Learning to play Tetris with Deep Reinforcement Learning

Kyriacos Papayiannis

Master of Science in Data Science
The University of Bath
2020-2021

Learning to play Tetris with Deep Reinforcement Learning

Submitted by: Kyriacos Papayiannis

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

In this dissertation we investigate Deep Reinforcement Learning methods to create an algorithm that would be able to play the game of Tetris. The game of Tetris is a sequential decision-making problem with a large state space, approximately $7x2^{200}$ states can be encountered in this game, that also has the difficulty of sparse rewards but maintains the properties to be considered a Markov Process. Solving a complex environment such as Tetris with Reinforcement Learning (RL) methods, which appear to be well-suited to this kind of problem, could provide significant insights on how these methods can be applied to real life problems with the same properties. In this project we experiment with various elements of the environment such as the state and action representation to understand how they affect the performance of learning and how can we utilise them the most. Also we demonstrate how changes in the parameters of the algorithm can influence the policy that will be developed and how we can influence the learning of the agent by adapting the reward function in what is expected to help the agent the most. Finally we experiment with different variations of DQN agents to explore which one is more effective in this environment and why its methods are superior to the other agents and propose what could be further done to increase the performance of the optimized agent.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Scope . . . 2

2 Literature and Technology Survey **4**
- 2.1 Reinforcement Learning . . . 4
  - 2.1.1 Value – Based methods . . . 5
  - 2.1.2 Function Approximation . . . 7
  - 2.1.3 Policy Based and Gradient methods . . . 7
  - 2.1.4 REINFORCE: Monte Carlo Policy Gradient . . . 8
  - 2.1.5 Model Free vs Model Based Methods . . . 8
- 2.2 Deep Learning . . . 8
- 2.3 Deep Q Networks . . . 10
  - 2.3.1 DQN . . . 10
  - 2.3.2 Double DQN . . . 10
  - 2.3.3 Duelling Q Network . . . 11
  - 2.3.4 Proximal Policy Optimization . . . 11
- 2.4 Performance of various Deep Reinforcement Agents on Atari Environments . 12
  - 2.4.1 DQN . . . 12
  - 2.4.2 Double DQN . . . 13
  - 2.4.3 Duelling Network Architecture . . . 13
  - 2.4.4 Rainbow DQN . . . 14
  - 2.4.5 Proximal Policy Optimization . . . 15

3 Tetris **16**
- 3.1 The game of tetris . . . 16
- 3.2 State Space . . . 17
- 3.3 Previous Attempts to solve Tetris . . . 17

4 Design of the Experiment **19**
- 4.1 Environment . . . 19
- 4.2 State Representation . . . 20
  - 4.2.1 State as a Matrix . . . 20
  - 4.2.2 State as a vector of metrics . . . 21
- 4.3 Action Representation and Selection . . . 22
- 4.4 Reward Functions . . . 23
- 4.5 Design of the Agent . . . 24

4.6 Building the Environment and Smaller Board . . . 26
4.7 Software and Hardware . . . 27

5 Preliminary Experiments and Results 28

6 Main Experiment 34
6.1 Hyperparameter Tuning . . . 34
6.2 Reward Function Experimentation . . . 37
6.2.1 Results in the Small Board . . . 37
6.2.2 Results in the Full Board . . . 39
6.3 Double DQN Agent Experimentation . . . 42
6.4 Duelling DQN Agent Experimentation . . . 44
6.5 Double Duelling DQN Agent Experimentation . . . 46
6.6 Comparison of the Agents . . . 48
6.7 Further Training . . . 50

7 Conclusions 51
7.1 Further Work . . . 52

Bibliography 54

A Dictionary of action correction 58

B Gameplays 59

C Ethics Form 60

List of Figures

2.1 The agent-environment interaction in reinforcement learning. (Sutton Barto,2018, Fig:3.1) . . . 4
2.2 Example of an Artificial Neural Network (Ognjanovski,2020, Image26) . . . 9
2.3 Evaluation of DQN in seven Atari games (Minh et.al.2013, table 1) . . . 13
2.4 Comparison of the estimated vs Real value (Van Hasselt et al., 2015, Figure 3) 13
2.5 Comparison Single with Dual stream Network (Wang et. al 2016, Figure 3) . 14
2.6 Rainbow DQN performance (Hessel et al.,2017, Figure 1) . . . 14
2.7 Games Each Algorithm performed better (Schulman et al., 2017, table2) . . 15

4.1 State Representation as Matrix . . . 20
4.2 State Representation as vector of Metrics . . . 21
4.3 Action Correction . . . 22

5.1 Shape of Transition Tensors . . . 28
5.2 Variation A Learning Curve . . . 30
5.3 Variation B Learning Curve . . . 31
5.4 Variation C Learning Curve . . . 32
5.5 Variation D Learning Curve . . . 32

6.1 Training and Evaluation of Various Learning rates with gamma=0.99 . . . 35
6.2 Training and Evaluation of Various Learning rates with gamma=0.5 . . . 36
6.3 Reward Functions on Small Board . . . 38
6.4 Reward D with different Discount Rates . . . 38
6.5 DQN-Reward A Training on the Full Board . . . 40
6.6 DQN-Reward B Training on the Full Board . . . 40
6.7 DQN-Reward C Training on the Full Board . . . 41
6.8 DQN and Double DQN training on the small Board, and the evaluation of the DDQN . . . 43
6.9 Double DQN -Training on the Full Board . . . 44
6.10 Duelling DQQ, DQN and Double DQN training on the small Board, and the evaluation of the Duelling DQN. . . . 45
6.11 Duelling DQN- Training on the Full Board . . . 46
6.12 All the agents Training on the Small Board, and the evaluation of Double Duelling DQN. . . . 47
6.13 Double Duelling DQN Training on the Full Board. . . . 47
6.14 Learning Curve of all the Agents on the Full Board. . . . 48
6.15 Evaluation Metrics of all the Agents . . . 49
6.16 Duelling DQN further Training . . . 50

A.1 Tetris Pieces with their corresponding piece ids 58

Acknowledgements

I extend my deep gratitude to my supervisor, Dr.Guy McCusker, for his guidance, assistance and support throughout this project. I would also like to thank my technical supervisor, Matt Hewitt, for providing support and advice for the technical part of this project.

Special thanks to my family for their love, support and encouragement during this project and my friends for the happy distractions. Finally, i would like to thank my friend Efi for helping me proof-read this document.

Chapter 1

Introduction

1.1 Background

Most of the decisions the average human is forced to make in their daily life are of sequential nature, where the outcome of the final decision depends on a series of related decisions and where each decision is taken by considering different parameters and trying to predict the near future. For example, while driving a car and deciding to make a turn, you have to make a sequence of successful decisions such as lowering speed, changing line and looking in both directions before the decision to turn is made. Some other sequential decision-making problems are managing a stock portfolio, playing cards and betting.

These sequential decision problems cannot be easily solved by any supervised or unsupervised machine learning algorithm since the algorithms often require prior knowledge to predict the future. This prior knowledge in these types of problems is not obtainable due to the constant change of environment and large state spaces. Thus, the focus of this project turns to the game of Tetris, which has the properties in order to be considered as a sequential decision problem with a very large state space and Reinforcement Learning which can learn by interaction with the environment and no prior knowledge. (Simsek, Algorta and Kothiyal, 2016)

Tetris is an arcade game created in 1984 by Alexey Pajitnov and has become very popular throughout the years. There are currently more than 67 variations of the game. The game of Tetris simulates an empty board which has pieces falling at a random order and the player must place these pieces in order to fill the line and clear it. The game is endless until the player fails to clear the lines which results into a full board. The game of Tetris is very simple to understand and play but yet very difficult to master. This is one of the main reasons why the game became very popular and addictive for many people but it also sparks the interest of many researchers in the field of Artificial Intelligence.

One well established branch of Artificial Intelligence is Reinforcement Learning which is neither a supervised learning paradigm nor an unsupervised paradigm but a third machine learning paradigm. Reinforcement learning is so far, the closest kind of learning to the type of learning structure that humans and other animals do. This is because the reinforcement learning agent interacts with its given environment and upon a choice of action, observes the response, or change that happens in that environment and based on the reward of that interaction learns to maximise that reward when training through trial and error. This branch of AI, and especially Deep Reinforcement Learning is said to have so much potential in the future of AI as it

can deliver decisions by simulating complex environments, testing different techniques and maximizing the reward in order to select the optimal policy. One of the most recent successes of Reinforcement Learning was in 2016 where an algorithm was created called Alpha Go that defeated the world champion at the board game Go. (Silver et al., 2017)

This success has sparked an interest in Deep Reinforcement Learning by many researchers since it indicates the potential of these algorithms to learn to play in even more difficult environments and surpass human performance in areas where previously this was not achievable.

1.2 Scope

This project aims to explore Deep Reinforcement learning methods in the environment of Tetris by implementing various agents, evaluating their performance, and analysing why each agent learns in a different way based on the architecture of their algorithm. Also, this project aims to discover how changes in the representation of the environment can affect the learning rate and final performance of each algorithm, such as different reward functions, state representation and board configurations. Comparison between the performance of the implemented algorithms versus the algorithms discussed in the literature could provide very informative insides to explain the behaviour of each agent.

There are lot of different algorithms that follow similar methods but take slightly different approaches. Achieving better understanding on how each agent performs in the environment of Tetris could be a big step into extending the use of the Deep Q learning agents in environments of real-world problems and classifying which algorithm is best suited for different problems that can occur.

Learning to play the game of Tetris with machine learning has the potential to help the better development of the field of AI in topics such as feature discovery, autonomous learning of action hierarchies and sample-efficient reinforcement learning.(Algorta and Şimşek, 2019)

Following the paper of Mnih et al. (2015), where the first DQN agent was introduced combining Neural Networks, which gave a huge power in the field of Artificial Intelligence, and the already well-established Q-learning methods that are used in the field of Reinforcement learning. This agent was trained in 49 different Atari 2600 games and achieved very impressive scored by surpassing the human performance in 30 of these games without optimising the agent to each individual game. This means that the same exact agent was able to play different games and have impressive performances. This work showed the power of Deep Reinforcement Learning and especially the combined power of Neural Networks and Q – Learning methods which caught the attention of many researchers in the field, to develop more advanced Deep-Q agents and improved the already outstanding performance of the basic DQN agent.

Previous attempts of Artificial Intelligence playing the game of Tetris were very successful by achieving particularly high scores on average without the use of Deep Reinforcement Learning but with methods of handwritten controllers, built and optimised to play the game of Tetris. These attempts focused on finding the perfect weights of different metrics using different methods in order to make the agent understand the environment and by using these metrics make the most promising action. There are not many high achieving agents published learning to play the game of Tetris with the use of Deep Reinforcement Learning and especially Deep-Q. Since Deep – Q agents have huge promise to succeed in even more complex environments than the Atari games, the main focus of this project will be Deep-Q agents and whether it is

possible to achieve high scores with model-free algorithms that can readily be redeployed in other sequential decision problems with larger state spaces.

This project initially describes the fundamentals of Reinforcement Learning, the different methods that contribute to this field, a brief review and introduction to Deep Learning as well as mentions to some well-established agents that use Deep Reinforcement Learning methods. Then, the implementation of the specific environment used in this project is discussed and explored in order to select the state representation and action selection methods to continue with. The specific project focuses on exploring deep reinforcement learning and specifically the implementation of a model-free algorithm that would be able to perform well in this complex environment. While there are more challenges when exploring deep reinforcement learning under the scope of a model-free algorithm instead of a model-based one, the adaptability that these model-free methods have makes it worth discovering their potential. By experimenting with different hyperparameters and observing how each of them alters the final policy and performance of the agent, while using various different Reward functions to influence different game styles for the agent to play the game, this project is able to implement a model free agent that is able to clear 250 lines on average in the Full 20x10 Tetris Board. Improvement to this agent is possible if further tuning and training is performed, and it is theorised that this agent could potentially play this implementation of the game almost perfectly if the optimal parameters of the reward function are found. This work could indicate the potential of model-free agents, that can be deployed in various environments with very few changes in their implementation, such as tailoring the Reward function to fit the purposes of the environment. This can be significant as it demonstrates that it can be successful in real life sequential decision-making problems.

Chapter 2

Literature and Technology Survey

2.1 Reinforcement Learning

According to (Sutton and Barto, 2018) Reinforcement learning describes a technique for solving problems rather than describing all of the information involved in solving the problem from creation to completion. It is known as an agent interacting with its environment where the agent is both the learner and the decision-maker, and the environment is everything outside the agents that holds the information of the problem and responds accordingly to the actions of the agent. The agent stores the information it gains from the interaction with the environment in a value function which hold all the information of the various states that are anticipated so far, in this value function each state-action pair is represented with a value. This value function is modified as the agent receives a reward after each action selection and interaction with the environment. The reward function is declared manually at the start of the problem based on the goals that the agent is aiming to learn and can produce negative or positive values (Carr, 2005).For example, in a problem which aims to finish a task in an optimal time step, it is common that each time step the agent receives a small negative reward due to the consumption of time plus the reward based on the resulting state of the environment.

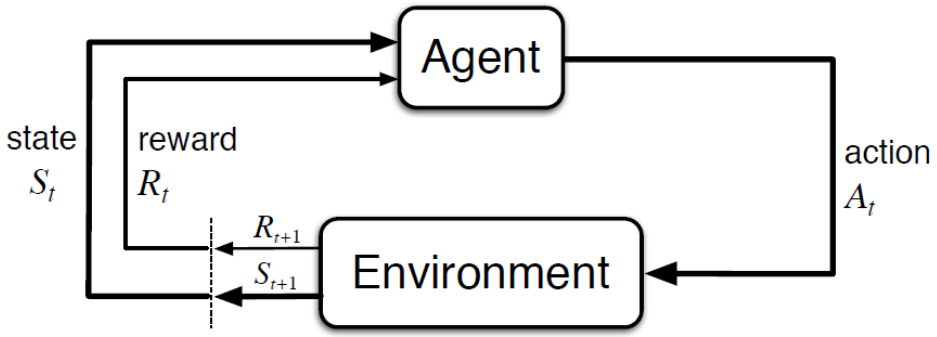

Figure 2.1: The agent-environment interaction in reinforcement learning. (Sutton Barto,2018, Fig:3.1)

The agent interacts with the environment in a sequence of discrete time steps t, starting with t=0, and at each time step the agent receives the current state from the environment S_t, chooses an appropriate action A_t at that current state and in the next time step receives the

corresponding reward of the environment R_{t+1} and the resulting state S_{t+1}.This is described in Figure 2.1. The agent follows a policy that maps states to actions and works in tandem with the value function to determine the agent's action selection. The agent's objective is to maximize the long-term accumulated reward. As the agent interacts with the environment and receives rewards for its actions through trial and error, the agent changes its policy accordingly trying to maximise that reward and ultimately learn to solve the problem in the most efficient way. There are two main categories of methods an agent improves its policy, Value-based methods and Policy-Based methods.

ε-greedy policies

Most of the algorithms follow an ε-greedy policy which aims to balance the exploration-exploitation dilemma. This means that the action selection is dependent on the value of epsilon. Epsilon determines the probability that the agent selects an action randomly instead of utilising what has already learn in order to explore completely new pathways that it may never had the opportunity to see or exploit what it has already learn and improve on that. Epsilon can be a constant or it can start with a high value and decay to a minimum value as the training progresses.

Markov Process

A Markov process is defined as a stochastic process that satisfy the Markov property. The Markov property states that the evolution of the current state (next state) depends only on the current state and not the history of the previous states.(Markov and Nagorny, 1988)

Most of the reinforcement learning problems can be represented as Markov processes even if they do not fully satisfy the Markov Property. (Sutton and Barto, 2018)

In discrete time, the probability of transitioning from a current state S_t with the selected action A_t, to another state S_{t+1} is defined by Sutton and Barto (2018) in the following way:

$$p\left(s', r \mid s, a\right) = Pr\left\{S_{t+1} = s', R_{t+1} \mid S_t = s, A_t = a\right\} \tag{2.1}$$

2.1.1 Value – Based methods

In this method, the agent in order to evaluate the policy at a current state needs to calculate a value that informs him how much the expected future rewards of the that state is. This expectation of future rewards given a policy π and a state s is given by the value function $V_\pi(s)$.

$$V_\pi(s) = \mathrm{E}_\pi\left[G_t \mid S_t = s\right] \tag{2.2}$$

Where Gt is the sum of the discounted Rewards until time t, including a discounting term $\gamma, 0 \leq \gamma \leq 1$.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

Another helpful definition is the action-value function $Q_\pi(s, a)$ which represents the expected return starting from state s and taking the action a following the policy π.

$$q_\pi(s, a) = E_\pi\left[G_t \mid S_t = s, At = a\right] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, At = a\right] \tag{2.4}$$

Finite Markov Decision processes have an optimal policy that we can define in the following way, we can also define q* in terms of v*. (Sutton and Barto, 2018)

$$V_*(s) = \max_\pi V_\pi(s) \quad \textit{for all } s \in S \tag{2.5}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \textit{for all } s \in S \textit{ and } a \in A(s) \tag{2.6}$$

$$q_*(s,a) = E\left[R_{t+1} + \gamma V_*\left(S_{t+1}\right) \mid S_t = s, At = a\right] \tag{2.7}$$

By using these functions to find the expected reward at each state-action pair several tabular algorithms emerge which, with the help of Temporal Difference, solve the Reinforcement Learning problem by finding the optimal policy even in Stochastic environments. The most popular algorithm of those is Q-learning, firstly introduced by Watkins (1989) and extended to prove that Q-learning converges to the optimal policy with probability 1 as long as the action-values are represented discretely, by Watkins and Dayan (1992). The Q- learning agent chooses an action each step by using an ε-greedy policy and update the value of q(s,a) at each step by using the equation (2.8) where α is the learning rate and δ is the temporal different presented in the equation (2.9).

$$q_\pi\left(s_t, a_t\right) = q_\pi\left(s_t, a_t\right) + \alpha * \delta \tag{2.8}$$

$$\delta = R_{t+1} + \gamma \max_{a_t} q_\pi\left(s_{t+1}, a\right) - q_\pi\left(s_t, a_t\right) \tag{2.9}$$

One other similar algorithm is called SARSA (Rummery and Niranjan, 1994), which has similar update equation as it is an on-policy algorithm which estimates the value of the policy being followed instead of Q-learning which is an off-policy algorithm.

LEARNING RATE

The parameter α, learning rate or step-size, is a small positive parameter which can take values from 0 to 1 and it influences the rate of learning or else how much impact on the q-values does each update have. The value of α has huge impact whether a policy converges or not since small updates may never converge to the optimal policy and big updates can make the policy change very drastically each iteration and affect the convergence. This parameter also has a big impact in the updates of the weights in a Deep Neural Network.

Discount Term-GAMMA

The discount term γ, determines the value of the future rewards since the value of the reward received in k steps into the future worth only γ^{k-1} times the original value of the reward. This discount rate makes the agent give value to the rewards of the future. A value of gamma closer to 1 means that the agent plans for future rewards, in comparison to a value of γ=0 which makes the agent only plan for the current reward and future rewards are worth 0. This parameter usually is closer to 1 for sequential decision problems since the environment is dependent on a series of decisions which can impact the future of the environment.

2.1.2 Function Approximation

In the previous section the discussed value-based methods assume small finite spaces where the agent visits each state-action pair multiple times and evaluates the q value until it converges. In most of the tasks we would like to apply reinforcement learning, real-world problems, the environment is infinite with very large numbers of state-action pairs. Thus, such tabular method algorithms will never converge to an optimal policy since even if infinite amount of memory and time is allowed, they will never visit all state-action pairs.
To solve this problem, we combine reinforcement learning methods with pre-existing generalization methods called function approximation, which is already used in other fields of AI.

In order to integrate that, we change the previously stated value function into an approximation value function using weights. This value function could be a linear function with different weights, a neural network with its corresponding weights, a decision tree with the weights specifying the splits or any other function approximation method used in AI. When a new update is made into this function that changes the weights from a state, it affects other similar states more through the generalisation that has been made, thus the learning rate become much faster and can achieve an optimal policy into a reasonable amount of training. Such value function approximation could be represented as $\hat{V}(s, w)$ for the approximation of state s with the given weights w that are a vector.(Sutton and Barto, 2018)

2.1.3 Policy Based and Gradient methods

In comparison with the value-based methods which select actions based on the output of the value function and update their policy accordingly, policy-based methods select an action without considering the value function and they use a parameterized policy. They can still use a value function under the notation (2.10) but it is not used for action selection.

$$\hat{V}(s, w) \tag{2.10}$$

The parameterized policy is represented as a probability that an action a is taken at time t given that the environment is in state s with parameter θ at time t, where $\theta \in R^d$.

$$\pi(\alpha \mid s, \theta) = Pr\{A_t = \mathrm{a} \mid S_t = s, \theta_t = \theta\} \tag{2.11}$$

Policy-based methods can naturally handle continuous action spaces and learn the probabilities for taking the actions, while also learn the correct amount of exploration and exploitation to approach deterministic policies asymptotically. (Sutton and Barto, 2018)

Policy gradient methods make use of the gradient of a performance measurement J(θ) with respect to the policy parameter to learn the policy parameter by seeking to maximize J(θ). So their update equations are in the form of (2.12) where the gradient of $J(\theta_t)$ is the gradient with respect to θ_t of an expectation of the performance and α is the learning rate.

$$\theta_{t+1} = \theta_t + \alpha \, \nabla \mathrm{J}(\theta_t) \tag{2.12}$$

2.1.4 REINFORCE: Monte Carlo Policy Gradient

One algorithm that makes use of the gradient policy is the Monte Carlo Policy Gradient, with the use of the gradient ascent and the policy gradient theorem we can write the gradient of the performance measurement as (2.13) and then the update equation (2.12) becomes (2.14). Then in order to have an update equation that does not involve q and just At we result to the REINFORCE update equation (2.14) as stated in Sutton and Barto (2018) (Chapter 13.3 ,2018).

$$\nabla J(\theta_t) = E_\pi \left[\sum q_\pi(S_t, a) \nabla \pi(a \mid S_t, \theta) \right] \tag{2.13}$$

$$\theta_{t+1} = \theta_t + \alpha \sum \hat{q}(S_t, a, w) \nabla \pi(a \mid S_t, \theta) \tag{2.14}$$

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta)} \tag{2.15}$$

2.1.5 Model Free vs Model Based Methods

The difference between these two categories of methods is mainly determined by the information available upon action selection and policy optimization.

Model-Free algorithms estimate the optimal policy by making use only the information that was captured from their experience with the interaction of the environment. Upon action selection these types of algorithms are unable to make predictions about the next state or reward and base their selection from that information, but the selection happens only from their expected reward using the value function that was calculated from experience.

On the contrary **Model-Based** algorithms make use of transition functions or probabilities in order to predict the future states and rewards and these functions are either learned by the agent from its interaction of the environment or may be computed manually. These algorithms use this information or other information provided by external sources that may have additional knowledge for the specific environment to select the optimal action and estimate the optimal policy. This is risky since if some of the additional information that is provided manually is not 100% correct then the agent may never converge to an optimal policy.

2.2 Deep Learning

Deep learning is a branch of Machine Learning which is mainly constructed by Artificial Neural Networks. Artificial Neural Networks are inspired by the biological neural networks in the brain of humans, and they try to mimic their structure (Marblestone, Wayne and Kording, 2016). Neural Networks are constructed by multiple layers. While the first layer is the Input Layer and the last one is the Output Layer, in between there are the hidden layers, without any constrain on how many hidden layers have to exist in the Network. The word deep in Deep Learning refers to the multiple layers of neurons in the network as an artificial neural network with just one hidden layer that is called a shallow network. Inside each layer there are Neurons that have an input and produce an output that is sent to other neurons, with the help of the connections and the weights. Further to this, activation functions also have a significant role in the Neural Network. The most popular activation functions are ReLu, SoftMax, Tanh and Sigmoid, each of them is best for a specific problem. The Neural Network makes use of a loss function and backpropagation to update its weights and ultimately improve its predictions by minimizing the loss function.

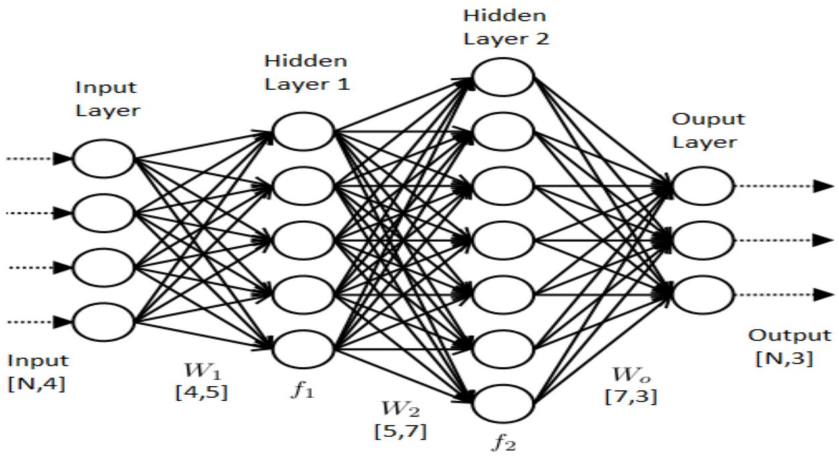

Figure 2.2: Example of an Artificial Neural Network (Ognjanovski,2020, Image26)

Neural Networks require a very small amount of pre-processing of the input data since through the network they automatically interpret the features of the data and can generate new features without any extra training. This is a big advantage of the Neural Networks in machine learning that has a significant impact on image classification. Artificial Neural Networks was first introduced as an idea in 1943 (McCulloch and Pitts, 1943) however, they did not have any significant success due to the lack of technology but as technology was improving this idea of Artificial Neural Networks started shaping up and becoming very powerful.

Deep Reinforcement Learning is a combination of Neural Networks and Reinforcement learning techniques. There are many algorithms that combine the two and the most relevant to our problem will be discussed in section 2.3.

The first big breakthrough of an RL making use of a Neural Network was the TD-Gammon algorithm (Tesauro et al., 1995) which had learned to play the game of backgammon and had an impressive performance, using Neural Networks in combination with Temporal Difference methods issued by (Sutton and Barto, 1987).

Convolutional Neural Network

Most of the algorithms that are mentioned and the agents that is used in this project, have a similar architecture of Neural Network. This architecture is called Convolutional Neural Network (CNN) and it specializes in analysing images and extracting the most important information. This architecture of Neural Network is assembled with a series of convolutional and pooling layers followed by fully connected layers and activation functions. Each convolution layer has a filter of a different or same size ($n \times n$) that slides through the input and creates a feature map, each convolution layer can have multiple feature maps that are stack together and produce the output of the layer. The pooling layers are performed in order to reduce the dimensionality of the output the convolutional layer produced, by reducing the width and height of the feature maps but keeping the depth intact. After these layers, there exist fully connected layers which perform matrix multiplications of the input vectors and weights matrix, fully connected layers expect a 1D input, so before the fully connected layer the output of the convolutional and pooling layer is flattened.(Dertat, 2017)

2.3 Deep Q Networks

2.3.1 DQN

Mnih et al. (2013) created a Deep Q Network (DQN) that was tested on seven Atari 2600 games with no adjustment in the architecture of the DQN algorithm. This algorithm combines the ability of neural networks to learn from raw input data (images/video) with a loss function of a Q learning algorithm. The agent plays the game normally choosing actions based on an ε- greedy strategy while also following a behaviour distribution and collects the transition probabilities and rewards but also saves them in an experience replay variable. Then it samples uniformly at random from that experienced replay, to minimise corelation between the samples performing the update it gives equal importance to all samples, the agent updates its policy by performing stochastic gradient descent to the Loss Function with respect to the weights (2.18).

The Loss function used is a Mean Squared error function $L_i(\theta_i)$ (2.18) by following the behaviour distribution, while y_i (2.17) is the is the target variable of the iteration. While the gradient descent was performed, the previous $L_i(\theta_i^-)$ was statically stored to prevent the algorithm from divergence. (Mnih et al., 2013)

$$L_i\left(\theta_i\right) = E_{\left(s_t, a_t, r_t, s_{t+1}\right)}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right] \tag{2.16}$$

$$y_i = E_{\left(s', \alpha, r\right)}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right)\right)\right] \tag{2.17}$$

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = E_{\left(s, \alpha, r, s'\right)}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right] \tag{2.18}$$

Where θ_i^- denotes the weights of the statically stored model (target network).

2.3.2 Double DQN

Van Hasselt, Guez and Silver (2015) suppose that the DQN agent suffers from overestimations that can lower the performance of the agent in trying to achieve the optimal policy. This happens since the DQN algorithm uses the target value to both select and evaluate an action and this results to overoptimistic value estimates. In their article they prove that these overestimations happen in some of the Atari 2600 games, and they introduce a new algorithm Double DQN which has its base to the original idea of Double Q learning of Van Hasselt (2010).

Double DQN uses the exact same structure as the original DQN agent with the only difference being a changed target value which result to a slightly changed Loss Function.

$$y_i = r_{i+1} + \gamma Q\left(s', argmax_a Q\left(s', a'; \theta_i\right); \theta_i^-\right). \tag{2.19}$$

With this change in the target update, the double DQN aims to reduce overestimations since the target value is now decomposed into action selection and action evaluation since the evaluation of the action is computed by the static target network and the action selection from

the online network. These two networks are the same as the target network is updated with the weights of the online network after some iterations but at the time of the evaluation the target network and the online network have different weight values. When tested Double DQN performed better in most of the games in comparison with standard DQN, but it also showed that in some environments overestimation did not affect the agent from learning the optimal policy. (Van Hasselt, Guez and Silver, 2015)

2.3.3 Duelling Q Network

Wang et al. (2016) improved the DQN algorithm of Mnih et al. (2015) by changing the convolutional network of the model. It was changed by separating the value function into two functions, the value function and the advantage function, this kind of idea was originally presented by Baird (1995). This change in the architecture was made so the model could learn more easily which states are important without considering the effect of the action since there are many states in the environment that are not affected by the action. This algorithm also maintains a lot of other attributes presented by the DQN agent such as experienced replay buffer and freezing the parameters of a secondary network for a certain number of iterations before performing updates to the primary network. The convolution network as DQN outputs Q values that are the estimated values for each action on the current state, in this duelling architecture the network outputs the Q values that are represented as (2.20) which is the aggregation of the value function and advantage function. The Duelling Q Network Agent outperformed the DQN while it was also tested in more games.(Wang et al., 2016)

$$Q_{\pi}(s,a) = V_{\pi}(s) + A_{\pi}(s,a) \tag{2.20}$$

2.3.4 Proximal Policy Optimization

Proximal Policy Optimization algorithm (PPO) is an algorithm introduced by Schulman et al. (2017) which makes use of policy gradient methods while also combining significant information from Trust Region Policy Optimization (TRPO) (Schulman et al., 2015). The aim of the team that created this PPO was to develop an algorithm that performs as good as TRPO or better, to be more easily comprehensive, to be sample efficient and easier to tune its hyperparameters. PPO makes use of the policy gradient (2.21) but slightly modified using an alternative method of TRPO 'Surrogate' objective.

$$L^{PG}(\theta) = \widehat{E}_t \left[\log \pi_\theta \left(a_t \mid s_t \right) \widehat{\mathrm{A}}_\mathrm{t} \right] \tag{2.21}$$

Where $log\pi_\theta \left(a_t \mid s_t \right)$ is the logarithmic probability of the policy and $\hat{A}_t$ is the advantage function which estimates the relative value of the selected action. Instead of the logarithmic probability, PPO uses a clip function on the ratio of the probabilities of the new policy over the probabilities of the old policy (2.23).

$$L^{CPI}\left(\theta\right) = \hat{E}_t \left[r_t \left(\theta\right) \hat{A}_t \right] \quad , where \quad r_t \left(\theta\right) = \frac{\pi_\theta \left(a_t \mid s_t \right)}{\pi_{\theta_{old}} \left(a_t \mid s_t \right)} \tag{2.22}$$

$$L^{CLIP}\left(\theta\right) = \hat{E}_t \left[min(r_t \left(\theta\right) \hat{A}_t, clip(r_t \left(\theta\right), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right] \tag{2.23}$$

Where ε is a hyperparameter usually close to 0,2. The clip function ensures that the algorithm does not change its value by a lot and stays only in the interval $[1-\varepsilon, 1+\varepsilon]$ for each update.

The minimum function is included to ensure that the update takes only the worst update of the two, as a result the policy does not change significantly by one random run.

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{2.24}$$

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{targ}\right)^2 \tag{2.25}$$

The finalised equation used from PPO algorithm to update its policy is (2.24), where c_1, c_2 are hyperparameters, the second term is a squared error loss (2.25) and the third term is an entropy bonus. (Schulman et al., 2017)

2.4 Performance of various Deep Reinforcement Agents on Atari Environments

After the great success of the DQN Agent at playing Atari games by Mnih et al. (2013) many more different variations and improvements of these agents have been made.

- Double DQN (Van Hasselt, Guez and Silver, 2015) tried to solve the overestimation problem of the standard DQN agent.
- Prioritized experience replay (Schaul et al., 2016) optimised the replay buffer sampling in order to achieve faster and more robust learning.
- Duelling Network Architecture (Wang et al., 2016) that split the value function to achieve better action selection in states that are not affected by actions.
- A3C (Mnih et al., 2016) which makes use of the Actor-critic methods.
- Distributional Q-learning (Bellemare et al., 2017)
- Noisy DQN (Fortunato et al., 2019)

These algorithms are also combined in various instances with one another such as Prioritized Double DQN with a Duelling network architecture, creating many combinations of algorithms. Each of those combinations suits a specific problem in the best possible way based on the structure of the problem's environment.

2.4.1 DQN

As discussed in section 2.3.1, Mnih et al. (2013) introduced the first deep reinforcement learning agent combined with Q learning methods. This method aimed to utilise the ability of deep Neural Networks to learn from raw inputs as images with the reinforcement learning methods which learn from interaction with their environment. This algorithm was tested without a change in its architecture on seven popular Atari games (Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders) and outperformed the linear policies SARSA and Contingency however, the DQN agent surpassed expert human performance in only 3 of the 7 games. The evaluation metrics of that research paper can be seen in Figure 2.3. Mnih et al. (2015) re-tested this algorithm in 49 Atari games where results from other algorithms were also available and the DQN algorithm outperformed the other algorithms in almost all the other games and surpassed the average human performance in more than the half games.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690

Figure 2.3: Evaluation of DQN in seven Atari games (Minh et.al.2013, table 1)

2.4.2 Double DQN

As discussed in section 2.3.2, the improvements made in the original DQN was in order to reduce the overestimation problems that was happening due to the structure of the target value. In the Figure 2.4 we can see this overestimation happening in the DQN and how it is improved in the Double DQN agent. From 2.4 it is clear that in the specific environment

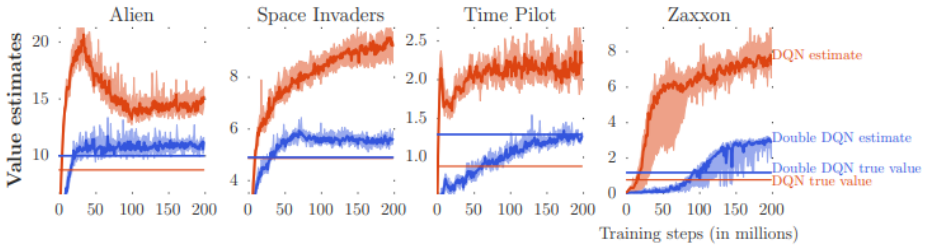

Figure 2.4: Comparison of the estimated vs Real value (Van Hasselt et al., 2015, Figure 3)

the changes in the algorithm of Double DQN helped the agent reduce the overestimations and be closer to the true value which can result in converging to the optimal policy much faster than the original DQN. Nevertheless, improving the overoptimism of the DQN agent does not always affect the final quality of the policy but it stabilizes the learning of the agent. (Van Hasselt, Guez and Silver, 2015)

In the paper where we can see the performance of the Double DQN in comparison with the standard DQN tested in 57 Atari 2600 games, we observe that the Double DQN agent outperforms in terms of score the DQN agent in most of the 57 games.

2.4.3 Duelling Network Architecture

Duelling Network Q learning introduced by Wang et al. (2016) and as mentioned in section 2.3.3 is an alternate version of the DQN agent. The experiments showed that the Duelling Network converged at the same rate as the single network for environments with less than 5 actions but as the number of actions was increased, a much faster convergence rate was observed from the Duelling Network Figure 2.5.

Furthermore, the duelling network was tested in 57 games against a single network and performed better than the single network in 43 out of 57 games. In the games with 18 actions the duelling network performed better 80% of the time which confirms the above statement that as the actions increase, the performance of the duelling network is better. Finally, this algorithm has achieved better performance than the average human player in 42 of those 57 games. (Wang et al., 2016)

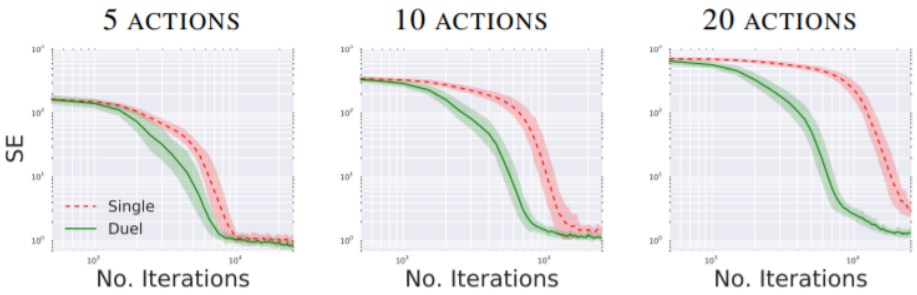

Figure 2.5: Comparison Single with Dual stream Network (Wang et. al 2016, Figure 3)

2.4.4 Rainbow DQN

As mentioned above there are many different variations of the DQN agent, each utilising a specific method to improve the DQN agent. This Rainbow DQN (Hessel et al., 2017) combines all of the following methods with the original DQN agent to produce a new optimised algorithm, Double Q-learning, Prioritized replay, Duelling Networks, Multi-step learning, Distributional RL and Noisy Nets. This integrated agent was evaluated on the 57 Atari 2600 games in the same procedures as the original DQN agent and compared with all the other agents that had evaluations in those 57 games.

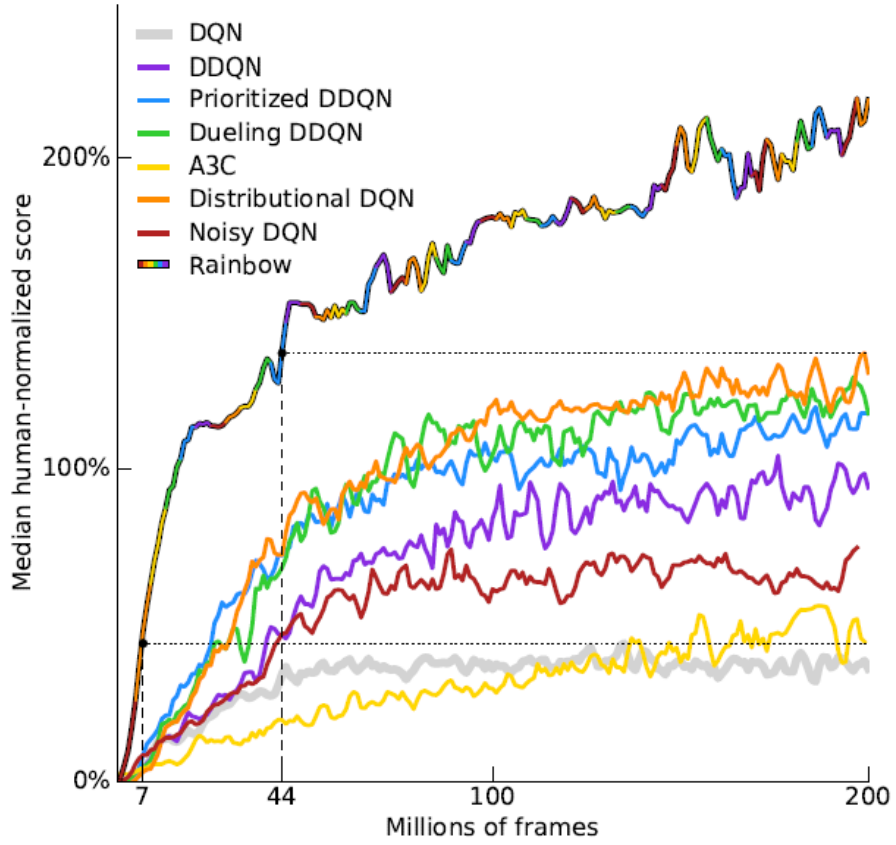

Figure 2.6: Rainbow DQN performance (Hessel et al.,2017, Figure 1)

Figure 2.6 displays the averaged performance on the 57 Atari games, the y axis corresponds to the score of an average human play. From this graph we can observe that Rainbow DQN surpasses the average performance of all the other agents and achieves their convergence state much faster. The hyperparameter tuning of this algorithm was not feasible to try each possible combination due to the large combinatorial space of hyperparameters and thus the creators of this algorithm first chose the optimal hyperparameters of each algorithm as stated in their original papers and then tuned accordingly manually to achieve the presented performance.

2.4.5 Proximal Policy Optimization

As mentioned in section 2.3.4, PPO is an algorithm introduced by Schulman et al. (2017). In their paper PPO was evaluated and compared with other similar algorithms of policy methods like A2C (Mnih et al., 2016) and ACER (Wang et al., 2017) both of which use actor critic methods. Moreover, all these three algorithms optimised their hyperparameters to perform well in the 49 Atari games and are presented in the following table using two score metrics. The first represents the number of games that each algorithm had the highest performance averaged in all training episodes and the second is the number of games that each algorithm had the highest performance in the last 100 episodes.

	A2C	ACER	PPO	Tie
(1) avg. episode reward over all of training	1	18	**30**	0
(2) avg. episode reward over last 100 episodes	1	**28**	19	1

Figure 2.7: Games Each Algorithm performed better (Schulman et al., 2017, table2)

The PPO algorithm achieved better results than the ACER in the metric that averaged the reward over all training steps which indicates that PPO is a faster learning algorithm than the other two. Despite this, in the final performance the ACER algorithm surpassed the performance of PPO in most of the games.

Chapter 3

Tetris

3.1 The game of tetris

Tetris is a popular arcade video game developed in 1984 by a Russian puzzle-loving software engineer named Alexey Pajitnov while working for the Soviet Academy of Sciences' Dorodnitsyn Computing Centre, a government-run research and development centre in Moscow. He invented the game of Tetris without the intention to profit from it but rather for a hobby, while he was inspired by a puzzle game called "pentominoes". (Weisberger, 2016)

The game of Tetris is played on a two-dimensional grid, in most cases 20x10, initially empty when objects composed by 4 1x1 squares in 7 different orientations, called Tetromino, fall from the top of the grid one by one at random sequences. The pieces are, ''I", ''O", ''T", ''J", ''L", ''S" and ''Z" and they are named after their corresponding shape. The Tetrominoes ''I", ''O", ''T" have reflectional symmetry, where the Tetrominoes ''J", ''L" are reflections of each other as well as ''S" and ''Z".

The player is able to control the Tetromino horizontally and also rotate it 90, 180 or 270 degrees without a limit until the Tetromino touches the end of the grid or another already placed Tetromino. When a row of the grid is full of tiles, then the line is cleared and the tiles filled on top of that row move down to get the place of the cleared row. The goal of the game is to clear as many lines as you can before the grid becomes full. Every time the player clears 10 lines, the difficulty of the game increases by one level until it reaches the maximum level 10 after clearing 90 lines. As the level increases the falling speed of the Tetromino quickens making it more and more difficult for human players to decide where to place the falling Tetromino. The Scoring of the game and the levels of difficulty can be different in each variation of the game, but the main two scoring systems is either one point given per line cleared, or more points given when clearing multiple lines simultaneously. (Tetris Wiki, n.d.)

According to Official Guinness Records, the most lines cleared were 4,988, and the highest possible score of 9,999,999, was achieved by Harry Hong, an American gamer playing "Tetris DX" in a competition on 13th of September 2007.

3.2 State Space

Despite the fact that the rules of Tetris are simple, and everyone is able to play the game without any difficulty, Tetris becomes very complex when trying to learn how to achieve constantly high scores. It is estimated to have $p \times 2^{h \times w}$ states, where p is the number of pieces, h is the height and w is the width of the board, meaning that usually this number corresponds to 7×2^{200}. It was proven mathematically by Burgiel (1997) that a game of Tetris eventually ends with probability 1. This happens because if a player receives a sufficiently large sequence of alternating S and Z Tetrominoes, no matter how well they are placed, the player is forced to leave gaps that eventually terminate the game.(Burgiel, 1997) This is why modern Tetris games use a bag-style randomizer, which ensures that players never get more than four S or Z pieces in a row by shuffling Tetrominoes of all types for each 7 pieces.(Tetris Wiki, n.d.)

Due to the large state space and the fact that Tetris has been proven to be NP-complete (Breukelaar et al., 2004) it is not efficient or even possible to find all the combinations of state-action pairs and search for an optimal policy. This is the reason why tabular Reinforcement Learning methods cannot be applied to this game and instead we will make use of function approximation techniques. As discussed above, reinforcement learning problems are better solved if they satisfy the Markov Property and can be modelled as Markov Processes. The environment of the Tetris game satisfies the Markov property since at any given point in the game, you can continue playing without the need of any information from previous states since previous actions are reflected on the current board and all the information is presented at any given time.

Solving a game like Tetris with reinforcement learning which can be modelled as a sequential decision problem (Simsek, Algorta and Kothiyal, 2016) where the agent accesses its environment and makes a sequence of decisions to maximise its reward, have the potential to be a great impact on the evolution of learning algorithms. Developing better performing algorithms, could help humanity solve real -life problems that have complex environments and cannot be solved optimally without the support of AI.

3.3 Previous Attempts to solve Tetris

Between the years 1996 to 2006, various attempts on solving Tetris were made by Tsitsiklis and Van Roy (1996), Bertsekas and Tsitsiklis (1995), Kakade (2001),Farias and Van Roy (2006) who used a variety of policy methods while exploring different features of the environment of Tetris.

Tsitsiklis and Van Roy (1996) managed to clear 30 lines by only using two sets of features, the number of holes and height of the highest column. Bertsekas and Tsitsiklis (1995) later added two more features, the height of each column and the difference between columns which helped them achieve a score of 2800 lines. Kakade (2001) by using the same features but changing to policy-gradient algorithm from lambda-policy iteration, scored an average of 6800 lines. However, Farias and Van Roy (2006) utilizing the same set of features but changing to an algorithm that makes use of the Bellman equation, found a policy that cleared an average of 4500 lines.

In 2005 with the introduction of genetic algorithms to this problem, Böhm, Kókai and Mandl (2005) created a Tetris Controller with an exponential function that cleared 34,000,000 lines. They also made use of new features more specifically the number of connected holes (wells), the

number of occupied cells and the number of occupied cells weighted by its height. The game of Tetris they implemented this algorithm to, was able to know the falling piece as well as the next piece. This gave an advantage to this algorithm in comparison to the others previously stated, that were not able to know the next falling piece, subsequently making it not a fair comparison. Boumaza (2009) introduced another evolutionary algorithm to Tetris, the covariance matrix adaptation evolution strategy (CMA-ES) which resulted to 35,000,000 cleared lines on average making use of the features introduced by Pierre Dellacherie. These features included number of holes, cumulative wells, row transitions, column transitions, landing height and eroded cells. Thiery and Scherrer (2009) following the cross-entropy work of Szita and Lörincz (2006), while introducing two new features, hole depth and rows with holes, developed the BCTS controller achieving 35,000,000 cleared lines on average.

The first reinforcement learning algorithm that had very high performance was the classification-based modified policy iteration (CBMPI) algorithm by Gabillon, Ghavamzadeh and Scherrer (2013) which cleared 51,000,000 lines in the original 10x20 board. This algorithm was inspired by the Approximate Dynamic Programming (ADP) algorithm of Lagoudakis, Parr and Littman (2002) which was a value function-based algorithm. Instead, the CBMPI uses a similar ADP algorithm that searches in the policy space instead of the value function space and in combination with the CMA-ES algorithm it updates its policy. This was proven to be the best performing algorithm in playing the game of Tetris known to this day.

This review of related work indicates a gap in the literature when applying Deep Q learning algorithms to solve the game of Tetris. All of the Deep Q Learning algorithms presented were evaluated in the 49 or 57 Atari 2600 games using the environment implementation of (Bellemare et al., 2017). As indicated, neither of those algorithms had higher performance in all of the games if compared to each game individually since each algorithm utilizes the environments differently and subsequently each environment can be solved optimally by using a different method. The game of Tetris was not included in the library of the 57 Atari games and thus an official implementation of the authors of each of the algorithms tested in the specific game could not be found. There are many implementations of Deep Q learning algorithms tested in the game of Tetris in the GitHub website, but neither is presented in an academic paper that describes implementation and performance. Two implementations of a DQN algorithm tested in the game of Tetris were found in the literature (Liu and Liu, n.d.) and (Stevens and Pradhan, n.d.) that demonstrated their methodology and their results, but neither had any significant performance compared with the work discussed previously in this section.

Chapter 4

Design of the Experiment

After reviewing the relevant work on Tetris and the theory behind the DQN agents, this section introduces how the game is build and played in this project, the problems faced and how they were managed, the different variations of state representation and action selection that are available and implemented, concluding with how the agent is built to utilise the properties of this environment.

4.1 Environment

As mentioned in section 3.1 the game of Tetris has many variations, in board configuration, scoring system, increasing levels of difficulty, the ability to see the next falling piece or even to hold a piece and use it later. In this project the environment that is used includes a board of height 20 and width 10 with the normal 7 pieces. The version of the game that this project explores does not have the option to neither see the next falling piece nor hold a piece, even if this could potentially help the agent to perform better. This is because the aim was to employ the most simplistic environment possible. The scoring system is irrelevant since the evaluation of the performance of the agent is based on clearing lines and not the score of the game, scoring is discussed in the section of Reward shaping (4.4) since it can hold significance for the performance of the agent. This implementation of the game does not have increasing difficulty as it progresses, since the time constraint that the piece is falling does not impact the learning nor the performance of the agent.

There are two ways to implement the playing style of the game.

1. The game is played live, meaning that the rotation of the Tetromino happens at each time step which the piece is falling from the top one step per frame, and it is placed into a certain spot. This implementation makes advance moves possible, for example T-spins and overhangs, but creates numerous more states since there is a required action to be taken at each time step and that increases the difficulty for the agent to learn the game.
2. The game progresses from one state to another without the piece actually falling. For example, the current state of the board is presented with the falling piece. Then, the selection of the desired final position or column and rotation of the current piece continues to the next state by automatically placing the piece from one frame to another without the need to perform rotations and wait for the piece to fall all the way to the bottom. This implementation significantly speeds the learning process, but the agent

does not learn to perform advance moves which may limit its ability to play the game for longer.

From those two options, this project continues with the implementation of the second option. The first option has a sparse reward structure, meaning that the agent is required to take a lot of actions before receiving any feedback about its actions. This happens because multiple actions are required before placing a piece and receiving a reward. Thus, this game style would require much more training time in comparison with the second style of game mentioned, and even more complex environment to learn overall. The second game style which is selected, is also the main environment used in other projects, specified in section 3.3, that aimed to learn to play the game of Tetris due to its simpler complexity and the fact that it does not limit the reinforcement learning aspects that the environment of Tetris can offer.

4.2 State Representation

In order to deploy an agent that can learn to play the game, the state representation is a very important aspect since it is the input value that the agent can retrieve all the information it needs. There are two main ways to represent the state at this stage.

4.2.1 State as a Matrix

One way to represent the state of the board is in the form of a Matrix, where the placed pieces are represented with ones, the empty tiles with zeros and the falling piece with twos. This state representation is pure and does not require any computations in between state transitions but it is much harder for the agent to extract information, something that can result in much more slower learning rate. In Figure 4.1 it is presented how each state is represented in the form of a matrix, the agent receives a matrix input per time step.

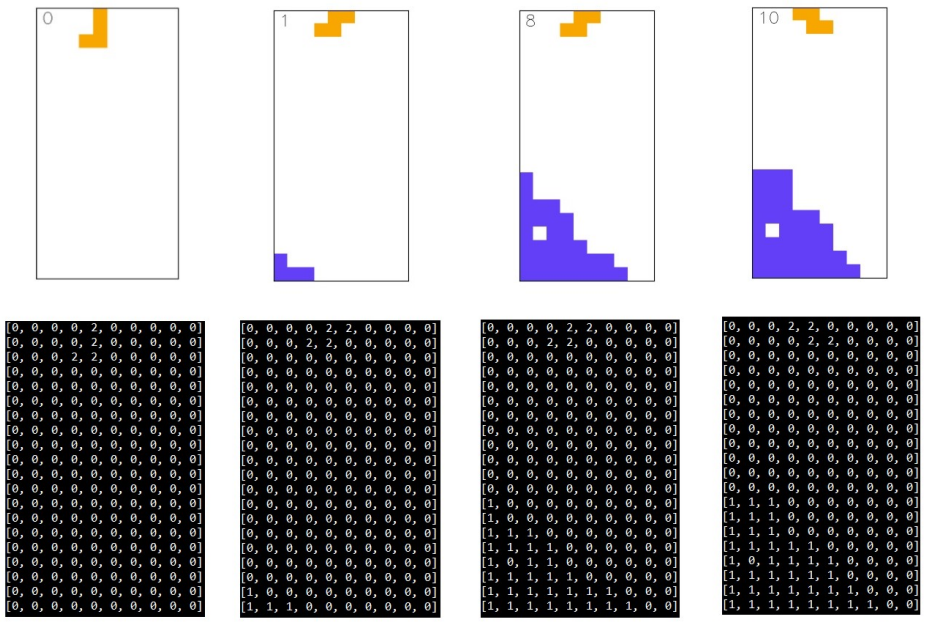

Figure 4.1: State Representation as Matrix

4.2.2 State as a vector of metrics

The second way the state can be represented is as a vector of metrics. These metrics can vary and are the same features discussed in section 3.3 where in the past a variation of set of features were deployed and had different performance. In this project the metrics used initially were Lines Cleared, Total number of Holes, Bumpiness and Summation of the total height.

- Lines Cleared: The number of total lines cleared in the placement of the last piece.
- Total Number of Holes: The number of holes currently on the board, a hole is defined by an empty tile surrounded by piece, if two empty tiles are together and both surrounded by placed pieces, in the literature this is defined as a well, but in this implementation, it counted as two holes.
- Bumpiness: Was calculated by the total difference of height between consecutive columns on the current board.
- Summation of the Total height: Was calculated by the total height of each column after the last piece was placed.

These four features created a vector of size 4×1 that was passed as the state representation. Figure 4.2 shows how these values change as the game is played, considering that the calculations do not include the falling piece, only the pieces that are already placed. The agent only receives a single 4×1 vector at each time step.

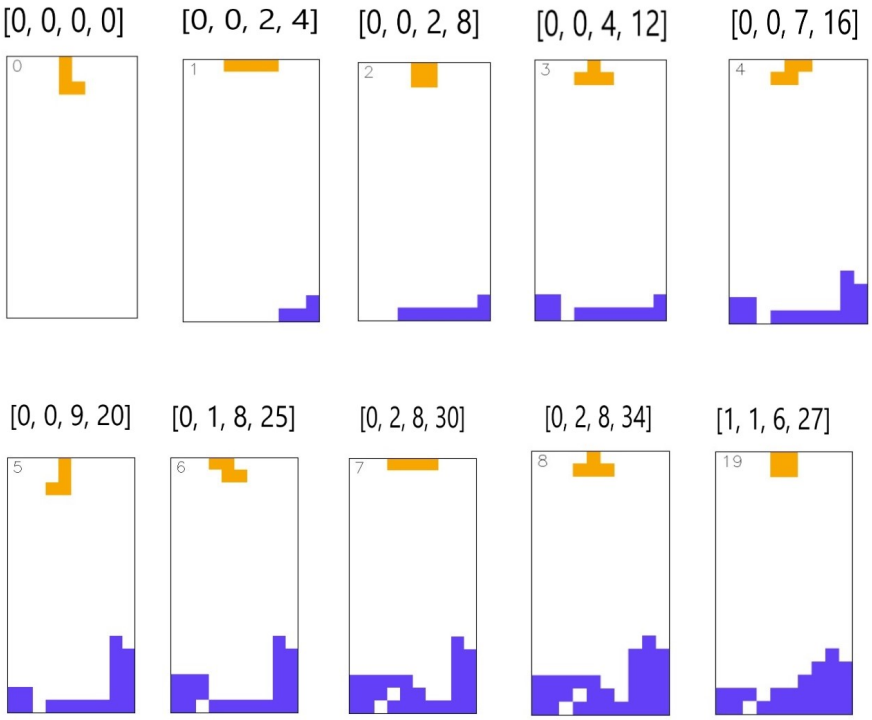

Figure 4.2: State Representation as vector of Metrics

4.3 Action Representation and Selection

The way actions are represented and selected also plays a significant role in the learning process of the agent since it is the way the agent interacts with the environment and receives feedback about its selections. There is only one way the actions can be represented in this implementation but two different ways on how the action selection can happens.

Firstly, the action is represented in a form of a tuple with two elements, where the first element represents the column that is selected and the second the rotation to be performed. For example, an action indicated by (3,180) specifies that the piece will be dropped into the fourth column, since the indexing starts with column 1 as 0, with a rotation of 180° around its axis. This leaves us with a total of 40 different actions since there are 10 columns and a maximum of 4 rotations for each piece. The pieces, like the square, where the rotation does not change the way it is dropped, still has the option to be rotated but with no different outcome at the end. This way of action selection creates a significant problem that breaks the environment since some of the pieces are unable to be placed in certain columns with some rotations.

This problem was initially tackled inside the class of the agent during the action selection process, by discarding the selected action if that action was invalid and select the next maximum-valued action. This made the agent never update its information about selecting these invalid actions and selecting them even more frequently as the training was progressing since they never received any negative impact. That slowed down the training of the agent and also did not help him to learn the correct action selection. Therefore, in order to speed up the training process and remove the limitations from the agent, this problem was mended in a simpler way inside the environment of Tetris. There was a dictionary created that mapped all the piece-actions pairs that would break the game into their corresponding ones that would happen if a human tried to place the piece in that place. For example, as shown in Figure 4.3, trying to place the ''I'' piece in the 8th, 9th or 10th column with a rotation of 0° or 180° , that would cause a line piece to exceed the board boundaries and break the game. When this action is selected, the game automatically maps that action to the action (6,0) which places the piece in the rightest place with the specified rotation. This happens also with other pieces; The full dictionary of action correction can be found in Appendix A.

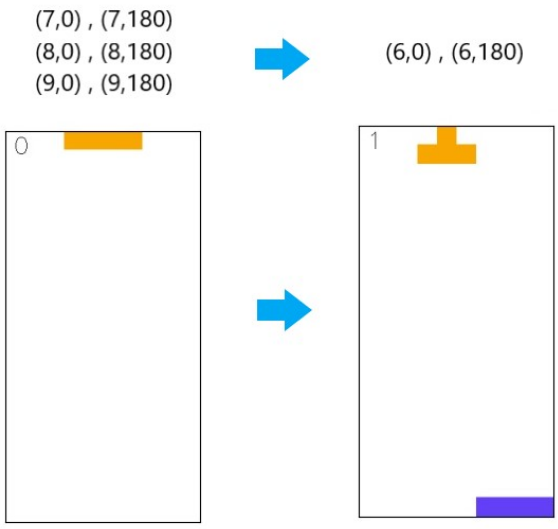

Figure 4.3: Action Correction

After dealing with the action representation, action selection can happen in two main ways. The first and obvious way is that after the agent sees the state, can choose any of the 40 available actions and after that action is selected the agent receives the corresponding next state and continues to play the game regularly.

In the second way, instead of having the option to choose any of the 40 actions, the agent receives the current state and then receives all the possible resulting states. Then the agent selects which next state wants to follow and after that choice is made, the corresponding action is selected, and the game continues likewise. In this implementation the agent has an advantage in learning since it can see the near future and select the action instead of trying to predict it.

Both of these ways will be implemented with the two different state representations in order to compare the difference in the training of the agent using all of the combinations. Each of these combinations are explained in section 4.5

4.4 Reward Functions

The Reward function is a major key in the training of the agent as changing slightly the reward can alter completely the game style that will be developed, since each different reward function encourages specific tasks for the agent to complete and receive the highest reward possible.

There are four different Reward functions implemented in order to try and maximise the performance of the agent and observe the different techniques learned.

Reward A

The first reward function is (4.1) where 1 point is given for each Tetromino placed plus the number of lines cleared squared at the placement of the last piece times the width of the board. The square of lines cleared encourages the agent to try to clear lines simultaneously since it results to a larger score. Also, if the game comes to a terminal state the agent receives a reward of -2.

$$R_t = 1 + (LinesCleared)^2 * (BoardWidth) \tag{4.1}$$

Reward B

The second reward function implemented (4.2) does not reward the agent for the placement of each piece. The only reward received is lines cleared times 10 which does not encourage the agent to clear multiple lines at the same time, but the only objective is to clear as much lines as possible in the long run. The agent receives a reward of -1 when the game is terminal.

$$R_t = (LinesCleared) * (10) \tag{4.2}$$

Reward C

The third function (4.3) is a function that makes use of the metrics discussed in section 4.2.2 and aims to make the agent understand and place the pieces more uniformly without creating a bumpy terrain and leaving holes. The coefficients of these metrics are subject to be tuned. The initial hypothesis in order to select these coefficients was to penalise the creation of holes the most, since the holes are the reason the lines cannot be cleared in the long run and the

major reason a game of Tetris ends. Secondly, to penalise the total bumpiness the second most so that the agent tries to place the piece where the pieces fit together the best. Lastly, to penalise the total summation of the Height the least and reward the clearing of a line in order to encourage the agent to keep the height low and clear lines. A reward of -100 is received when the game is terminal.

The initial coefficients selected were a=10, b=5, c=2, d=1.

$$R_t = a * (LinesCleared) - b * (TotalHoles) - c * (Bumpiness) - d * (SumHeight) \quad (4.3)$$

Reward D

The fourth reward function (4.4) mimics the exact same mechanics as reward function (4.3) but with much more encouragement to clear multiple lines at the same time but still understand the game and keep the board at low height with no holes.

The initial coefficients selected were a=10, b=5, c=2, d=1.

$$R_t = a * \left(10^{LinesCleared}\right) - b * (TotalHoles) - c * (Bumpiness) - d * (SumHeight) \quad (4.4)$$

4.5 Design of the Agent

The main class of the agent will be the baseline for all the agents to follow and the various implementations based on the state and action representation. The structure of the implemented agents follows the structure of the original DQN agent specified in Mnih et al. (2013).

The agent starts and selects actions following an ε-greedy policy and stores the transitions it observes into a Buffer as (state, action, reward, done, next state). The Buffer contains 5 different lists, state, action, reward, done, next state and stores the transition into the corresponding list so that the indexes of each list match with the transition observed. Then the agent continues to play the game while storing the observed transitions. Once the size of the buffer size reaches a certain number the agent begins to start learning. The learning happens with a random sample of a specific batch size from the Buffer. Once this sample is generated, with the use of the loss function and backpropagation techniques the weights of the Neural networks are updated. This update is different for each agent and it's the main difference between the various agents that are implemented.

The size of the batch size, the size of the memory of the Buffer will be treated as hyperparameters, as well as the number of epsilon greedy frames where epsilon will decay uniformly until reaching the specified number of greedy frames and then will be constant to a minimum value of 0.1. Some of the hyperparameters will be optimised and some will be taken as granted from relevant papers.

As discussed in the sections 4.2 and 4.3 there are two different ways to represent the state and two different ways to perform the action selection process, this leaves us with four different combinations of implementation for the agent's environment. Each of these variations is tested with the basic DQN Agent and reward function (4.1)

Variation A

The first combination is the state representation as a matrix and the action selection with the agent choosing the next resulting state. This combination forces the agent to have a

Convolutional Neural Network since the input is a 2D matrix of size 20x10 and an output of a single value. With this structure the agent receives the resulting states as matrices and predicts a single value for each of them, which selects the maximum value based on an ε-greedy policy.

Variation B

The second combination is still the state as a matrix, but the action selection happens by choosing one of the 40 available actions without seeing the resulting state. This implementation also uses a Convolutional Neural Network with the input being a 2D 20x10 matrix, but the output is a vector of size 40, where each index represents an action. Using this variation, the agent receives the current matrix and produces the vector of 40 values, which uses to select the maximum value action based on an ε-greedy policy.

Variation C

The third combination is the state as a vector of the four metrics, and the action selection with the agent choosing the next resulting state and not the action. This variation requires a Linear Neural Network since the input is a vector of size 4x1 and output a single value. With this structure the agent receives the resulting states as vectors of metrics and predicts a single value for each of them, which then selects the best resulting state based on an ε-greedy policy.

Variation D

The fourth combination receives the state as a vector of the four metrics plus another value which corresponds to the index of the current piece, but the action selection happens by choosing one of the 40 available actions without seeing the resulting state. This implementation also uses a Linear Neural Network with the input being a vector of 5x1, but the output is a vector of size 40, where each index represents an action. Using this variation, the agent receives the current state as the vector of metrics plus the piece id and produces the vector of 40 values, which are used to select the best action based on an ε-greedy policy.

From those four variations, only variation B can be characterized as model free implementation while the other three variations are characterized as model based. This is because as mentioned in section 2.1.5 model-based methods get extra information from sources that are not the agent. In the case of Variation A, the agent receives the resulting states as a given and not as a prediction from its experience. In the case of variation C and D the state is represented as a vector of metrics, these metrics come from external functions and cannot be computed from the learning experience of the agent. This leaves us with variation B, the only implementation where the learning of the agent happens purely from its experience.

4.6 Building the Environment and Smaller Board

This project makes use of the python language only. The main part of the Tetris game is from an author found on GitHub which is the implementation of the game that fits the criteria mentioned.(Faria, 2019)

This implementation was playing the game by representing the states as a vector of the four metrics discussed in section 4.2.2. Then, the game was computing the next possible states, in the form of metrics as well, with their corresponding actions as a dictionary, then the action was selected from that dictionary based on the selected resulting state. Then the action was extracted from that dictionary as a form of a tuple of column and rotation and placed in the board creating the resulting state. This is the exact way the variation C was implemented to play the game. This environment did not have any functionality to play the game without seeing the next state and just selected an action. This created problems since invalid actions were still able to be placed even when exceeding the board. In order to implement the other three variations, there were additional functions that were required to be added to the game.

- A function that returns the next possible boards in a form of a dictionary but represents the state as a matrix which was needed for variation A.
- A function that selects different reward functions based on the initialisation of the environment.
- A function that returns all the possible actions.
- A function that checks if the piece placed exceeds the board and terminates the game if that happens.
- A changed function of the main play to map pieces with the dictionary mentioned in section 4.3 and return the lines cleared after a piece placement.

In order to assess the performance of the agents before being tested in the main environment, a smaller board had to be implemented. This smaller board was used since it required much less computational time to understand whether the agent shows signs of learning and fix the bugs that may appear in later stages of training.

The board was of width=6 and height=8 with only two pieces, composed of three 1x1 cubes creating the pieces ''I'' and ''L'' since they are the only two pieces that can be created with three 1x1 cubes. Pieces with size of four 1x1 were not included since it would make the complexity of the environment much more difficult which was not the reason of the implementation of this board. All of the previous functions, action selection, state and action representation were the same as the full board, and the action selection dictionary was implemented as well as to fit the corresponding pieces of the smaller board.

The overall states of this environment are $2x2^{48}$ in comparison with the $7x2^{200}$ of the full board, meaning that the complexity of the environment was reduced by $7x2^{151}$ times. This massive reduction of the states and pieces made the training time of the agent to reduce significantly, and a successful agent could show indications of learning from the first 30 minutes of training on an average of 50 thousand steps.

Another helpful aspect of the smaller board could be the use of tuning hyperparameters and reward functions, since it is almost impossible to tune the hyperparameters properly with the full board, with the current time constraints and hardware availability. But this tuning

by using the smaller board raises some concerns since the complexity of the environment is much different and the optimal policy that the agent has to develop could be much different. Hyperparameter tuning is discussed in section 6.1.

4.7 Software and Hardware

This project is based on the programming language python which has a large number of different libraries that help the development of the environment and the implementation of the different learning agents. For the implementation of the environment the libraries used were:

- NumPy, for mathematical operations,
- Random, to import randomness into some operations.
- Time, to use the sleep function in rendering the game.
- CV2, to pre-process the image and render the game.
- PIL, to convert the matrix to an image.

These are all open-source libraries available in python. The implementation of the agent was built following the implementation of Tabor (2020) built for playing the cartpole Atari game and changed in order to fit the purposes of this project based on the variation tested each time. The main structure of the agent was kept intact.

The python files that were used to build the agent made use of the libraries:

- Collections, in order to use the deque function for the implementation of the buffer.
- Pickle, to store and load different parameters or the buffer.
- NumPy, for mathematical operations.
- Torch, for the implementation of the Neural Network.
- TensorFlow, for the implementation of the Neural Network.

The Neural Network of the agent was initially built with the library of TensorFlow but was then changed to Torch since the library of torch was easier to implement the GPU functionality and ultimately had much faster computational time. The environment and the various agents were constructed in different python files which then were imported into the Google Collaboratory Notebook where the training of the agent was implemented. Google Colab Notebook is very similar to the known Jupyter notebook but is running in the cloud and can offer GPU functionality, which can help speed the process of learning significantly. Also, by using a cloud service it was easier to continue working since the RAM of the laptop used was free during the process of training the agents. The disadvantages of using Google Collaboratory were that you can only run two different notebooks simultaneously and there were some disconnections because of time limits or idle timeouts. Hence, I upgraded to a paid subscription to enjoy more running time in the server and less idle timeouts in order to complete the training in the time constraints of this project.

Chapter 5

Preliminary Experiments and Results

The start of the experimentation begins with the implementation and training of the DQN agent to play the game of Tetris in the small 8x6 Board with all of the four different variations of playing the game discussed in section 4.5.

The DQN agent is constructed as described in section 2.3.1 following the implementation of Mnih et al. (2015) by building on the design of the class as mentioned in section 4.5. The agent starts by choosing actions based on an ε-greedy policy and stores the transitions observed in the Buffer. Once the size of the buffer is at least the size of the batch size which is set to 32 then the agent samples these transitions and uses them to update the weights of the Network. These transitions are sampled in the form of tensors. Figure 5.1 shows the shape of each tensor for each Variation.

	Variation A	Variation B	Variation C	Variation D
State	32x1x20x10	32x1x20x10	32x1x4	32x1x5
Next State	32x1x20x10	32x1x20x10	32x1x4	32x1x5
Actions	-	32	-	32
Rewards	32	32	32	32
Dones	32	32	32	32

Figure 5.1: Shape of Transition Tensors

Taking the indices one by one corresponds to a transition of the game. For example when taking the first index of states with the first index of actions, results to the first index of next states with the reward of the first index of rewards where first index of Dones shows if it was a terminal state or not. In the case of Variation A and C the actions are not sampled since they are irrelevant for the learning of the agent.

The architecture of the Network for Variation A and B where a Convolutional Neural Network is used, is similar to the one described in Mnih et al. (2015) but it was changed properly since the input matrix used in that paper was much bigger (84x84x4) and their filter size of 8x8 with stride 4 was too big for the input size of this project which was 20x10. Hence, the structure of the network is as follows. The input of the Neural Network was a matrix of size 20x10, the first hidden layer convolves 32 filters of 3x3 with stride 1 with the input matrix and applies the

ReLu activation function. The second and third hidden layer convolve 64 filters of 3x3 with stride 1, followed by the ReLu activation function. The final hidden layer is a fully connected one consisting of 512 rectified units. Further to this, the output layer is a fully connected layer with a single output for each one of the actions which were 40 in variation B and 1 in variation A. The numbers mentioned above are implemented for the Full Board, in order for this Neural Network to work on the smaller board where the input size of the matrix is smaller, the filter kernel was reduced to 2x2.

The Neural Network used in variation C and D, was a Linear Neural Network following a very simplistic approach trying to imitate the architecture of the Convolutional Neural Network specified. The input of this Network was a vector of size 4x1. The first hidden layer had 32 filters followed by a ReLu activation function, while the second hidden layer had 64 filters followed by a ReLu activation function. The output layer was a fully connected layer with a single output for each one of the actions which were 1 in variation C and 40 in variation D.

The Loss function used for the update of the weights in the Neural Network was equation (2.18) which can be denoted as the Q target values (5.2) minus the Q predicted values (5.1).

$$Q_{predicted} = Q\left(s, a; \theta_i\right) \tag{5.1}$$

$$Q_{target} = r + \gamma max_{a'} Q\left(s', a'; \theta_i^-\right) \tag{5.2}$$

Using the online model and passing the state tensor as input we get the 40 predicted q values for each of the 32 states passed, then we select only the q value for the action selected for each of the 32 states, resulting to a tensor of size 32.

Using the target model (or static model as mentioned in previous sections) and passing the next states as input the result is still a 32x40 tensor which is filtered by taking the maximum value corresponding to the best action, resulting to a tensor of size 32. Then the q-value for the states that were terminal is set to zero since the game terminates and no additional rewards are expected. Hence, the q-value should be 0. Then each of these 32 values is multiplied by the discount parameter gamma and the corresponding rewards for each of these 32 values are added, resulting to the final Q target values.

For the cases of variation A and C where there are no actions, the action selected for the Q predicted values and the maximum operation for the Q target values are excluded, because the output of the Network in those cases produces a single value.

Then the backpropagation of the model optimizing its weights happens with passing the Q predicted and the Q target tensors into the Loss function. After the completion of each training step the epsilon value is decreased by the specified decaying rate and the weights of the target model are updated each some step count which is treated as a hyperparameter and for this particular experiment set to 1000 training steps. With this hyperparameter set to 1000, the update of the target network happens every 150 episodes at the start of the training and every one or two episodes in later stages of training when the agent plays the game much better and each episode lasts much longer. This hyperparameter is set particularly high and what may seem strange is that the update at the start of the training happens not so frequently. However, it is meant to prevent the update of the target network from happening multiple times in each episode for later stages of training.

In order to assess the implementation and performance of the agent of each of the variations without biases we perform the training with the same hyperparameters for each trial. The

initial hyperparameters used were inspired by those used by Mnih et al. (2015) in the training of their DQN agent for the Atari 2600 games. Some of the hyperparameters were changed since they tended to perform better in the environment of this project.

Hyperparameters used:

- Learning rate: 0.0001
- Discount Rate (gamma): 0.99
- Batch size: 32
- Target Network update Frequency: 1000 training steps
- Exploration frames: 50000 frames until epsilon reaches its minimum value
- Starting Epsilon: 0.9
- Epsilon minimum Value: 0.1
- Buffer Memory length: 100000 transitions

All of the variations were tested with these specific hyperparameters for 15000 episodes to the Small 8x6 board with the same Reward function (4.1)

Variation A (State as Matrix– Action Selection from receiving resulting states)

As we can see from the learning curve in Figure 5.2 the agent learns to play the game and converges to a final performance of approximately 22 lines cleared from the first 100thousand steps. This performance converges to that value since epsilon is still 0.1 and the agent continues to select on a random action with percentage 10% during the training which results to terminate the game or not being able to recover from that mistake and eventually end the game early. This can be confirmed since in the evaluation when the epsilon is 0, and the agent never takes a random action, we see the agent playing the game endlessly in 96 of the 100 games tested. It would require more computational time to leave the agent play endlessly until it loses, which may never happen, and therefore it is assumed that after the agent surpasses the 50 lines cleared mark that it will not lose. It is also notable that the agent was only trained for 50 minutes while it managed to reach its convergence value on the first 9 minutes.

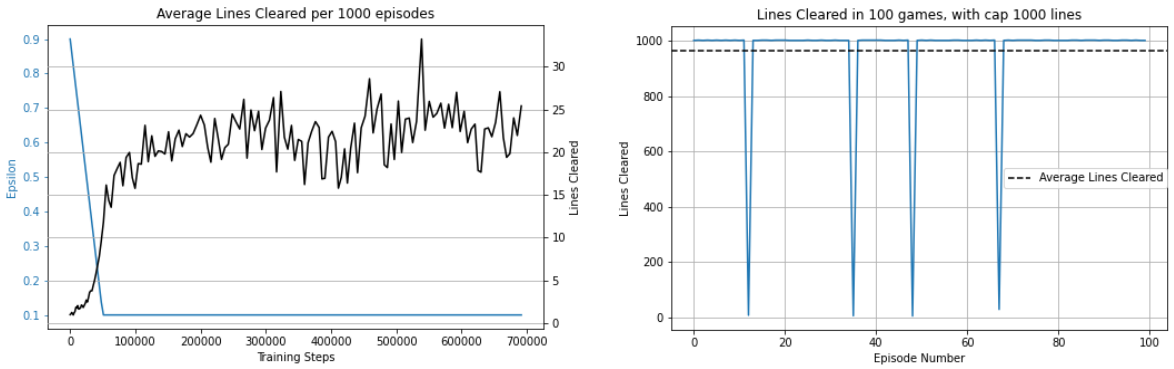

Figure 5.2: Variation A Learning Curve

Variation B (State as Matrix – Action Selection from 24 possible actions)

In the training run shown in 5.3, the agent converges to a value of approximately 14 lines cleared in a similar training step as the previous one. The value of epsilon that is kept constant at 0.1 is the reason why the agent does not clear more lines in the training process, and behaves in the same manner as in Variation A. The difference of the convergence value of Variation A and Variation B seems to not affect the performance in the evaluation part as both of these variations are clear endlessly 96% of the time. This training run took 42 minutes since it was clearing less lines on average than the previous one and managed to reach the 100 thousand training steps in 10 minutes.

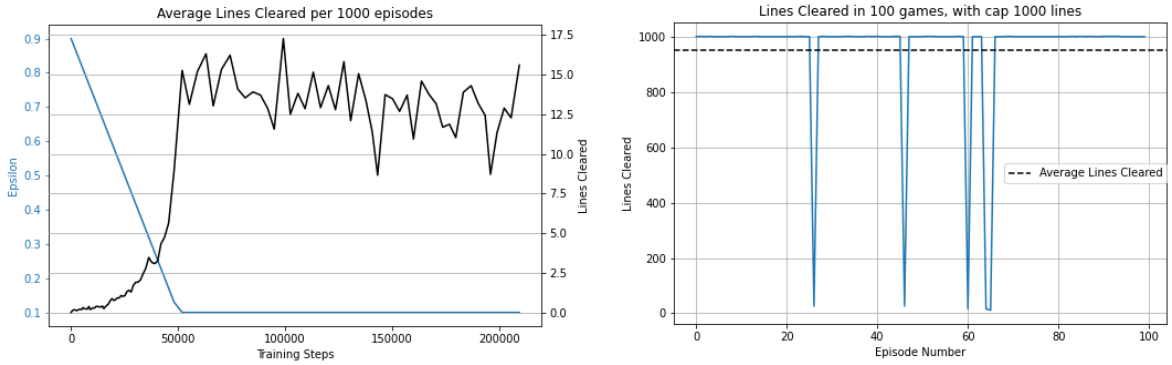

Figure 5.3: Variation B Learning Curve

Variation C (State as vector of metrics – Action Selection from receiving resulting states)

By observing the training run in 5.4 what is noteworthy is that it is much different from the previous two. Firstly, the agent seems to converge clearing 10 lines at very early stages of training but during the last 100thousand training steps this value spikes at 20 lines cleared. This probably happens because the agent finds a new policy that is better than the previous one by a random exploration step and starts learning and improving that policy. This seems to be the reason why the evaluation of this agent performs much worse than the previous two, due to the fact that this new policy it discovered at later stages of training, is not final and creates controversy between the old and new policy that leads to mistakes that ultimately terminate the game. If the training was left for another 5000-10000 episodes, then the evaluation of the agent would likely be the same as the previous two variations. This agent completed training at 43 while it reached 100 thousand frames in 7 minutes.

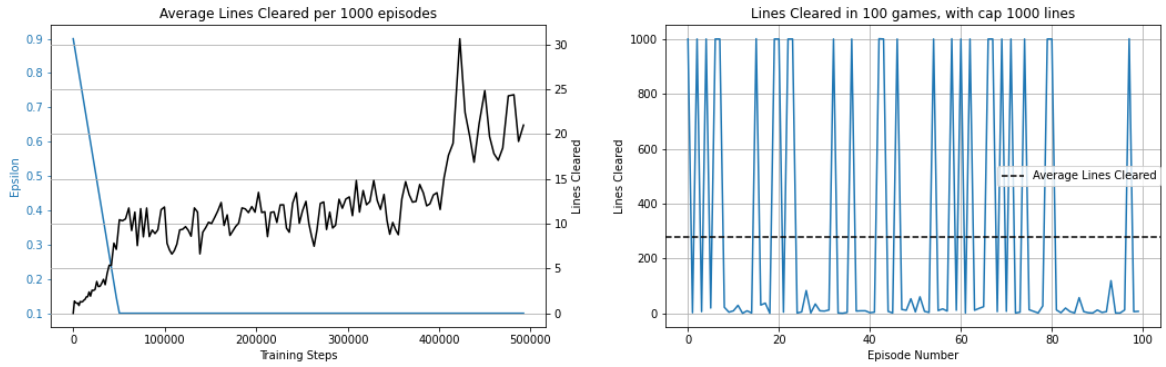

Figure 5.4: Variation C Learning Curve

Variation D (State as vector of metrics with piece id – Action Selection from 24 possible actions)

In the run shown in 5.5, we can see a very different run from the previous ones. This agent does not seem to have converge to a final value and performs very poorly in the training run in comparison to the other agents. It is also possible that in this variation the agent needs much more episodes in order to converge to a value or its policy is the most susceptible to the random action selection that happens in the training process due to the epsilon staying constant at 0.1. Although this agent had the worse training performance, in the evaluation process managed to reach the cap of 1000 lines in all 100 games without losing. This can give the intuition that the performance of the agent in the training process, while epsilon is still influencing actions, may not reflect on the quality of the final policy.

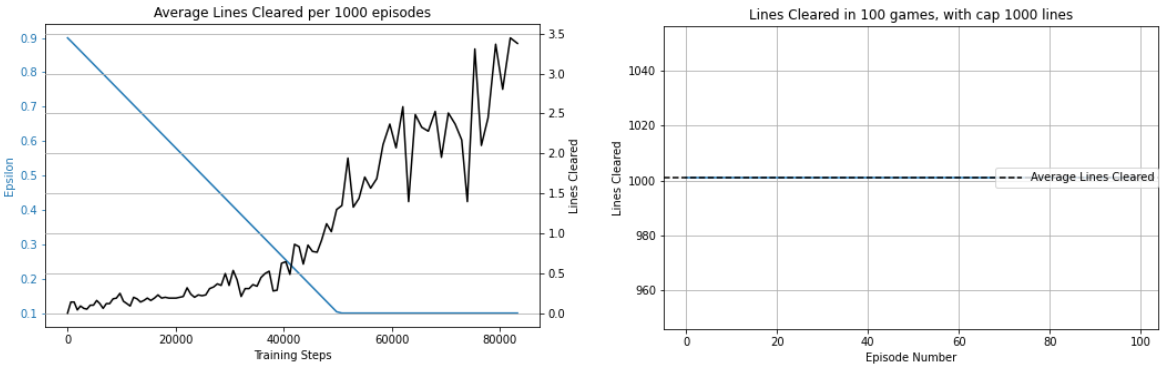

Figure 5.5: Variation D Learning Curve

After observing how the agent performs in all four variations it is not straight forward to select the best performing variation to continue with since in variations A and B the agent plays the game perfectly 96% of the time, in variation D the agent plays the game successfully 100% of the time while in variation C the agent was very inconsistent. The main difference of these variations, as mentioned earlier, is that only one of them is model-free while the others can be considered as model-based methods. It was very surprising that the model-free method could compete with the performance of the model-based methods, but this is because the complexity of the small board is rather simplistic. The translation of this performance in the bigger board is expected to decrease significantly for this model-free method in comparison with the other

three variations. Nevertheless, this project onwards focuses only on experimentation with the Variation B, the model-free method, since it represents a much purer form of Reinforcement Learning that could be more interesting to explore. Additionally, this model-free method can be used in other environments with very few changes, contrary to the other three methods that may be completely useless in other environments since they are built specifically to make use of metrics and information of the Tetris environment.

Finally, as mentioned in section 3.3 previous attempts to solve Tetris, made use of model-based methods, and succeeded impressively by finding the optimal weights to use for the metrics and play the game.However, there are not many succeeding agents making use of a model-free method and it would be interesting to investigate if a model-free method could succeed in such a complex environment as Tetris.

Chapter 6

Main Experiment

As specified earlier, the focus of the project onwards will be specifically in the implementation called variation B. This means that the state representation is a Matrix the size of the board with 0,1,2 indicating empty cells, placed pieces and falling pieces correspondingly. The agent selects an action based on the 40 possible actions of the full board by following the action correction mentioned in section 4.3.

It is also notable that this method of action correction, mentioned in section 4.3, results to a specific mapping of the actions which ''pushes" pieces to the rightest column most of the time. For example, out of the 40 actions of the line piece, 4 actions per column, 8 of those are mapped to the same action and place the line piece horizontally in the rightest position on the board. This does not affect the ability of the agent to learn to place the pieces correctly, but it may affect the learning process since it is biased towards the selection of that action and eventually is selected randomly more times than the other actions. We can observe this hypothesis in later stages when the agent completes its training and the gameplay is observed.

The project aims to optimize the parameters and improve the agent's structure with more advanced agents than the DQN. This happens in order to maximise the performance of the final agent playing the game on the Full Board. In the given time constraints, it is very time expensive to experiment with various parameters, reward functions and different algorithms of agents on the Full Board since the training time could take more than 48hours to complete and show significant insights. Thus in this project the aim is to use the smaller board to experiment and optimise the parameters of the agent hoping these optimal parameters reflect to improve the performance on the Full Board as well.

6.1 Hyperparameter Tuning

As mentioned in section 5 some values were selected as the hyperparameters of this project. We have 8 different hyperparameters that could be tuned in order to fit the purposes of this exact environment and help in the learning process. Nevertheless, it is almost impossible to tune all of them given the current computational power available in this project. Only two of them, gamma and learning rate, are explored and tuned since it is proposed that these two parameters may have the biggest impact on the learning performance of the agent.

The most straight-forward way to attack this optimization problem would be to train one agent for each of the combinations of hyperparameters and observe how each combination affects

the learning. This again would take a lot of computational time which is not the optimal scenario, hence a more simplistic approach to tune these two parameters from pure intuition of the results trying to achieve a better gameplay is followed.

The learning rate parameter can be a huge part of the learning process of the agent, having a wrong value set for this parameter can result to no learning at all since this learning rate tunes how much the weights of the Network change after each update. A very small value could result to longer training time in order to find the optimal policy since the change at each update will be very small and thus take more time to get there, or the agent will never converge to an optimal policy. A large value of learning rate could cause a very volatile learning where each update could result to a much different policy. This may cause the agent to an early convergence of a sub-optimal policy or to never converge to any policy. (Brownlee, 2020)

Hence the first parameter to optimise is the learning rate. By keeping the other parameters constant and gamma to 0.99, five different values of learning rates, 0.01, 0.001, 0.00025, 0.0001, 0.00001, were tested. The Figure 6.1 shows the learning curve of each agent trained with the different learning rates. From these curves we can observe that the middle three values (0.001, 0.00025, 0.0001) follow a very similar trend where they have a huge spike in performance at the episode 5000, where the epsilon reaches its minimum value and stays constant at that value afterwards. Then their performance is slowly decreasing until the end of the training. This may be happening because the agent is over optimistic about a certain policy which cannot be constantly successful due to the stochasticity of the environment or the randomness of the epsilon. In contrast, observing the minimum and maximum value of learning rate, 0.00001 and 0.01 correspondingly, we can spot a linear trend in the learning process where the time when epsilon reaches its minimum value at 6000 episode the spike is much smaller than the other three learning curves. At the end of the training, we can see that the smallest value of the learning rate parameter has the most lines cleared and still holds the linear upward trend it had during the whole training where the other values of learning rates seems to have converged.

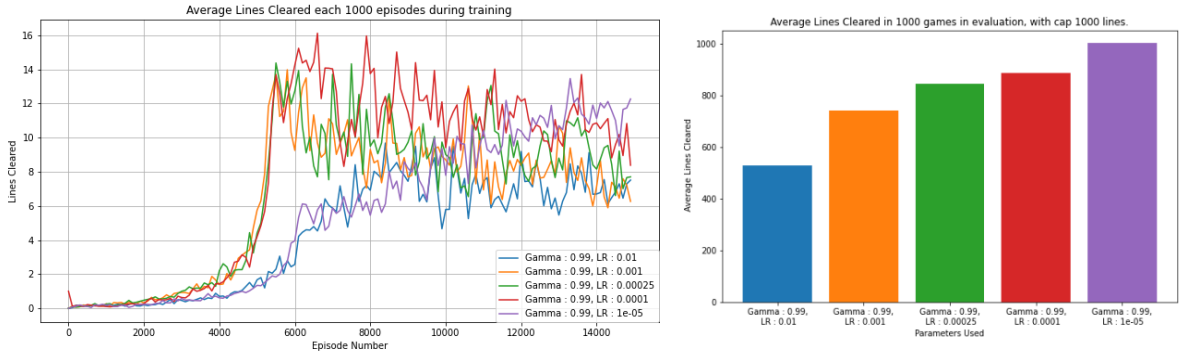

Figure 6.1: Training and Evaluation of Various Learning rates with gamma=0.99

Nevertheless, as discussed in section 5 the overall quality of the policy is not fully translated by the learning curve since some policies may be more or less susceptible to the randomness of epsilon. Hence, we use each of those agents trained while setting epsilon to 0 and evaluated their performance in 1000 games, where a maximum cap of 1000 is set since some of the agents tend to play forever. In Figure 6.1 we can see the average of the numbers each combination of parameters managed to clear in 1000 games. In this case the learning curve is pretty accurate with the order of the final performance as the best performing combination in the learning

curve was also the best performing in the evaluation where it reaches the cap of 1000 lines in all of the 1000 games.

At this point, the best learning rate seems to be 0.00001, but the discount rate, gamma has to also be taken into consideration. Gamma as also specified in section 2.1.1 is the ''value'' that the agent gives to future rewards in a way that is not just biased for the immediate reward only. This term, tunes how much the policy that the agent learns plans for future rewards. In the game of Tetris planning for future rewards is important since in most cases the placement of multiple pieces is required in order to receive a reward. This is the reason why initially the value of gamma was set to 0.99, in order to maximize future rewards and achieve a better overall policy.

However, in the game of Tetris, there is a maximum lines cleared per piece placement, which is 4 lines and only happens with the placement of the line piece in a horizontal position which completes a line of 4 rows. This is the optimal policy that pro players use since it is the move that gives the maximum score available in the game each time, In human competitions the final goal is to reach the highest score in a certain time limit. In this project, the overall score does not have a significant role since the aim is to create an agent that can learn the game and clear lines for as long as it cans rather than maximising the score per piece placement. This thought led to the hypothesis that consistently clearing lines and keeping the board at low column heights could reduce the risk of terminating the game and achieving an overall higher number of average lines cleared per game. Since planning for clearing multiple lines with one piece or planning for rewards too far ahead into the future increases the risk of losing, for example if a very difficult sequence of pieces occurs or if waiting for the line piece for too long. This hypothesis led to the conclusions that the value of gamma should be decreased and an agent that focuses on clearing lines more frequently should be built. Therefore, the value of gamma was changed from 0.99 to 0.5 and tested with the same learning rates as before. The results of this could be seen in Figure 6.2 .

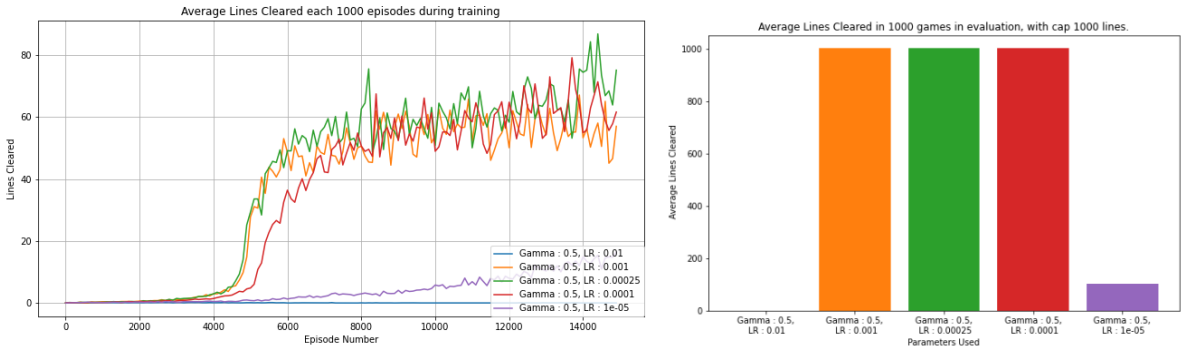

Figure 6.2: Training and Evaluation of Various Learning rates with gamma=0.5

Similarly, as before the middle three values of learning rate appear to follow the same trend and the minimum and maximum value of learning rate follow a very similar trend between them. As previously seen, with the value of gamma set to 0.99, a large spike occurs and then there is a decrease of the learning curves. In this case, the middle three values of learning rates still appear to have a big spike of increase around the episode 5000 but then they keep increasing until converging to a value of approximately 60 lines cleared on average of 1000 episodes while epsilon is still 0.1. The learning rate with the minimum value has the same linear upward trend as before but is much slower. It also seems that it needs much more

training episodes to reach a better policy, while the maximum value of learning rate did not learn to play the game at all. Both of the three middle value learning rates (0.001, 0.00025, 0.0001) achieved the maximum performance in the evaluation process.

We can see that these policies in comparison with before, almost triple their performance in the training process where the agent takes actions randomly with probability 10%. This indicates that these policies are much less susceptible to epsilon, meaning that they can recover better from the mistakes the agent makes when acting randomly. Having that in mind, gamma=0.5 helps the agent develop better policies and is preferred over gamma=0.99 for the small board. As for the learning rate in the case of gamma=0.5 the learning rate=0.00025 seems to converge faster than the other two and to a higher value, but in case of gamma=0.99 the learning rate=0.0001 converged faster and to a higher value than the learning rate of 0.00025. Thus, there is no obvious pick of which is the best learning rate parameter to choose since it seems that this parameter is affected by the other parameters as well as with the complexity of the environment. Hence this project continues with the value of learning rate 0.0001 due to the increase of the complexity and stochasticity of the bigger board, a smaller changing rate on the weights seems more reasonable. It would be more preferable if these parameters were tuned in the bigger board.

6.2 Reward Function Experimentation

The reward function as mentioned in section 4.4 is a very important part of the training and the final policy the agent learns. Especially in the case of the model-free method that this project focuses on, the reward function is the only ''external information'' we can give the agent to help the development of a good policy. Thus, this reward function should be optimised in order to achieve a better performance. This optimization happens through the DQN agent in the small board with the parameters selected above, but this raises some concerns since the policy the agent should develop in the full board could possibly have a lot of differences with the policy that will be optimised in the small board due to the complexity of the environment. Also, after optimising the parameters in the small board, changing the reward function results to the agent learning to play the game perfectly in almost all the cases, which does not reflect the same in the full board and does not offer enough information in order to choose the best reward function to continue with. The way to obtain the most important information is by observing the agent play the actual game and understand the policy that is developed and how each reward function influences it.

6.2.1 Results in the Small Board

As we can see from Figure 6.3 the DQN agent has very good performance with all reward functions except the reward function D. In the evaluation process the agent performs excellent in all cases excluding reward function D. The convergence in cases A and B is almost identical with reward B having a slightly faster convergence but both end up in the same final value. In the case of reward C the convergence is slower than the other two but with a sudden spike to a higher value than the others. In the last reward function D, the agent has a much slower learning rate and seems that it has already converged to clearing 3 lines on average which can be considered very poor performance for the small board.

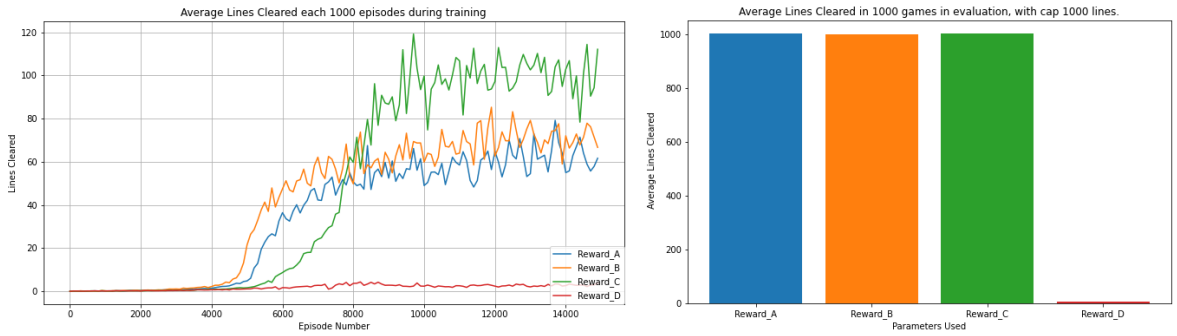

Figure 6.3: Reward Functions on Small Board

The idea behind this reward function D is to encourage multiple line clearing which contradicts with the idea behind choosing a smaller value of gamma to focus more on the current reward, causing the agent to have poor performance. Indeed, as show in Figure 6.4, when tested with gamma=0.99 and reward function D, the agent learns to only clear multiple lines simultaneously and never lose when is evaluated, something that clearly indicates that this reward function performs better with gamma=0.99.

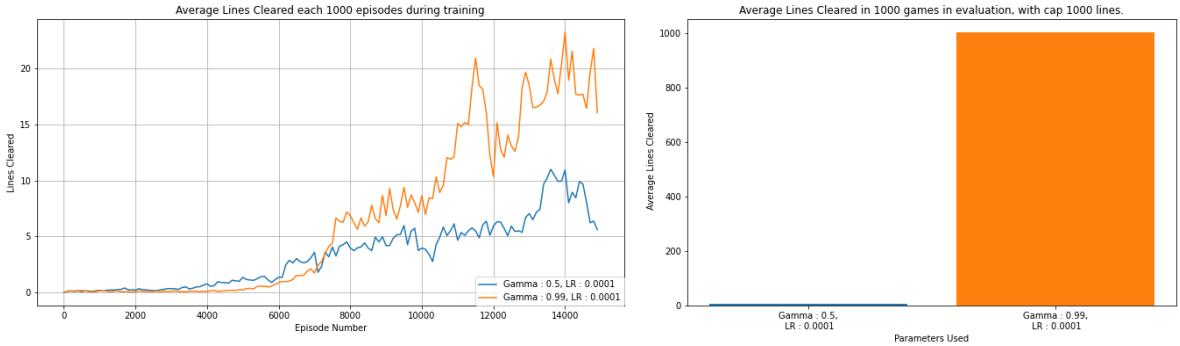

Figure 6.4: Reward D with different Discount Rates

The main difference of these reward functions is the game play. In reward A and B we see a very similar technique of placing the pieces with the difference that in the case of reward B the agent clears the line immediately when it is possible. This happens because the only reward it gets is from line clearing and not for placing a piece, in contrast with reward A which does not behave in the same manner and can plan to clear even two lines simultaneously. The agent with reward C seems to have developed the exact same technique that the agent with reward A has develop which is very interesting since their reward functions are much different. This may occur because the small board has only two pieces and uses similar techniques to keep the board ''tidy''.

By observing the gameplay of the agent with reward D, and gamma=0.99, a big difference of policy can be observed. This policy plans to create a well, wait for the line piece to clear three lines simultaneously and almost never chooses to clear less lines even if it is possible. At this point, it would be optimal to use the smaller board to optimise the parameters of reward functions C and D and select one reward function to continue this project. However, this is not possible since every reward function works on the small board but is not easy to predict which one could be more efficient in the full board. Since the aim is to reduce the risk for the

agent while playing the game and discourage multiple line clearing, the reward function D is not further used.

The gameplay of these agents with various reward functions can be found following the link in the Appendix B.1.

6.2.2 Results in the Full Board

Hence, in order to choose the most promising reward function, the DQN agent is trained in the full board with both of these three-reward functions and its performances are observed. The agents with different reward functions are left training for different amounts of training episodes since based on their performance the training time is more expensive in some cases than others. The agent with reward A was trained for 520 thousand episodes, the agent with reward B for 380 thousand episodes and agent with reward C for 220 thousand episodes.

Hyperparameters used:

- Learning rate: 0.0001
- Discount Rate (gamma): 0.5
- Batch size: 32
- Target Network update Frequency: 5000 training steps
- Exploration frames: 3000000 frames until epsilon reaches its minimum value
- Starting Epsilon: 0.9
- Epsilon minimum Value: 0.1
- Buffer Memory length: 300000 transitions

Reward A

In Figure 6.5 we can see the learning curve of the DQN agent using the Reward function A (4.1) and the evaluation of the agent after training. In contrast with the smaller board, this performance is very poor since it took the agent 6 million training steps in order to clear 1 line on average with epsilon 0.1. At 11 million training steps the epsilon of the agent was further decreased to 0.01 in order to help the agent reach higher scores and learn to play the game better. The policy does not seem to have converge and if further training is made, the agent could reach clearing more lines on average. However, due to time constraints, the training was stopped earlier than it should for both this agent and for the next ones. In the evaluation process we can see that the agent clears 2 lines on average, reaching a maximum of 7 lines once but only once it did not clear a line.

Through observing the gameplay, the agent seems to have learned to place the pieces not completely randomly and understands that it needs to clear a line. This can be observed since every time there is an opportunity to clear a line, the agent selects it but creates a lot of holes. Consequently, when multiple holes are created the agent stacks the pieces in one or two columns and loses. This indicates that further training is required in order to understand the basics of the game but also that the agent did manage to learn that the goal is to clear lines.

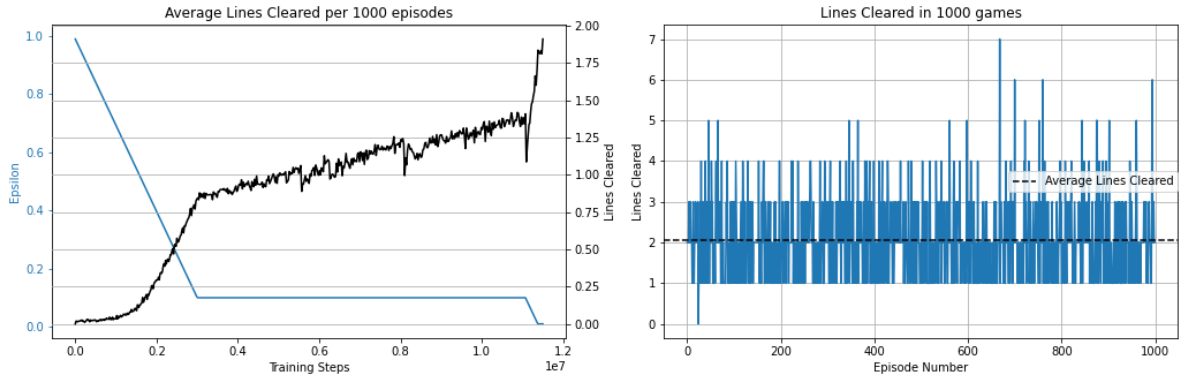

Figure 6.5: DQN-Reward A Training on the Full Board

Reward B

The following Figure 6.6 shows the learning process and evaluation of the agent using the reward function B (4.2) on the Full Board. The learning curve of this agent is very similar to the previous one where it was using the reward function A, and similarly the performance on the full board is much poorer than its performance on the small board. We can observe that this agent reached the mark of one line cleared on average at the same step as the last one but then it begun improving its performance faster. The agent did not converge to a final policy and was still improving its performance, but the training had to be terminated due to time limits. In the evaluation process the average lines cleared in 1000 games are slightly higher than the average of the agent with reward function A with some higher performances. Although on the contrary, this agent failed to clear a line many more times than the previous agent. This shows that the policy of the previous one is less risky but fails to clear lines when the height of the board is increased. This policy which focused on clearing lines immediately also indicates that it manages to clear more lines but has less understanding of the game which results to losing without clearing a line more frequently.

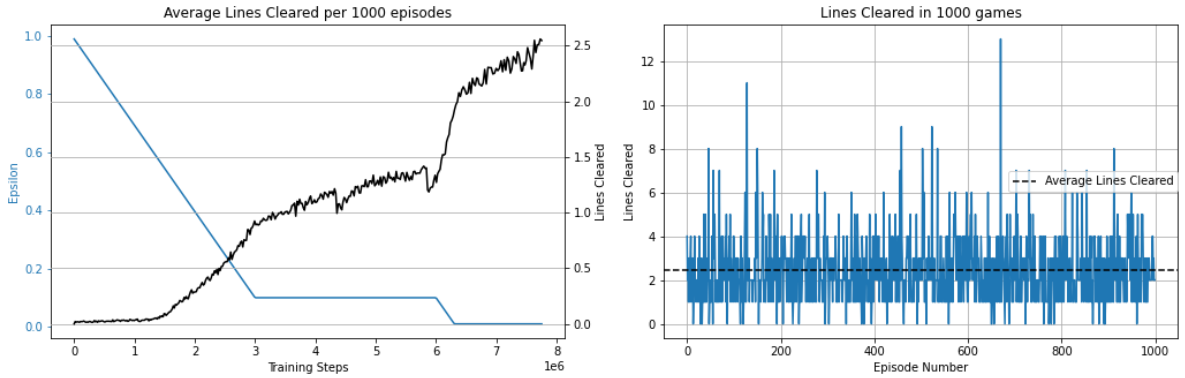

Figure 6.6: DQN-Reward B Training on the Full Board

When observing the agent play the game, it seems that the agent understands some of the basics of the game but focuses on clearing lines rather than creating a more seamless board without leaving holes and creating additional bumpiness in cases that it can. In comparison with the previous agent, one could say that this reward function makes the agent play less randomly, however this can still vary between games and thus further training is required.

One common mistake the agent tends to do, is that it places more pieces in one column and stacks them which results to the termination of the game while still other options are available. This column is usually the last column, hence, it could be assumed that this happens due to the action correction specified earlier which ultimately creates a bias for the agent to choose that action more times than others and thus create this false style of gameplay.

Reward C

The Figure 6.7 shows the training and evaluation of the DQN agent with reward function C (4.3) on the Full board.The agent trained for 11 million training steps and at the stage of 3 million frames the epsilon was at its currently minimum value of 0.1 for another 3 million frames where it was reduced to 0.01 for the rest of the training. Reducing the epsilon from 0.1 to 0.01 made a huge difference in the training of the agent which indicates that the policy of the agent is very susceptible to the randomness of epsilon. The epsilon was reduced to almost zero in order to help the agent encounter more advanced states more frequently and improve its policy for those states as well. At the end of the training the agent seems that it started to converge but it is not certain that if the training was continued the agent would not improve further. We can see in the evaluation of the agent that the average lines cleared in 1000 games is 124 , which is a huge improvement from the agent with reward A that had average 2 lines and from the agent with reward B that had average 2.3 lines cleared. This shows that the particular reward function really helps the agent understand the game and perform much better.

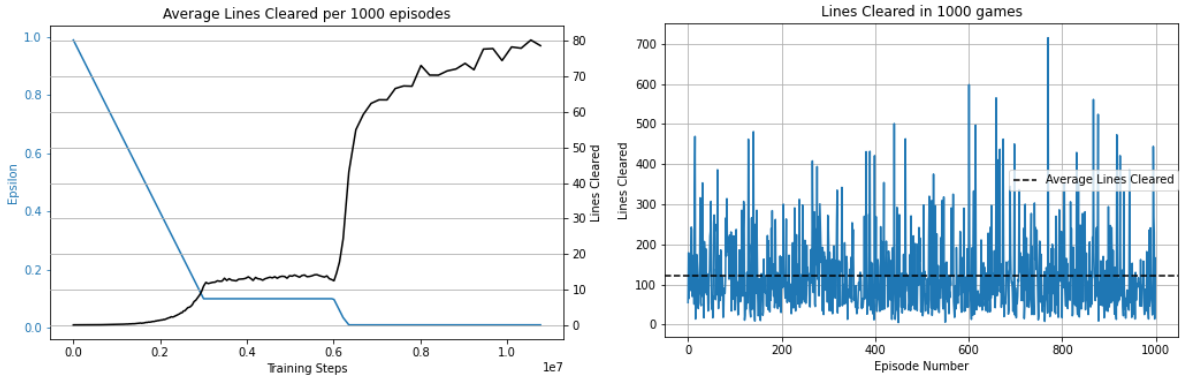

Figure 6.7: DQN-Reward C Training on the Full Board

Observing the agent play the game, indicates that the agent knows to play the game well enough by clearing lines and keeping the board without many holes. As the holes are eventually created and the height of the board increases, the agent does something peculiar most of the times which causes the termination of the game, something clearly wrong in the eyes of a human. It starts to stack pieces in the rightest column of the board while having enough space in the rest of the board and loses although there are ways even at the final step to prevent the termination of the game. This could be explained in three different ways, or it is maybe a combination of those three reasons.

1. The agent is biased towards picking an action that results to a placement of the piece in the rightest column due to the action correction implementation specified in 4.3 and the problem caused from that fix is mentioned in section 6.The issue is that the agent has encountered the action that places the piece to the last column much more times since

there are more actions that perform this placement. However, in the overall gameplay of the agent, it does not seem to be that biased towards selecting the placement of the piece in that column.

2. The placement in that place is considered the best move that maximises the reward that the agent receives at that moment since placing a piece in any other place would result to the creation of more holes and bumpiness. Thus, the agent learns to choose to place the piece into one column. If this is indeed the case it could be explored by tuning the coefficients of the reward function used (4.3) so that the particular event is fixed.

3. The agent needs more training time. The agent did not encounter this problem enough times and explored a solution and this is why it did not learn to solve it. My hypothesis is that even if the reasons 1 or 2 have played a role into the creation of this problem, the agent if trained for much longer period of time, will eventually be able to understand that this technique is wrong and result to lower rewards in the long run, and therefore could be able to find a solution that will lead to an outstanding performance.

Since this similar behaviour is also observed in the agent with Reward function B, where there are no metrics involved in the Reward function, we can assume that the second reason mentioned has the least role in this behaviour.

The gameplay of these agents with various reward functions can be found following the link in the Appendix B.2.

At this point is clear that the reward function C (4.3) results to a much better performance of the agent and helps the agent significantly to get a better overall understanding of the game. From this point onwards this project continues with this reward function. It would be interesting to try different variations of the coefficients of this reward function to observe the different policies the agent would deploy and to maximise the performance of the agent hopefully to a never-ending gameplay. But instead, this project investigates as a priority, the performance of various similar DQN agents and whether these agents with the same reward function could overcome some of the problems the standard DQN agent has and achieve better performance or more stable policies.

6.3 Double DQN Agent Experimentation

Starting with the implementation of the Double DQN agent, as mentioned in section 2.3.2 this agent was introduced by Van Hasselt, Guez and Silver (2015) hoping to reduce the overestimations that the standard DQN is susceptible to. Meaning this double DQN aims to estimate the true value of the expected reward better than the standard DQN. In this project we hope that this agent would be able to estimate the value of the true expected reward that the previous DQN agent may have overestimated and led to take actions that seemed to have better rewards than the actions that actually had the best reward. If this agent estimates the expected rewards better, we should be able to see the agent learn to play the game better and achieve higher rewards.

The implementation of the agent is the exact same as the previous agent with the only difference in the update function, where the equation of Q_{target} (5.2) is modified. In the DQN agent, the calculation of the Q target values was happening by taking the predicted maximum action for each next state from the target model. In this agent, the calculation happens by

passing to the online model the next states tensor and selecting the indices of the maximum action for each state. These indices are used to select the corresponding action value that the target model predicted for that next states tensor. Hence the Q_{target} equation changes from (5.2) to (6.1).

$$Q_{target} = r + \gamma \; Q\left(s', argmax_{a\prime}\left[Q\left(s', a'; \theta_i\right)\right]; \theta_i^-\right) \tag{6.1}$$

At first, this agent was tested on the small board, in the exact same way that the previous DQN agent was tested and with the selected hyperparameters and reward function as mentioned in the previous section. The training on the small board showed that the implementation of the agent was correct and that the learning was indeed happening. Comparing this agent with the previous DQN on the small board, it is observed that both of the agents have very similar learning curves with both of them converging to the same value with the only difference that the Double DQN had a slightly faster learning rate. This agent still plays the game on the small board 100% of the times perfectly without losing. Figure 6.8 shows the evaluation of the agent clearing 1001 or 1002 since the cap is at 1000 lines with the final step sometimes clearing one or two lines. Observing the gameplay of this Double DQN does not add anything new since it developed the same policy as the standard DQN agent. Thus, this agent could be tested in the full board to observe if this change at the loss function creates a difference in learning.

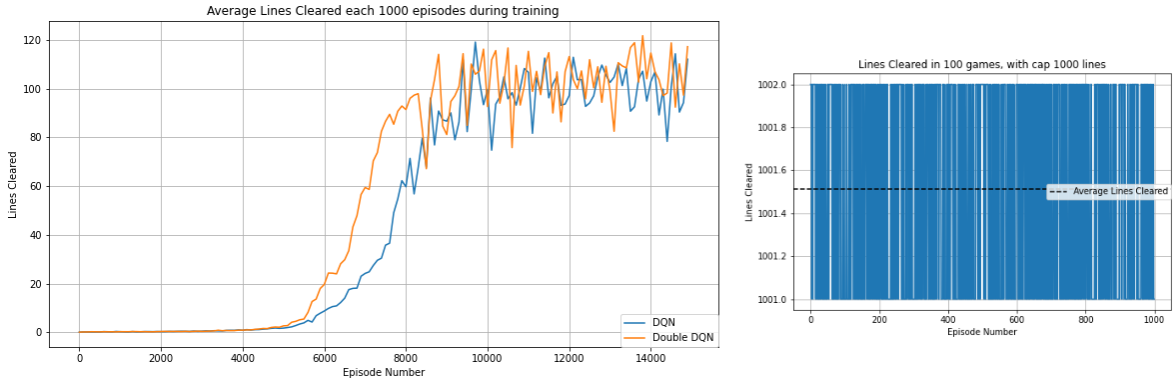

Figure 6.8: DQN and Double DQN training on the small Board, and the evaluation of the DDQN

In Figure 6.9 the training of the agent on the Full Board with the same hyperparameters and the same reward function for 220000 episodes is shown. The results of this training and evaluation are very similar to the results of the DQN agent, as they both converged to the same value at very similar training steps and with not much different rate. Similarly, in evaluation they had very similar average with a difference of six lines, the maximum lines cleared in the Double DQN were a bit higher, but the minimum lines cleared of the standard DQN were better, which raises some concerns. From this metrics it is very hard to identify if the Double DQN agent managed to stop the overoptimism on the true expected reward. However, even if it did manage to predict that value better, it did not make any significant change to the final performance and maybe the overoptimism of DQN helped to score higher minimum lines cleared than the Double DQN.

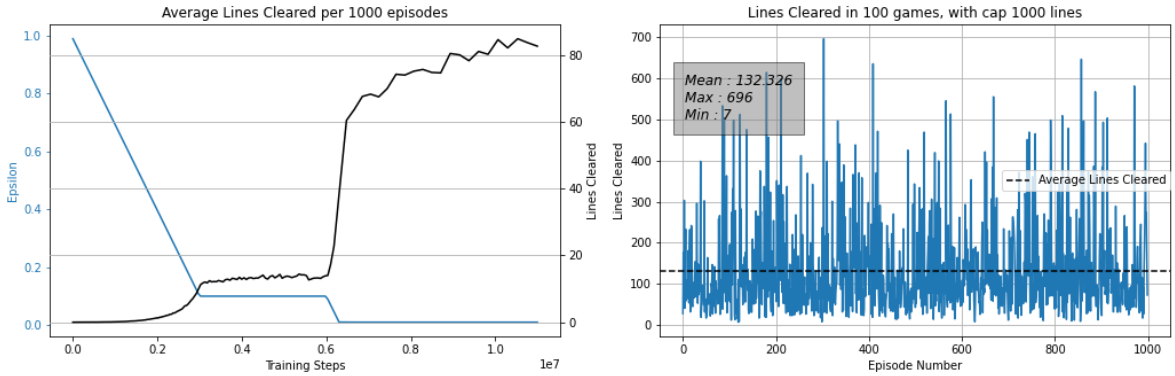

Figure 6.9: Double DQN -Training on the Full Board

Observing the agent play the game, it seems to have developed the same or a very similar policy as the DQN agent since their gameplay is very similar. The only significant change that is shown, is that the agent loses by stacking pieces in one column where more options are available that can prevent the termination of the game. This is similar to the DQN agent, but in this agent the last column is not always selected, and this incident happens to other columns as well. This gives the intuition that the action correction may not affect the policy of the agent in the long run, but this is maybe caused due to the limited training time or the coefficients of the reward function.

The gameplay of this agent can be found following the link in the Appendix B.3.

6.4 Duelling DQN Agent Experimentation

Considering that the Double DQN Agent did not really improved the performance of the DQN Agent, this project is going to experiment with the implementation of the Duelling DQN agent hoping that this will help the agent speed the process of learning and discover more robust policies that will lead to a better performance. As mentioned in section 2.3.3 this duelling network architecture that uses a slightly different convolutional network that separates the value function into a value and advantage function. It was introduced by Wang et al. (2016) with the intention of creating an algorithm that can recognise the importance of each state without considering the results of each possible action. In the case of Tetris and specifically in the environment implemented in this project, at each different state is required to take an action and the evaluation of the next state is highly susceptible by the action taken, because of that this duelling architecture may not be the most suitable architecture. Nevertheless, as mentioned in Wang et al. (2016) for bootstrapping algorithms, it is important that the agent can estimate the state values for each state, as this can help the agent of this project aim to recreate the states that had the highest values and achieve higher rewards in the long run.

In order to implement this agent, the main class of the DQN agent was used with some modifications in the network of the agent, the action selection process, and the learning process. Starting with the network having the exact same architecture with the previous network used (section 5) but the fully connected layer prior to the last one instead of connecting to another fully connected layer to result to the final 40 values that represent the actions. This one is split into two different fully connected layers, the one called the value stream which had as an output one value that represents the estimation value of the state and the second one called

the advantage stream which had as an output 40 different values that represent each action available. Hence the network has two streams of output, one value coming from the value stream and 40 values coming from the advantage stream. The action selection was being fulfilled using only the advantage stream and selecting the maximum resulting action based on an ε-greedy policy.

The update equation is the same as (5.1) for the Q-predicted and (5.2) for the Q-target and similarly the loss function as used in the same way as the standard DQN. The difference comes in the to how these Q values are calculated since in this architecture there are two streams of output. These Q values are calculated using the (6.2) equation where the V corresponds to the values produced by the Value stream and the A the values produced by the Advantage stream.

$$Q(s, a; \theta_i) = V(s; \theta_i) + \left(A(s, a; \theta_i) - \frac{1}{|A|} \sum_{\alpha\prime} A(s, a'; \theta_i) \right) \tag{6.2}$$

For start this agent was trained in the small board as always to observe if the implementation is correct and fix any bugs that may appear, and test whether the agent can display learning in a simpler environment before the training starts at the Full board. In Figure 6.10 is displayed the learning curve of this Duelling agent as well as with the learning curves of the standard and Double DQN for comparison, and on the right of the Figure is the evaluation of the agent in 1000 games with a cap of 1000 that was reached in every single one of them. From this comparison is shown that the implementation of this agent works, and it is very similar to the performance of the other two agent, since all of them converge to the same value and with approximately the same rate. Observing the gameplay of the agent does not give any significant insights since it developed the same policy as the previous agents, so in order to assess better the performance of this agent it is tested on the Full Board.

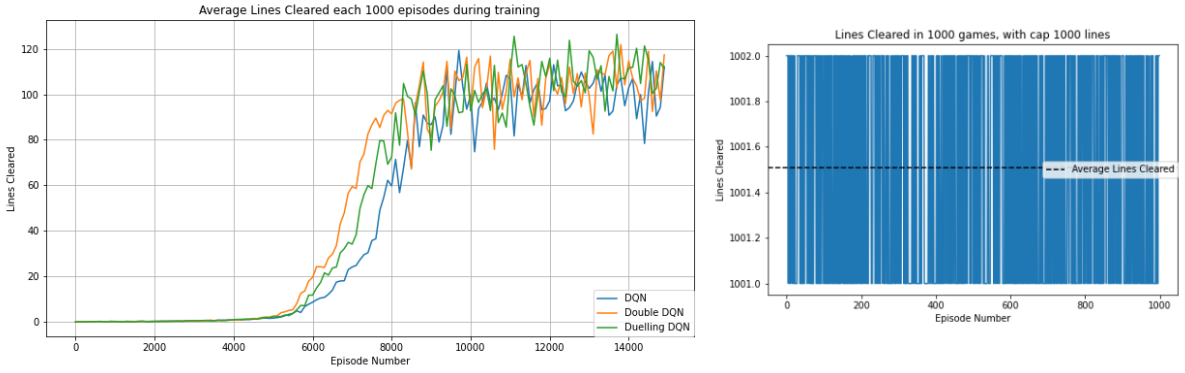

Figure 6.10: Duelling DQQ, DQN and Double DQN training on the small Board, and the evaluation of the Duelling DQN.

With the use of the same hyperparameters for the Full board as the previous agents and the same reward function C the training on the full board of the Duelling DQN agent is displayed in Figure 6.11. The learning curve is again similar with the previous agents with the difference that this agent was able to reach their maximum value that was 80 lines cleared much earlier in the training process and at the end of 220000 episodes, the same episodes that all of the agents are trained for, was able to reach 100 lines cleared and shows that still have not converged to a final policy. This shows that this agent could potentially increase its performance from further training. In the evaluation of the agent it is observed that its performance is much

better than the previous two since its average value is approximately 30 lines cleared more and it seems that can reach much higher values than the others. This is probably caused from the duelling architecture of the network that can estimate the state values and understand the game and objective of the game faster which results to reaching higher values earlier in the training process that benefit the overall training.

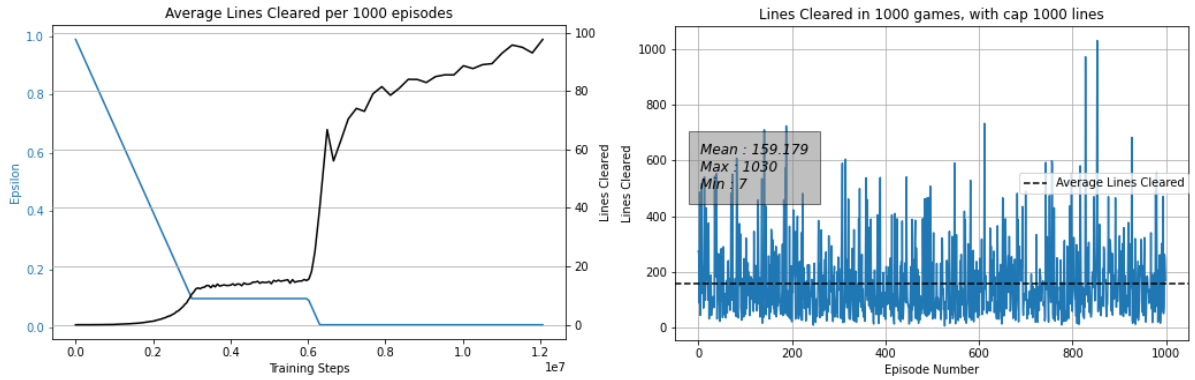

Figure 6.11: Duelling DQN- Training on the Full Board

The gameplay of the agent does not provide any significant information to understand what was changed in the policy of the agent that result to better performance since the gameplay is very similar to both of these three agents. This agent still causes the termination of the game due to stacking pieces to one column when other actions are available that can prevent the agent from losing, but this stacking is arbitrary to which column it happens, so it is probably caused, as mentioned before, from the coefficients of the reward function or the lack of enough training.

The gameplay of this agent can be found following the link in the Appendix B.4.

6.5 Double Duelling DQN Agent Experimentation

The performance of the Double DQN agent was slightly better than the performance of the standard DQN agent, and the performance of the Duelling Agent was better than both of those agents using an update function similar to the DQN rather than the Double DQN. A combination of these two, the Double DQN update with the Duelling network of the Duelling DQN could result to a much better policy and overall performance of the agent. As mentioned, the Double Duelling DQN agent will make use of the Duelling Network , the action selection of the Duelling DQN and the update equation of the agent is a combination of the (6.1) equation with (6.2) equation.

Testing the agent in the small board at first result to a very similar learning curve with all of the other three agents as shown in Figure 6.12. The convergence is very similar to the other agent with the difference of a spike at the very last stages of training which surpasses all of the other agents. In the evaluation of the agent, as expected the agent reaches the maximum lines for each one of the 1000 games. The gameplay of this agent is the same as the previous agents, so it is assumed that the policy developed for the small board is the same.

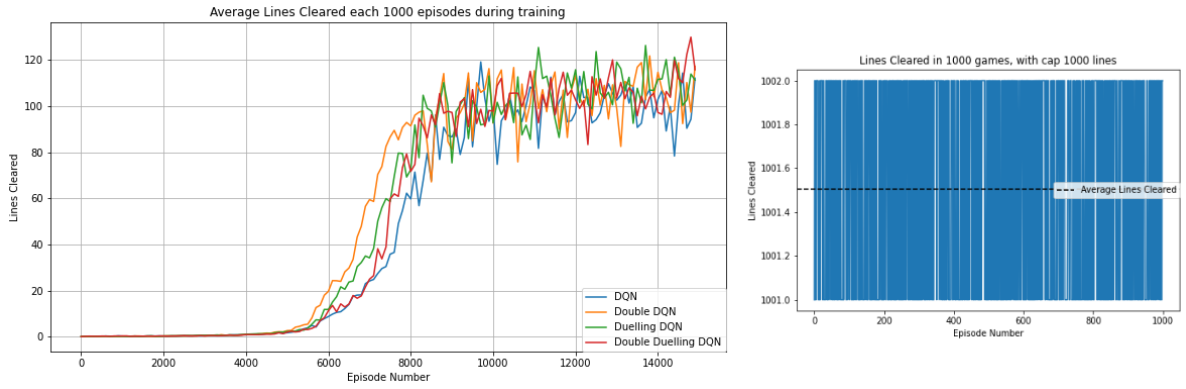

Figure 6.12: All the agents Training on the Small Board, and the evaluation of Double Duelling DQN.

Training the agent in the full board with the same hyperparameters and reward function C, it is expected to observe a better performance than the other agents since it is a combination of the two best agents so far and this could help the agent understand the game better and achieve the best performance. But as the Figure 6.13 shows, the agent reaches 90 lines and seems that it did not converge, which is 10 lines less than the Duelling DQN agent. Also, in the evaluation of the agent the average lines are approximately 15 lines less on average than the Duelling DQN and the maximum lines cleared are also significantly less. This agent surpasses the performance of the standard and Double DQN but fails to develop a better policy and achieve better results than the Duelling DQN agent. This could be potentially caused from the influence that the Double DQN has in the update function which is the only difference this agent has with the Duelling agent. It seems that if this agent fixes the overoptimistic behaviour of the agent it does not help the overall performance. This overoptimistic evaluation of the expected reward could help the agent clear more lines and keep the game without ending for more time. But it could also mean that this method as the Double DQN just need more training time to reach a better policy and even surpass the performance of the Duelling DQN agent.

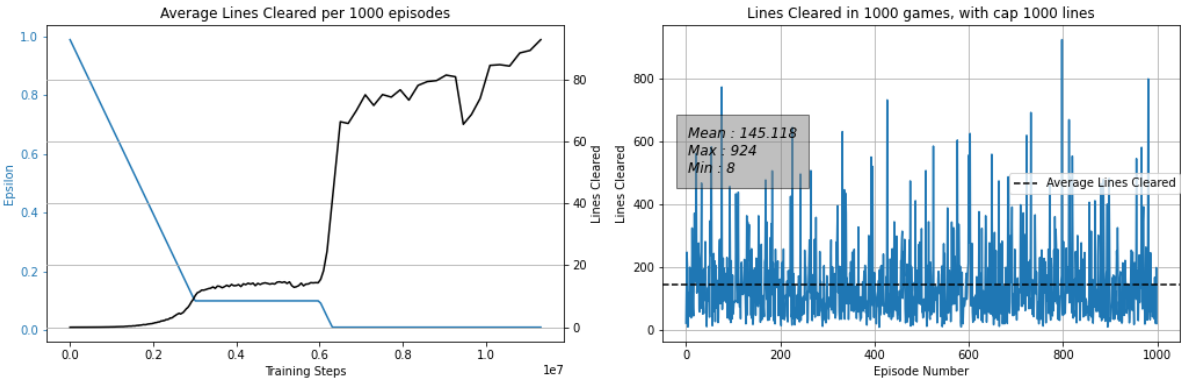

Figure 6.13: Double Duelling DQN Training on the Full Board.

Observing the agent play the game, there is not any significant playing style that can lead to the conclusion that this agent developed a different technique than the other agents. The same way of terminating the game is also observed in this agent, with not any bias to select a specific column to stack the pieces.

The gameplay of this agent can be found following the link in the Appendix B.5.

6.6 Comparison of the Agents

As specified before all of those four agents were trained with the same hyperparameters mentioned in section 6.2.2, same reward function (4.3) with the coefficients mentioned and all were trained for 220 thousand episodes in order to be compared equally. These training runs are all displayed in Figure 6.14 for easier comparison, and as shown their learning curves are very similar until the final 20 thousand episodes where the difference of their performance is made. All of the agents have a small spike at around episode 140 thousand where the epsilon reaches its first minimum value 0.1 and then a much bigger spike of performance at roughly 190 thousand episodes where the epsilon is further decreased at 0.01. It is also observed that during the episodes where epsilon was set to 0.1, between episode 140 and 190 thousand, no significant learning is shown from any of the agents, however after the epsilon is reduced to 0.01 an upward linear trend for all the agents is observed and none of the agents appear to have converged. Additionally the Duelling DQN seems to have a slightly faster learning rate and also the higher value of average lines at the end of the 220 thousand episodes. These runs needed from 36 to 48 hours to complete training based on the performance of the agent, since the better agent was playing for longer time each episode. It would be optimal to leave each of the agents complete their training when they converge to a final value, but going from 220 thousand episode to 230 thousand episode needed approximately 8 hours.

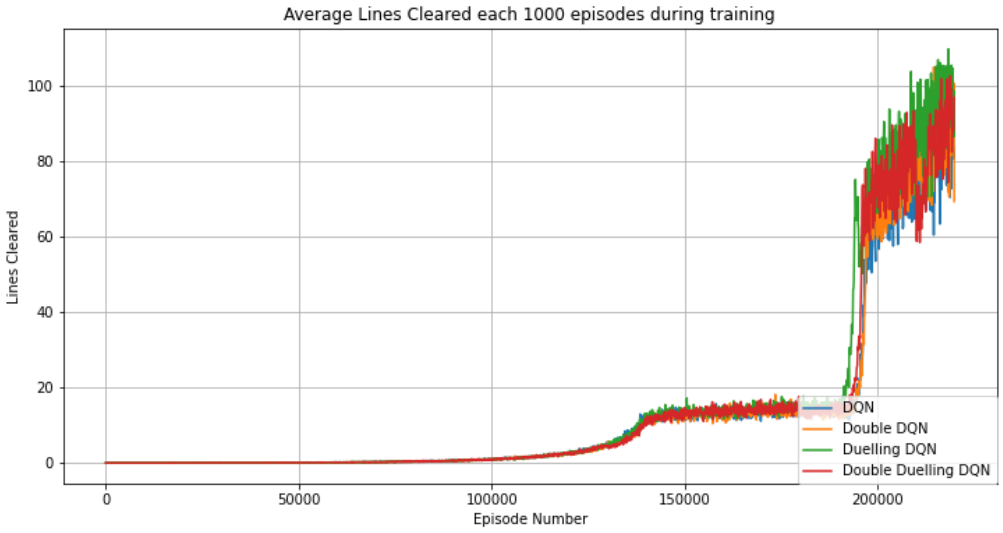

Figure 6.14: Learning Curve of all the Agents on the Full Board.

The following Figure 6.15 shows a boxplot for each one of the agents representing their performance in the evaluation run, where the epsilon value was set to 0 and each agent played 1000 games. The orange line each box represents the median value of the 1000 games , and the green arrow the mean value and its corresponding value in text. The two values on the right of each box indicate the value of Quartile 1 and Quartile 3 correspondingly. From these metrics in the evaluation run it is confirmed that the agent has similar performance as in the training. The Double DQN agent is slightly better than the standard DQN, the Duelling DQN

is much better than the Double DQN having both of its values higher than the Double DQN as well as its outlier show some very high-performance games. The combination of those agents, the Double Duelling DQN, did not perform as good as it was expected since it had lower values than the Duelling DQN at all of the metrics but better results than the Double DQN. Finally, the best performing agent for this specific environment is the Duelling DQN agent with reward function C.

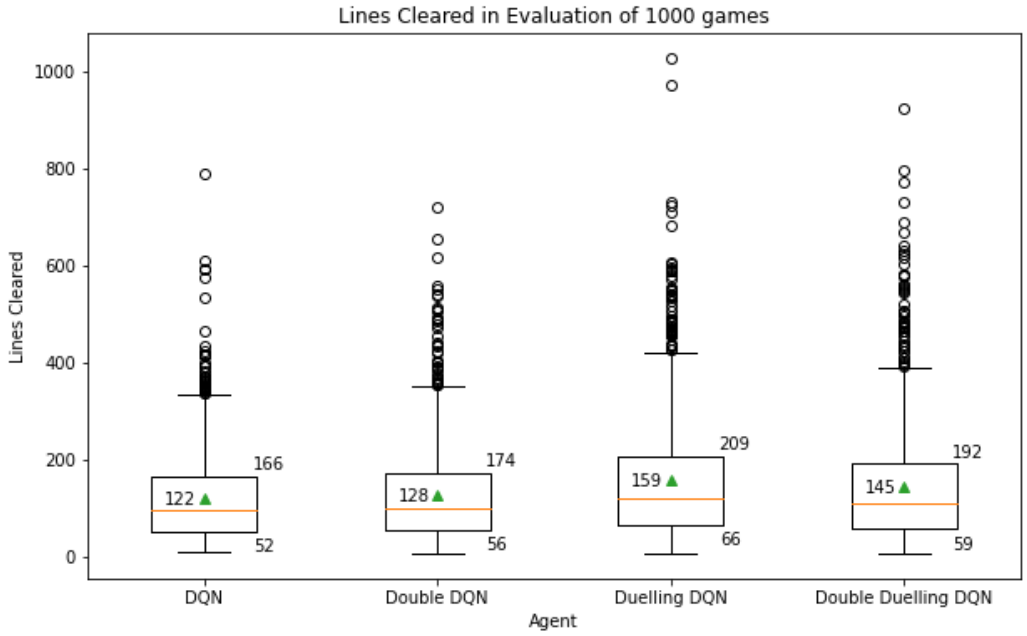

Figure 6.15: Evaluation Metrics of all the Agents

6.7 Further Training

In order to improve this agent, the problem of column stacking should be fixed. Further training of the Duelling DQN agent with the same reward function was made, to test whether with the same reward function this agent could fix that problem.

Figure 6.16 shows the learning and evaluation of this agent that was trained for 25 million training steps, the training was resumed for another 40 thousand episodes from the training in Figure 6.11. The further training improved the overall performance and shows that it did not converge to a final value but the problem that causes the early termination of the game, where more moves are available and the agent chooses to place the piece in a spot that terminates the game, still happens.

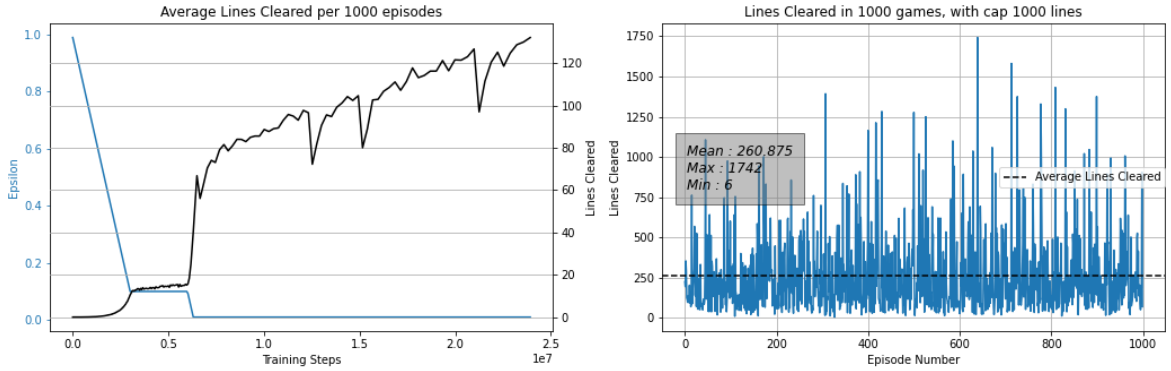

Figure 6.16: Duelling DQN further Training

The gameplay of this agent can be found following the link in the Appendix B.6.

It is also worth mentioning that a similar reward function was implemented, that considers the holes, bumpiness and total height that are created by just the last piece placement and not the total board, solved the problem that caused the early termination of the game and played until the board was completely full. This agent with that reward function did not manage to surpass the performance of the agent with reward function C since it cleared 175 lines on average but proved that this problem of column stacking was caused by the coefficient of the specified reward function 4.3.

Chapter 7

Conclusions

Overall, the results demonstrate that model-free Deep Q learning algorithms are able to learn to play the game of Tetris. The overall performance on the Small board is exceptional from all of the agents deployed with minimal tuning of the parameters, which indicates that these methods are extremely effective in simpler environments, but still this environment has 2×2^{48} states that are not considered a small state space. While the small board was also used for tuning the hyperparameters and experimenting with various reward functions, the results were not scalable to the Full Board, and it is not certain that the optimal parameters found in the small board are the same for the agents on the Full board. This limitation could show that the small board should only be used to fix bugs and see if the agent indicates signs of learning but not for tuning hyperparameters.

The same agents did not perform that good in the Full board without the help of a more advanced reward function that helps them to understand and perform at the game better. This demonstrates the huge significance of the Reward function. Changing the reward function to a more suitable one for this specific environment skyrockets the performance of the agent from 2 lines cleared on average to 120 lines cleared on average. With the limited time and hardware available, the coefficients of this reward function were not able to be tuned and were selected out of pure speculation on what could work, which indicates the potential this reward function has if the right coefficients are found. We conclude that, even in a model-free approach, supplying some domain knowledge to the agent via the reward function is a valuable tool in assisting learning.

What is also worth mentioning, the ε-greedy policy which works in order to balance the exploration-exploitation the agent performs on the action selection process, had to be refined to further help the development of a better policy. The initial policy was to decrease the epsilon to the value of 0.1 after 3 million training steps, after observing that all of the agents did not show significant improvement for further 3 million steps the epsilon was further reduced to 0.01. All of the agents after the epsilon was reduced to 0.01 showed an upward learning rate which shows that the policy was further improving with very limited exploration. This happened because the agents at the training steps with epsilon 0.1 were not able to reach higher scoring states due to randomness of the action selection which capped their learning at that point, lowering the epsilon helped the agents reach and observe different states with the board having higher height and optimise their policy to overcome this difficult states as well. From this we can conclude that care is needed when setting epsilon for optimal exploration-exploitation trade-off based on the nature of the environment.

The implementation of the Double DQN showed that it improves the performance of the standard DQN and this is probably due to the correction of the overoptimism that the initial DQN agent had.

The Duelling DQN has a much better performance than both the standard and Double DQN agents, this indicates that the Duelling network that makes the agent estimate the value of each state, helps the agent speed its learning and achieve a better policy. This finding was not expected, because the specific duelling network thrives better into environments where some states are not affected by the selected action, and the environment of Tetris does not have such states.

The Double Duelling DQN performs better than the Double DQN as the Duelling network had much more success over the standard network. Although, it does not manage to perform better than the Duelling Network. This indicates that the combination of the Double DQN and Duelling network do not work that optimally together for this particular environment and achieve lower performance.

Finally, the performance of the Duelling DQN, was trained for approximately 25 million frames, without finding the optimal parameters and not optimising further the coefficients of the reward function. The performance was proven to have an average performance of 260 average lines cleared in 1000 games with some outliers of games achieving as high as 1742 lines cleared. This shows the potential of the agent to be trained further and optimised to play this game almost perfectly. Taking this and the huge complexity of the environment of Tetris into consideration, indicates the potential of this model-free agents to succeed in other complex environments that have the properties of a sequential decision problem.

Having that in mind, these model-free agents to be successful still need the supervision in some degree of a human that can understand the basics of the environment and can tailor the parameters and the reward function based on their understanding and experimentation of the environment. Nevertheless, using a model-free method that gets all the 'external' information in the form of a reward function makes this agent more versatile and transferable to other environments.

7.1 Further Work

This implementation of the agent could be further improved in various ways that can boost the performance of the agent and achieve consistently high scores.

- Optimising the coefficients of the reward function used and including more variables to the computations such as maximum height, number of wells , Landing Height, Row and Column transitions.
- Implementing the rainbow DQN agent that utilises more properties from various different DQN agents and that has shown signs that it outperforms the rest of the agents as mentioned in section 2.4.4
- Making use of a prioritized Replay Buffer that samples the states that had the bigger loss. Through that sampling, the weights of the network could be updated more frequently to overcome the most difficult states that most of the time lead to the termination of the game. Also, with that replay buffer, the states that are reached in later stages of the

game close to termination could be used more to learn from, which can help to increase the performance.

- Experimenting and optimising the hyperparameters of the agent in the Full Board, since more hyperparameters such as the exploration frames or the size of the batch size could increase performance if optimised.
- Training the agent for at least 100m frames with a replay buffer of a size of 1 million transitions is the specified factors in the papers of Deep Mind which work on the implementation and experiment with DQN agents. The agent should be trained with these parameters in order to converge to a final policy.
- Optimising the code of the agent and the environment used could help to significantly speed the learning of the agent.

Achieving an outstanding performance by any of the methods proposed, could lead to a model-free algorithm with great potential to succeed in a very complex real life sequential decision making problem.

Bibliography

Algorta, S. and Şimşek, , 2019. The Game of Tetris in Machine Learning. *arxiv:1905.01652 [cs]* [Online]. ArXiv: 1905.01652. Available from: http://arxiv.org/abs/1905.01652 [Accessed 2021-04-23].

Baird, L., 1995. Residual algorithms: Reinforcement learning with function approximation. *Machine learning proceedings 1995*. Elsevier, pp.30–37.

Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M., 2017. enThe Arcade Learning Environment: An Evaluation Platform For General Agents (Extended Abstract). p.5.

Bertsekas, D.P. and Tsitsiklis, J.N., 1995. Neuro-dynamic programming: an overview. *Proceedings of 1995 34th ieee conference on decision and control*. IEEE, vol. 1, pp.560–564.

Böhm, N., Kókai, G. and Mandl, S., 2005. An evolutionary approach to tetris. *The sixth metaheuristics international conference (mic2005)*. Citeseer, p.5.

Boumaza, A., 2009. On the evolution of artificial Tetris players [Online]. *The IEEE Symposium on Computational Intelligence and Games*. Milan, Italy, pp.387–393. Available from: https://hal.archives-ouvertes.fr/hal-00397045 [Accessed 2021-04-28].

Breukelaar, R., Demaine, E.D., Hohenberger, S., Hoogeboom, H.J., Kosters, W.A. and Liben-Nowell, D., 2004. Tetris is hard, even to approximate. *International journal of computational geometry & applications* [Online], 14(01n02), pp.41–68. Publisher: World Scientific Publishing Co. Available from: https://doi.org/10.1142/S0218195904001354 [Accessed 2021-05-05].

Brownlee, J., 2020. Understand the impact of learning rate on neural network performance. Available from: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

Burgiel, H., 1997. enHow to lose at Tetris. *The mathematical gazette* [Online], 81(491), pp.194–200. Publisher: Cambridge University Press. Available from: https://doi.org/10.2307/3619195 [Accessed 2021-04-28].

Carr, D., 2005. enApplying reinforcement learning to Tetris. p.15.

Dertat, A., 2017. *Applied deep learning-part 4: Convolutional neural networks*. Available from: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

Faria, N., 2019. Tetris-ai. https://github.com/nuno-faria/tetris-ai/blob/master/tetris.py.

Farias, V.F. and Van Roy, B., 2006. enTetris: A Study of Randomized Constraint Sampling [Online]. In: G. Calafiore and F. Dabbene, eds. *Probabilistic and Randomized Methods for Design under Uncertainty*. London: Springer, pp.189–201. Available from: https://doi.org/10.1007/1-84628-095-8_6 [Accessed 2021-04-28].

Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C. and Legg, S., 2019. Noisy Networks for Exploration. *arxiv:1706.10295 [cs, stat]* [Online]. ArXiv: 1706.10295. Available from: http://arxiv.org/abs/1706.10295 [Accessed 2021-05-05].

Gabillon, V., Ghavamzadeh, M. and Scherrer, B., 2013. enApproximate Dynamic Programming Finally Performs Well in the Game of Tetris [Online]. Available from: https://hal.inria.fr/hal-00921250 [Accessed 2021-05-05].

Hasselt, H., 2010. Double Q-learning [Online]. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., vol. 23. Available from: https://papers.nips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html [Accessed 2021-08-12].

Hessel, M., Modayil, J., Hasselt, H. van, Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arxiv:1710.02298 [cs]* [Online]. ArXiv: 1710.02298. Available from: http://arxiv.org/abs/1710.02298 [Accessed 2021-04-27].

Kakade, S., 2001. A Natural Policy Gradient. *Advances in neural information processing systems* [Online]. Available from: https://repository.upenn.edu/statistics_papers/471.

Lagoudakis, M.G., Parr, R. and Littman, M.L., 2002. enLeast-Squares Methods in Reinforcement Learning for Control [Online]. In: I.P. Vlahavas and C.D. Spyropoulos, eds. *Methods and Applications of Artificial Intelligence*. Berlin, Heidelberg: Springer, Lecture Notes in Computer Science, pp.249–260. Available from: https://doi.org/10.1007/3-540-46014-4_23.

Liu, H. and Liu, L., n.d. enLearn to Play Tetris with Deep Reinforcement Learning. p.9.

Marblestone, A.H., Wayne, G. and Kording, K.P., 2016. Toward an integration of deep learning and neuroscience. *Frontiers in computational neuroscience* [Online], 10. Available from: https://doi.org/10.3389/fncom.2016.00094.

Markov, A.A. and Nagorny, N.M., 1988. en*The Theory of Algorithms* [Online], Mathematics and its Applications. Springer Netherlands. Available from: https://www.springer.com/gp/book/9789027727732 [Accessed 2021-08-29].

McCulloch, W.S. and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* [Online], 5(4), pp.115–133. Available from: https://doi.org/10.1007/BF02478259.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous Methods for Deep Reinforcement Learning. *arxiv:1602.01783 [cs]* [Online]. ArXiv: 1602.01783. Available from: http://arxiv.org/abs/1602.01783 [Accessed 2021-05-02].

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning. *arxiv:1312.5602 [cs]* [Online].

ArXiv: 1312.5602. Available from: http://arxiv.org/abs/1312.5602 [Accessed 2021-04-27].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. enHuman-level control through deep reinforcement learning. *Nature* [Online], 518(7540), pp.529–533. Available from: https://doi.org/10.1038/nature14236 [Accessed 2021-05-01].

Rummery, G. and Niranjan, M., 1994. On-Line Q-Learning Using Connectionist Systems. *Technical report cued/f-infeng/tr 166*.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritized Experience Replay. *arxiv:1511.05952 [cs]* [Online]. ArXiv: 1511.05952. Available from: http://arxiv.org/abs/1511.05952 [Accessed 2021-05-03].

Schulman, J., Levine, S., Moritz, P., Jordan, M.I. and Abbeel, P., 2015. Trust Region Policy Optimization. *arxiv:1502.05477 [cs]* [Online]. ArXiv: 1502.05477 version: 1. Available from: http://arxiv.org/abs/1502.05477 [Accessed 2021-04-27].

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal Policy Optimization Algorithms. *arxiv:1707.06347 [cs]* [Online]. ArXiv: 1707.06347. Available from: http://arxiv.org/abs/1707.06347 [Accessed 2021-04-27].

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G.v.d., Graepel, T. and Hassabis, D., 2017. enMastering the game of Go without human knowledge. *Nature* [Online], 550(7676), pp.354–359. Number: 7676 Publisher: Nature Publishing Group. Available from: https://doi.org/10.1038/nature24270 [Accessed 2021-04-23].

Simsek, O., Algorta, S. and Kothiyal, A., 2016. enWhy Most Decisions Are Easy in Tetris—And Perhaps in Other Sequential Decision Problems, As Well [Online]. *International Conference on Machine Learning*. PMLR, pp.1757–1765. ISSN: 1938-7228. Available from: http://proceedings.mlr.press/v48/simsek16.html [Accessed 2021-04-28].

Stevens, M. and Pradhan, S., n.d. enPlaying Tetris with Deep Reinforcement Learning. p.7.

Sutton, R.S. and Barto, A.G., 1987. A temporal-difference model of classical conditioning. *Proceedings of the ninth annual conference of the cognitive science society*. Seattle, WA, pp.355–378.

Sutton, R.S. and Barto, A.G., 2018. en*Reinforcement learning: an introduction*, Adaptive computation and machine learning series. Second edition ed. Cambridge, Massachusetts: The MIT Press.

Szita, I. and Lörincz, A., 2006. Learning Tetris Using the Noisy Cross-Entropy Method. *Neural computation* [Online], 18(12), pp.2936–2941. Available from: https://doi.org/10.1162/neco.2006.18.12.2936 [Accessed 2021-05-05].

Tabor, P., 2020. Deep-q-learning-paper-to-code. https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code/tree/master/DQN.

Tesauro, G. et al., 1995. Temporal difference learning and td-gammon. *Communications of the acm*, 38(3), pp.58–68.

Tetris wiki, n.d. Available from: `https://tetris.fandom.com/wiki/Tetris_Wiki`.

Thiery, C. and Scherrer, B., 2009. enBuilding Controllers for Tetris. *Icga journal* [Online], 32(1), pp.3–11. Available from: `https://doi.org/10.3233/ICG-2009-32102` [Accessed 2021-04-28].

Tsitsiklis, J.N. and Van Roy, B., 1996. enFeature-Based Methods for Large Scale Dynamic Programming. *Machine learning* [Online], 22(1), pp.59–94. Available from: `https://doi.org/10.1023/A:1018008221616` [Accessed 2021-05-05].

Van Hasselt, H., Guez, A. and Silver, D., 2015. Deep Reinforcement Learning with Double Q-learning. *arxiv:1509.06461 [cs]* [Online]. ArXiv: 1509.06461. Available from: `http://arxiv.org/abs/1509.06461` [Accessed 2021-05-01].

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K. and Freitas, N. de, 2017. Sample Efficient Actor-Critic with Experience Replay. *arxiv:1611.01224 [cs]* [Online]. ArXiv: 1611.01224. Available from: `http://arxiv.org/abs/1611.01224` [Accessed 2021-05-02].

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. van, Lanctot, M. and Freitas, N. de, 2016. enDueling Network Architectures for Deep Reinforcement Learning. p.9.

Watkins, C.J.C.H., 1989. Learning from delayed rewards.

Watkins, C.J.C.H. and Dayan, P., 1992. enQ-learning. *Machine learning* [Online], 8(3-4), pp.279–292. Available from: `https://doi.org/10.1007/BF00992698` [Accessed 2021-08-29].

Weisberger, M., 2016. The bizarre history of 'tetris'. Available from: `https://www.livescience.com/56481-strange-history-of-tetris.html`.

Appendix A

Dictionary of action correction

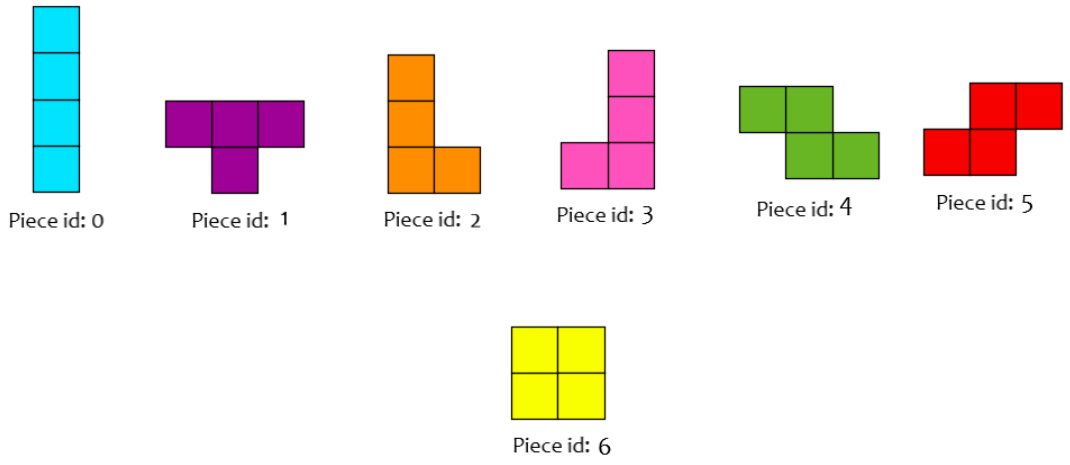

Figure A.1: Tetris Pieces with their corresponding piece ids

piece 0= {(7,0):(6,0),(7,180):(6,180),(8,0):(6,0),(8,180):(6,180),(9,0):(6,0),(9,180):(6,180) }
piece 1= {(8,0):(7,0),(8,180):(7,180),(9,0):(7,0),(9,180):(7,180),(9,90):(8,90),(9,270):(8,270)}
piece 2= {(8,90):(7,90),(8,270):(7,270),(9,0):(8,0),(9,180):(8,180),(9,90):(7,90),(9,270):(7,270)}
piece 3= {(8,90):(7,90),(8,270):(7,270),(9,0):(8,0),(9,180):(8,180),(9,90):(7,90),(9,270):(7,270)}
piece 4= {(8,0):(7,0),(8,180):(7,180),(9,0):(7,0),(9,180):(7,180),(9,90):(8,90),(9,270):(8,270)}
piece 5= {(8,0):(7,0),(8,180):(7,180),(9,0):(7,0),(9,180):(7,180),(9,90):(8,90),(9,270):(8,270)}
piece 6= {(9,0):(8,0),(9,90):(8,90),(9,180):(8,180),(9,270):(8,270)}

Appendix B

Gameplays

1. DQN Small Board with various Reward functions - https://drive.google.com/drive/folders/16t1sD0vWZR8Rzvo3LX9mtmNZ79kV6IRH?usp=sharing
2. DQN Full Board with various Reward functions - https://drive.google.com/drive/folders/1nZrZ-vpvr0KhQ-IkgO2nKPn7DSmBlxZK?usp=sharing
3. Double DQN, Small and Full – https://drive.google.com/drive/folders/1cEcQro9XRtatClTzLaexFSlACbi_L6Sc?usp=sharing
4. Duelling DQN, Small and Full - https://drive.google.com/drive/folders/1ktbH5JByFmyZNMxCU3CblHnDcxq8ID8_?usp=sharing
5. Double Duelling DQN, Small and Full - https://drive.google.com/drive/folders/1GRgCDCqscuqTcC4gQ6xftHL4ilFKGZVp?usp=sharing
6. Best Performing agent in the Full Board - https://drive.google.com/file/d/1MNQQUCHKBwwUV1CJok_UMixIKkp9u0pL/view?usp=sharing

Appendix C

Ethics Form

Department of Computer Science

12-Point Ethics Checklist for UG and MSc Projects

Student	Kyriacos Papayiannis
Academic Year or Project Title	Learning to play Tetris with Deep Reinforcement learning 2020/2021
Supervisor	Dr.Guy McCusker

Does your project involve people for the collection of data other than you and your supervisor(s)? NO

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Will you prepare a Participant Information Sheet for volunteers?* YES / NO
   This means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO
   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Will there be any intentional deception of the participants?* YES / NO
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO
   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature

of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* YES / NO
Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete. If your data collection is entirely anonymous and does not include collection of personal data you do not need to collect a signature. Instead, you should include a checkbox, which must be checked by the participant to indicate that informed consent has been given.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* YES / NO
Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Will you be offering any incentive to the participants?* YES / NO
The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Will you be in a position of authority or influence over any of your participants?* YES / NO
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Will any of your participants be under the age of 16?* YES / NO
Parental consent is required for participants under the age of 16.

10. *Will any of your participants have an impairment that will limit Their understanding or communication?* YES / NO
Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* YES / NO
All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. *Will you have a data management plan for all recorded data?* YES / NO
Personal data is anything which could be used to identify a person, or which can be related to an identifiable person. All personal data (hard copy and/or soft copy) should be anonymized (with the exception of consent forms) and stored securely on university servers (not the cloud).