris. The comparison and evaluation of the different RL algorithms will be done by training the algorithms in the simulated game environment.

The RL agents will be evaluated based on the training time and computational power, as well as the methods' efficiency and complexity. Once the testing is done, the best RL methods will be identified for playing Tetris.

1.2. Project Objectives

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

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 Reinforcement Learning.

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 Reinforcement Learning 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 Reinforcement Learning agent can be applied to real-life situations, such as teaching a child to improve at reading or writing.

Chapter 2

Reinforcement Learning

Reinforcement Learning is a powerful and effective Machine Learning tool that has a wide range of applications and uses in fields such as Atari gameplay, flight operations, or self-driving cars. RL learns based on a trial-and-error approach and aims to map certain scenarios to actions to maximize the reward signal in an environment, thereby learning the optimal control in an incompletely known Markov Decision Process.

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 as to what the agent should learn by changing the reward function, but the agent will only learn from the experience that it has generated. [5]

RL uses an agent to interact with an environment, in this case, the Tetris game. The agent is a combination of Machine Learning functions that are used to 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 Markov Decision Process.

The agent will then receive a scalar reward value upon reaching the next state S_{t+1}. The essence of Reinforcement Learning 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 that are 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. This control loop can be seen in Figure 2.1.

2.1. Markov Decision Processes

An important property of Reinforcement Learning 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, $\{S_0, S_1, S_2, \ldots, S_{t-1}, S_t\}$. [2]

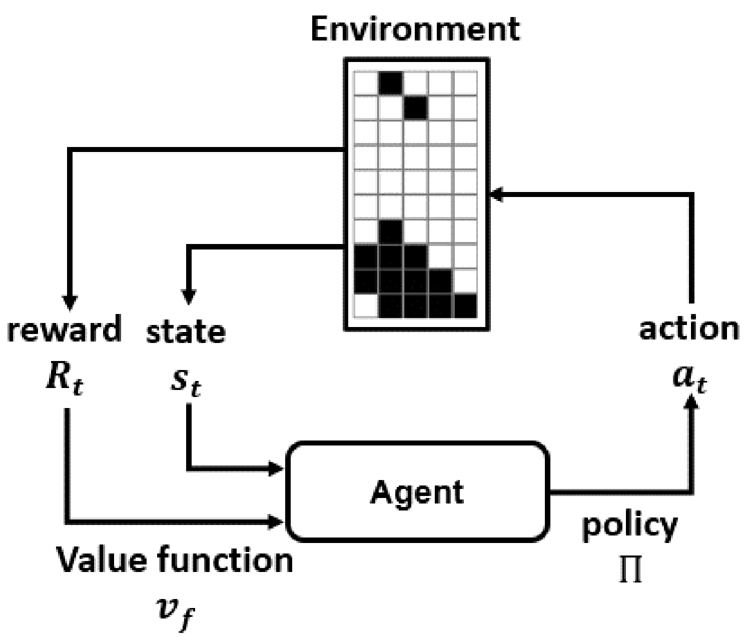


Figure 2.1: Reinforcement Learning Loop [1]

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

2.1.1. Structure of a Markov Decision Process?Q

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 as seen in Equation 2.1.

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

Next, a reward is introduced to indicate how good or bad it is to reach a specific state. The initial state will have no reward, however 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 behavior of the agent 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.

$$G_t = R_{t+1} + R_{t+1} + R_{t+1} + \cdots \tag{2.2}$$

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

$$G_t = R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+1} + \cdots \tag{2.3}$$

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 π, which in turn triggers state transformations to S_{t+1}, yielding a reward R_{t+1} for reaching this state.

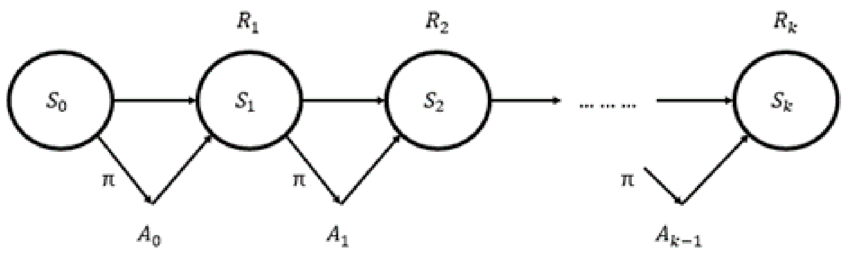


Figure 2.2: Markov Decision Process How do I reference the RL course??? Prof Hugo

2.1.2. 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 does this by estimating 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 need to 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

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

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

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 on the basis that 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.

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

The goal is to solve the Bellman Equation for all the states, allowing the agent to follow the states in increasing value to solve for the environment state and maximize the expected return for the environment.

2.1.3. 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 G_t for taking a certain action A_t given the state S_t. This is denoted by Equation

$$q_*(s) = \rho(s,a) + \gamma \max_{a'} \sum_{s'} q_*(s',a')P(s'|s,a) \tag{2.7}$$

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 Reinforcement Learning methods will specify how the agent's policy is changed as a result of its experience. This will change the actions that an agent takes for the same observations and improve the performance of the agent in the environment.

In some cases, the policy works with the value function to choose the best possible actions for the input states to maximize the reward in the environment. This is called the greedy policy where the agent will choose the action that will lead to the greatest expected reward as seen in Equation 2.8.

$$\pi(s) = \arg\max_a q_\pi(s,a) \tag{2.8}$$

2.2. Basic Concepts

The main objective for RL is to train a policy that will maximize the cumulative expected reward by choosing the best-suited action A_t for a given state S_t. The policy will thus take the state as an input and use this to generate an action. The value of the selected action is described by the reward signal received, which is in turn used to update the policy of the agent and optimize the action selections.

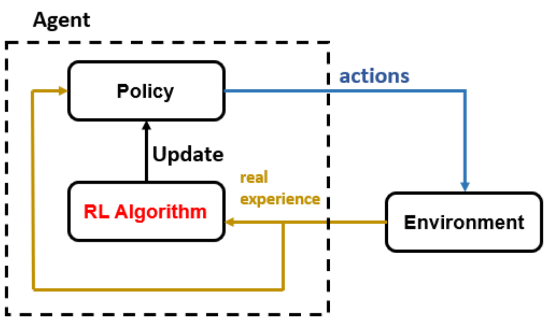
For favorable actions ($R_t > 0$), the updated policy would increase the probability of choosing the same action again in that state. For actions that negatively affect the environment ($R_t < 0$), the updated policy will decrease the probability of choosing the same action for the given state. Through these interactions and updates, the policy will converge towards an optimal policy for the environment which it is controlling.

Some important concepts of RL affect key features of the model performances such as training time, stability, and the accuracy of the trained model.

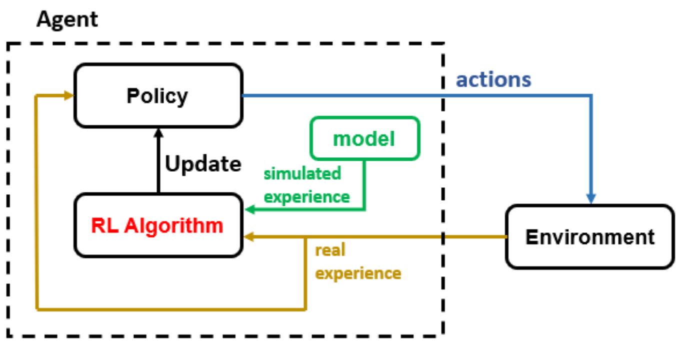
2.2.1. Model-Free and Model-Based Reinforcement Learning

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.

The two models can be seen in Figure 2.3.



(a) Model-Free Control Loop



(b) Model-Based Control Loop

Figure 2.3: Structure of Model-Free vs Model-Based Reinforcement Learning [2]

2.2.2. 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.

Thus, 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 ϵ-Greedy policy chooses the action by sampling a random value Sample $= [0, 1]$ and comparing this to the threshold value. This can be seen in the Algorithm 2.1. [3]

Algorithm 2.1: ϵ-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 $\leftarrow A_t =$ Random Action
**else**
    Return $\leftarrow A_t = \pi_\theta(S_t)$
**end if**
```

2.2.3. Online vs. Offline Learning

The basis of RL is that the agent generates the experience that 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 be made to the current policy and results in increased 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. The use of older data also allows for various collection methods and can be used to implement imitation learning. This method can be dangerous though as large changes to the policy can lead to policy crashes and performance errors.

Offline learning is easier to implement and more generalized than its counterpart. It can be used in a variety of different environments and will always achieve some convergence. Policy crashes can be avoided by engineering the reward function and the hyperparameters of the environment and the model.

2.2.4. 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 pick actions. 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 is used in methods such as PPO and A2C.

Off-Policy methods work differently as the agent in this case can't pick 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 ϵ-Greedy policy, where the agent has no control over the action that will be chosen.

This method will train with exploration and be used to play without an exploration term ($\epsilon = 0$). This method is also usually used as an Offline method and can be trained using recorded data, however, this recorded data might limit the agent's performance since the expert data might be imperfect.

Off-Policy methods include the Q-Learning algorithms and will be used extensively during the project. [6]

2.3. Types of Reinforcement Learning Algorithms

Different methods of RL incorporate different combinations of the basic concepts that were described in the previous section. The RL models have different properties and advantages that make them suitable for different applications and use cases. For example, an Off-Policy algorithm can be applied to general Atari games without major modification and still yield convergence. Whereas an On-Policy algorithm will be used in an engineered case where stability is crucial, such as a self-driving car.

There are many different types of RL, but this report will focus only on methods in the Q-Learning and Policy Gradient Methods classes.

2.3.1. 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, 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 2.9. [5]

$$Q_k(S_t, A_t) \longleftarrow Q_k(S_t, A_t) + \alpha_k(R_{t+1} + \lambda Q_{k+1} - Q_k(S_t, A_t)) \tag{2.9}$$

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 Q-Learning and Deep Q-Learning, the state-action value function aims to approximate the expected cumulative return G_t for taking each action given a certain 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 ϵ-greedy policy to guide the agent towards convergence to find the optimal solution for the environment. This is done by iteratively updating the Q-value for each state until the algorithm converges. In Tabular Q-Learning, convergence can be guaranteed as long as the agent does more exploitation than exploration.

Q-Learning is an Off-Policy method, meaning that experience from all versions of the policy can be used. Thus, this method requires large memory space for experience to be stored and access to train and update the policy. This also means that 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. [7]

2.3.2. Policy Gradient Methods

Policy Gradient 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, Policy Gradient methods differ from Q-Learning in the sense that the network that chooses the action, outputs a probability distribution of the set of available actions, rather than the expected reward for each of these actions.

Policy Gradient Methods use an Actor-Critic structure, where the Actor Network will be the mapping of the state input to the probability distribution of the actions. The critic network will estimate the value of the state that it has reached thereby serving as a reference to the quality of the action that was chosen. In Policy Gradient methods the action is then selected by categorical sampling from this probability distribution output given by the Actor Network. [8]

For example, the policy is described by the Actor Network whose input is a representation of the state, whose output is the action selection probabilities, and whose weights are the policy parameters. For this case, if θ denotes the vector of the policy parameters and ρ denotes the performance of the current policy, then the policy gradient method, the policy parameters are updated proportional to the gradient as seen in equation 2.10:

$$\Delta\theta \approx \alpha\frac{\partial\rho}{\partial\theta} \tag{2.10}$$

where α is a positive-definite step size - or the learning rate. This will ensure that θ will converge to a local minimum if the equation can be achieved. Unlike the value-function approach, here small changes in θ can cause only small changes in the policy and the state-visitation distribution.

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 π_θ will be able to train on is the experience gained from the previous model $\pi_{\theta_{old}}$. [9]

Chapter 3

Reinforcement Learning Methods Development and Testing on Melax's Tetris

3.1. OpenAI Gym Environment ?Q

How do I cite the OpenAI Gym environment structure??? To be able to design and test RL algorithms, a testing environment is required. This environment is where the agent will take action and receive rewards and observations. There are many open-source environments that are available to test RL algorithms, and these are designed by Open AI. 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 will also be used to test the performance of the custom RL models.

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

Reset function

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

Step function

This function will take an action that was given by the agent. This action will then change the environment and return 4 pieces of information to the agent.

1. The *observation* of the current game state S_t will be formatted and returned by the Step function for the agent to use. This includes flattening and preprocessing to Tensor values if required by the agent.

2. 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.

3. In the case of the environment reaching a terminal state, the Step function will set and return a *terminal flag* to indicate to the agent that the reset function needs to be called to continue.

4. An *info* list is also returned and can contain any extra information, such as game scores of 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. The render function significantly reduces the model's performance when used in series with the training of the RL models and therefore needs to be run in parallel. This is done by loading the most recent version of the RL model and running one episode in the environment.

The basic control loop of an RL model interacting with a Gym environment follows the same structure as given in Algorithm 3.2.

Algorithm 3.2: Gym Environment Control Loop

```
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 ???

Should I add: How were the models saved and loaded

Background

Reinforcement Learning follows a trial-and-error approach to learn features from an environment and requires many iterations to find useful information and correct actions for specific states. This makes it difficult to test Reinforcement Learning methods on physical equipment and often requires the simulations to be run on models of an environment. Atari games have been the environment of choice for testing and validating the created RL algorithms as they are safe and easy to operate whilst allowing for precise feedback for developers. s

In this project, the test environment of choice will be Tetris. Originally 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) in a grid environment with the goal of filling an entire row with these blocks, which results in the line being cleared and increasing the score of the player. The game will continue until the placed blocks exceed the playing field height limit, ending the game and giving the player the final score.

Development

For the Tetris game to be useful 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 is developed following the structure given by an example from Timur Bakibayev [10]. The is divided into 3 different classes combined to form the final game of Tetris.

1. Figure class: This class represents the different figures and the important information such as the x, y - positions and rotation.

2. Interaction class: This includes the motion of the pieces, the rotation, and the stacking. In this class, the shapes will be checked for interference and operate the game field as necessary. This is also where the rules and the scoring of the game will be dealt with.

3. Control class: This class will render the game environment and allow the player to choose and interact with the game. The game clock and user interface data will reside in this class.

4. OpenAI Gym class: Once the RL models are ready to be tested, the Gym class must interact with the agents. This class will follow the structure described in Section 3.1.

Game Representation

The game field is represented by a 20×10 matrix of 1's and 0's, representing filled blocks and open spaces, respectively. The current falling piece can be represented by 2's in the matrix to give the player an indication of the current playing piece. The block falls constantly, and a player can make multiple moves in a single downward step. Seven different Tetrominos exist in the Tetris game and will be given as input to the game in random orders and rotations. This makes it difficult to plan as the model does not know what the next pieces will be. The game field and shape of the different Tetrominoes are seen in Figures 3.1 and 3.2, respectively.

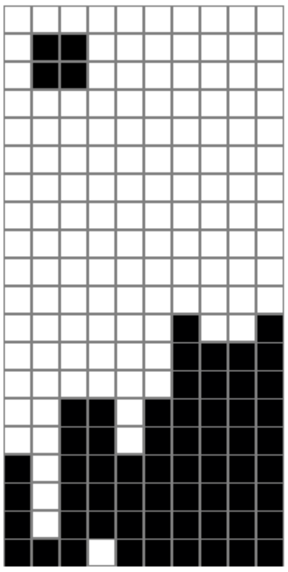


Figure 3.1: Full 20 × 10 Tetris Environment

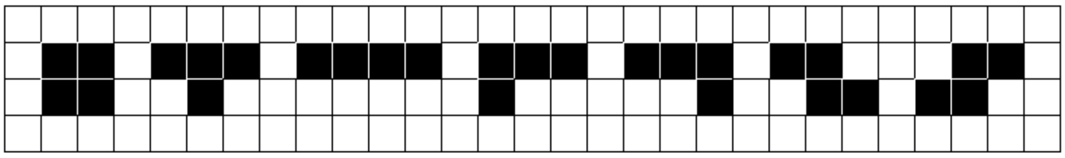


Figure 3.2: Full 20 × 10 Tetris Shapes

A human player would observe the game as the 20 × 10 matrix and make decisions based on this information. This is fed directly to the RL model as an input and the reward of the previous action. 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 will serve as the main reward and can be used in the sparse reward setup; however, the reward function can be used to 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.

Tetris setup for Reinforcement Learning development

This 20×10 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 10×5 matrix to test the different models and compare the results before moving to the 20×10 Tetris. This smaller version of Tetris, as seen in Figure 3.3 can be referred to as Melax's Tetris and was developed for testing AI in the Tetris environment.

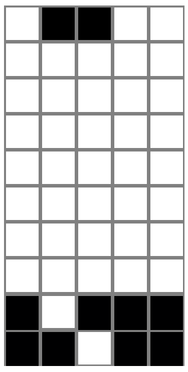


Figure 3.3: Melax 10×5 Tetris Environment

In this environment, the shapes are simpler and the drop area smaller, resulting in better performance in sparse reward environments. There are four blocks that can be played in this environment, which represent the simplified versions of the 20×10 Tetris pieces as seen in Figure 3.4.

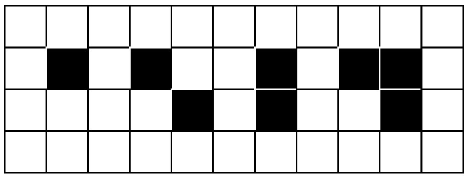


Figure 3.4: Melax 10×5 Tetris Shapes

3.3. Applying Reinforcement Learning to Melax's Tetris

Melax's Tetris, just like any Atari game, can be seen as a Markovian by nature. This is because the next state can be estimated by only knowing the information about the previous state. The transition is controlled by the chosen action for the specific state and the reward for this action. This will provide the agent with feedback to evaluate the chosen actions with the goal of maximizing the reward or the score of the game. After consideration, the algorithms chosen to solve Melax's Tetris environment were Deep Q-learning (DQN) and Proximal Policy Optimization (PPO), which incorporate both types of Model-Free RL.

3.3.1. Melax's Tetris Observation Space and Action Space

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

For the smaller version of Tetris, it is not needed for a Convolutional Neural Network. Thus the only preprocessing done on the matrix is flattening the 10×5 shape to a 50-element array and converting this to a tensor before it is given as an input to the RL algorithm.

Arcade Action Space

The first type of action space consists of a single discrete number between 0 and 4. This will represent the available actions in the environment and allow the agent to choose actions:

0. Rotate the piece
1. Move the piece left
2. Move the piece right
3. Drop the piece into the position
4. No action

This is similar to how a human would interact with the environment. However, a human only uses this to move the piece to the desired location. This is to say, the human player decides the piece's location rather than the series of actions needed to reach this location.

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 block type). Thus, the model output will be converted from the integer value to the x-position and rotation of the shape. The shape will then be moved to these values and placed into position.

The position and rotation of the shape are calculated using the Game Field Width $\mathcal{W}$ and a number of rotations available for the current piece $\mathbb{R}$ to evaluate the current action A_t in range $[0, 19]$. Equation 3.1 describes this process:

$$\text{x\_pos} = A_t \% \mathcal{W} \tag{3.1}$$

$$\text{rot} = \min(\mathbb{R}, \text{floor}(A_t/\mathcal{W})) \tag{3.2}$$

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 block to the same position using the controls as a human would. The effect of this simplification is that the model can learn important moves quicker and train faster with better accuracy and efficiency.

3.3.2. Tetris Reward Function

The reward system for the Tetris environment must be engineered to allow the model to effectively learn the environment. Two types of reward systems were tested with Melax's Tetris environment. A sparse reward system and a shaped reward system.

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.

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.

The best weights for the reward function are shown in the equation 3.3.

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

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 an input to the Reward Function.

3.3.3. Model Validation and Progress Visualization

The verification for the training of the models is done by logging the total episodic rewards, the episode lengths, and the epsilon thresholds 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 ϵ-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. The epsilon threshold value indicates what percentage of steps the agent will take random actions to explore the environment at a certain timestep.

An example of a TensorBoard plot and a good trend can be seen in Figure 3.5 with the yellow line yielding higher episodic returns and a positive trend compared to the pink line, which stays at a relatively constant episodic return. This shows us that the yellow model is worth training, whereas the pink model can be stopped and re-evaluated.

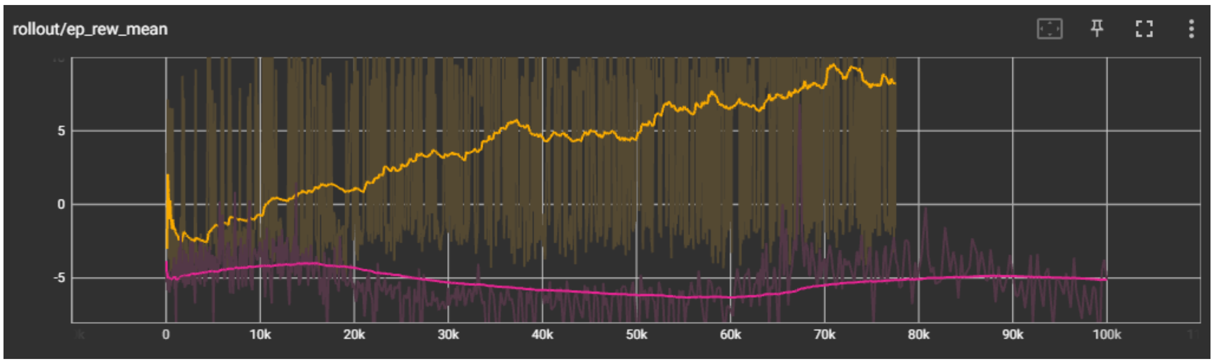


Figure 3.5: TensorBoard Log Example - Episodic Reward

This TensorBoard data can be extracted and saved as a .csv (comma-separated values) 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 Baselines v3 versions of the algorithms. This library contains various RL algorithms that can be used to test custom environments and learn about the algorithms.

The Stable Baseline 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 algorithm 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.

These 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.

3.4. Tabular Q-Learning

The first step to implementing and understanding the Q-Learning algorithm is to start on a very small scale. The first version of the Tetris environment will be a 5×3 playing field with only one piece available. This implementation also only has three actions available left, right, and drop.

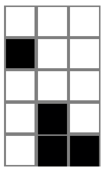
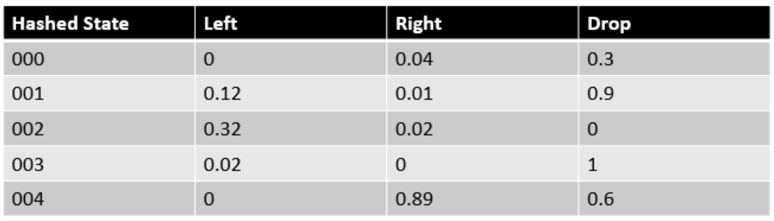


Figure 3.6: Simplest version of Tetris for Tabular Q-Learning

Tabular Q-Learning is the simplest version of Q-learning and works well for small observation spaces and action spaces. The algorithm creates a Q-Table that stores the q-values for all the state-action pairs in the environment. This is done by converting the state observations to hashed states, meaning each unique state will have a unique state key. This state key will be the input to the Q-Table, with the Q-values for each action being the output of the Q-Table. The Q-Table structure is shown in figure 3.7.



Hashed State	Left	Right	Drop
000	0	0.04	0.3
001	0.12	0.01	0.9
002	0.32	0.02	0
003	0.02	0	1
004	0	0.89	0.6

Figure 3.7: Q-Table Structure with example Q-values for the Simplest version of Tetris

The output values for the actions will then be used to select the best action for the given input state. The action will then trigger a reward and a next-state observation. This reward and observation are used to update the Q-value for the action taken. This update occurs following equation 2.9 and will create the values as shown in the Q-Table of Figure 3.7.

The Q-table initiated and compiled with values using a pessimistic approach, setting all the values in the table to $Q(s, a) = 0$. Once the RL algorithm starts, the Q-table is then updated with the values from the rewards for reaching the next state S_{t+1}. This is done through an iterative function that originates from the Bellman Optimality Equation 2.9, where the action-value function $Q(s, a)$ is updated at intervals throughout the training of the RL agent. This will then be an effective mapping of the input observation space to the action space.

The implementation of this method can be seen in algorithm 3.3

Algorithm 3.3: Custom Tabular Q-learning Algorithm

```
Initialize Q-Table with Pessimistic values  $Q(s, a) = 0$
for $e$ in episodes do
    Initialize environment and first observation $s_1$
    Convert $s_1$ into hashed key and create an entry in Q-table if the key does not exist
    while Not Gameover do
        With state $s_t$ and probability $\epsilon$ select random action $a_t$
            or policy action $a_t = \arg\max_a(Q(s, a))$
        Use $a_t$ within environment and generate $r_t$ and $s_{t+1}$
        Convert $s_{t+1}$ into hashed key and create an entry in Q-table if the key does not exist
        Choose next action $a_{t+1}$ with Q-Table and state $S_{t+1}$
        Update the Q-Table entries based on the Bellman Optimality Equation 2.9
            $Q_k(S_t, A_t) \longleftarrow Q_k(s_t, a_t) + \alpha_k(r_{t+1} + \lambda Q_{k+1} - Q_k(s_t, a_t))$
        Anneal probability $\epsilon$
        $s_t \leftarrow s_{t+1}$
        $a_t \leftarrow a_{t+1}$
    end while
end for
```

The result is that the Tabular method converges to the correct solution of the simple Tetris environment and can play without losing a game.

However, increasing the observation space to Melax's Tetris size shows that the Tabular method struggles with the larger input space, and therefore a better approach to estimating the return values for the action is required.

This solution comes in the form of Deep Q-Learning, which uses Neural Networks to estimate the value of each action and can be used in situations with larger action and observation spaces. [7]

3.5. Deep Q-Learning ?Q

A custom DQN model was written and implemented with the help of a notebook by Herman Engelbrecht. This notebook was originally designed to solve the Gridworld problem but was converted and adapted to the Tetris environment. How do I cite Prof Engelbrecth This was done by changing the input space and the action spaces and redoing the preprocessing of the observations.

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. [6] The training loop was designed based on the before-mentioned paper's structure, and the model validation was custom-designed.

This DQN model was first tested on the Cartpole and the Gridworld environment to test the functionality and tune the model's hyperparameters. After the algorithm was confirmed, it was 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 ϵ-greedy policy.

3.5.1. 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 a flattened version of the Tetris game field, relating to an 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 3.8.

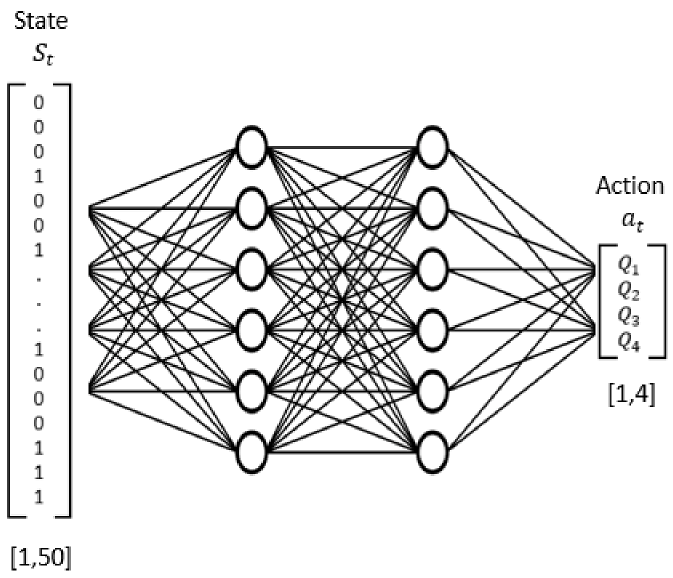


Figure 3.8: Policy Network Structure for the DQN - Note not to scale

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.

3.5.2. 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, 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, and the reward received. This information is pushed to the memory for each step in the environment and will store a default of 50000 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.

3.5.3. ϵ-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 ϵ-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.

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 ϵ-Greedy policy strikes a fine balance by defining a parameter epsilon ($\epsilon << 1$), which then plays a role in the action selection. The epsilon (ϵ) 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 ϵ-Greedy policy and the annealing on the choice between exploration and exploitation can be seen in Figure 3.9.

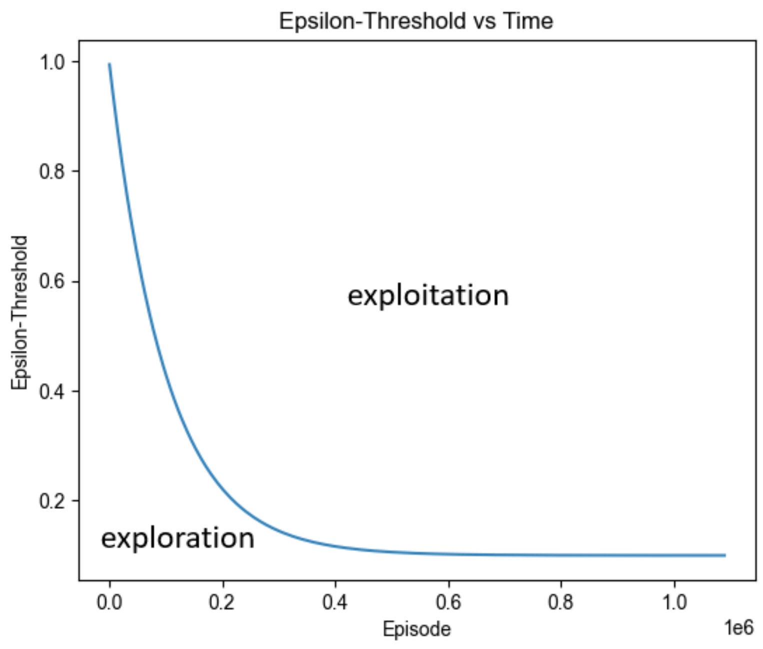


Figure 3.9: Exploration vs Exploitation with Annealing Threshold [3]

3.5.4. Policy and Target Network Optimization

The networks are updated using the loss generated between the Policy Network and 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.

A simplified representation of the Optimization function can be seen in Figure 3.10.

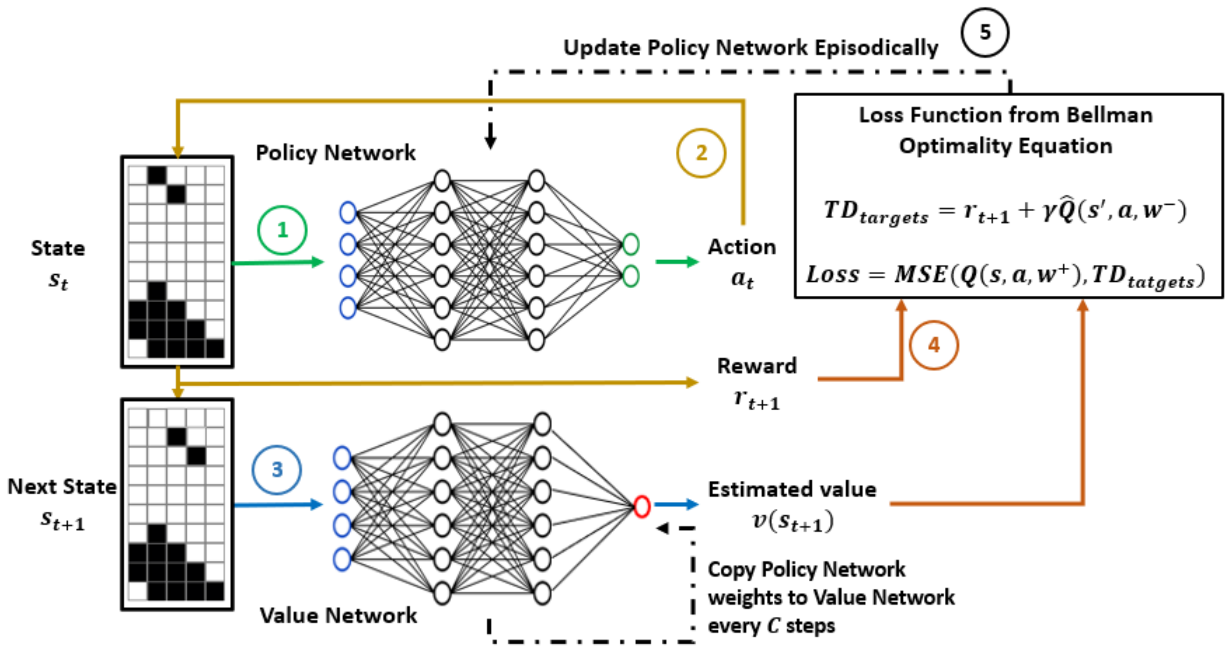

Update Policy Network Episodically
5
Loss Function from Bellman Optimality Equation
$TD_{targets} = r_{t+1} + \gamma\hat{Q}(s', a, w^-)$
$Loss = MSE(Q(s, a, w^+), TD_{tatgets})$
Policy Network
2
1
State
s_t
Action
a_t
Reward
r_{t+1}
4
3
Next State
s_{t+1}
Estimated value
$v(s_{t+1})$
Copy Policy Network weights to Value Network every C steps
Value Network

Figure 3.10: Optimization Function for the Policy Network

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 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 Value Networks and determines the expected return for the next-state S_{t+1}. This is important and explained in the Q-Learning Section 2.3.1 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 2.1.2.

$$\mathrm{TD}_{\mathrm{targets}} = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \tag{3.4}$$

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 α.

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.

3.5.5. 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.

This is 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. This flow diagram for this loop can be seen in Figure 3.11.

In this figure, section 1 refers to the ϵ-greedy policy, section 2 refers to the Replay Memory and section 3 refers to the Policy and Value Networks and the updating thereof.

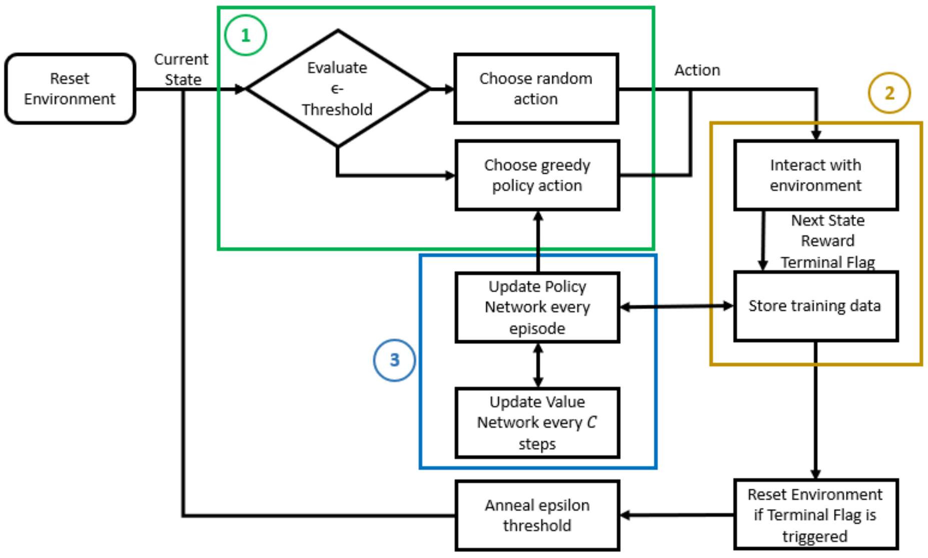

1
2
3
Reset Environment
Current State
Evaluate ϵ- Threshold
Choose random action
Choose greedy policy action
Action
Interact with environment
Next State
Reward
Terminal Flag
Store training data
Update Policy Network every episode
Update Value Network every C steps
Anneal epsilon threshold
Reset Environment if Terminal Flag is triggered

Figure 3.11: Training Loop for the Deep Q-Learning Algorithm

The pseudo-code for the training loop can be seen in algorithm 3.4 following the work from the Mnih paper [6].

3.5.6. Tetris Specific Changes and Results

For the model to interact with the environment and find a solution to 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 20×10 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 1000000 timesteps 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. After the 1000000 timesteps, the final model is used to play Tetris, and the videos of the models can be seen here:

Tetris Videos

Stable Baselines vs 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

Algorithm 3.4: Custom Deep Q-learning Algorithm

```
Initialize *Replay Memory* with capacity $\mathcal{D}$
Initialize *Policy Network* with random weights $w$
Initialize *Value Network* with random weights $w^-$
for $e$ in episodes do
    Initialize environment and first observation $s_1$
    while Not Gameover do
        With state $s_t$ and probability $\epsilon$ select random action $a_t$
            or policy action $a_t = \arg\max_a(Q(s, a)$ from *Policy Network*
        Use $a_t$ within environment and generate $r_t$ and $s_{t+1}$
        Store the transition data $(s_t, a_t, s_{t+1}, r_t)$ in the *Replay Memory*
        Sample the transition data from the *Replay Memory* in batches
        Calculate the Network Loss for the batched transition data
            and update the weights of the *Policy Network* $w$
        Every $C$ steps update the *Value Network* weights $w^-$
        Anneal probability $\epsilon$
        $s_t \leftarrow s_{t+1}$
        $a_t \leftarrow a_{t+1}$
    end while
end for
```

trained for the 1 million steps, and the results can be seen in Figure 3.12.

The custom model performed better than the generalized SB3 model since it has a network structure that caters to the 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.

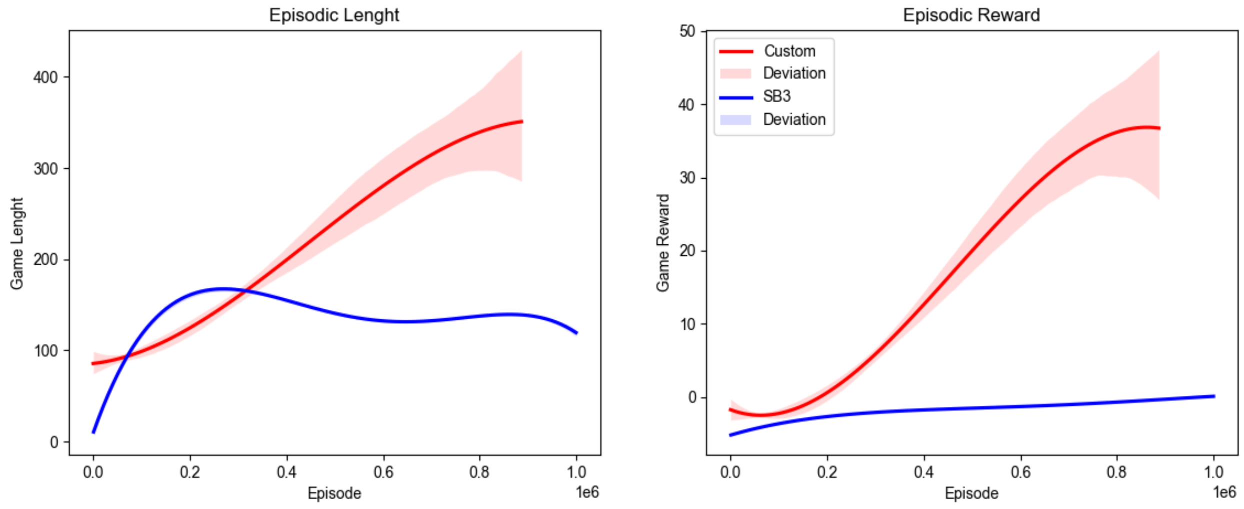


Figure 3.12: Custom DQN model vs Stable Baselines v3 General Model

Sparse Rewards vs Reward Shaping

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 3.3.2 and will be the criterion by which this test is run.

The comparison can be seen in Figure 3.13 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 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 and could outperform the reward-shaped function. However, this is out of the scope of this project and will not be investigated.

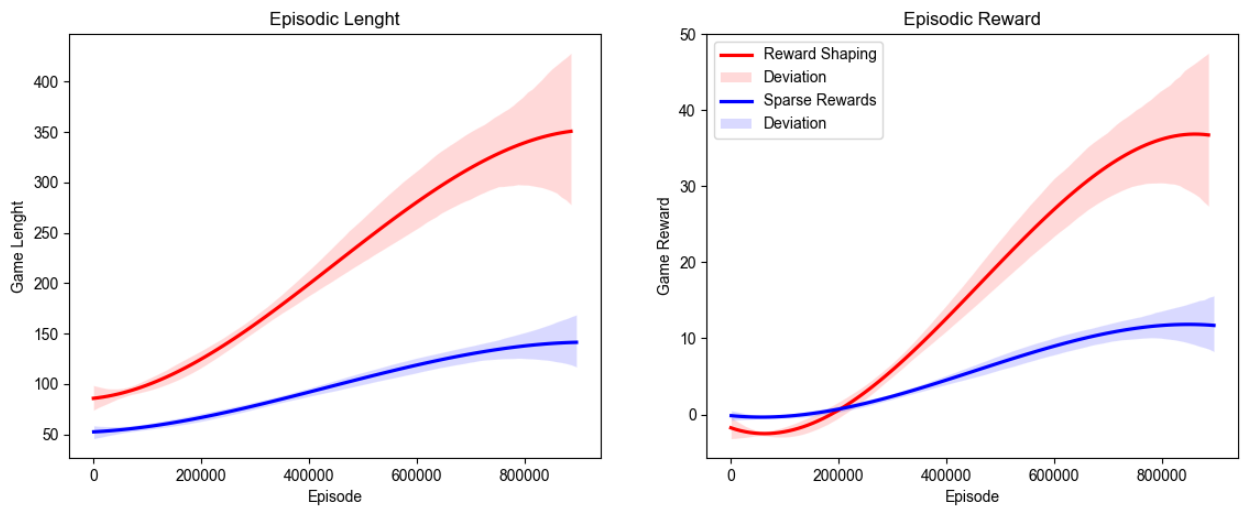


Figure 3.13: Sparse Rewards vs Shaped Rewards System

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 ϵ-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 Tetrominos – 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. The function to determine the best position is given in Figure 3.14.

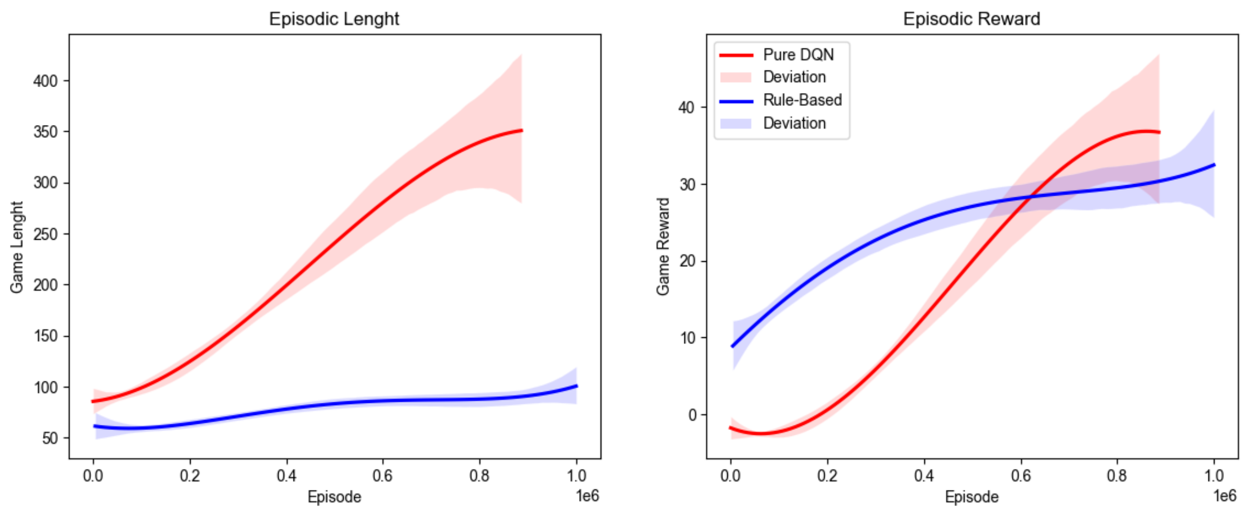


Figure 3.14: Pure DQN vs Rule-Based Imitation Learning

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 600000 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 play the game.

ϵ-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 ϵ-Greedy performs better by finding the best actions and the Greedy policy makes little to no progress in increasing the total reward.

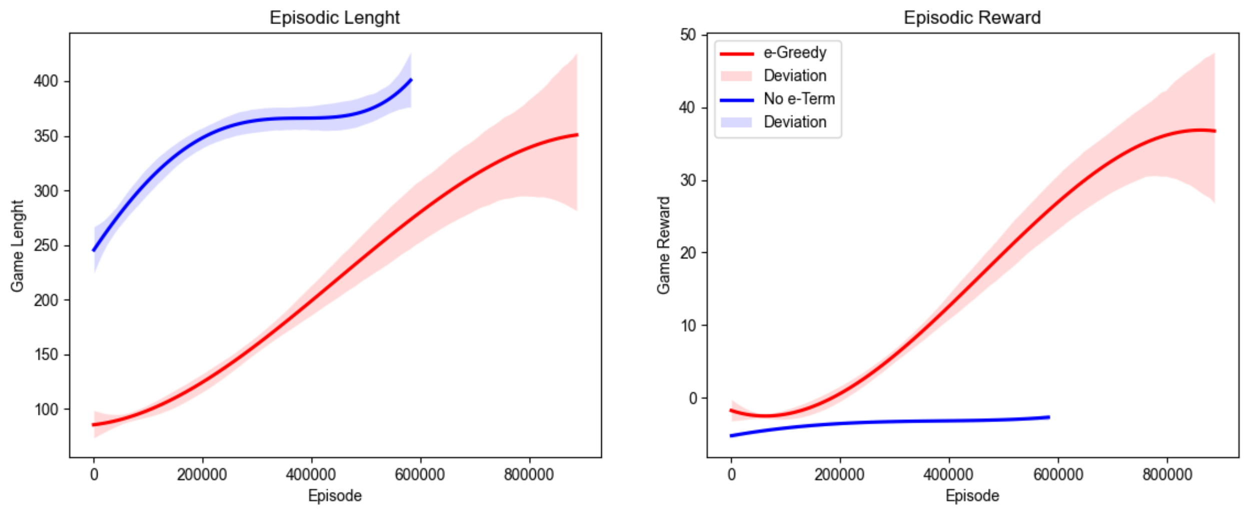


Figure 3.15: Use of ϵ-Greedy vs No Exploration Function

Lastly, the difference between the Direct and Arcade Action space is tested. The results can be seen in Figure 3.16 where the Red line represents the arcade action space, and the Blue line represents the placement action space. The Direct agent learns much quicker than the Arcade counterpart, even though they have the same reward function.

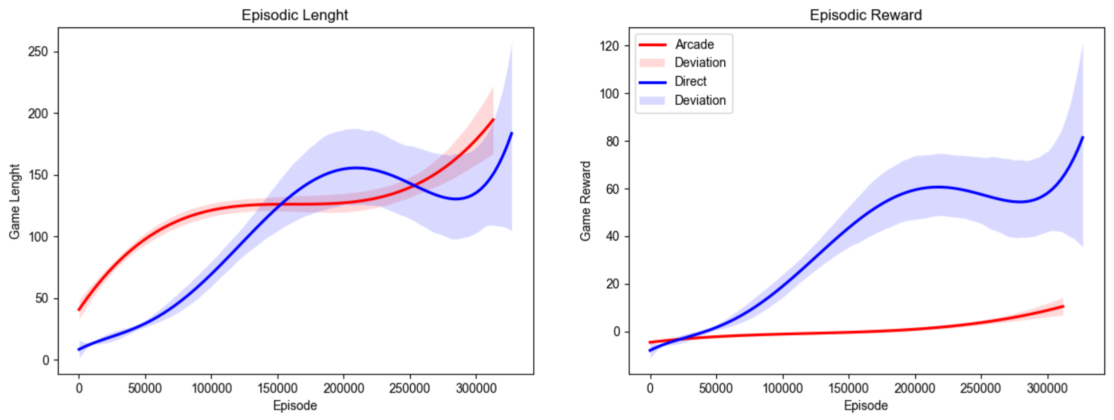


Figure 3.16: Direct Action Space vs Arcade Action Space for Melax's Tetris

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.

3.5.7. Final Deep Q-Learning Results and Initial Conclusion

The final model was trained using the Reward Shaping system with the ϵ-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 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 3.1.

Table 3.1: Hyperparameter Values for the final DQN Model to be used for Melax's Tetris

Parameter	Min	Value	Max
Learning Rate (α)	0.0005	0.001	0.003
Discount Factor (γ)	0.90	0.90	0.99
Start ϵ	0.7	1	1
Stop ϵ	0.01	0.1	0.2
ϵ 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

The Arcade model is trained for 12 million steps and yielded the results as seen in Figure 3.17 when applied to Melax's Tetris.

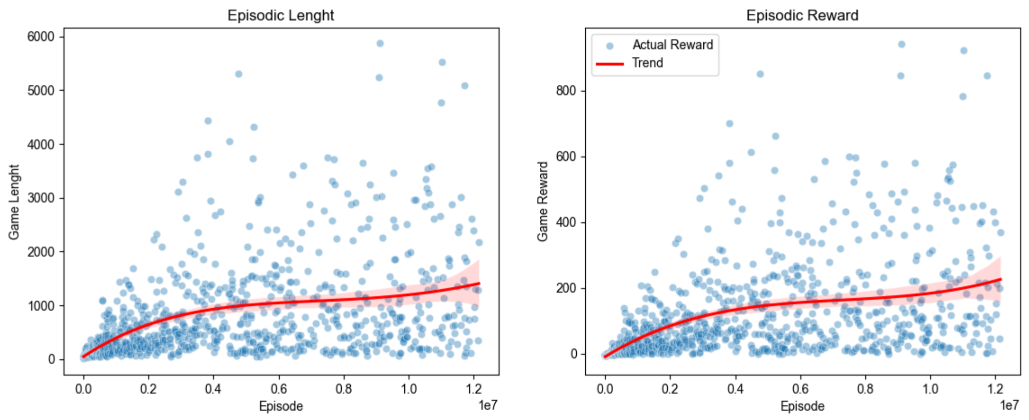


Figure 3.17: Final DQN model Training Graph

The model was then used to play the Tetris game and produced the results as seen in Table 3.2.

Table 3.2: 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 3.18 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.

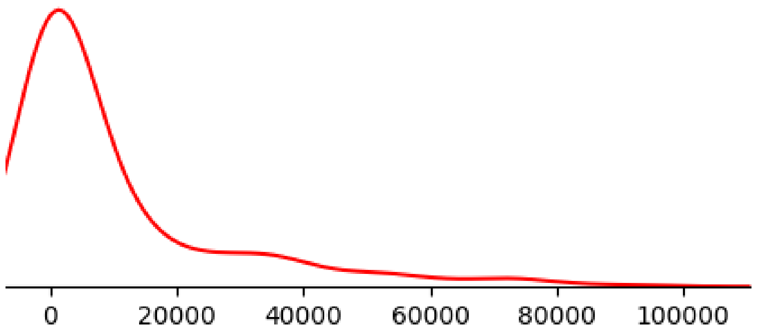


Figure 3.18: Distribution of the DQN model scores

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 had a similar training trend but was only logged for 300000 training steps as seen in figure 3.19 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.

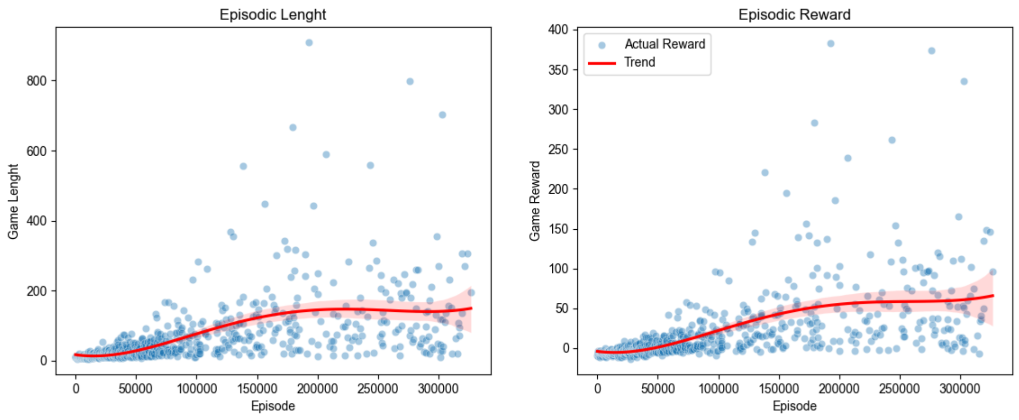


Figure 3.19: Final DQN model Training Graph

The Direct Action space is validated and produced the results from Table 3.3.

Table 3.3: Results produced by the Direct DQN model

Total Validation Games	7
High Score	404561
Average Score	105670

3.6. Proximal Policy Optimization

To cover all bases, a Policy Gradient method is tested on the Tetris environment to compare the capabilities of the Policy Gradient Methods to the Q-Learning methods. Proximal Policy Optimization is the chosen Policy Gradient 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. [9]

The custom implementation of the PPO algorithm includes work from different sources to improve 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 the model, however, the methods are fully understood and explained in the subsections below.

3.6.1. 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 was 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 3.6.5.

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. This is combined with Advantage Normalization and KL-divergence penalties to ensure stability in the algorithm as seen in section ??.

A simplified version of the Training Loop is shown in Figure 3.20

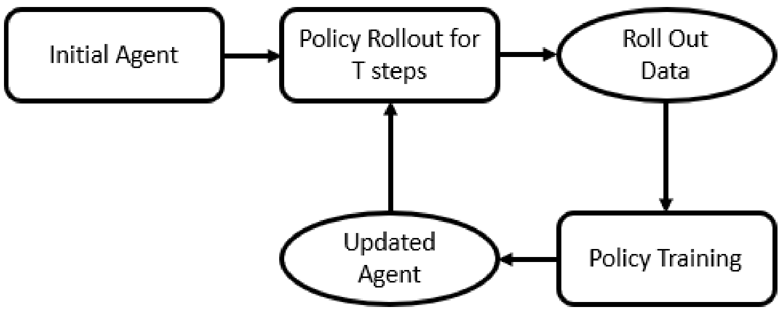


Figure 3.20: Training Loop for the Proximal Policy Optimization Algorithm

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 3.5 and is the building block for the model.

Algorithm 3.5: 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 $\mathbb{A}_1, \mathbb{A}_2, \ldots, \mathbb{A}_t$
    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
```

3.6.2. 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 policy gradient 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.

3.6.3. 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 determines 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.

3.6.4. 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 3.6.6.

3.6.5. Advantage Estimate Calculation

The Advantage Estimate $\hat{\mathbb{A}}$ 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 3.5.

$$\hat{\mathbb{A}} = \sum_{t=0}^{T} \gamma^t R_t - \mathbb{V}^\pi(s) \tag{3.5}$$

The effect of the advantage function on the updates of the network is further discussed in Section 3.6.6.

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.

3.6.6. Proximal Policy Optimization Theory

Developing the PPO model requires a deep understanding of the mathematical concepts and the main surrogate objective.

PPO works on the basic idea of TRPO in that the algorithm 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 easier to implement than TRPO and is an elegant solution to the problem at hand.

The main objective for PPO is described in Equation 3.6.

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}[\min(r_t(\theta)\hat{\mathbb{A}}_t, \text{clip}(1-\epsilon, 1+\epsilon)\hat{\mathbb{A}}_t] \tag{3.6}$$

This formula does the same as the TRPO main objective and 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 $\hat{\mathbb{A}}_t$. 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 $\hat{\mathbb{A}}_t > 0$, 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 $\hat{\mathbb{A}}_t < 0$ 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 3.21.

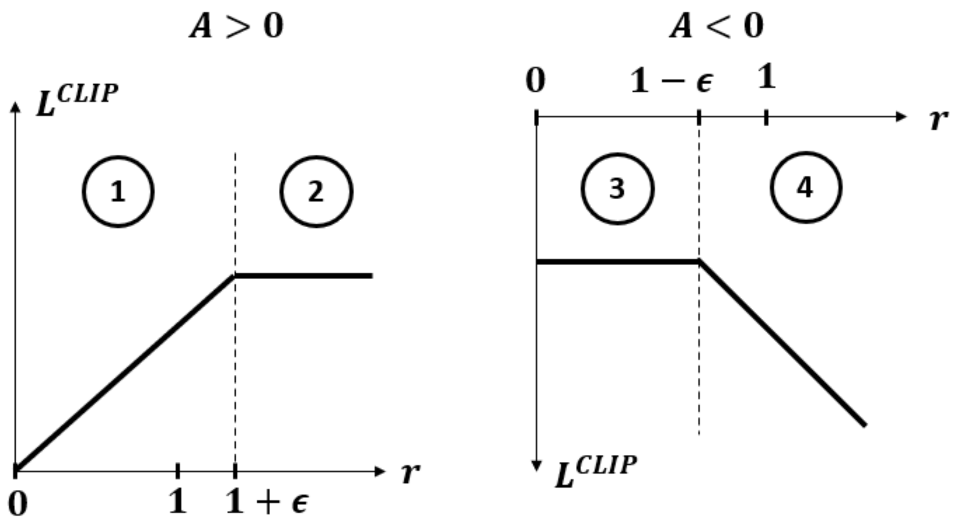


Figure 3.21: Visual Representation of the PPO Main Objective

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 policy gradient 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 $\hat{\mathbb{A}}_t > 0$ graph, if the action was more probable for the current policy compared to the old policy, the Loss would be in Section 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.

In the case where $\hat{\mathbb{A}}_t < 0$, 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 section 3 of the graph. In this case, the policy gradient 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 section 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 3.7

$$L_t^{\text{total}} = \mathbb{E}[L_t^{\text{clip}} - c_1 L_t^{\text{vf}} + c_2 L_t^{\text{entropy}}] \tag{3.7}$$

3.6.7. 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 3.6.6 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 phase 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 largest 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 γ and follows the pseudo-code as written in Algorithm 3.6.

Algorithm 3.6: Caclulating 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. The clipping function is completed by using the clip range value (usually 0.2) and the *Torch.clamp* function to create the effect described by Figure 3.21.

3.6.8. Tetris Tests and Results

The PPO model is applied to Melax's Tetris and trained for 1 million steps to determine the trend and the feasibility of the model. The first test in the environment uses the Arcade Action space to interact with the environment and generate the necessary data for the Network updates. This test compares the Sparse Reward environment with the 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 3.22 which clearly shows that the 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 towards the line clears.

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 3.23 this comparison is shown. The conclusion is that the direct method allows the agent to learn from more important experiences and can understand the environment faster.

A final test is done to test the effect of the learning rate and the total timesteps that the agent will interact with the environment. The learning rate is important as it will

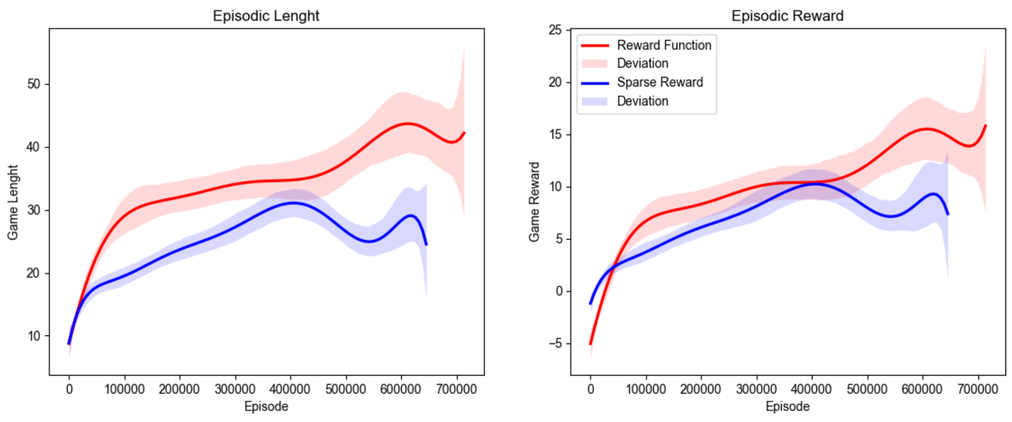


Figure 3.22: Sparse Rewards vs. Reward Shaping for PPO

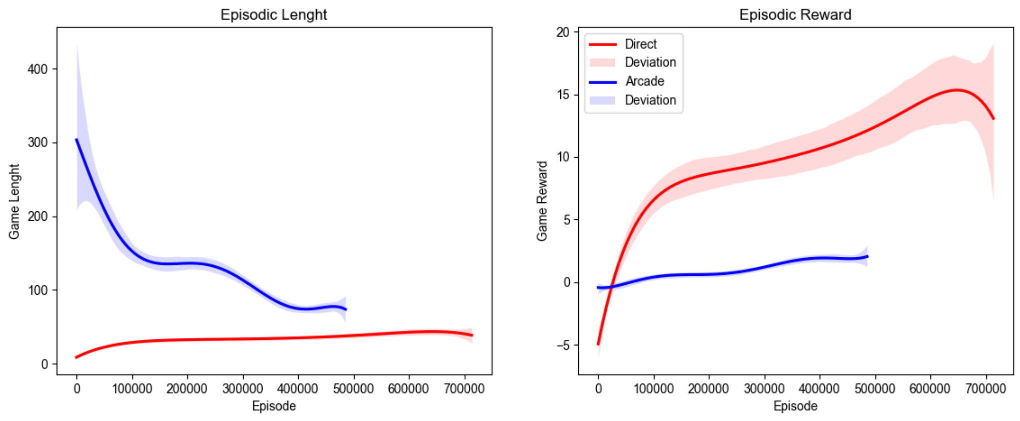


Figure 3.23: Direct vs. Arcade Action space for PPO

contribute immensely to the stability of the PPO model and the amount of timesteps will allow the agent to update more or less often. The results are seen in Figure 3.24.

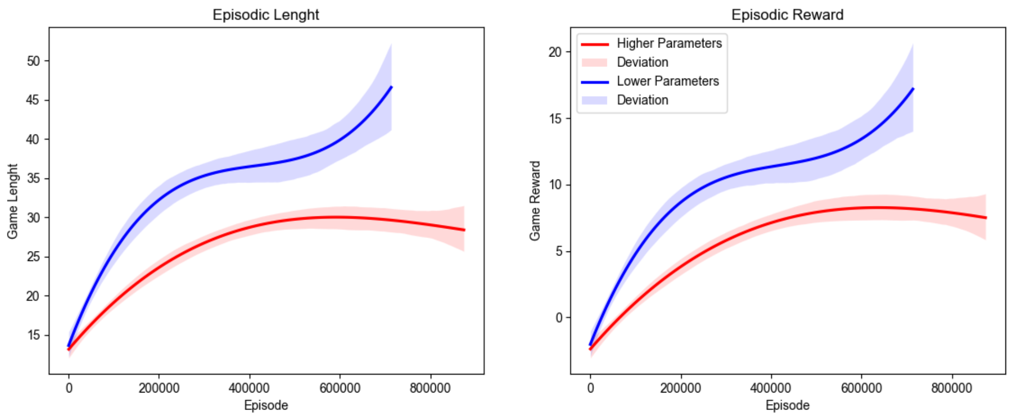


Figure 3.24: Effects of parameter changes to the PPO model

The effect is that a lower learning rate in the range of ≈ 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.

3.6.9. 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 3.4.

Table 3.4: Hyperparameter Values for the final PPO Model to be used for Melax's Tetris

Parameter	Min	Value	Max
Learning Rate (α)	0.0005	0.001	0.003
Discount Factor (γ)	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

The final model is trained for 4.5 million training steps and the final results of this are shown in Figure 3.25.

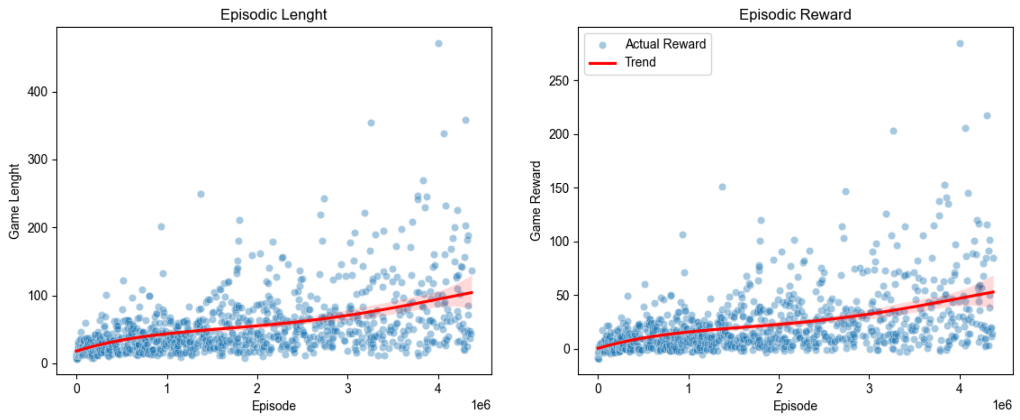


Figure 3.25: Final Training Trend for the PPO model

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 model.

The final model reached the results given in Table 3.5 and the distribution plot in Figure 3.26 which describes the performance and the stability of the model.

Table 3.5: Results produced by the PPO model

Total Validation Games	513
High Score	182
Average Score	42

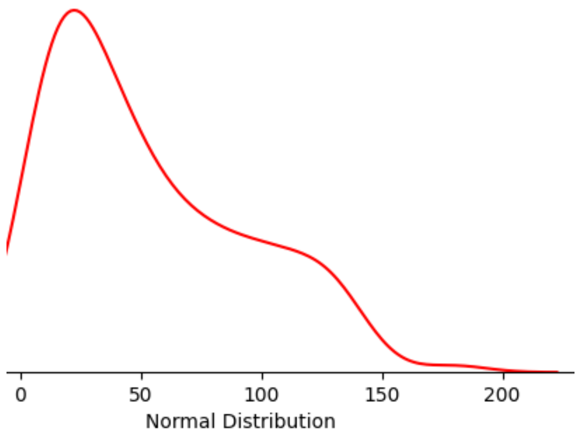


Figure 3.26: Results Distribution for the PPO model

3.7. 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 × 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 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 vectorized environment is a large contributing factor to these results, but this puts emphasis on the next point; that 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. 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 3.27. The clear choice for the completion of the project is to continue with the DQN model for the Full 20×10 Tetris since the performance of the custom DQN model is better than the PPO counterpart.

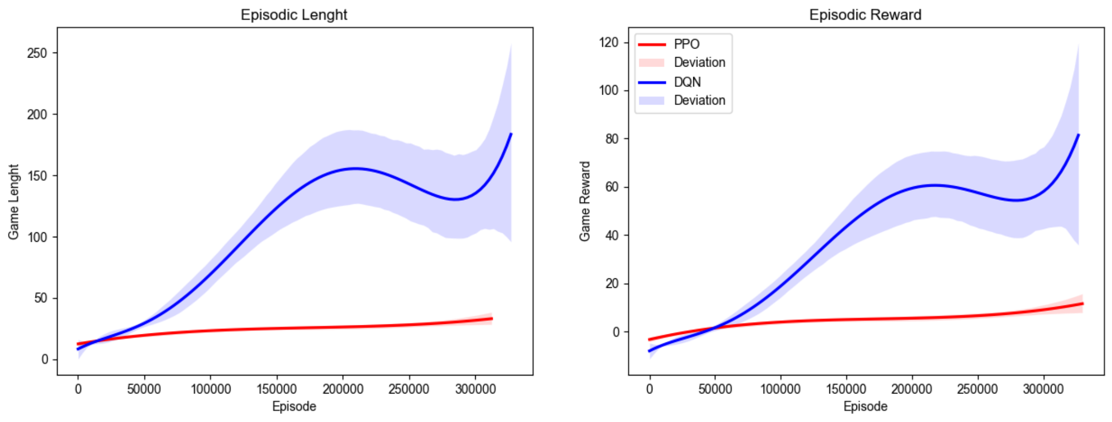


Figure 3.27: PPO vs DQN using the Direct Action space

The results of the gameplay are displayed in table 3.6 to verify the choice of algorithm.

Table 3.6: 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

Chapter 4

Reinforcement Learning Full Tetris

4.1. Full Tetris Challenges

Once the method has been finalized, it can be used to attempt to solve the Full 20×10 Tetris environment. This environment is 4 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 7 different pieces with more rotation options, leading to an increased number of possible states that the agent will have to learn. The use of CNNs could help with this problem, introducing feature extraction and improving the scalability of the network.

Another method will simplify the observation space 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 pieces – compared to 5 pieces – of larger sizer and more rotation options. This, combined with the size of the game field, means that the chances of clearing lines by chance through the ϵ-Greedy policy are very small compared to the Melax counterpart.

The reward function of the environment will 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.

Sample Efficiency

Many actions will be taken in the environment, which will receive little to no reward. Therefore, the Replay Memory of the agent needs to be optimized for the agent to learn information that updates the network weights with the most critical information in the data set. This will be done by introducing the concept of prioritized sweeping and updating the priority of the updates that have the most significant impact on the networks.

4.2. Different approaches for the DQN model for Full Tetris

4.2.1. Convolutional Neural Networks and Frame Stacking

Due to the complexity of Full Tetris vs. Melax's Tetris, a few critical changes needed to be made to improve the performance and accuracy of the model.

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 Convolutional layers to the network and update the size of the hidden layers. As stated in section 4.1, this will introduce the concept of feature extraction to the matrix environment and generalize the input space. An important note is that Convolutional 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's output will be multiple channels, each extracting different features from the model.

Frame stacking is a concept 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 Neural Network 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.

Using frame stacking, the structure of the CNN model can be described in Figure 4.1 and shows how the input states are processed to produce the output actions.

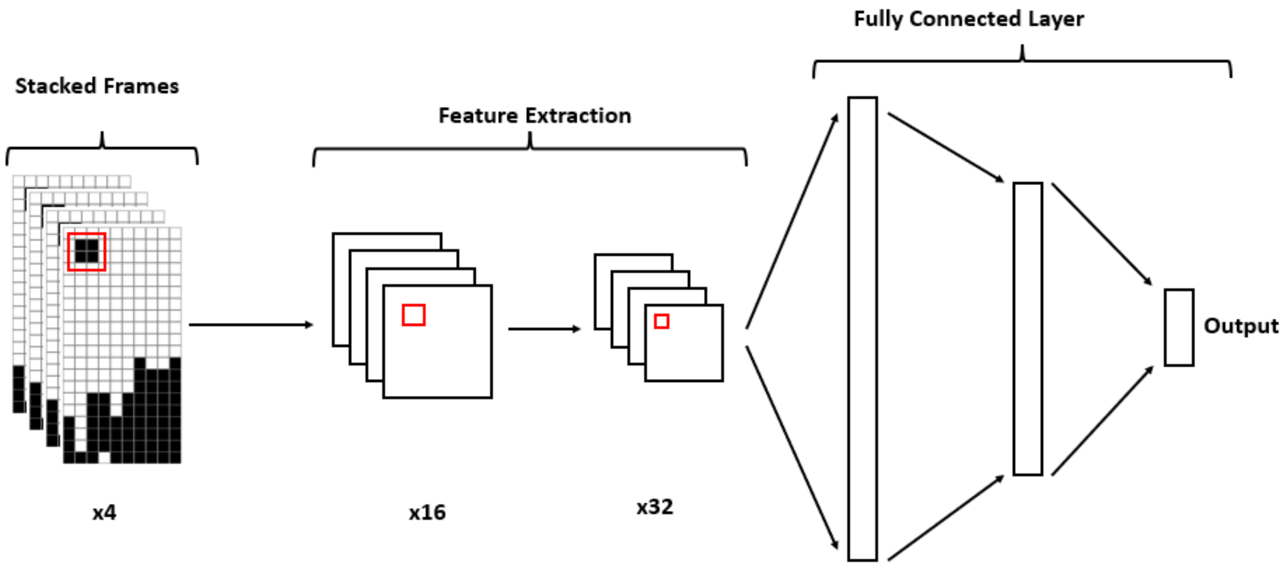


Figure 4.1: General Structure of CNN Setup [4]

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.

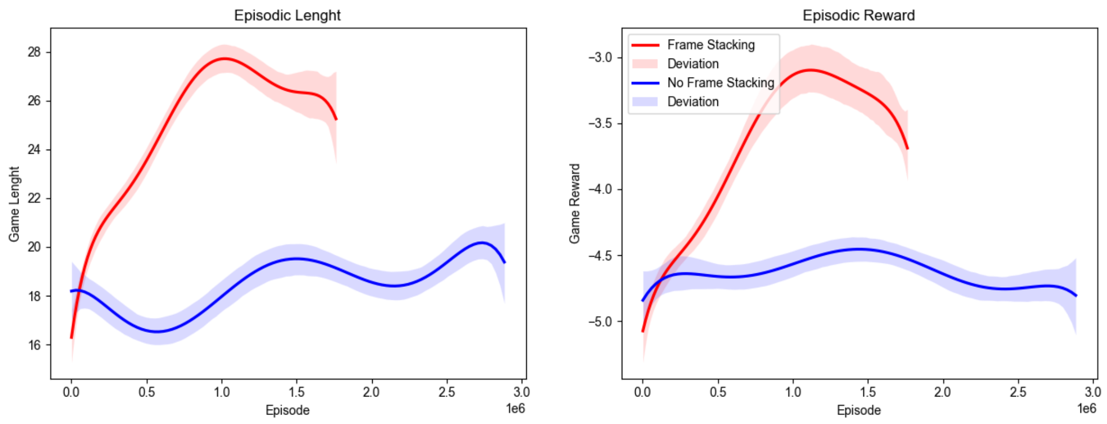


Figure 4.2: Effect of Frame Stacking on the performance of the CNN model

The CNN model is then compared to the Linear model. Figure 4.3 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 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.

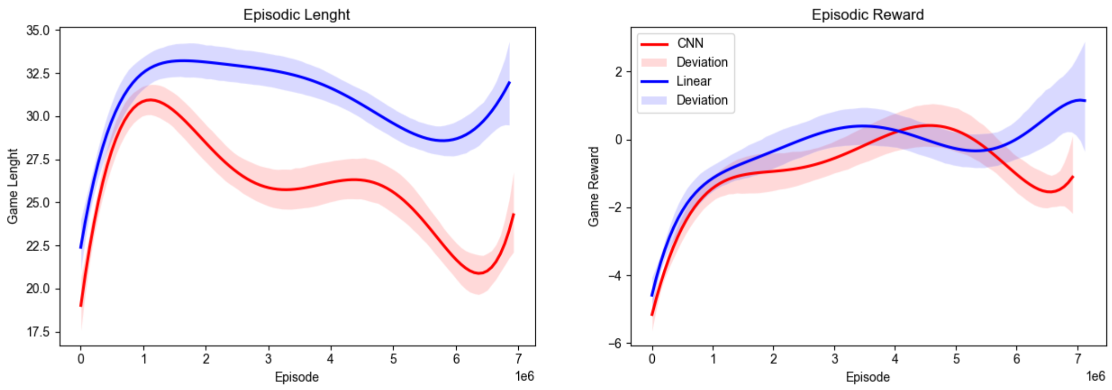


Figure 4.3: CNN Structure vs. Linear Structure

4.2.2. Simplified Observation Space

Noticing that the Linear model outperformed the CNN model, 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 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.

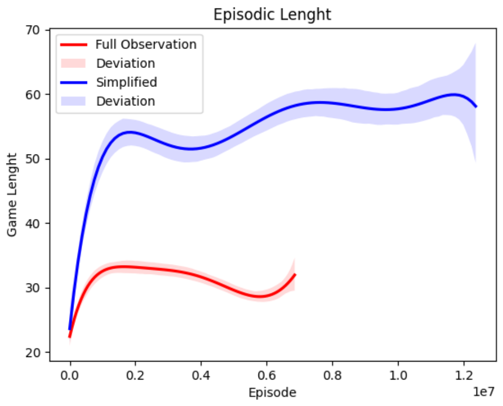


Figure 4.4: Simplified vs. Full Observation space

In Figure 4.4 only the episodic length is compared as the reward function plotting was different for both of the models, but the reward function stayed the same.

4.2.3. Exploration and the Action Space

Exploration is essential as the agent must navigate the environment to find the best actions to yield the most reward. This exploration is challenging with the current action space (rotate, left, right, and drop), as the total area of the game field and the immense number of states create an infinite combination of pieces and actions that can be taken to reach the goal of a clear line.

After many attempts to create a working model with the current action space, the agent often moved the pieces around the playing field and seldom dropped them into the correct locations.

With enough Model-Based finetuning, this would have been solvable. Still, due to time constraints, it was decided that a different approach would be needed to solve the environment and improve the exploration of the placing pieces, which was the most important information.

This is resolved by converting from the Arcade Action space (with 4 different actions) to the Direct Action (with 40 different placements) space. This allows the agent to learn directly from every placement rather than the trivial moves to position the shape into the correct location.

This was proven in Melax's Tetris Section 3.5.6.

4.3. Final Model Structure and Hyperparameters

Network Structure

The final model used to play the Full Scale 20×10 Tetris environment is derived from the previous sections' remarks and will pivot on the results collected. Even though the CNN does show some improvement in the agent's performance, the final model will not be using a CNN as the size of 200 input space is small enough for the network to handle and train faster than the CNN algorithm. This is based on the results from section 4.2.1.

Observation Space

The Observation space of the environment is chosen to be the simplified version as explained in Section 4.2.2. This setup significantly improved the gameplay results and allowed the agent to score higher than the Full 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 3.16.

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 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 4.1.

$$\begin{aligned}\text{Reward Function} = {} & (\text{Previous Height - Current Height}) \\ & + 0.1 \times (\text{Previous Standard Deviation} - \text{Current Standard Deviation}) \\ & - 0.05 \times (\text{Holes Created}) \end{aligned} \tag{4.1}$$

Model Hyperparameters

The final model is trained with the hyperparameters as shown in Table 4.1. 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.

Table 4.1: Hyperparameter Values for the final DQN Model to be used for Full Tetris

Parameter	Min	Value	Max
Learning Rate (α)	0.0005	0.001	0.003
Discount Factor (γ)	0.90	0.90	0.99
Start ϵ	0.7	1	1
Stop ϵ	0.01	0.15	0.2
ϵ 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

4.3.1. Results of the Final Model

The model is trained for 7 million in steps; however, the model plateaus at around 2 million in training steps. Figure 4.5 shows this trend.

The final DQN Model was trained and produced the results from Table 4.2. The model performed well compared to human players, who managed an average high score similar to the model average score. The model beat the best human player by 179 line clears and reached the goal of winning human players in Full Tetris.

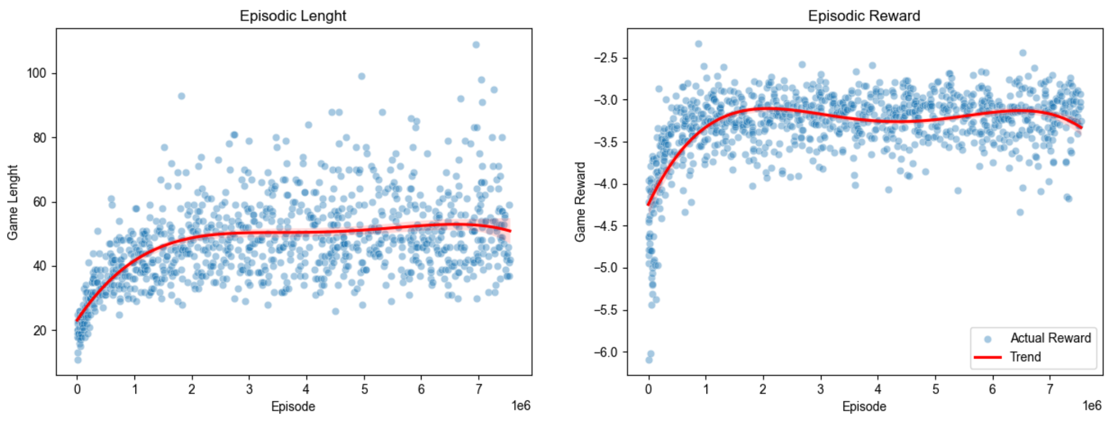


Figure 4.5: Final DQN model for Full Tetris

Table 4.2: Results produced by the Final DQN model

Total Validation Games	500
High Score	280
Average Score	64
Standard Deviation	37

The distribution of the game scores is shown in Figure 4.6, indicating that the model is stable enough for consistent gameplay.

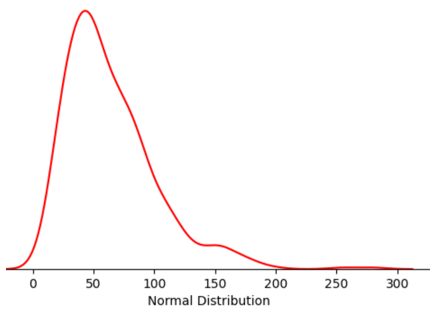


Figure 4.6: Results distribution for Final DQN model in Full Tetris

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 environment and proved that the model could reach and surpass human capabilities at this task.

Chapter 5

Summary and Conclusion

5.1. Project Outline

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 [10] and converted to an OpenAI Gym environment cite for the 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 entire project as different situations required updated reward 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 20×10 Tetris environment, in this case, the DQN model.

Melax's Tetris environment was modified to the Full Tetris environment, and the DQN model was applied. 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.

Applying a Convolutional Neural Network with Frame Stacking to the input space was among these solutions but was less effective than the Linear Networks. Another solution was to use the simplified Observation space and the Direct Action space, which largely improved the performance and accuracy of the model.

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

5.2. Findings and Discussion

This project forces the developer to think about the inner workings of the RL algorithms rather than just the implementation thereof. The following findings for the algorithms are discussed.

Tetris Environment

For both the DQN and the PPO model, it was extremely important to keep the observation space simple. This means there are fewer possible states to learn from, and the model could ”focus” on learning from the important states.

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 agent was not scalable and was required to change the environment and model functions to allow the agent to learn and play the Tetris environment. The Network sizes for the model were kept to a minimum as the smaller Networks outperformed the larger Networks.

Importantly, the DQN algorithm was the easiest to understand and implement correctly. This means it could benefit developers with little knowledge about Reinforcement Learning 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 3.6.5).

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.

Bibliography

[1] 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, p. 689934, 08 2021.

[2] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” *The MIT Press*, 2017.

[3] I. Sajedian, H. Lee, and J. Rho, “Double-deep q-learning to increase the efficiency of metasurface holograms,” *Scientific Reports*, vol. 9, p. 10899, 07 2019.

[4] 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, p. 4500, 10 2019.

[5] 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

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, JoelVeness, 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

[7] 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

[8] Y. Wang and S. Zou, “Policy gradient method for robust reinforcement learning,” 2022.

[9] 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

[10] T. Bakibayev, “How to write tetris in python,” *Medium*, May 2020.

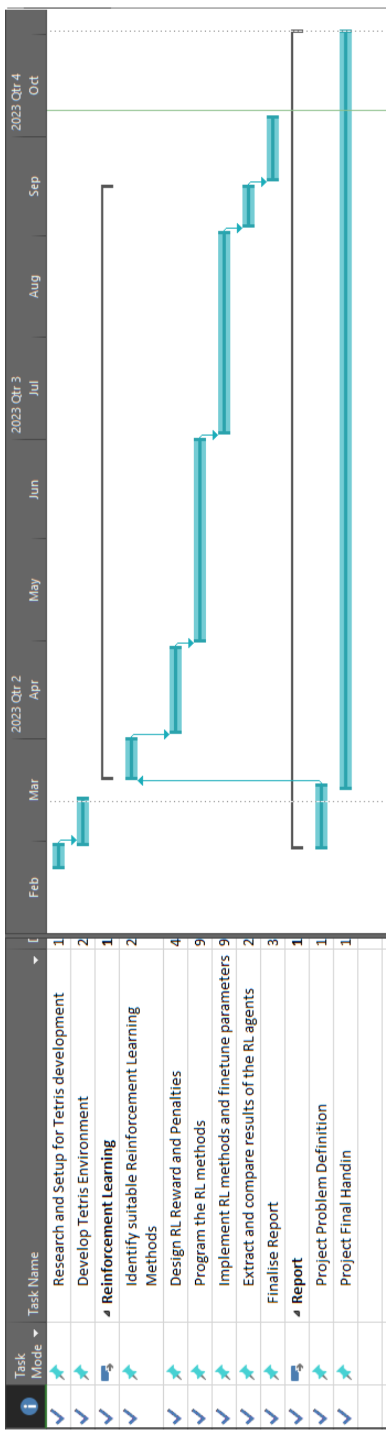
Appendix A

Results ???

Should I put my results here to save space???

Appendix B

Project Planning Schedule ???



	Task Mode	Task Name	[
✓		Research and Setup for Tetris development	1
✓		Develop Tetris Environment	2
✓		**Reinforcement Learning**	1
✓		Identify suitable Reinforcement Learning Methods	2
✓		Design RL Reward and Penalties	4
✓		Program the RL methods	9
✓		Implement RL methods and finetune parameters	9
✓		Extract and compare results of the RL agents	2
✓		Finalise Report	3
✓		**Report**	1
✓		Project Problem Definition	1
✓		Project Final Handin	1

Appendix C

Techno Economic Analysis

C.1. Budget

Table C.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 4 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.

C.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.

C.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.

C.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 reinforcement learning continues to grow and evolve, there is significant potential for commercialization in a wide range of industries.

Appendix D

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 Git.

Safety Report???

Appendix E

Responsible Use of Resources and End-of-Life Strategy

E.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.

E.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.