Perfecting Tetris using Deep Reinforcement Learning

Andreas Kafkalias

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

Perfecting Tetris using Deep Reinforcement Learning

Submitted by: Andreas Kafkalias

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

This study attempts to crack the game of Tetris using model-free Deep Reinforcement Learning and compares the performance of value-based methods against policy-based methods. In this project, we define perfecting the game of Tetris as clearing simultaneously as many lines are possible. The main algorithms used are Deep Q-Networks, DQN, and Proximal Policy optimizations, PPO. We experiment with different state representations of the game, the effect of the discount rate and the importance of a good reward function. Most of the early implementation and testing is done on multiple small size boards, instead of the normal 20x10, to increase training efficiency and allow for parameter tuning. The policy-based methods are able to perfect the game of Tetris using a smaller board clearing the maximum amount of lines simultaneously and overall 1000 lines per episode, but fail to replicate their performance on the big board, clearing less than 10 lines. The value-based methods create an endurance behavior and clear lots of single lines to maximize game play time. This results in clearing over 2000 lines per episode on the small board and over 10 lines on the big board. Overall, this project showcases the difficulties and complications that model-free Deep Reinforcement Learning has to face. Any code or relevant information about the implementations can be found in https://github.com/Andreas430/Tetris

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my awesome supervisor Dr. Guy McCusker for his in-valuable guidance and help throughout every step of the project. I would also like to thank my family for supporting and helping throughout this tough journey.

Chapter 1

Introduction

The game of Tetris is a significant benchmark for research in artificial intelligence and machine learning. It is considered one of the most challenging games for artificial intelligence due its enormous search space and the difficulty of being a sequential decision problem. The main aim of the project is to create a computer program that plays the game at a professional level and understand the critical point of success or failure of the algorithms.

This project will attempt to crack the game using Reinforcement Learning Methods. Reinforcement Learning is the area of Machine Learning where an agent interacts with an environment and updates its strategy based on the feedback received. The goal of the agent is to maximize future rewards. More specifically, this project will use Deep Reinforcement Learning, which combines the fields of Deep Learning and Reinforcement Learning. The main objective is to understand and visualize the policy (strategy) of such a simple yet complex game.

The agents that are used throughout the project fall in the model free area of Reinforcement Learning. One of the main issues of such agents is the computational expense and long-training times. The preliminary analysis, implementation and testing will be on boards with smaller dimensions than the actual Tetris board, which are computationally cheaper to solve and require less-training. This will allow for easier parameter tuning and testing. Furthermore, multiple state representations will be investigated. Reward functions based on future rewards and board stability will be implemented.

Early attempts to solve the game date back to the 1990s, a few years after the release of the game. Even after all those years, solving the game of Tetris today is not trivial. Various approaches proved to be successful over the past decades. However, the game is still used to research sequential decision making under uncertainty. After an analysis of the numerous methods of Deep Reinforcement Learning and the previous attempts of solving the game, various approaches that showcased great potential will be implemented while the difficulties and obstacles encountered will be discussed.

Chapter 2

The game of Tetris

2.1 A brief History

In 1984, Alexey Pajitnov and Dmitry Pavlovsky, were working as computer engineers at the Computing Center of the Russian Academy of Sciences (Fahey, 2003). The two young Russians were interested in developing and selling computer games. After testing numerous different games, Alexey was inspired by the Ancient Greek puzzle game, Pentominos. The game involved arranging puzzle pieces made of five squares. Alexey quickly realized the geometry introduced by the twelve different five-square shapes was too complex, so the game was modified to use Tetraminoes instead, which were made of four squared blocks. Even though Pajitnov was originally assisted by Dmitry Pavlovsky, later he worked on the project with Vadim Gerasimov, a 16-year-old high school boy at the time. Gerasimov was in charge of the graphical design of the game. By 1987 Tetris started to spread globally. Nowadays, Tetris can be found on almost every computer platform.

2.2 The rules

The game is played on a two-dimensional grid which is initially empty (Demaine, Hohenberger and Liben-Nowell, 2003). The standard grid size is 10 cells wide and 20 cells high. Blocks of different shapes, called Tetriminos, fall on top of each other, stacking up on each other. During the game, the player must strategically choose one of the following actions:

1. Request to translate left by one column.
2. Request to translate right by one column.
3. Request to do a counterclockwise rotation.
4. Request to instantly drop the piece.

The goal is to clear as many lines as possible by completing full horizontal rows of blocks. Clearing many rows at the same time gives the player a higher score. In fact, clearing four lines at the same time, which is the maximum lines that can be cleared in a single turn, is called “Tetris”. The game finishes when the Tetriminos are stacked up to a height exceeding the top of the grid. Figure 2.1 shows the standard board during a game when a player makes a ‘Tetris’ and all seven tetriminos along with their respective names.

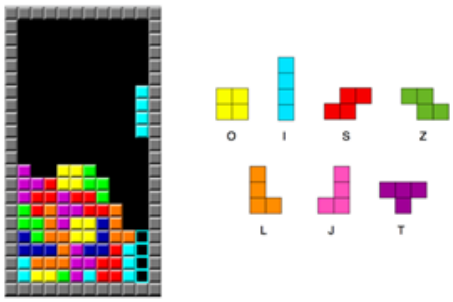


Figure 2.1: A classical board during the game about to make a 'Tetris' (left). The seven possible Tetriminos and their names (right).

2.3 Projects and events inspired by Tetris

Even though Tetris was created as a two-dimensional game, over the years different versions of the game were produced, always based on Tetris' rules. Some adaptations introduce dimensionality to the game, where it can be played in three or four dimensions (Frear, 2009). In fact, even one-dimensional Tetris exist, which was created simply as a math joke.

Over the years different individuals manage to turn buildings into their Tetris board (Fahey, 2003). In 1995, researchers at the University of Delft, played Tetris on a two thousand m2 squared surface area building. In 2000 Steve Wozniak, cofounder of Apple Computers, played Tetris at Brown University, using light in the windows. However, Drexel University's Frank Lee, holds the record as the creator of the worlds 'largest architectural videogame display'. In 2014 Lee, managed to play Tetris in Philadelphia's Cira Centre (Faultstick, 2014), totaled nearly eleven thousand square meters.

In 2010 the Classic Tetris World Championship (CTWC) was hosted for the very first time by the Portland Retro Gaming Expo (Olson, 2020). The contestants go heads up, competing for the highest score in the 1989 Nintendo Version of Tetris. Since then, professional Tetris players and enthusiasts gather every year to crown a Classical Tetris champion.

Figure 2.2: Jonas Neubauer. 7-times Tetris world champion. He recently passed away at the age of 39 due to health issues.

2.4 Tetris effect

Tetris gave its name to a syndrome, the Tetris Effect (Earling, 1996). This occurs when people devote so much time and attention to a single activity that it begins to affect them and pattern their thoughts, dreams, and mental images. Many who end playing the game too much often visualize the game in the real world, thinking about the ways that different shapes can fit together.

Chapter 3

Literature and Technology Survey

3.1 The agent environment Interface

In mathematics, a Markov decision process (MDP) is a discrete-time stochastic control process. MDPs are a formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards (Sutton and Barto, 2018). This is the kind of problem the project is trying to tackle.

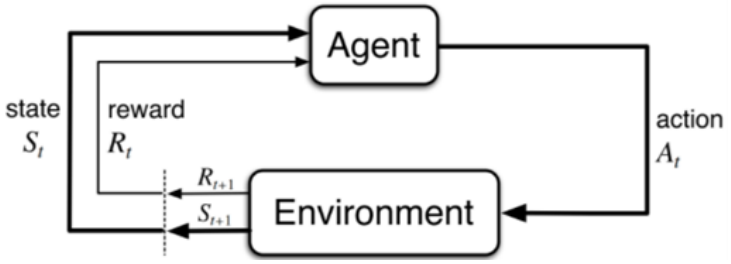


Figure 3.1: The agent-environment interaction in an MDP. Image courtesy of Reinforcement Learning: An Introduction, p38, R. Sutton A. Barto (2018).

Reinforcement Learning control problems that have the markov property are formulated as a Markov Decision Process. In the reinforcement learning framework, an agent is trained to perform a task by interacting with an unknown environment (Sutton and Barto, 2018). The agent can be described as the decision maker and the environment is the everything that the agent interacts with. For every action the agent makes, it receives feedback from the environment in the form of rewards, which are special numerical values. The notion of the RL framework is focused on gradually improving the agent's behavior and estimating its policy by maximizing the total long-term expected reward.

The two main approaches for Reinforcement Learning use Dynamic Programming, DP (Bellman, 1952), and Monte Carlo methods, MC (Metropolis and Ulam, 1949). DP requires a complete knowledge of the environment or all possible transitions while MC works on a sampled state-action trajectory, τ, of a full-episode.

$$\tau = (S_0, A_0, R_0, \dots, S_{T-1}, A_{T-1}, R_{T-1}, S_T, A_T, R_T)$$

where S represents the state, A represents the action and R represents the reward. This implies that Learning with DP requires just a one-step transition while MC methods waits for episodic ends. Temporal Difference learning, TD (Sutton and Barto, 1998), are a hybrid method of both MC and DP. TD methods sample from the environment like MC methods and perform updates based on the current estimates, like DP methods.

A dilemma with Reinforcement Learning is the exploration vs exploitation tradeoff (Bubeck and Cesa-Bianchi, 2012). The agent must explore the environment and gain information that may lead him to a better decision. On the other hand, the agent wants to achieve the highest possible outcome by following the optimal decisions it has learned so far. This issue is addressed by choosing actions based on their probabilities of occurring or introducing a method called Epsilon-Greedy. In Epsilon-Greedy methods, the agents choose actions based on an epsilon, ε, percentage. The higher the ε, the more likely the agent chooses a random action. By choosing a lower epsilon, the agent can explore while choosing the optimal decisions most of the time. Another similar method to the Epsilon-Greedy is the Epsilon Decreasing Methods which have a very high ε at the start to allow the agent to explore, and then gradually decay it to maximize the outcome.

3.2 Value based methods

3.2.1 Tabular Methods

The goal of a Reinforcement Learning agent in a value-based method is to find the policy that optimizes the expected return. Value based methods achieve that by calculating values for each state or state-action pair and then following the path that gives the highest values (Fleming and Rishel, 1975). The value function for a given state s, $V^{\pi}(s)$ can be defined as:

$$V^{\pi}(s) = E_{\pi}\{G_t \mid S_t = s\}$$

where V^{π} is the value of state following policy π and E_{π} denotes the expected value under policy π. S represents a set of all the states in the environment while state s, implies whatever information is available to agent at given time t. G represents the expected reward and can be defined as some specific function of the reward, R, sequence. Another additional concept that must be considered is discounting. The discount rate, $\gamma \in [0, 1]$, determines the value of future rewards. If γ is zero, then the agent maximizes only the next rewards and if $\gamma < 1$ will result in a finite value. For future references, the discount rate will referred as gamma.

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

The optimal expected return for a state can be defined as:

$$V^* = \max_{\pi} V^{\pi}(s) \quad \forall s \in S$$

Another form of the of the value function, is the quality value function, $Q^{\pi}(s, a)$. The quality value function is the expected return starting from that state, taking that action α, and thereafter following π:

$$Q^{\pi}(s,a) = E_{\pi}\left\{G_t \mid S_t = s, A_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right\}$$

Every finite MDP must have at least one optimum policy π^*, which will yield the maximum expected return. The optimal expected return for a state action pair is denoted as:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in S, \forall a \in A$$

The optimal action-value function follows a critical identity known as the Bellman Equation (Bellman, 1952).

$$Q^*(s,a) = E_{s' \sim \varepsilon}\left[r + \gamma \max_{a}(s', a') \mid s, a\right]$$

This is because if the optimal state-action value at the current state was known for all possible actions, then the optimal strategy is to select the action that maximizes $r \ + \gamma Q^*(s\prime, a\prime)$.

The agent improves its policy and value function by constantly updating them through feedback given from the environment.

Another useful function is the advantage function $A^{\pi}(s,a)$, which is a relative measure about the importance of each action relative to its state and can be denoted as the difference between $Q^{\pi}(s,a)$ and $V^{\pi}(s)$:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Popular value-based algorithms include Q-learning (Watkins and Dayan, 1992) and State-Action-Reward-State-Action, SARSA (Rummery and Niranjan, 1994). Both algorithms update their state-action value function based on the next state-action value. This technique is called bootstrapping. Both techniques are very similar but differ in one way. Q-learning is an off-policy that updates the values using a greedy policy based on the current state-action values while SARSA is an on policy which updates the values based on the action chosen by the current policy. The general form for both equations can be represented as:

$$A^{\pi}(s,a) \ = \ Q^{\pi}(s,a) \ - \ V^{\pi}(s)$$

Where α is the learning rate and δ is the temporal difference.

Q-learning:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \ (\ r_t + \gamma \ \times \max a \ Q(s_{t+1}, \ a) - Q(s_t, a_t))$$

SARSA:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \ (\ r_t + \gamma \ \times \ Q(s_{t+1}, \ a_{s+1}) - Q(s_t, a_t))$$

The previous algorithms work best when the number of states is small and finite because the agent must experience every state-action pair to learn about it. These methods are called

Tabular. Most tabular methods use tables and arrays to store all the possible state-value actions. In most cases of practical interest there are large state spaces. The memory required is too large and not all state-action pairs can be reached so tabular methods are impractical most of the time.

3.2.2 Function approximation methods

Large state spaces require a lot of time and data to fill them accurately. Such problems require to find a good approximate solution. Function approximation methods address this issue. It is important to think each update as an example of desired input-output behavior. Similar state-actions will exhibit similar behavior and therefore the desired action should be the same. An example of a function approximation method is the Linear function approximation (Sutton and Barto, 2018). It is one of the most important and most basic function approximations:

$$\hat{v}(s, w) = \sum_{i=0}^{d} w_i x_i(s) = w^T x(s), \quad x(s) \in R^d, w \in R^d$$

Each state pair s can be represented with a feature vector $x(s)$ and the function parameter vector w, map states to values and has the same length as $x(s)$. The performance of the function is assessed using a loss function, such as the Mean Squared Error (MSE), and are usually optimized using stochastic gradient descent (Robbins and Monro, 1951). For SGD to take place, the function must be differentiable to calculate the gradient. Step functions and delta functions are not a valid choice of function approximation since they are not differentiable. The gradient of the loss functions is calculated with respect to each of the w parameters and the function shifts in the direction which lowers the loss.

This is very familiar to world of Machine learning due to its similarity to Supervised Learning. Function approximation methods expect to receive examples of the aforementioned input-output behavior of the function they are trying to approximate. Some other examples of the different functions that could be used are Decision Trees and Neural Networks, also known as Deep Reinforcement Learning and will be discussed in depth later in section 3.5.

3.3 Policy based methods

Value-based methods derive their policy based on action-value function. This assumes that the optimal policy is deterministic. However sometimes, it can be stochastic. Policy-based methods directly search the policy-space for the optimal policy $\pi^*(a|s)$ (Sutton and Barto, 2018).

Policy based methods also follow the input-output behavior. The policy can be represented as a differentiable function of the state s, with parameters θ. The agent chooses actions based on probabilities. The probability of choosing an action a given state s and policy parameters θ can be denoted as $\pi_\theta(a|s)$. The action probabilities are adjusted based on whether the policy performed worse or better than expected. Unlike function-based approximation, policy-based methods optimize the policy using stochastic gradient ascent, by maximizing an objective function. The main issue is finding a good loss function to compute the effectiveness of the policy.

3.3.1 Policy gradients

Policy gradient methods update the policy by maximizing some objective function, $J(\theta)$ (Baxter and Bartlett, 2000). MSE in not suitable here since the outcome of the function is not a simple value. A more appropriate loss function is the undiscounted expected return. The better the agent performs and the better the policy, the higher the expected reward.

$$J(\pi_\theta) = E_{\tau\sim\pi_\theta}[G(\tau)]$$

The policy parameters can be updated to maximize $J(\theta)$ are updated using gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla J(\theta_t)$$

where α is the learning rate and $\nabla J(\theta_t)$ is the gradient of the objective function at point t. Assuming undiscounted expected reward as the objective function, the policy gradient theorem (Sutton et al., 1999) estimates the policy gradient to be:

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{=0}^{T} \nabla_\theta \log \pi_\theta(A_t \mid S_t)\, G(t)\right]$$

By taking the sample mean over many trajectories, $\tau \in D$, the policy gradient transforms to:

$$\nabla_\theta J(\pi_\theta) = \frac{1}{|D|} \sum_{\tau\in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t \mid S_t)\, G(t)$$

This procedure is the REINFORCE algorithm (Williams, 1987), otherwise known as Monte-Carlo policy differentiation. The learning in a REINFORCE algorithm happens offline, as the agents waits for the total expected reward $G(t)$, which implies that the agent updates its policy after an episodic end. REINFORCE method scales policy gradient by total rewards. This introduces high variance in the magnitude of the updates.

Agents should only reinforce actions based just on the consequences of those actions. Rewards obtained before actually taking an action should have no influence on how good that action was. Using the total expected reward $G(t)$ is a flaw in the REINFORCE algorithm. Therefore, the total reward can be modified, and base actions or rewards obtained after they are taken.

$$G(t) = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$$

Overall, policy gradients methods work by computing an estimator of the policy gradient and optimizing it through gradient ascent with the most common gradient estimator having the form of:

$$\hat{g} = \widehat{E_t}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \widehat{A_t} \mid\right.$$

where $\widehat{A_t}$ stands for an advantage function.

3.3.2 Actor-Critic Methods

The variance of the updates can be reduced by adding or subtracting a baseline function. Any function will do, as long as the expectation remains the same. In fact, any function which depends only the state will do because:

$$E\left[\nabla_\theta \log P_\theta(x)\right] = 0$$

where P_θ is a parameterized probability distribution over random variable x.

Methods that use value-based functions as the baseline are called Actor-critic methods, AC (Sutton and Barto, 2018). This allows bootstrapping with policy-based functions taking away their monte Carlo effect. Actor-critic methods approximate the optimal action value function while searching the policy space for the optimal policy at the same time. The general form of an AC method is:

$$\nabla_\theta J\left(\pi_\theta\right) = E_{\tau\sim\pi_\theta}\left[\sum_{=0}^{T} \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right)\left(\sum_{t'=t}^{T} R\left(s_{t'}, a_{t',s_{t'+1}}\right) - V_\pi\left(s_t\right)\right)\right]$$

3.4 Model Free vs Model-Based RL

A very critical point in a Reinforcement Learning algorithm is whether the agent knows a model of how the environment works. In this case a model is simply a function that allows the agent to take an action from its current state and observe the next states along with possible rewards (Atkeson and Santamaria, 1997). The agent can use this information to 'plan ahead' and decide between its options. These methods are called Model-Based. A famous approach that uses Model-Based RL is AlphaGO that shook the world in 2015 when it was able to beat the world champion in the game of GO, a feat previously thought to be impossible decades ago. AlphaGo and its successors will be discussed later in section 4.1.2

However, Model-Based techniques suffer from model bias (Deisenroth and Rasmussen, 2011) because they assume that the model sufficiently and accurately resembles the real environment. This implies that the agent performs well based on the model and sub-optimally in the actual environment. Learning a model can be very computationally expensive, is fundamentally very hard and can still fail.

On the other Model-Free techniques do not have the possible benefits of Model-Based but tend to be simpler to tune and implement. Furthermore, Model-Free techniques are way more common, therefore the have been significantly more developed and examined.

3.5 Deep Reinforcement Learning

Neural Networks NNs, represent incredibly powerful machine learning techniques, and they are used to solve many real-world problems. NNs are the core of Deep Learning. The basic theory for Neural networks was first proposed in the 1940s (McCulloch and Pitts, 1943) and they were well established by the 1980s. Other key factors of NNs such as Stochastic Gradient Descent and the different architectures have been around for a while. However, Deep Learning was only really successful the past few years.

Deep Learning models, require a lot of data and are very computationally expensive (Schmidhuber, 2015). Thanks to the internet and powerful hardware/software developed over the last decade, neural networks were able to solve problems thought to be impossible. As discussed before in section 3.2.2, Neural networks can be used in function approximation methods. By feeding raw inputs into deep neural networks, it is possible to learn better state representations.

3.5.1 Deep Q-networks

Deep Q-Networks are value based, model free, bootstrapping deep Reinforcement Learning methods. DQN use the concept of Q-learning in combination with deep learning (Mnih et al., 2013). The action-value function is approximated using a deep neural network. It is an off-policy learning algorithm, learning once data has been gathered.

DQN use two network models, one is fixed and the other is constantly updating. The fixed network is used to calculate target values and the moving network is used to choose actions. For a given state s the network outputs a vector of action values $Q(s, .; \theta)$ where θ are the parameters of the moving network. The target network with parameters $\theta-$, is updated every τ steps so that the $\theta_t^- = \theta_\tau$. The target used by DQN can be denoted as:

$$Y_t^{DQN} = R_{t+1} + \gamma Q \max_a Q\left(S_{t+1}, a; \theta_t^-\right)$$

The loss can be computed by comparing the outputs of both networks. Large errors between the expected q value and the predicted q value can result in large updates to network weight. The Q-network can be trained by minimizing the loss at every iteration.

$$L_i\left(\theta_i\right) = E_{s,a\sim\rho(\cdot)}\left[\left(Y_i^{DQN} - Q\left(s, a; \theta_i\right)^2\right)\right]$$

The gradient of the loss function with respect to the weights can be derived by differentiating the loss function:

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = E_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right]$$

Computing the differentiated loss function is usually computationally intensive. It is expected to optimize the loss function through stochastic gradient descent. The Huber loss function HuL, can reduce the impact of such errors.

$$HuL_i = \begin{cases} 0.5(y-y)^2 & for|y-\hat{y}| \leq 1 \\ |y-\hat{y}| - 0.5 & otherwise \end{cases}$$

The Huber loss is less sensitive to outliers than the normal squared error loss because it is quadratic for small values of $y - \hat{y}$ and linear for large values.

The DQN behaves complete randomly for the first n number of frames, where n is a set hyperparameter. Then, it uses an epsilon decreasing method, discussed in section 3.1, where the epsilon falls from a set epsilon max to a set epsilon min, where both set values are also hyperparameters of the agent. This allows the agent enough exploration time, to understand the different state representations.

Furthermore, the most of the DQN agents use clipped rewards techniques, depending on the desired outcome, so that the learning is more consistent. For example, the game of Tetris bases its score based on how fast the blocks fall. If the purpose of the DQN is to clear as much lines as possible, the agent will consider all lines cleared as the same number of points, giving priority to the best position

As the DQN agent interacts with environment, it stores the (s, a, r, $s\prime$) in an experience replay buffer. Then the agent trains by sampling random batches of a fixed size (usually 32 or 64) from the replay buffer. Training in such a way gives the agent various advantages over standard online Q-learning. Firstly, every step taken by the agent, can potentially be used in many updates, which improves data efficiency. Furthermore, because of the random sampling, the agent is forced to learn from non-consecutive samples which breaks the correlation and reduces the variance of the update. In addition, the current parameters will determine the next data sample that the parameters will use to train.

However, a significant drawback of the DQN agent is the long training time. Having a lot of randomness in the environment means that training is slow, which is a critical component to the researcher's abilities to carry out experiments. Tuning the neural networks and the hyper-parameters takes up to 12-14 days, even with the most powerful GPU's (deepmind).

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3.2: Pseudocode for the Vanilla DQN in the original paper by Mnih et al. (2013)

Everything discussed so far about DQN can be considered the vanilla method. Over the years different variations of DQN were created, focusing on different aspects at a time.

Experience replay will randomly sample batches from its memory, regardless of their significance. However, a prioritized experience replay chooses which transitions are replayed and therefore can improve training efficiency (Schaul et al., 2015). The ideal criterion when sampling is the Temporal Difference error. The bigger the error, the more the agent can learn from of it. However, the TD-error has some drawbacks. It has a high sensitivity to noise spikes, such as stochastic rewards. In addition, high errors shrink very slowly which will result in some transitions being picked repeatedly. This introduces over-fitting. To solve this issue a stochastic sampling method was introduced. The probability of a sampling transition can be defined as:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

where $p_i > 0$ is the priority of transition i, and a determine how much prioritization is used, with $\alpha = 0$ corresponding to a uniform case. The general form of the sampling equation with probability p_t is:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}} \left(S_{t+1}, a' \right) - q_{\theta} \left(S_t, A_t \right) \right|^{\omega}$$

where ω is a hyper -parameter that determines the shape of the distribution.

The Double Q learning algorithm was first introduced to tabular methods and it was then generalized to work with deep neural networks, called the Double DQN (Van Hasselt, Guez and Silver, 2016). Normal DQN uses the same model to calculate the values and to select and evaluate an action. This introduces the following issue. Consider a network that produces an output for two actions, alpha and beta. If action alpha has a higher value, it will be chosen every time, resulting in higher alpha value. If for any reason in the future, action beta becomes the better choice under some conditions, the model will not adapt easily due to the high alpha value. Double DQN solves this problem by evaluating the greedy policy according to the online network, but the value is estimated using the target network. This reduces overestimations by decomposing the max operation in the target Y_t^{DQN} ,into action selection and action evaluation.

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q \left(S_{t+1}, \arg\max_a Q \left(S_{t+1}, a; \theta_t \right), \theta_t^- \right)$$

Dueling DQN networks (Wang et al., 2016) make use two different estimators, one of the state value function $V^{\pi}(s)$ and one for the state-dependent action advantage function A^{π}.

$$A(s, a; \theta, \alpha) = Q(s, a; \theta, \alpha, \beta) - V(s; \theta, \beta)$$

where α and β are the parameters of each stream.

The dueling architecture allows the agent to distinguish which states are valuable, without investigating the effect of each action on the state. This helps to identify the proper action faster in states where actions do not affect the environment in any way. Following the definition of advantage, it can be concluded that:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

However due to error of identifiability, where the value of V and A cannot be retrieved when given Q, the equation transforms by forcing he advantage function estimator to have an advantage of zero at the chosen action:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left[A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum A(s, a; \theta, \alpha) \right]$$

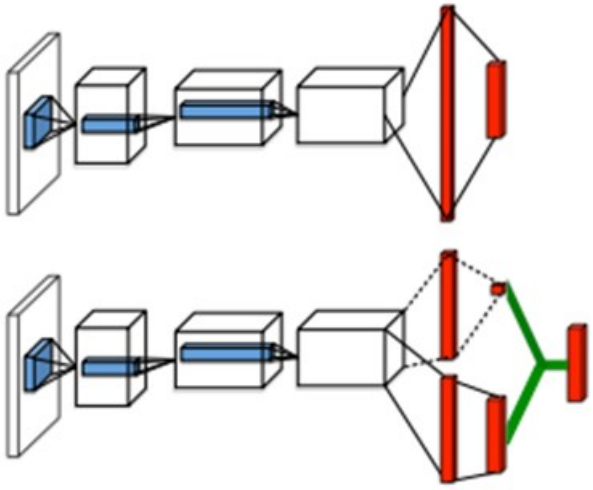


Figure 3.3: The general architecture of single stream Q-network (top) and the dueling Q-network (bottom). The two streams of the Dueling DQN represent the state value function and the advantage function. Published in the original Dueling DQN paper by Wang et al. (2016)

Multi-step learning is an extension to the DQN agent where instead of returning a single reward at the next bootstrap, the agent makes n steps and returns the n reward. This technique is known as Multi-step learning (De Asis et al., 2018) and can be defined by minimizing an alternative loss:

$$\left(R_t^{(n)} + \gamma_t^{(n)} \max_{ar} q_\theta\left(S_{t+n}, a'\right) - q_\theta\left(S_t, A_t\right)\right)^2$$

In environments where many actions must occur to obtain a reward, such as clearing a row in Tetris, there are limitations of exploring using the epsilon greedy policy. Adding parametric noise to the weight parameters of the neural network (Fortunato et al., 2017) introduces greater variability to the decision making which has potential for exploratory actions. In addition, the noise added can be tuned by the algorithm automatically which eases any hyper parameter tuning. This method is known as Noisy Nets.

The previous techniques mentioned above can be integrated and into a single agent, called Rainbow DQN (Hessel et al., 2018)

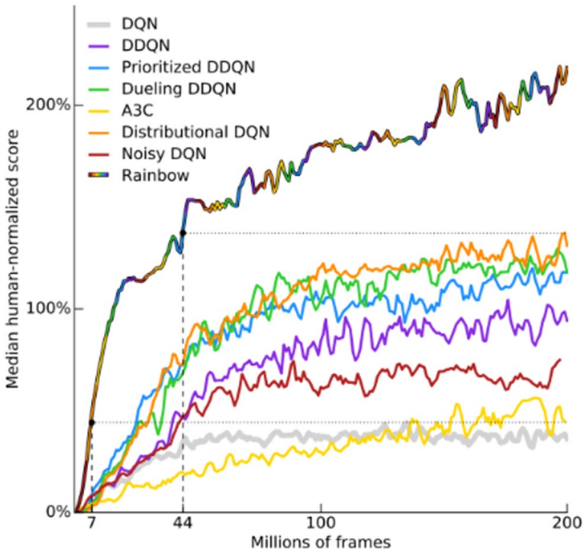


Figure 3.4: 8 Median huma-normalized performance across 57 Atari games. Comparison between Rainbow DQN and the other published baselines. Published in the original Rainbow DQN paper by Hessel et al. (2018)

3.5.2 Proximal Policy Optimization

Proximal Policy Optimizations, PPO (Schulman et al., 2017), is a policy gradient, model free deep RL method. The core purpose of PPO is to strike a balance between ease of implementation, sample efficiency and ease of tuning. PPO learns directly from whatever the agent encounters in the environment, unlike Deep Q learning methods which learn from stored offline batches of experience. PPO methods try to solve the policy by taking the biggest possible improvement from the data collected over set small number of actions, also called a minibatch, without shifting the policy to much to destroy the performance. Policy gradient methods are typically less sample efficient because they use the collected sample experience just once. There are two different variants of Proximal Policy methods, PPO-penalty and PPO-clip.

Both variants were designed based on another policy gradient method, Trust Region Policy Optimization TRPO (Schulman et al., 2015), due to its reliability and data efficiency. TRPO maximizes the following objective function:

$$\theta maximize = \widehat{E_t}\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{old}}\left(a_t \mid s_t\right)}\widehat{A_t}\right]$$

$$\text{subject to } \widehat{E_t}\left[KL\left[\pi_{\theta_{old}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right]\right] \leq \delta$$

which is identical to the vanilla objective function subject to a Kullback-Leibler divergence constraint, and it is one of the famous divergence measures between two probability distributions. By considering two probability distributions P and Q, where P is the data and the Q is a model of the data, the KL divergence can be explained as the average difference of the number bits needed for encoding samples of P using code optimized for Q. This makes sure that the new updated policy does not move far away from the old policy to avoid massive shifts in the policy.

However, the KL constraint adds additional overhead which can sometimes lead to undesirable training behavior. PPO includes this constraint directly into the optimization objective.

The ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, also written as r_θ, is the probability ratio between the updated policy output and the previous old version of the policy. If the action has a higher likelihood based on the current policy, $r_\theta > 1$. The main objective function of PPO-clip function can be denoted as:

$$L^{CLIP}(\theta) : \widehat{E_t}\left[\min\left(r_t(\theta)\widehat{A_t}, clip\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\widehat{A_t}\right)\right]$$

The first term inside the min operator is the vanilla objective function which pushes the policy towards actions that yield a high positive advantage. The second term is a clipped version of the objective function based on a hyperparameter ε, called the clipped range. Figure 3.5 demonstrates this effect and shows how the PPO clips the reward of high r_θ, with positive and negative advantage functions.

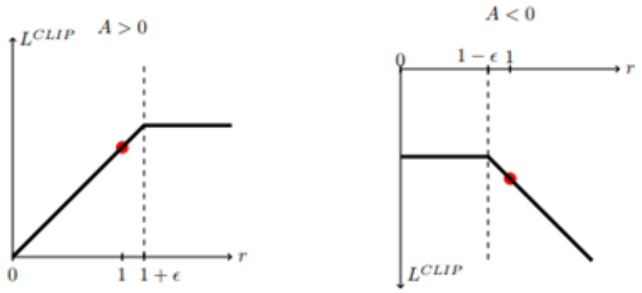


Figure 3.5: Positive advantages vs Negative advantages plots of the surrogate LCLIP as a function of the probability ratio r. The red circles demonstrate the starting point for the optimization. Published in the original PPO paper by Schulman et al. (2017)

The PPO-penalty variant is the alternative method, or sometimes in combination with the clipped version, and it uses a penalty on KL divergence. By the use of a penalty coefficient it achieves a target value of the KL divergence. However, it was found to perform worse than the PPO-clip and it was used as a baseline by its creators, to showcase the impressive results of the PPO.

A popular style of PPO algorithms with recurrent neural networks, uses the current policy for t timesteps and performs an update based on the collected samples. The advantage estimator in this case, does not look beyond timestep T.

$$\widehat{A_t} = -V(s_t) + r_t + \gamma r_{t+1} + ... + (\gamma\lambda)^{T-t+1} + \gamma^{T-t}V(s_T)$$

PPO algorithms use a fixed-length trajectory segments to train. The pseudocode for PPO is shown in Figure 3.6.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 3.6: Pseudocode for PP0 in the original paper by Schulman et al. (2017)

3.5.3 Asynchronous methods for Deep RL

The Asynchronous Advantage Actor-Critic algorithm A3C (Mnih et al., 2016), is a model free, policy based deep Reinforcement method that uses two Neural Networks as functions approximators for the actor and the critic. The policy of A3C is updated using the value function $V(s_t; \theta_v)$ as the baseline function. Like PPO, the policy is updated based on collected samples after t_{max} timesteps or when a terminal step is reached. The update can be denoted as:

$$\nabla_\theta, \log \pi \left(a_t \mid s_t; \theta'\right) A \left(s_t, a_t; \theta, \theta_v\right)$$

where θ are the parameters of the actor NN, θ_v are the parameters of the critic NN and $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function denoted by:

$$\mathrm{A}\left(\mathrm{s_t}, \mathrm{a_t}; \theta, \theta_\mathrm{v}\right) = \sum_{\mathrm{i}=0}^{\mathrm{k}-1} \gamma^{\mathrm{i}} \mathrm{r_{t+1}} + \gamma^{\mathrm{k}} \mathrm{V}\left(\mathrm{s_{t+k}}; \theta_\mathrm{v}\right) - \mathrm{V}\left(\mathrm{s_t}; \theta_\mathrm{v}\right)$$

where k is upper bounded from t_{max} and can possibly vary from state to state.

The main idea behind A3C is the asynchronous implementation. Consider Deep Q networks, where a single agent is represented by a single network which interacts with a single environment. A3C utilizes multiple incarnations based on this idea to learn more efficiently. A3C uses a global network and multiple worker agents which each have their own set of network parameters. Every agent interacts with a copy its own environment, and all the agents 'work' together at the same time. The experience available for training is more diverse since each of the agent gains experience independent of the others. This speed up the process because more work gets done at the same time.

3.6 Tetris in Reinforcement Learning

Tetris can be formulated as a Markov decision problem. Using the Reinforcement learning domain, the environment can be a simulation playing the game, the agent can be any algorithm choosing the current action of the falling piece and the state can be a way of describing the actual Tetris board, either by using important features or visual representations. Furthermore, the game can be described as a stochastic environment as the pieces sequence involves

randomness. Even though some versions of the game inform the player of the future shapes, he can never know the full list of blocks coming. This randomness makes every game unique, while increasing the complexity.

3.6.1 Possible Board Configurations

Even though the game has a simple gameplay, the different combinations of tetriminoes can produce a massive number of game states. Most AI algorithms in reinforcement Learning investigate all possible states and decide the optimal action. With a board of size of 20 x 10 tiles, the upper bound of possible game states is in the order of 2^{200} states (Algorta and Şimşek, 2019). This makes Tetris very computationally expensive to solve. Therefore, techniques of minimizing the number of states visited are crucial for efficiency.

However, the number of possible board states (configurations) is much less than the upper bound stated above, 2^{200} (Fahey, 2003). Given that each piece adds exactly 4 cells and each row completion eliminates 10 cells from the board means that the number of occupied cells will always be even. Therefore, the board states are halved (2^{200}-> 2^{199}). Furthermore, other excluded scenarios that must be noted are:

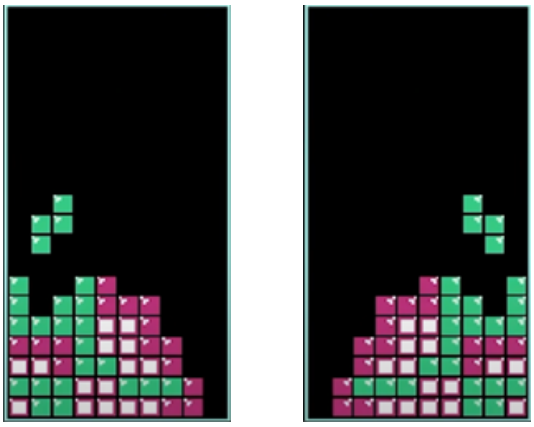
1. Filled rows as they are eliminated before the move is completed.
2. Configurations that include empty rows or empty space below non-empty rows or non-empty space.

Excluding the filled row case, the number of states in each row drops from 1024 (2^{10}) to 1022. Continuing this analysis, taking into consideration both points mentioned above, it can be computed that the maximum number of states are around 1022^{19} x 1023 which approximates 96.25% of 2^{199} of even states. In practice there is a small number of states that are impossible to reach due to the tetriminoes geometry. Since the grid is a rectangle with dimensions 20x10, it is clear to say that there are 2 lines of symmetry which divides it two identical parts. Symmetry is fundamental in creating patterns and organizing the environment. The vertical symmetry of the grid is very helpful in reducing the possible number of states reached. For example, consider the following example. Figure 3.7 (left) was taken from the Classic world Championship 2018 Final round, where Jonas Neubauer, found the optimal move to get a high score. Figure 3.7 (right) is an exact mirror of the board state. It is clear to say that the optimal move it is the same but mirrored. This divides the possible states by half again. Simply, there is no need to find the optimal move if its mirrored position is already known.

3.6.2 Score system

The standard game of Tetris has 10 difficulty levels depending on the number of lines cleared. The difficulty starts at level 1 and ranks up by clearing 10 lines reaching a max level of 10 after clearing 90 lines. The higher the difficulty, the faster the tetriminoes fall. The more cells the tetriminoes instantly drop and the higher the level, the higher the overall score the player achieves. Clearing a single line gives scores of 40 x level, double scores 100 x level, triple scores 300 x level and making a "Tetris" scores 1200 x level. Otherwise, the score function can be defined as:

$$Score = ((21 + (3 \times current\_level - Free\_Fall\_Iterations))$$



(a) Actual Position (b) Mirrored Position

Figure 3.7: Left - A board state reached by Jonas Neubauer during the 2018 finals | Right - The mirrored position showing that the optimal move is also mirrored.

where Free_Fall_Iterations are the number of cells the tetrimino was not hard dropped. If you are doing a primarily software development project, this is the chapter in which you review the requirements decisions and critique the requirements process.

Chapter 4

Related work in the field

4.1 Attempts at similar games

Reinforcement Learning was successful in mastering different games over the past decades. In fact, DeepMind Technologies, a British Artificial Intelligence subsidiary of Alphabet Inc., demonstrated the capabilities of Reinforcement Learning by creating powerful programs that can play the game of chess and GO. Additionally, they were able to make significant advances in the problem of protein folding, showing the effects of RL in real world problems.

4.1.1 TD-gammon

Perhaps the greatest early success of Reinforcement Learning it's the story of TD-gammon, a computer backgammon program developed in 1992 by Gerald Tesauro (Tesauro, 1995). TD-gammon is a model free RL algorithm that achieved a professional level of play exclusively on RL and self-play by approximating the value function, like Q learning.

However, it was mainly considered it as a special case, (Pollack and Blair, 1997), as similar attempts to replicate the success of TD-gammon on other games such as chess, checkers and GO failed. In fact, these following attempts showcased some of the key weaknesses in model free RL algorithms such as failure of convergence.

4.1.2 Chess and Go

Both the games of chess (10^{50} states) and Go (10^{172} states) exhibit the same computational issues with Tetris. The idea at the start was to search all possible moves in each state and evaluate them for a given depth. Even though some engines were able to perform well in the game of chess, after a lot of computationally expensive training of course, there were far from perfect. In fact, Stockfish 8 (chess engine), can calculate 70 million positions in just a second but can still be beaten. In 2015, DeepMind Technologies, (Silver et al., 2017b), were able to create the very first groundbreaking computer program that can play the board game Go, AlphaGo. The algorithm was first trained using supervised learning from human play, and then underwent extensive training using reinforcement of simulated computer play.

Its successor AlphaZero, (Silver et al., 2017a), is completely self-taught without learning from human games, using just reinforcement learning. AlphaZero is a model based deep RL algorithm that uses two deep neural networks, a policy-based and a value-based network. Once

the networks were trained, they were combined with a Monte-Carlo Tree Search, to provide a lookahead search and narrow down the possible moves. This was groundbreaking as it was able to beat Stockfish 8 in a time controlled 100-game tournament (22 wins, 0 losses, 78 draws). At the time this was outstanding since AlphaZero was able to calculate just around 80 thousand of positions in a second, which is around 1.1% of the moves Stockfish 8 can predict.

4.1.3 Atari games

Most of the agents discussed in section 3.5 were firstly designed by different research teams in DeepMind. The methods were implemented and tested on Atari 2600 games which is a challenging RL testbed with visual input of 210 x 160 RGB video at 60 Hz. Each game had a different set of tasks and rules and the goal of the agents was to successfully learn how to master the game without knowing the rules, just with self-play.

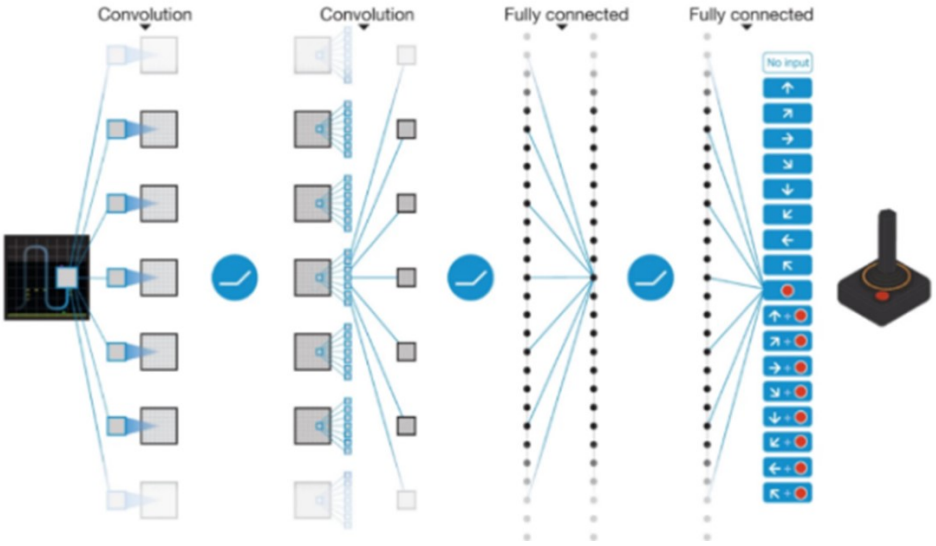


Figure 4.1: Illustration of the Convolutional Neural Network architecture used by the DQN for atari games. Published by Mnih et al. (2015)

The first successful deep learning model to learn control policies using raw data as input was presented and it was the Vanilla DQN discussed in section 3.5.1 (Mnih et al., 2013). At the beginning, the agent was tested on seven different Atari environments and was able to master the games withing a week of training on a single NVIDIA K40 GPU for each game. Later it was tested on multiple Atari environments where it was able to surpass the average human level in more than 60% of the games.

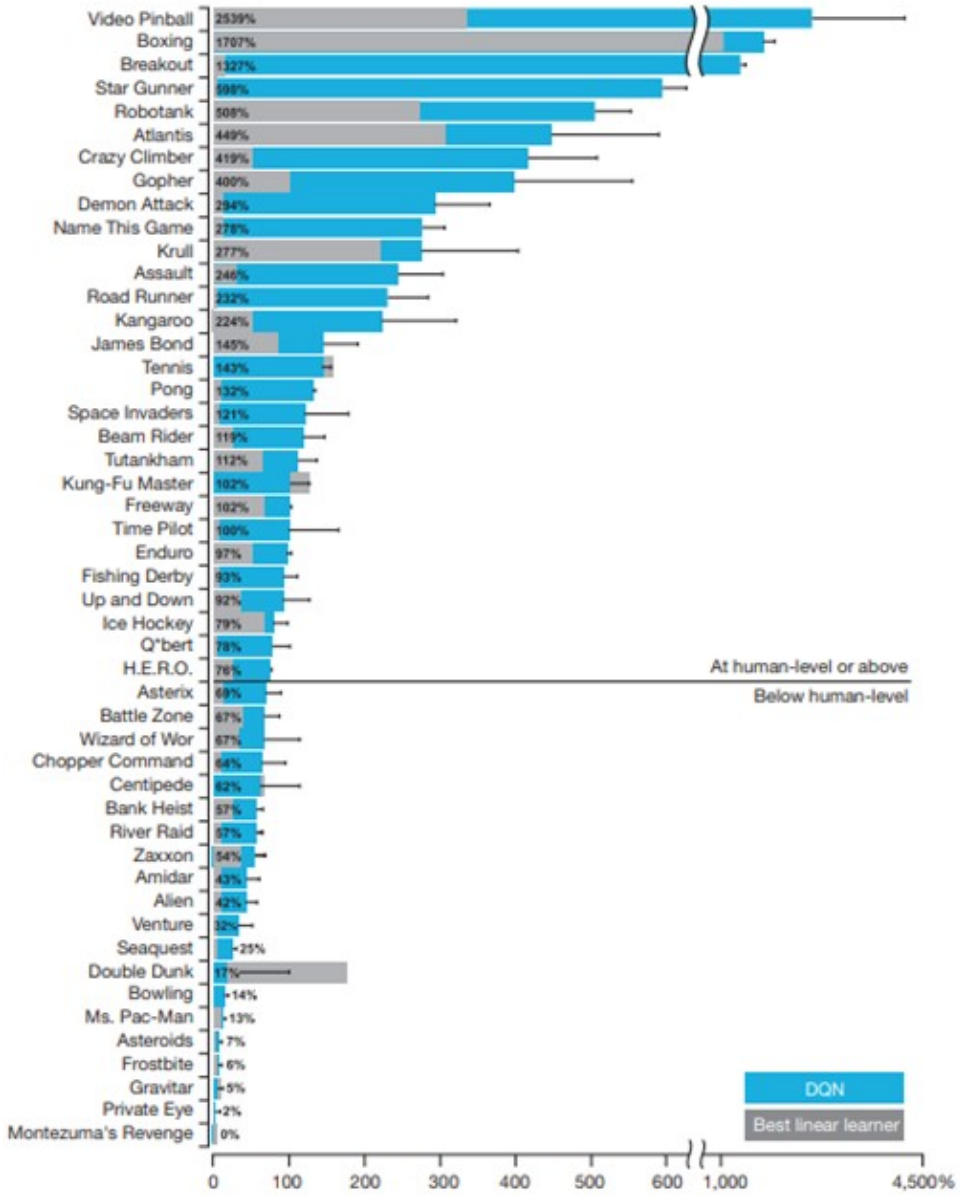


Figure 4.2: Comparison between linear learners and DQN between 49 Atari games. Published by Mnih et al. (2015)

Three years later, DeepMind, (Mnih et al., 2016), demonstrated the idea of asynchronous gradient descent by presenting four different variants of standard Reinforcement Learning algorithms and proved that parallel actor-learners have a stabilizing effect on training. They were able to surpass the current state-of-art (DQN) at the time in half of the training time using their asynchronous variant of actor-critic, the A3C agent discussed in session 3.5.3, by using 16 CPU cores.

The several improvements over the year of the DQN agent were combined to create the Rainbow DQN (Hessel et al., 2018). They were able to demonstrate that multiple improvements can be successfully integrated to a single agent. Rainbow DQN produced better results in a shorter time and proved the importance of such agents for future research.

A newly proposed family of policy gradient methods based on Trust Region Optimization

(TRPO); the Proximal Policy Optimization (PPO) were introduced that year as well (Schulman et al., 2017). They were able to demonstrate that PPO reaches a new satisfactory balance between sample complexity, simplicity, and wall time. By using multiple epochs of gradient ascent to complete each policy update, they were able to reach a better overall performance.

4.2 Attempts at solving Tetris

The game of Tetris presents several complexities for research (Algorta and Şimşek, 2019). Firstly, the score achieved in every game has a large variance which implies that many games must be played to accurately evaluate performance. Secondly, games may require a long time to complete. To solve this issue, many researchers have tried reducing the size of the grid to 10x10, in order to speed up the process. Furthermore, comparison between different research articles can be difficult. This is because small variations in the implementation of Tetris can lead to significant differences in score.

4.2.1 Early attempts

Tsitsiklis and Van Roy (1996) attempted to solve the game using feature-based dynamic programming on a non-standard 16 x 10 grid. They were able to clear 30 lines by using only two features: the number of holes and the height of the highest column.

Later that year Bertsekas and Tsitsiklis (1996) used a lamda-policy iteration and added another two extra features: the difference in height between each column and the height of all the individual columns. They were able to clear around 2800 lines

Using a least-squares policy iteration Lagoudakis, Parr and Littman (2002) added even more features: the mean column height and the sum of differences in consecutive heights. An average score of 1000 to 3000 lines was achieved.

4.2.2 Genetic Algorithms

Genetic Algorithms, sometimes referred as evolutionary algorithms, are search-based optimization techniques based on Natural selection and Genetics (Turing, 2009). In GAs, a population of possible solutions is given to the problem. The solutions that deliver the best results are combined to produce a new generation of solutions. This process is repeated over various generations. GAs have been used to give promising results in solving the game of Tetris over the past two decades.

One of the first successful GA was a simple approach by using a simple two-level search for every possible game-board (Böhm, Kókai and Mandl, 2005). This was done by rating all the boards and choosing the board path that will give the highest rating. Using a linear rating function, they were able to clear around 480,000,000 lines and 34,000,000 lines using an exponential function with displacement. The functions calculated the rating based on twelve different criteria: pile height, holes, connected holes, removed lines, altitude difference, maximum well depth, sum of all wells, landing height, number of blocks, weighted blocks, row transitions and column transitions.

Szita & Lorincz were able to clear 350,000 lines using the cross-entropy algorithm (Szita and Lörincz, 2006). Building on this work by adding extra features such as hole depth and rows

with holes,(Thiery and Scherrer, 2009), created the BCTS controller (Building Controllers for Tetris). Using BCTS they were able to clean 35,000,000 lines and win the 2008 Reinforcement Learning competition.

Another popular GA (Yiyuan, 2013) uses a look-ahead method of computing all possible moves and selects the one with the best score. In fact Yiyuan, claimed to have developed the near Perfect AI, always according to himself, and it can play the game of Tetris without losing. The score function is given using a linear combination of four properties, the aggregated height, the lines cleared, the number of holes and the bumpiness of the board. The score function and the final weights for each property can be defined as:

$$\vec{p} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}, \vec{x} = \begin{pmatrix} AggregateHeight \\ CompleteLines \\ Holes \\ Bumpiness \end{pmatrix}$$

$$a = -0.510066$$

$$b = 0.760666$$

$$c = -0.35663$$

$$d = -0.184483$$

4.2.3 Approximate Modified Policy Iteration

Motivated by Lagoudakis, Parr and Littman (2002) classification-based policy iteration algorithm, Gabillon, Ghavamzadeh and Scherrer (2013) discovered a vector of weights that was able to clear around 51,000,000 lines. This marked a landmark in Tetris AI since it was the first time Reinforcement Learning algorithms could be compared to Genetic Algorithms. The main idea behind Lagoudakis & Parr algorithm was to use classifiers within the loop of RL algorithms to recognize good actions. Using the CMA-ES algorithm Hansen and Ostermeier (2001), Gabillon et al. estimated the values of state-action pairs making use of rollouts and the reduced a complex function of these rollouts.

4.2.4 Deep Reinforcement Agents

Using DQN (Stevens and Pradhan, 2016) attempted to replicate of Atari games in Tetris but they were unsuccessful scoring an average of less than twenty cleared lines. However, the demonstrated that the key difficulty is the delay between the action and the reward. They proposed that this introduced overfitting to the model. Furthermore, they tried to minimize the time between actions and rewards by grouping the action. Even though it achieved a better performance, the agent still performed basic mistakes.

By using an improved state configuration (Liu and Liu, n.d.) applied several Deep Learning agents to the game of Tetris. They proposed that the state configuration will only change when the falling tetrimino is only stationary instead of every time the block drops. They were able to clear around 1100 lines using DQN and around 700 using PPO showing promising results. Their scores are shown in figure 4.3

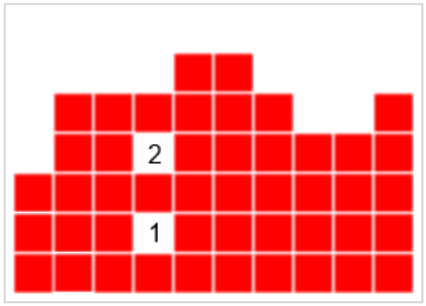
Method	Total step	#cleared lines in total
DQN	3 000 000	1163
PPO [22]	10 000 000	692
DrQ [13]	3 000 000	931
Dreamer [9]	300 000	55
Plan2xplore [23]	300 000	72
LucidDreamer	300 000	107

Figure 4.3: Comparison between different deep RL agents for Tetris. DQN performs the best. Results published by Liu and Liu (n.d.)

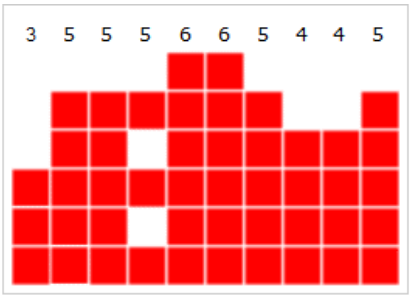
By building on the work of Yiyuan discussed in section 4.2.2, the GA algorithm was translated into a model-based deep Q-agent (Nuno, 2019). The succesfull agent simulates all possible moves and evaluates future state values. The agent uses as a state representation, four board features:

- Number of lines cleared
- Number of holes
- Bumpiness
- Total Height

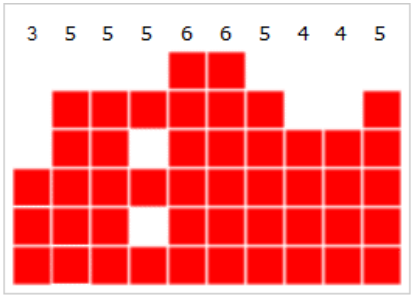
In just 1500 episodes, the agent was able to clear over 2000 lines per episodes. Detailed explanation of the features can be seen in the figure 4.4 below.



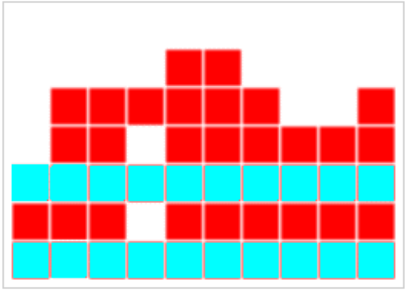
(a) Number of holes = 2



(b) Bumpiness= |3-5|+|5-5|+...+|4-4|+|4-5|=6



(c) Total Height = 48



(d) Completed Lines = 2

Figure 4.4: Popular properties used by multiple successfully Reinforcement Learning agents in the past. Image Courtesy of Code my Road. Yiyuan (2013)

Chapter 5

Methodology, Implementation and Development

The main aim of the project is to create a computer program that plays the game at a professional level and understand the critical point of success or failure of the algorithms. However, the main objective is to use the observations to help in understanding sequential decision problems and introduce key-decision making concepts.

The project will proceed by implementing different tabular and deep RL agents and compare their performance with each other. The various agents should have techniques from both value-based and policy-based methods.

The project should demonstrate clear understanding of why each agent performed well or poorly. Furthermore, their strategy (policy) will be visualized to understand the decision making of every agent. In this way, possible improvements can be introduced.

Most research on the game of Tetris work on smaller, more flexible boards which are computationally cheaper to solve and require less training (Algorta and Şimşek, 2019). This way, the agents will be tested in simpler environments which will help in investigating possible flaws in their implementation and minimize the errors. The best method will be applied to a board of standard size, 20x10 to check the consistency of the results. In fact:

1. Tabular methods will be used on a 4x4 version of the game. Using the results, it will allow for easy tuning of the environment and reward function and check for any complications in the code.
2. Value-based and policy-based deep RL methods will attempt to crack an 8x6 environment. This will allow easy parameter tuning of the agents, fast comparison between different state representations and even more reward shaping

5.1 Creating the environment

All this research will require access to a Tetris environment in which the agents can act. The environment was implemented using Python 3. The idea behind it is simple; create a two-dimensional array to store the blocks as they fall. The array is initially filled with zeros. As the blocks begin to fall, the array is populated with ones and twos, stationary and falling blocks respectively.

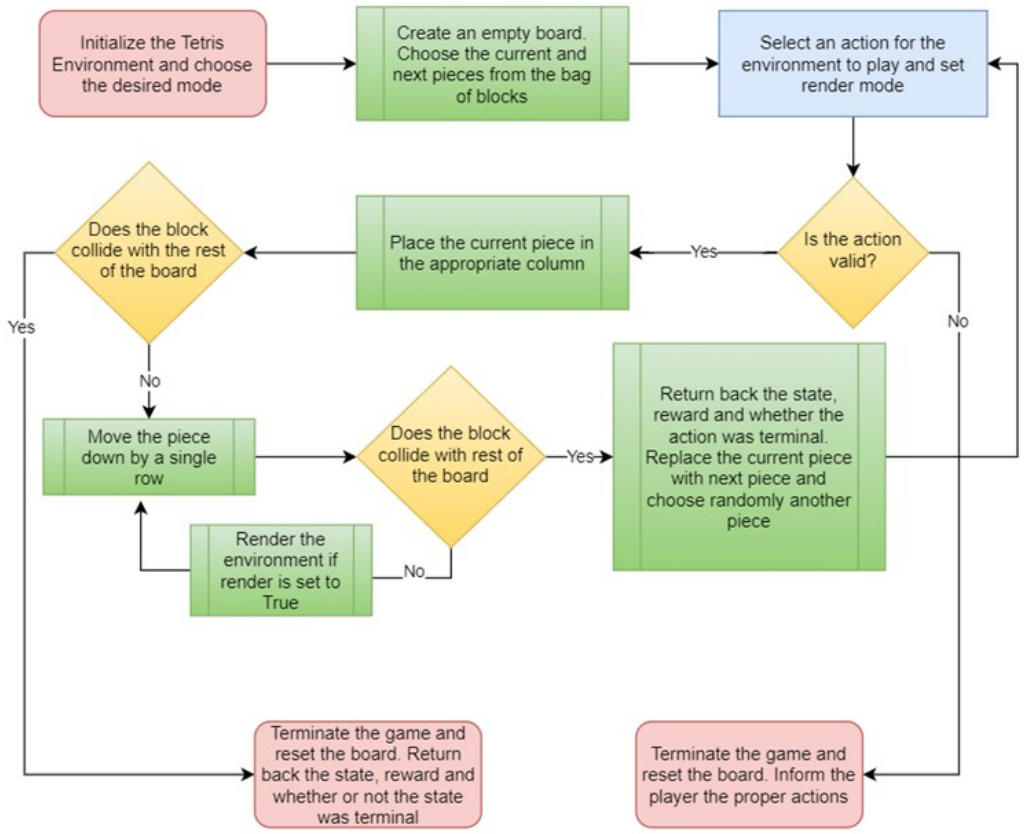


Figure 5.1: The regular steps that the environment follows when playing a single game of Tetris

The environment takes as input a location and rotation for the current piece to drop. It will not allow the agent or the player to change the rotation or displacement of the piece once it is dropped. This will reduce the action-reward delay encountered in the previous literature. However, it will not eliminate it since it will take a few actions to get the corresponding reward. The following modules were used:

1. NumPy: A collection of high-level mathematical functions supporting multi-dimensional arrays and matrices. Responsible for manipulating the array -grid- of the game.
2. OpenCV: An optimized Computer vision Library, used for rendering the array into a picture for easier visualization.

5.1.1 Tetris 4x4

Even though the state space of Tetris is much smaller than the upper bound as discussed in section 3.6.1, it is still enormous, so tabular methods are not a viable option. However, making the dimensions of the board small enough and reducing the complexity of the blocks will allow the use of tabular methods. Using such a board with well-known methods that guarantee results will allow for easy tuning of the environment and the reward function. The shapes used are going to be made up by two blocks which can create just a single geometry – a single line. Given the 4 columns and the 4 possible rotations of each shape, there should be 16 possible actions but due to the simplicity only 7 actions are viable and can be seen in figure 5.2.

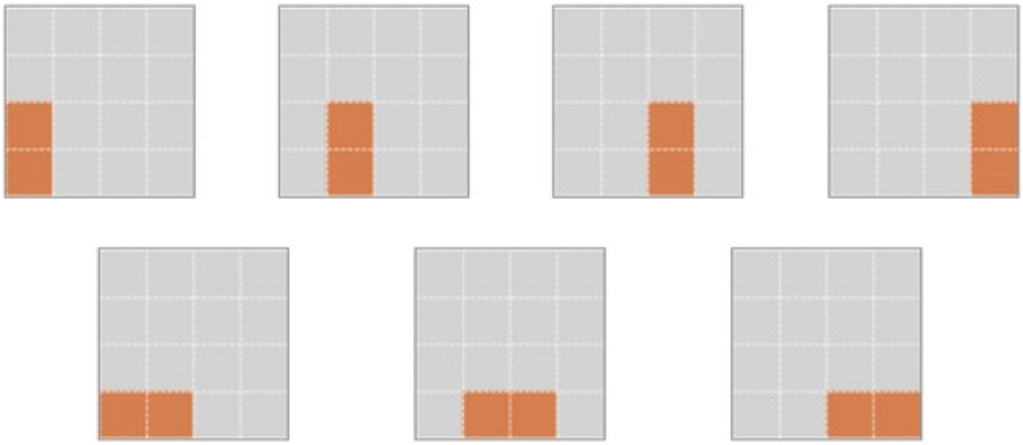


Figure 5.2: All possible 7 actions in the 4x4 environment. Action 0 - top left | Action 6 - bottom right

5.1.2 Tetris 8x6

The idea behind the 8x6 board is to implement the different deep RL agents and compare their performance with each other. The algorithms that will be used to crack the game of Tetris will utilize model-free Deep Reinforcement Learning techniques. The agents will begin the training knowing nothing about the environment and therefore can be thought of as "explicit" trial-and-error algorithms. In fact, the agents that are going to be used are DQN and PPO, the two most popular agents. This board will allow easy parameter tuning for the agents which would otherwise be almost impossible on the full grid, due to the high training periods. Once again, given the 6 columns and the 4 possible rotations there are 24 possible actions. In fact, the agent takes as an input a single number from 0-23 and maps it to the corresponding position and rotation, where 0 is column 1 – rotation 0 and 23 column 6 - rotation 270. This can be demonstrated in the table below. The shapes used for this implementation will be made up of three blocks, meaning only two possible geometries, the I shape and the L shape.

Action	Position - Rotation	Action	Position - Rotation	Action	Position - Rotation
0	1 - 0º	8	3 - 0º	16	5 - 0º
1	1 - 90º	9	3 - 90º	17	5 - 90º
2	1 - 180º	10	3 - 180º	18	5 - 180º
3	1 - 270º	11	3 - 270º	19	5 - 270º
4	2 - 0º	12	4 - 0º	20	6 - 0º
5	2 - 90º	13	4 - 90º	21	6 - 90º
6	2 - 180º	14	4 - 180º	22	6 - 180º
7	2 - 270º	15	4 - 270º	23	6 - 270º

However, this action space runs into issues where placing a piece is simply not possible due to the board geometry. For example, consider the I shape; Vertically it can be placed in all the columns but horizontally the board runs out of space. To fix this, the environment was designed to place the piece at the right corner. This can be demonstrated in figure 5.3:

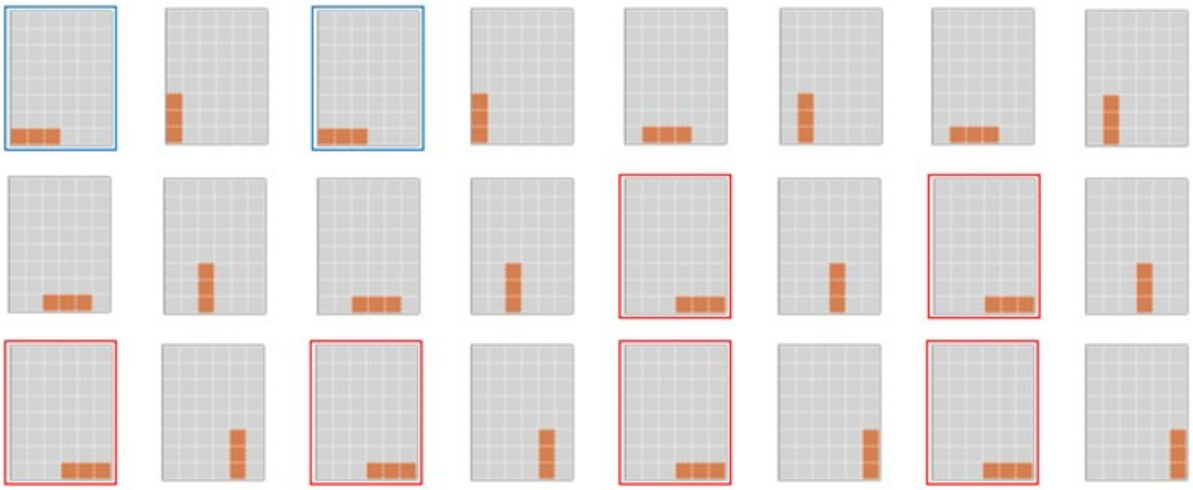


Figure 5.3: All possible 24 actions in the 8x6 environment. Action 0 - top left | Action 24 - bottom right

By considering figure 5.3, actions 0 and 2 (blue grid), rotations 0º and 180º respectively, is an example of how placements can be completely identical due the symmetry of the I piece. The same applies for every other action. However, actions 12,14,16,18,20,22 (red-grid) are identical not only by rotation symmetry, but also due to the fact that I piece cannot be placed horizontally in columns 5 and 6. Therefore, it returns back to the previous available horizontal position at column 4.

This introduces a factor of bias in the environment. At the start, the agents must understand that some actions produce the same results. Through exploring, the agents must learn to deal with it. On the other hand, random agents will have no idea about it and their recorded performance will slightly be worse that the actual. This will not be an issue since the averages of random agents are already extremely low.

5.2 Feature Engineering and State Representations

As discussed previously in section 4.2, past work showed that it is possible for RL agents to learn through low-dimensional state representations. The use of Deep Learning allows for these state representations to be learnt during training even with an input of raw visual data. In fact, it is possible for the agent to gain a better understanding through visual input. The environment will have different modes, each returning a different state representation, and the modes yielding the best results will be tested on the larger environment.

5.2.1 Mode 1: Feature Representation

The first feature mode of the environment returns several important properties discussed in the literature that will be used as the input to the Deep RL agent. The features returned by the environment are:

1. The number of cleared lines by the action taken
2. The number of holes in the board

3. The board bumpiness which is the sum of the differences of heights between the pair of columns
4. The summation of all the column's height

For future reference, this will be referred as the 4-feat state representation. Figure 5.4 demonstrates this representation on a random board:

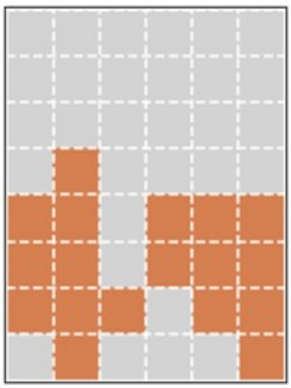


Figure 5.4: 4-feat Representation using the features **(Lines, Holes, Bumpiness, Height sum)** with respective state representation **(0,5,6,23)**

Even though the discussed properties are some of the most popular ones used for decades, the agent will be considered as 'blind'. This state representation does not inform the agent of what are the pieces it currently places, and it has no what next pieces to expect. Furthermore, it has no way of knowing how the actual board looks like. It only gives the agent the aforementioned properties. As the expectations are not so high, a second state representation was designed, to investigate this effect which in addition to the aforementioned properties, it will also return:

1. The number of blocks in each column
2. The current and the next piece that follows

For future reference, this will be referred as the 12-feat state representation. Therefore, the same board configuration translates to:

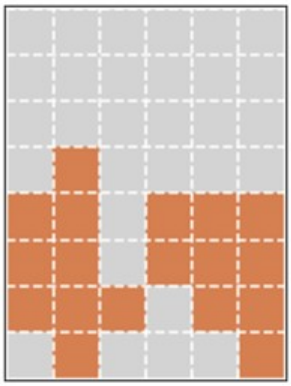
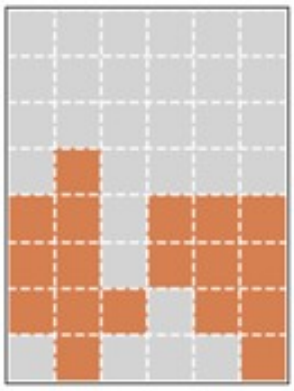


Figure 5.5: 12-feat Representation using the features **Lines, Holes, Bumpiness, Height sum, Blocks 1-6, Current piece, Next piece** with respective state representation **(0,5,6,23,3,5,1,2,3,4,1,0)**

5.2.2 Mode 2: Board Representation

The second mode of the environment will return the full array that holds the board's data as it's state. As demonstrated by DeepMind, the current state of art uses visual input to train the models. By using the full board and enough exploratory time for the algorithm, the network should have sufficient information to distinguish key features. Even though it receives some visual input, the agent will still be blind to the next pieces coming. For future reference, this will be referred as the board representation. The same board translates to:



(a) Left

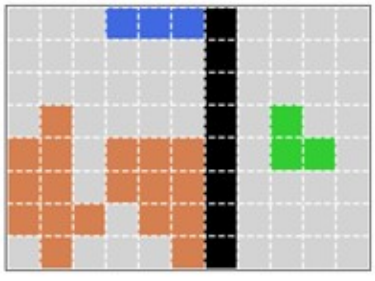
```
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 1 0 0 0 0]
 [1 1 0 1 1 1]
 [1 1 0 1 1 1]
 [1 1 1 0 1 1]
 [0 1 0 0 0 1]]
```

(b) Right

Figure 5.6: Board Representation | Left - A reference board | Right - The respective state representation

5.2.3 Mode 3: A glimpse into the future

The third and final mode of the environment will still return the full board, with the current falling piece as well the next piece appearing. Most Tetris consoles allow the player to know which block comes next. In fact there are games which allow the player to view up the next five blocks. Pro players where able to exploit this feature and created geometries which redefined the game of the Tetris like never before. Even though learning might be a bit slower at the start, due to higher volume of input data, it would be interesting to the result. For future reference, this will be referred as the glimpse representation.



(a) Left

```
[[0 0 0 2 2 2 3 0 0 0 0]
 [0 0 0 0 0 0 3 0 0 0 0]
 [0 0 0 0 0 0 3 0 0 0 0]
 [0 1 0 0 0 0 3 0 4 0 0]
 [1 1 0 1 1 1 3 0 4 4 0]
 [1 1 0 1 1 1 3 0 0 0 0]
 [1 1 1 0 1 1 3 0 0 0 0]
 [0 1 0 0 0 1 3 0 0 0 0]]
```

(b) Right

Figure 5.7: Glimpse Representation | Left - A reference board | Right - The respective state representation

5.3 Deep RL Agents

The two agents that will attempt to crack the game of Tetris are the actor critic PPO and the first successful deep Q Learning agent, the 'vanilla' DQN. The aim is to compare how a value-based agent performs vs a policy-based agent. How does each agent learn to play the game and how does the increased complexity affect their performance? The hyper parameters for each agent are determined by previous, successful literature and will be adjusted accordingly to the results of the 8x6 board.

The hyperparameters of the DQN agent are going to be based on the DQN by Mnih et al. (2015). However, the agents used to frame skip in order to remove a lot of unnecessary states from the buffer and improve sample efficiency. However, the Tetris environment representation can be also be considered frame skipping, as it does not show the pieces falling, a feature designed to solve the delayed action-reward phenomenon. Therefore the adjusted hyper parameters of the DQN are:

Hyper-parameter	Value	Comment
Gamma	0.99	High to increase the weight of future rewards
Max Epsilon	0.9	High to induce randomness at the start
Min Epsilon	0.1	Low enough to follow the policy but high enough to explore
Epsilon random frames	12500	Completely random first frames
Epsilon greedy frames	250000	Frames needed for epsilon to drop to 0.1
Buffer size	125000	Maximum memory. Adjusted from DeepMind
Batch size	32	Suggested by DeepMind
Update main model weights	After 1 frame	Adjusted from DeepMind
Update target model weights	After 2500 frames	Adjusted from DeepMind
Learning rate	0.00025	Adjusted from DeepMind
Loss function	Huber loss	Avoid overffiting

Table 5.1: DQN hyper-parameters

The hyper parameters of the PPO agent are adjusted from the popular implementation of the PPO agent by OpenAI. (2020)

Hyper-parameter	Value	Comment
Gamma	0.99	High to increase the weight of future rewards
Epochs	Depends on results	Higher epochs means more training
Steps per epoch	4000	Actions taken before learning
Clip ratio	0.2	Suggested by Creator of PPO
Policy learning rate	0.0003	Learning rate for policy optimizer
Value function Learning rate	0.001	Learning rate for value function optimizer.
Train policy iterations	80	Maximum number of gradient descent steps to take on policy loss per epoch. (Early stopping may cause optimizer to take fewer than this.) Adjusted from Open AI
Train value iterations	80e	Number of gradient descent steps to take on value function per epoch. Adjusted from Open AI
Lambda	0.97	Adjusted from OpenAI
Target KL	0.01	Adjusted from OpenAI
Loss function	Mean Squared error	Suggested by OpenAI

Table 5.2: PPO hyper-parameters

5.4 Neural Network Architecture

Tensorflow and more specically the Keras, the high-level API of Tensflow 2 which focuses on deep learning is used to built the models. Keras is simplistic, flexible and powerful used by organizations and companies including NASA, Youtube and Waymo.

Throughout the training a constant deep model architecture was used to ensure consistency of the results. A 3-layer Multi-Layer Perceptron (MLP) architecture with 2 hidden fully connected layers is used. Hyperbolic tangent activation functions are used for the hidden layers, and a no activation function used for the output layer (basically a linear regression). The input layer expects an input of of a flatten observation space and the two hidden layers contain 64 neurons each. The output layer is fully connected with an output dimension of the num of actions.

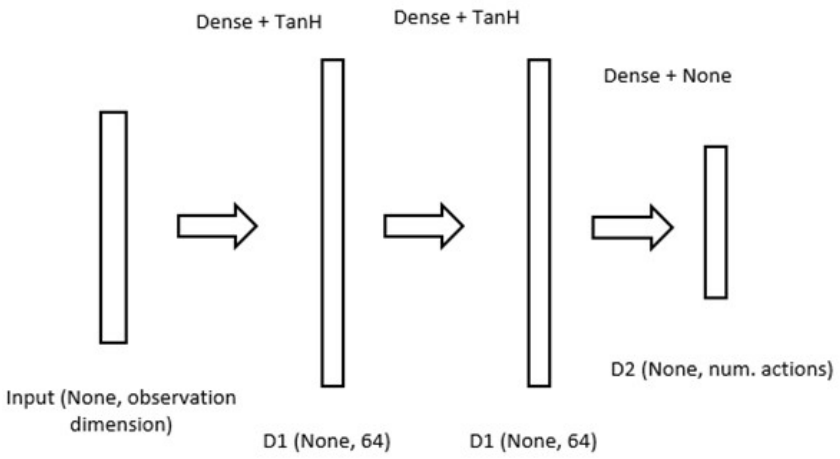


Figure 5.8: The MLP architecture used by the PPO and the 4/12 feat representations

The only exception where a different architecture was used are the DQN models around visual input. As explained in section 4.1.3, DeepMind was able to overcome successfully the barriers of deep Q Learning using visual input and convolutional layers. The architecture of the DQN models around vision data revolve around the convolutional and fully connected Dense Layers in order to approximate the q values. The first convolutional layer uses 32 3x3 filters with a stride of 3 and a Rectified Linear Unit (relu) activation function. The second convolutional layer uses 64 2x2 filters a stride of 2 similarly followed by relu activation. The output of the final convolutional is flattened and passed to a fully connected layer of 512 neurons with relu activation. Lastly the output layer is fully connected with a linear activation and an output of the number of actions. Figure 5.9 shows the architecture of the model.

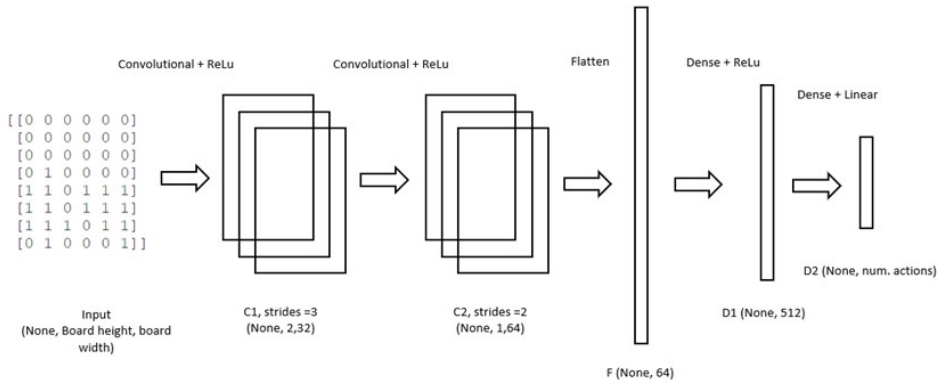


Figure 5.9: The convolutional architecture used by DQN on the Board and Glimpse Representations

5.5 Training and Resources

Most of the training will be done on Google Colab (GC) using a Colab Pro subscription due to its ease to use, interactive interface and its accessibility to GPUs and TPUs. Colab doesn't only work as intergraded development environment (IDE), but also as a presentation and

visualization tool that will allow live monitoring of the results. Furthermore, GC has direct access to the google drive, where the models, the hyperparameters and the DQN buffers will be stored as the agents' run. For reproducibility of the results, the model weights will be saved every 8000 actions. This will be helpful in visualizations of the agent's strategy as well as performance. As training requires a lot of time, especially the DQN, some agents will be stopped earlier than the pre-determined threshold when they show signs of convergence. This will help increase efficiency in the project and will make time to tune everything better.

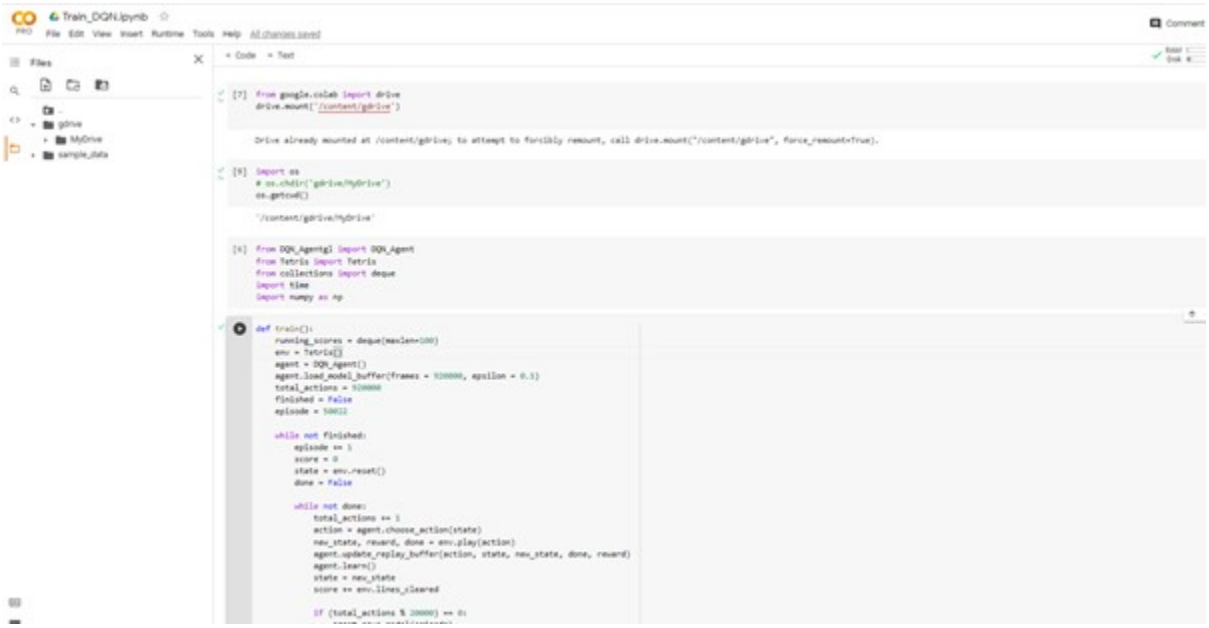


Figure 5.10: Google colab in Action. Mounted on the Google drive for easy file transfer and file storing.

Chapter 6

Testing and Experimentation

6.1 Preliminary exploration of the environment and Reward Shaping

Using the 4x4 environment, two tabular value-based methods will be used to ensure that the environment is working properly are SARSA and Q-Learning, discussed in section 3.2.1. Different reward functions will be implemented, and their performance will be evaluated. The algorithms were deployed using jupyter notebooks and the numpy package as the main core. Figures 6.1 and 6.2 demonstrate the pseudocode for both algorithms.

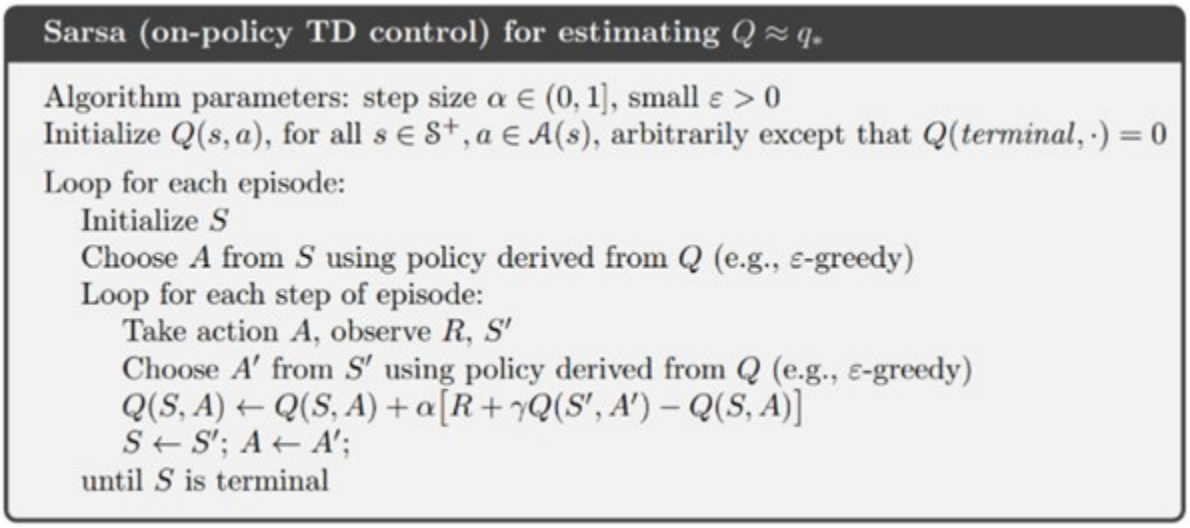


Figure 6.1: The SARSA pseudocode. Image courtesy of Reinforcement Learning: An Introduction, p130, R. Sutton & A. Barto (2018).

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 6.2: The Q-learning pseudocode. Image courtesy of Reinforcement Learning: An Introduction, p131, R. Sutton & A. Barto (2018).

Perfecting at a fast rate

Theoretically, the agent will master the game when it learns to clear as much multiple lines as possible given any scenario at a fast rate. Therefore, the agent should be rewarded only when lines are cleared. This will force the game to learn to play the game optimally. The following score system based on exponential scale of four is proposed:

$$Reward = \begin{cases} 4^{n-1} \quad for n > 0 \\ -1, \text{ for } n = 0 \\ -10 \text{ for terminal states} \end{cases} \quad \text{,where } n = \text{ number of rows of cleared}$$

The agent collects a reward of -10 for losing, -1 for not clearing any lines and a maximum reward of 64, when clearing the maximum 4 lines. Using this score function will force the agent to learn how to clear lines fast since it's the only time it collects a reward. Otherwise, the agent is punished. The results from the suggested reward function are shown below:

Figure 6.3: Q Learning RF 1, Performance and policy over 500 episodes

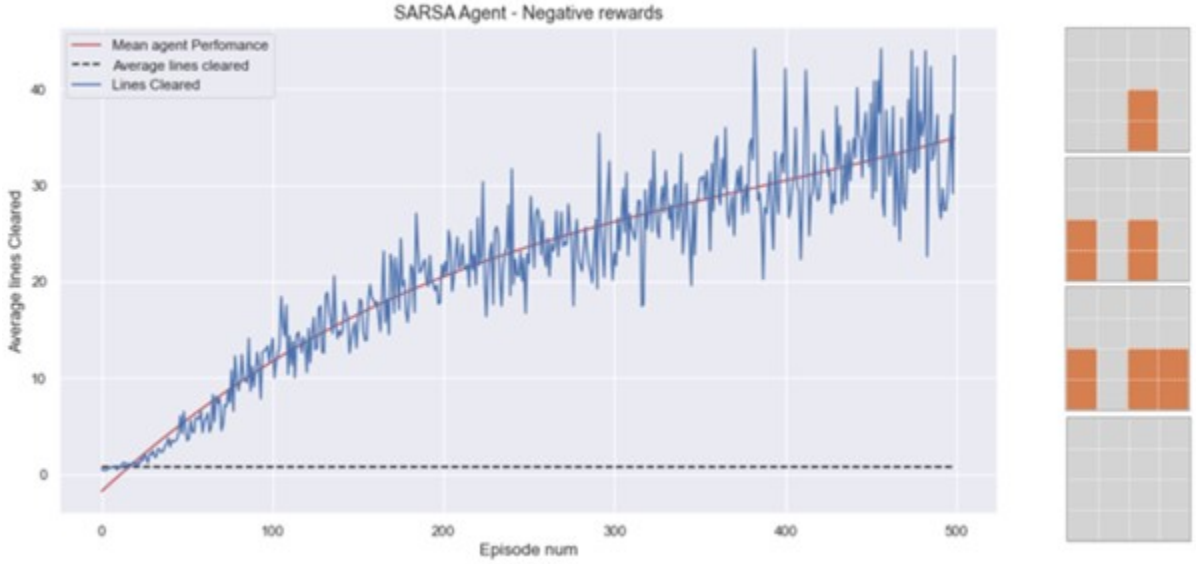


Figure 6.4: SARSA RF 1, Performance and policy over 500 episodes

Perfecting Reward Function

On the other hand, punishing the agent for not clearing any lines may discourage the agent in waiting for states that yield a higher reward. In much more complex environments, where clearing single lines is difficult, the agent may choose to lose fast in order to avoid collecting negative rewards. Therefore a second reward function is proposed where the agent is only punished for losing. For now on, this will be referred as the Perfecting Reward function:

$$Reward = \begin{cases} 4^{n-1} \quad for n > 0 \\ 0, \text{ for } n = 0 \\ -10 \text{ for terminal states} \end{cases} \text{,where } n = \text{ number of rows of cleared}$$

Figure 6.5: Q Learning RF 2, Performance and policy over 500 episodes

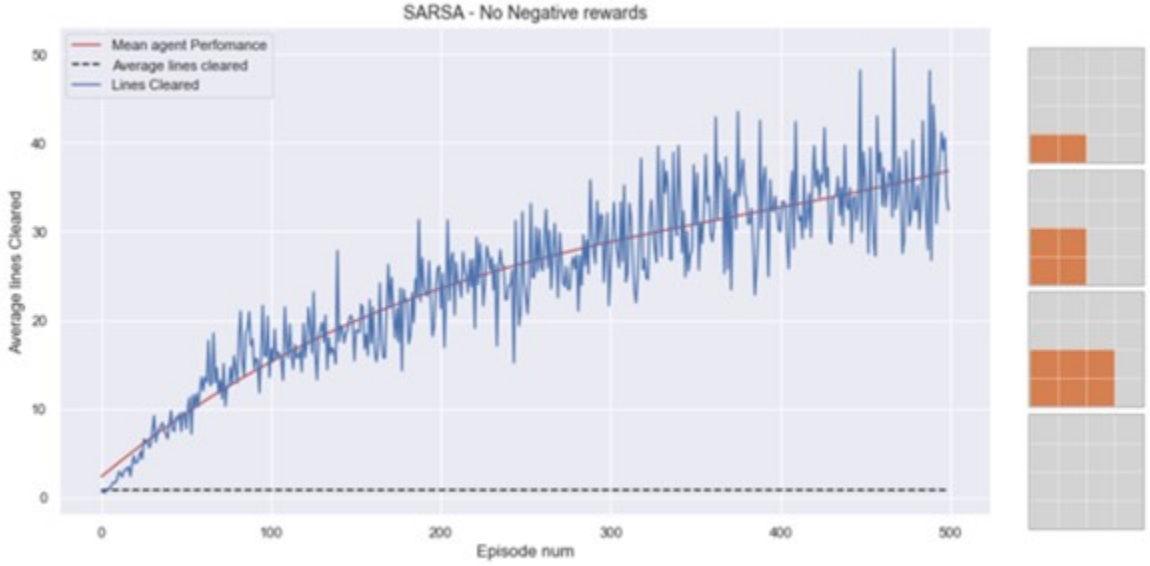


Figure 6.6: SARSA RF 2, Performance and policy over 500 episodes

As expected, due to the simplicity of the environment, both agents were able to to solve the board using both reward functions. The strategy varied from agent to agent but all of them were able to find the optimal strategy, clearing two lines at the same time. This shows that there are many ways of getting to a good solution, so the agents can attack the problem at many angles.

However, it is clear that the agents using the no punishment reward functions, learn to clear more lines over the first 100 episodes, even though the algorithms and hyperparameters are exactly the same. In fact, the punishing reward function clears almost no lines during the first 30 episodes even though the cleared lines by a random agent is 0.8 lines per game. This is because the agent collects negative rewards while exploring at the start, and it instead of trying

to clear lines, the agent tries to lose as fast as possible to avoid negative scores. As this is a improper repetitive behaviour, it will only translate to worse results when the complexity gets higher with bigger boards, making the parametric weights of possible future Neural Networks models converge wrongly. Therefore it must be avoided.

Despite the intuition that the reward function 2 should yield the best results, some agents may struggle with this reward function. The reward function can be adapted later to take into consideration some of the most important properties, always based on future results and a better understanding.

6.1.1 A bug/feature in the environment!

The different environments were left to run for several episodes to spot a bug in the code and flush out any unusual or unwanted positions. There are possible positions, where blocks can float. When the environment encounters a full line, it clears it and drops every block above by the number of lines it cleared. When there is empty space below the completed line, the blocks above end up floating. Figure 6.7 demonstrates this "floating effect".

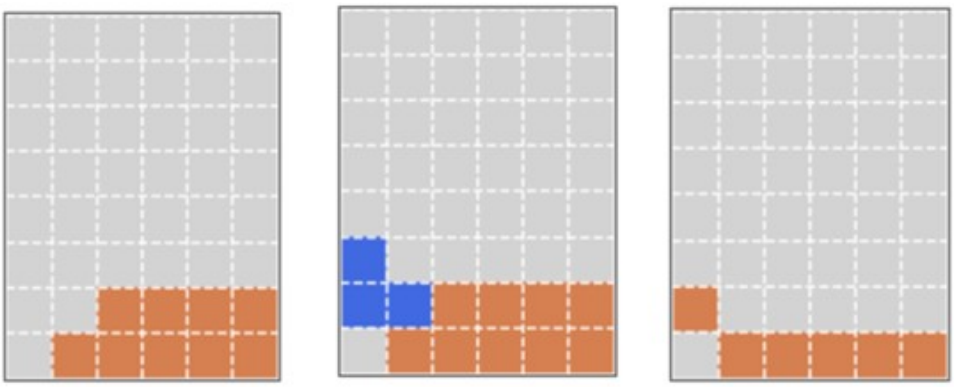


Figure 6.7: Floating blocks in the environment

According to Tetris wiki this bug existed all the way back in the original Tetris created by Pavlovsky and instead of correcting it, it was referred as an unexpected feature. Modern Tetris games still include this effect, with some few exceptions. Therefore, this bug/feature will exist in the environment as well and the agents must learn to approach it

6.2 Experiments and testing with Deep RL Agents

6.2.1 Feature Representation

The first representation tested with the deep algorithms is the 4-feat. Both algorithms were left to train for 1 million actions (250 epochs for PPO) for consistency. The results below are averaged over 10 agents using an epsilon (randomness) of zero. For reference, a random agent on the 8x6 board clears on average 0.16 lines per game.

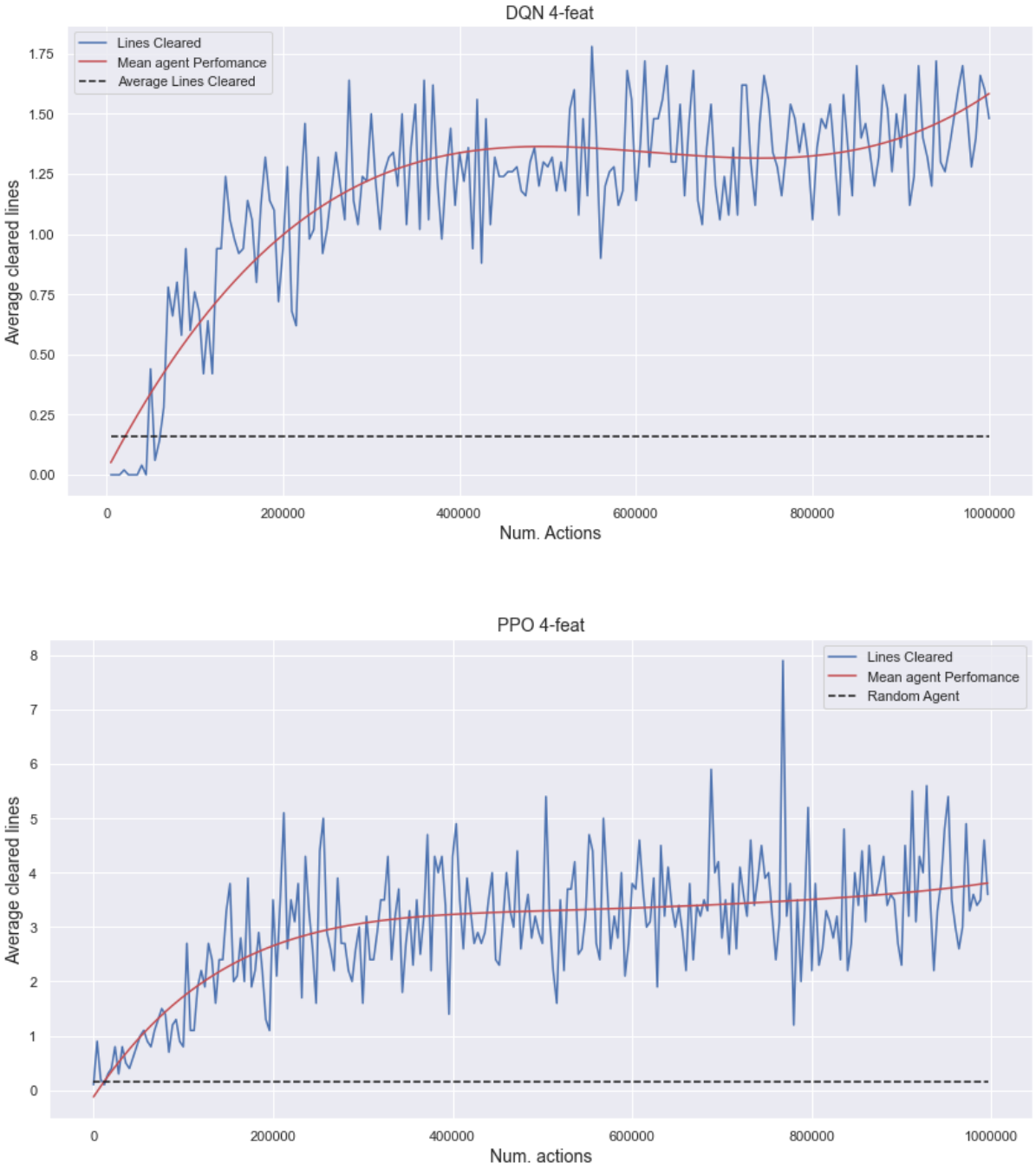


Figure 6.8: DQN's Performance (top) vs PPO's performance (bottom) over the course of one million actions using 4-feat

Both agents demonstrate some understanding of the game with the DQN agent clearing an average of 1.5 lines per episode the PPO clears almost 4 lines. The DQN agent seemed to converge after 250000 actions passed, where the epsilon reached its minimum. The same stage representation with the same model architecture, yielded a higher score for the policy-based agent. Even though both agents are built on keras and tensorflow, the buffer size for the DQN is significantly bigger than that of the PPO. As mentioned before the buffer and the models are constantly being saved for reproducibility. This slows down the already slow DQN agent by a lot. In fact, training for the PPO agent for one million actions requires a single hour while the DQN needs around a full day!

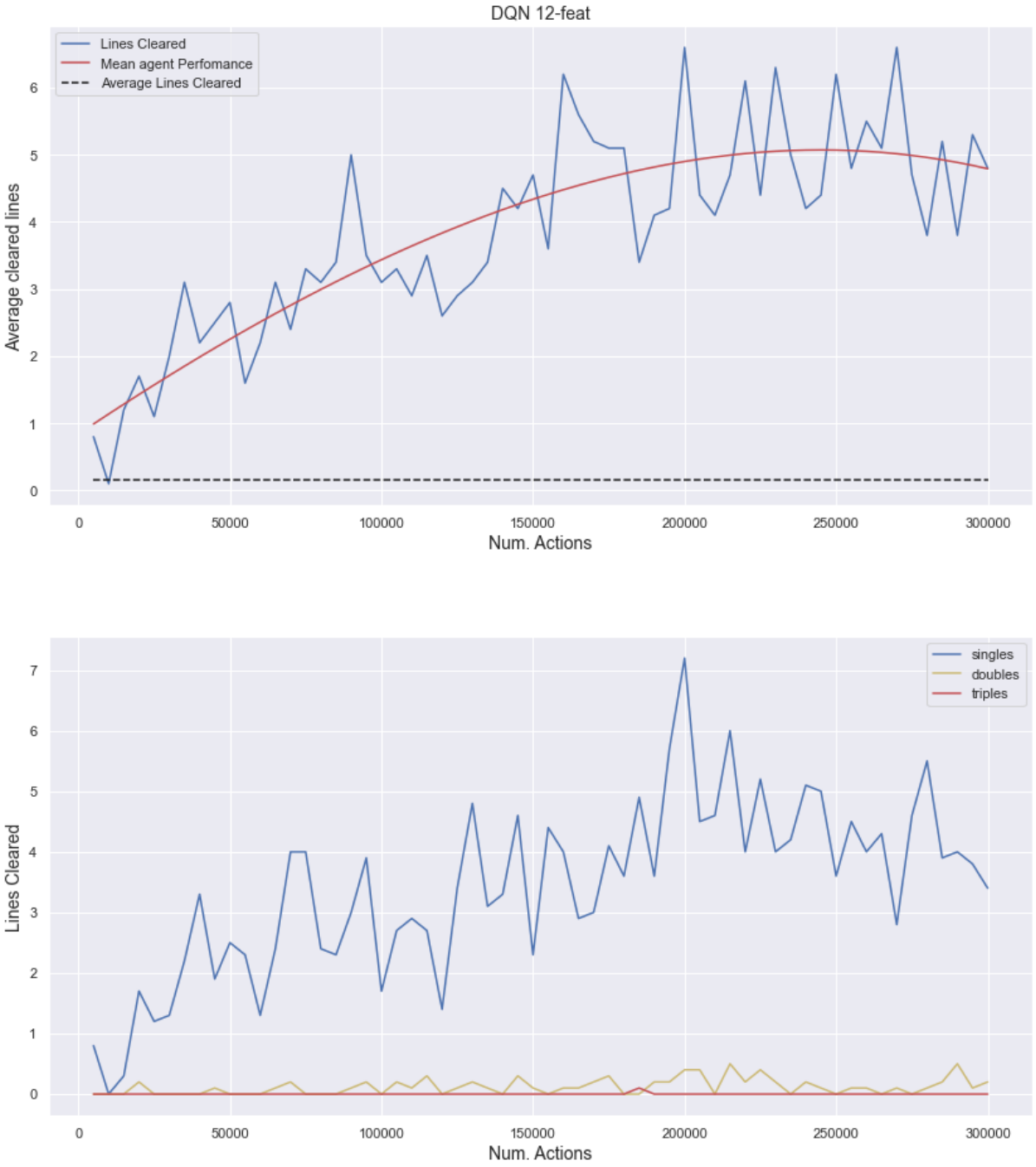


Figure 6.9: DQN's Performance for the 12 feat representation (top) and the type of lines cleared (bottom)

Changing the state representation from 4-feat to the 12-feat, the performance of the DQN agent increases from 1.5 lines to 5 lines in a shorter period of actions. The agent seems to peak at around 250 thousand actions, again where the epsilon drops the minimum, but then the performance starts to drop around 20%, falling from 5 average cleared lines to 4 lines. The agent was stopped earlier Even though some double lines are cleared, there is no clear evidence that the agent understands how to wait and clear simultaneously a higher number of lines. Instead, the agent seems to clear single lines to maximize the game play time. On the other hand, the results of the PPO agents are very promising:

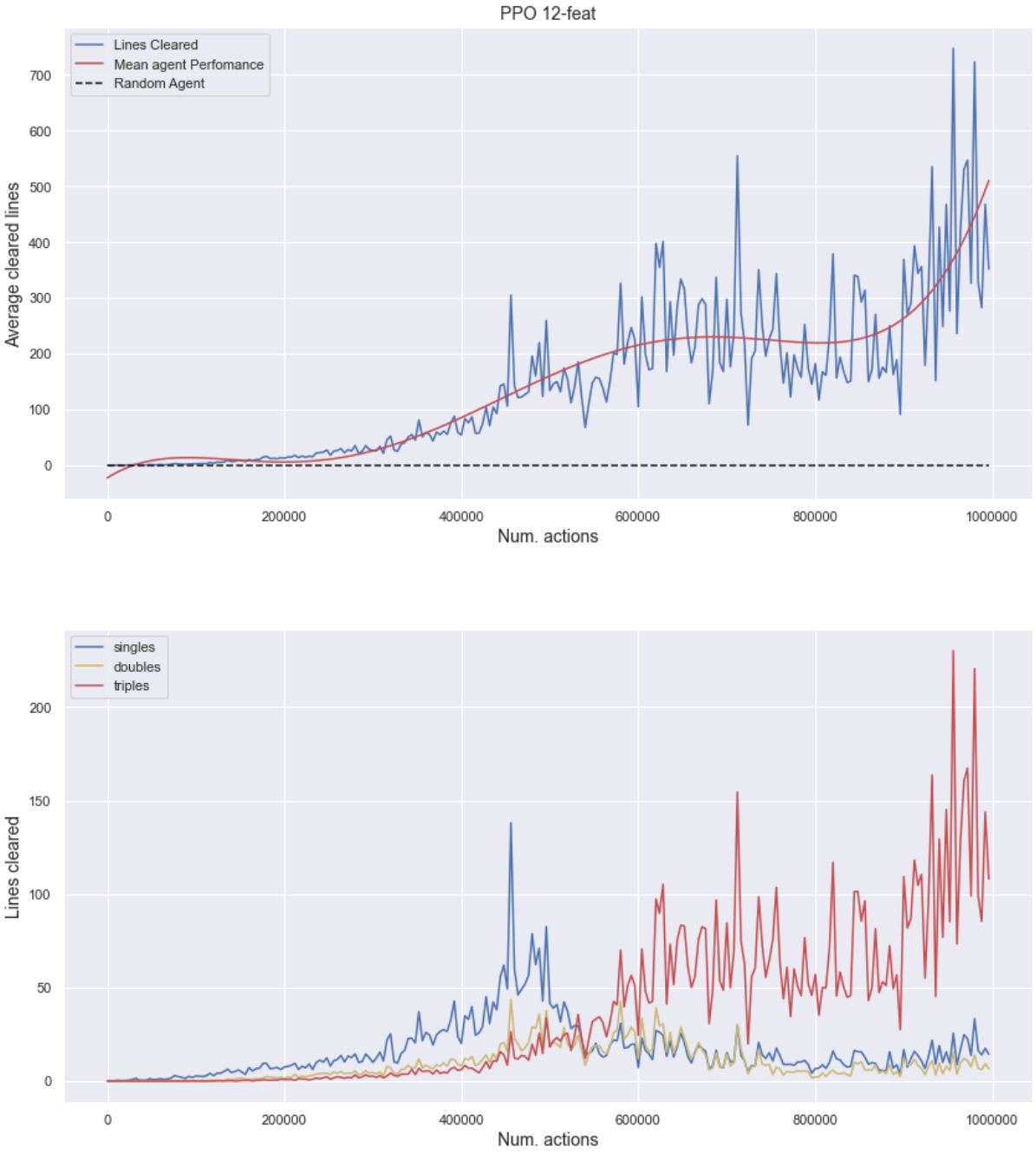


Figure 6.10: PPO's Performance for the 12 feat representation (top) and the type of lines cleared (bottom)

The PPO agent shows a very high level of understanding of the game using the 12-feat representation. It can clear over 500 lines on average at around 1 million actions, with the majority being triple lines. It achieves that by making a pitch in the middle of the board and then dropping a vertical I shape. The big break occurs after the half million threshold where the agent figures out that clearing simultaneously several lines will result in a much higher reward. Then gradually over the final 500 thousand actions it tries to perfect its strategy. The PPO agent along with the 12-feat representation are a strong contender to attempt clearing the 20x10 board.

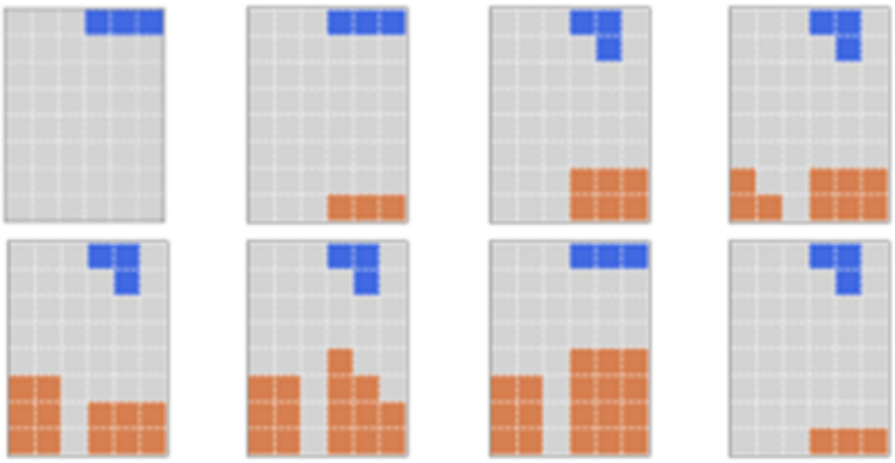
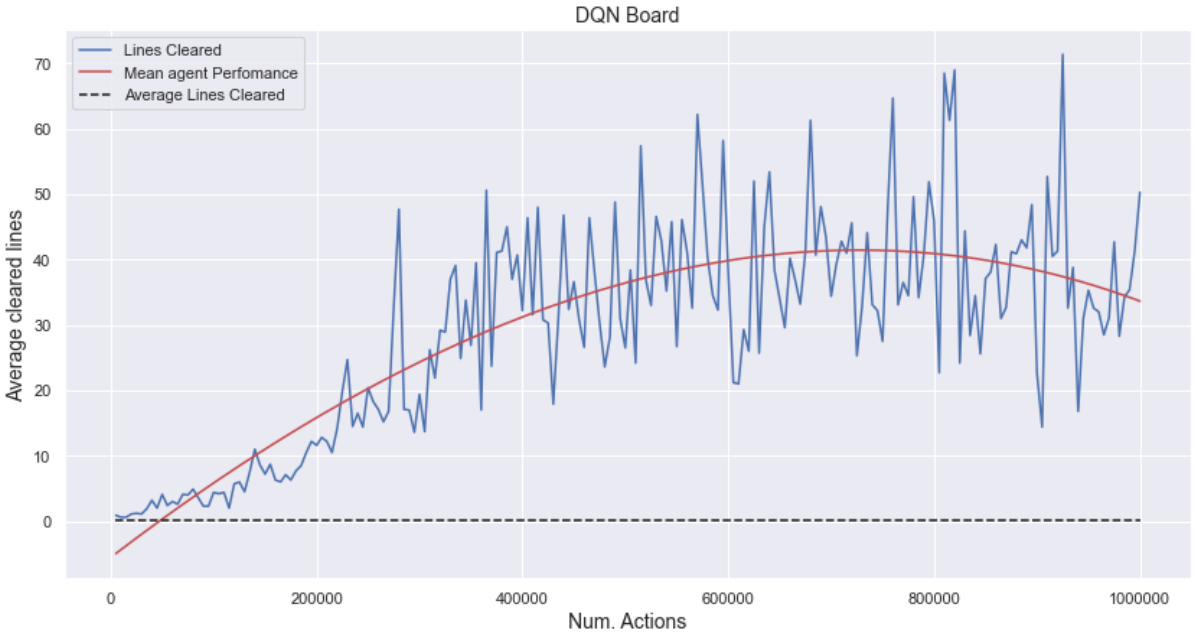


Figure 6.11: PPO 12-feat stacks the pieces and can clear triples lines

6.2.2 Board Representation

Even though the agents are technically blind in the board representations, not knowing which piece is currently falling or which piece is coming, the results are quite interesting.



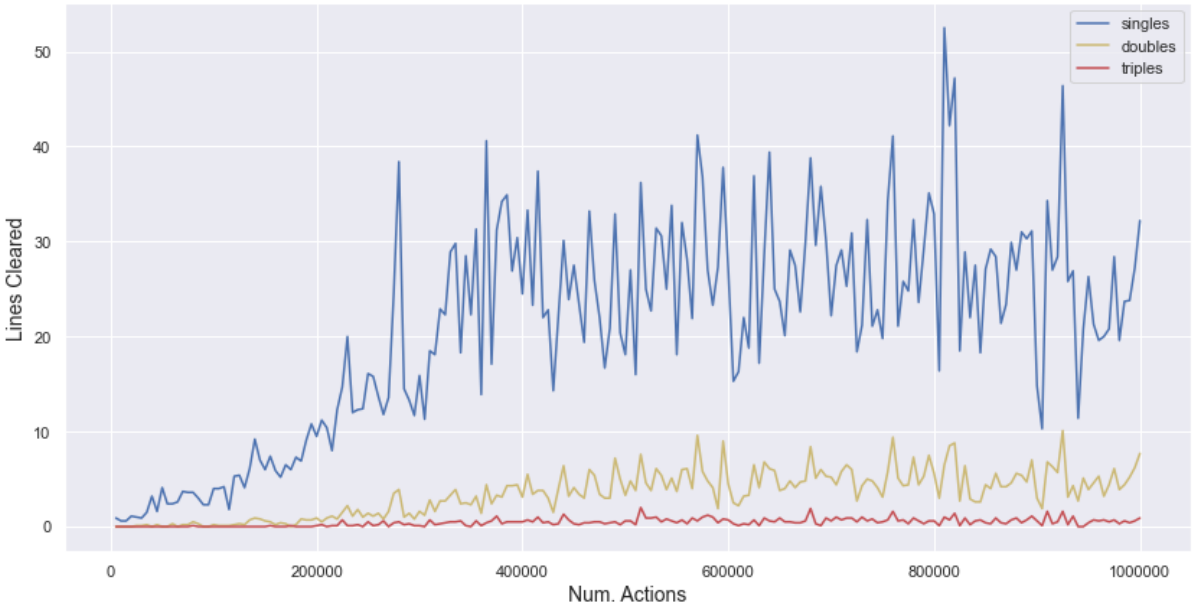


Figure 6.12: DQN's Performance for the board representation (top) and the type of lines cleared (bottom)

The DQN agent overperforms both the feature versions averaging at 40 cleared lines at its peak which is still 8 times better that its corresponding best feature version. The convolutional property of the deep architecture allows for a better understanding of the board and a better feature extraction. Some recurrent behavioral patterns start to appear regarding the DQN agent.

1. The performance starts to degrade over longer periods of training
2. The agent is not able to clear many multiple lines, mostly single lines. This results in an 'endurance' tactic of trying to play the game longer instead of trying to clear multiple lines
3. The agent's performance peaks just right after the epsilon has decreased to its minimum

One possible explanation for the drop is the performance can be the epsilon randomness itself. During the first 250 thousand actions, the agent cannot follow the optimal route because of the randomness. Figure 6.13 below shows the ratio of double lines to single lines that as time progresses.

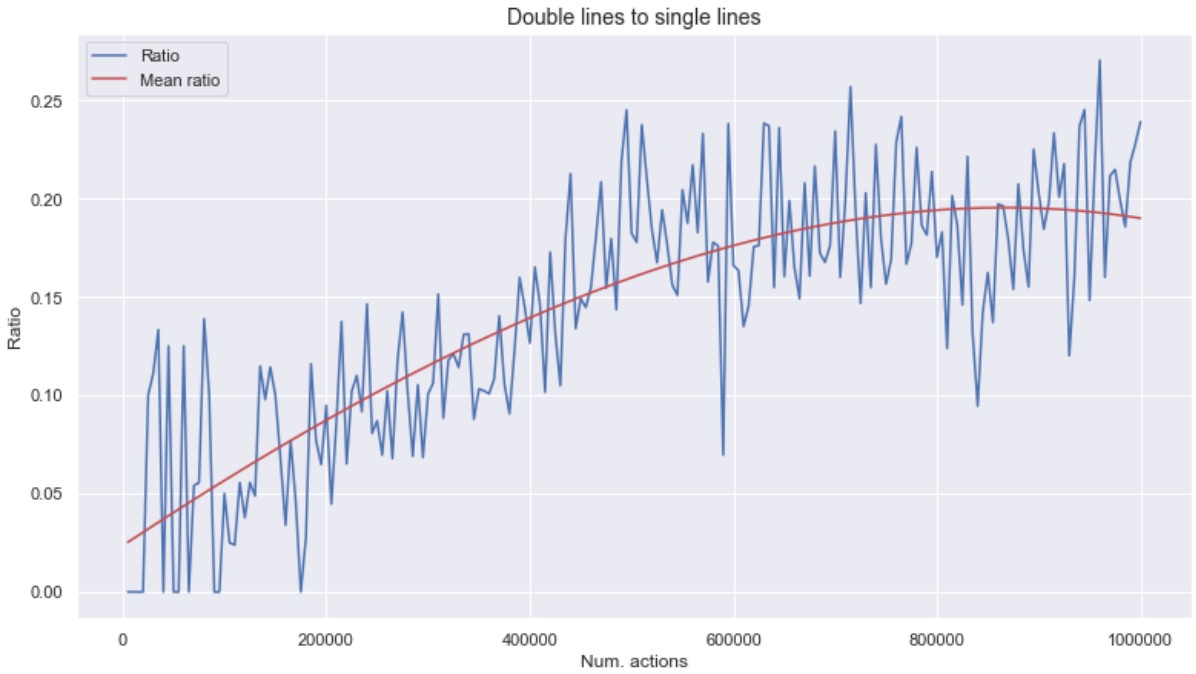
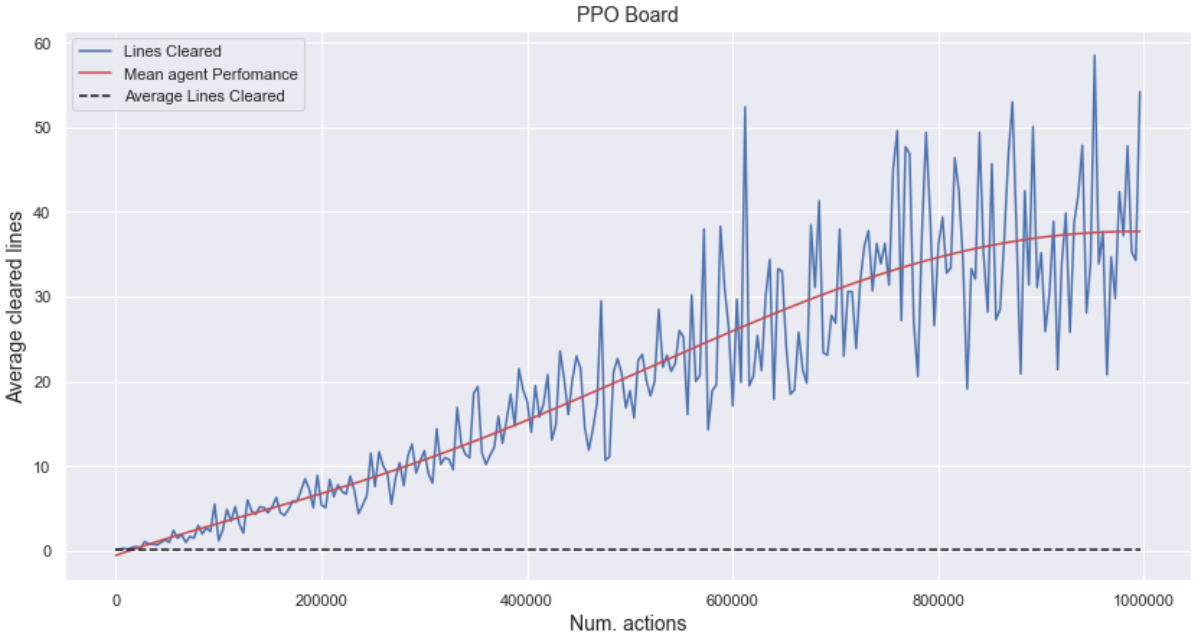


Figure 6.13: The ratio of double lines cleared to single. As it converges, the overall perfomance of the agent drops

The first few spikes arise from the low number of single lines cleared at the start of the game against the relatively high probability of getting a double line. As the epsilon falls to its minimum value, the agent slowly understands the concept of clearing multiple lines simultaneously. Therefore, it tries to adjust its weights to adapt. As a result, the agent unlearns the process of clearing single lines, justifying the drop in performance. The PPO achieves a similar performance:



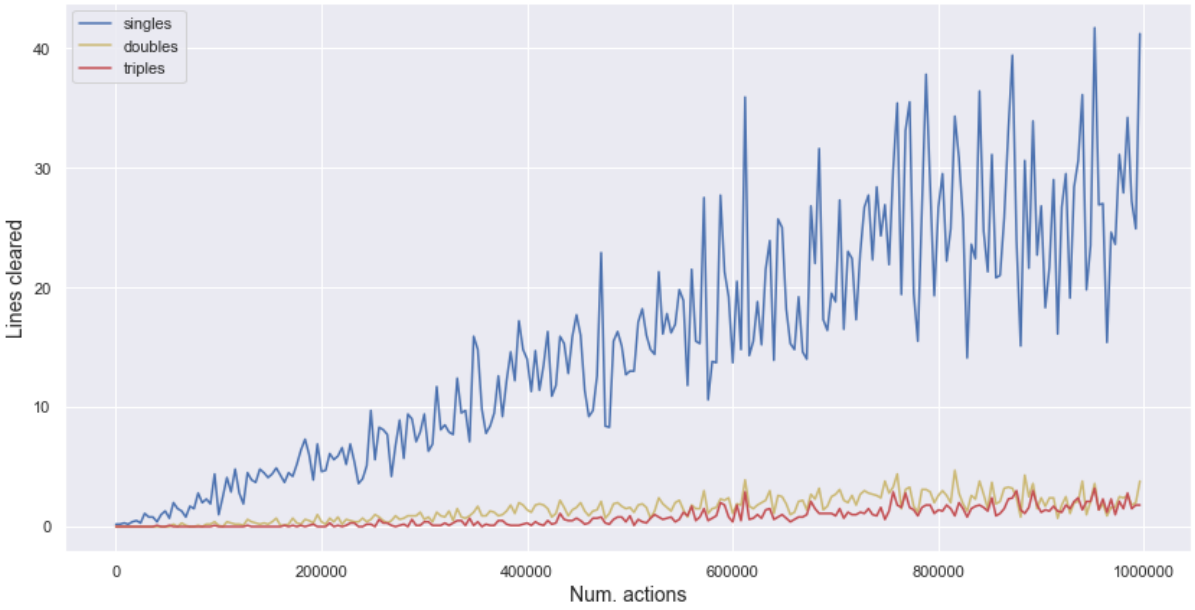
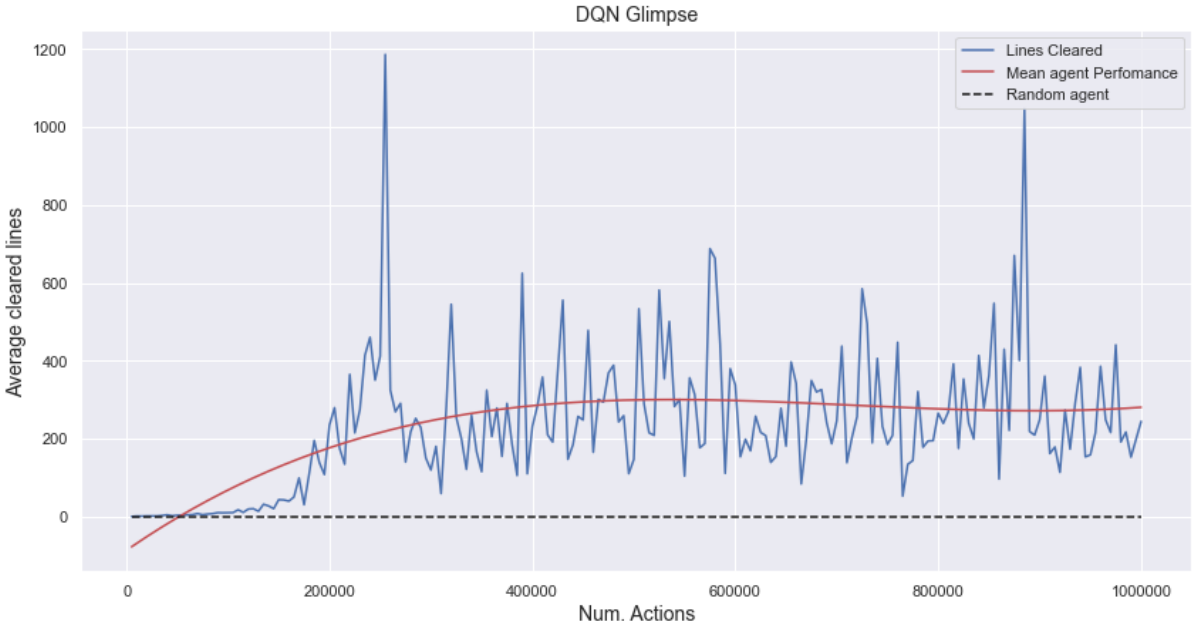


Figure 6.14: PPO's Performance for the board representation (top) and the type of lines cleared (bottom)

Both the agents clear around the same number of lines and exhibit the same behavior of clearing mostly single lines. However, the PPO agent shows a steady learning curve with no significant signs in learning how to clear more than one lines.

The overall behavior of both agents is to be expected as they have no idea what's coming next. Therefore, stacking the pieces appropriately is almost impossible.

6.2.3 Glimpse Representation



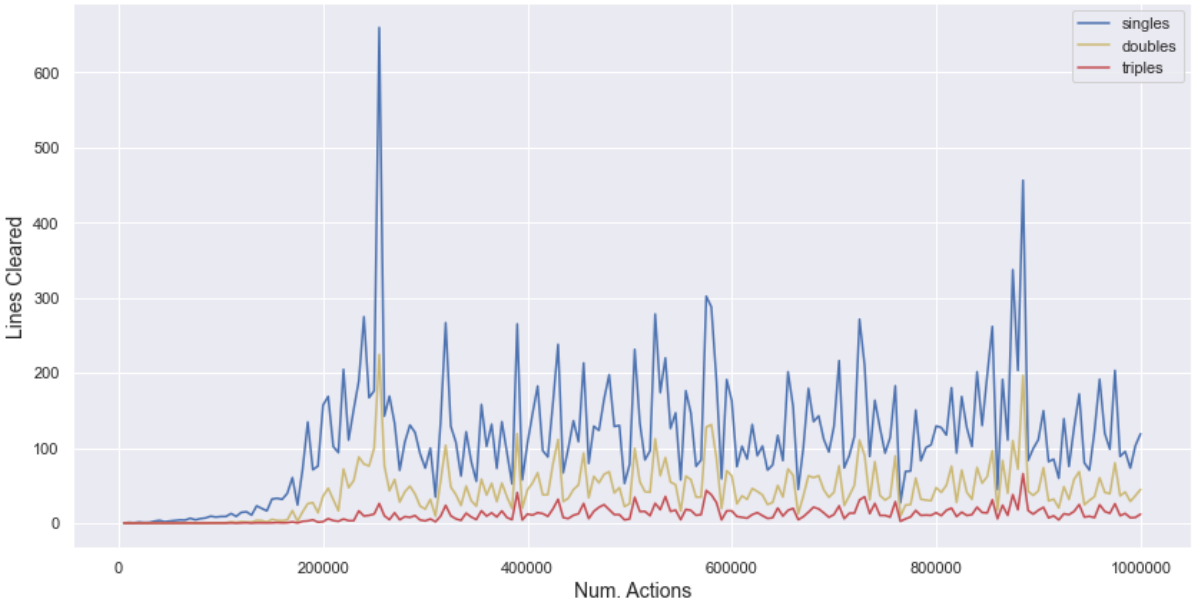
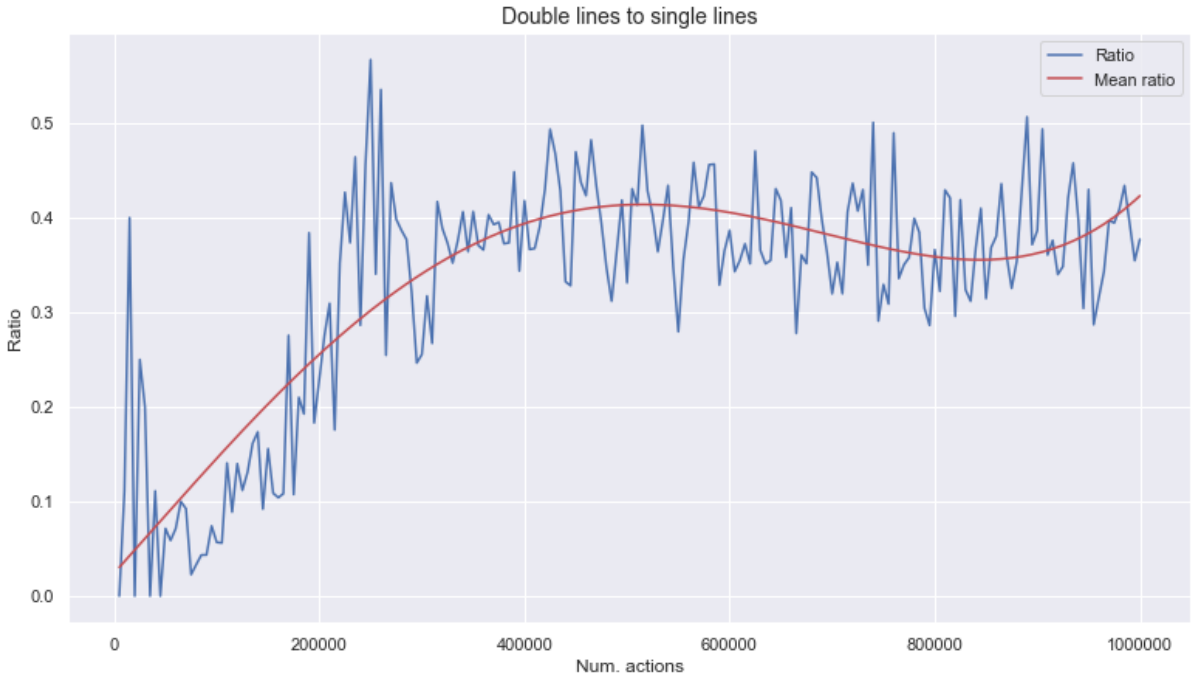


Figure 6.15: DQN's Performance for the glimpse representation (top) and the type of lines cleared (bottom)

By changing the state representation so that the agent knows which two pieces are following, the performance increases from clearing 40 at its previous peak to steadily clearing around 300 lines. The agent still 'prefers' its endurance policy and clears more single lines than doubles. At the same time, the same pattern in dropping performance persists to exist. After its peak around 250 thousand frames, the number of single lines cleared starts to slowly drop, while starting to clear simultaneously more lines.



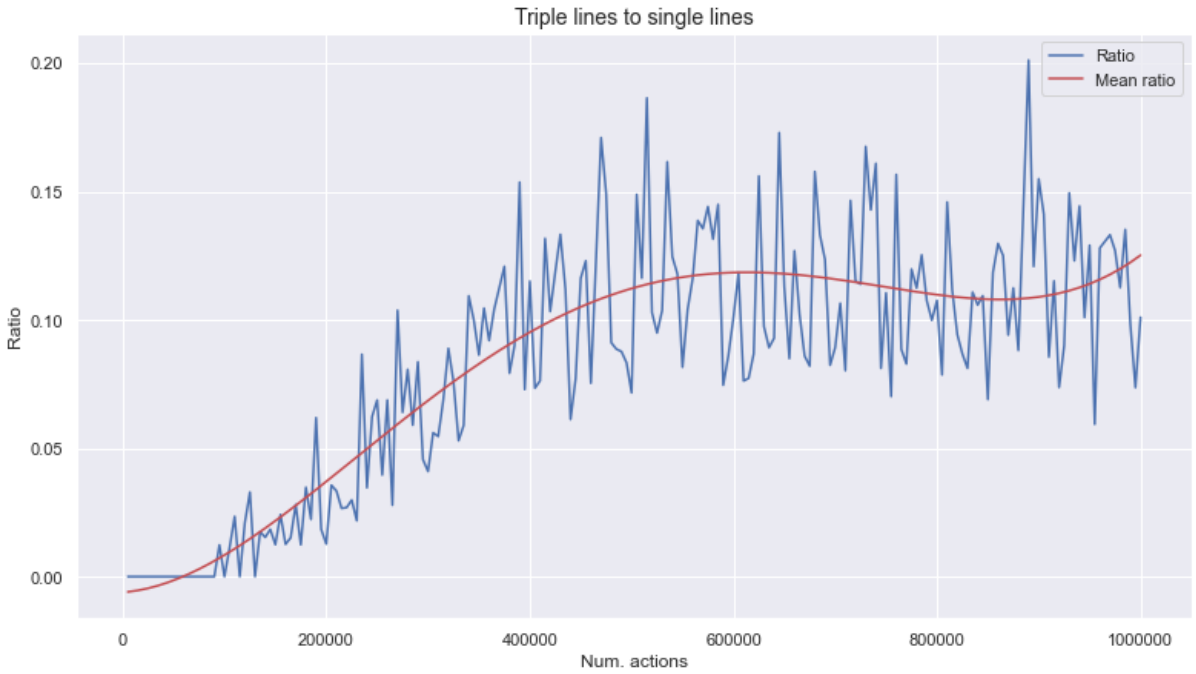
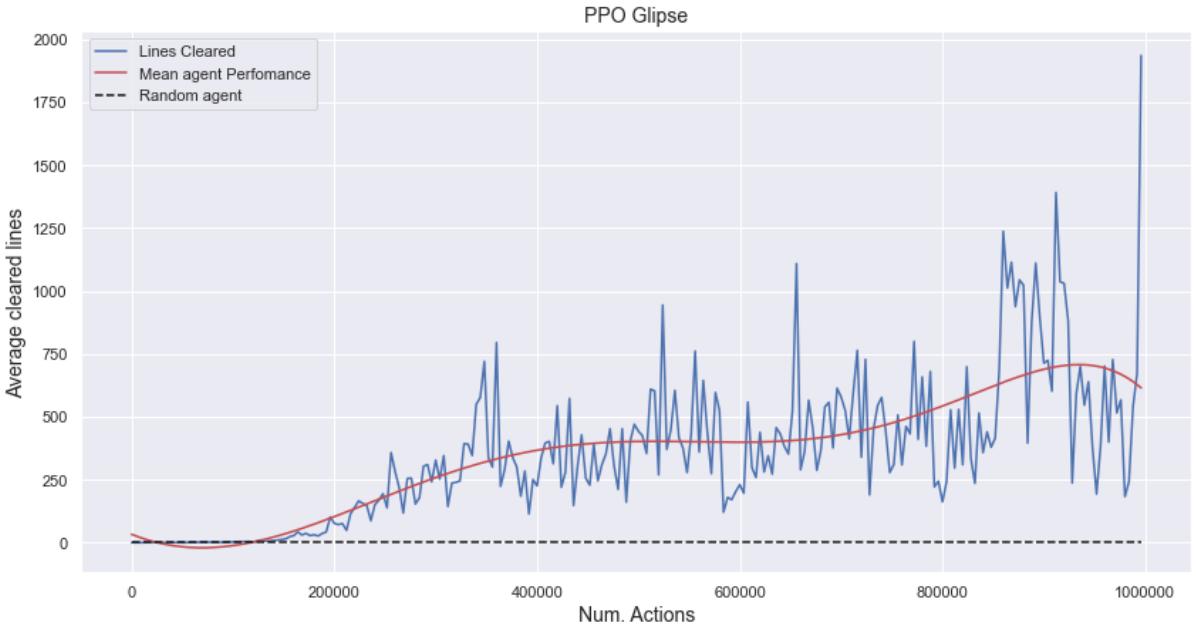


Figure 6.16: Ratio of double lines to single lines (top) and triple lines to single lines (bottom). Both graphs seem to converge around 500 thousand actions in

Even though the ratios are low in figure 6.16, the agent demonstrates an understanding of on clearing multiple lines. For the first time, the DQN shows that it can clear triple lines at certain occasions making its big breakthrough. However, the epsilon randomness, even though necessary to train the 'eyes' of the agent, seems to hinder its final performance. When the ratio of double lines to singles and triple lines to single converges, the agent forgets how to properly clear single lines. The adjustment in the parametric weights causes an imbalance in the agent's play style. On the other hand, PPO perfect the game:



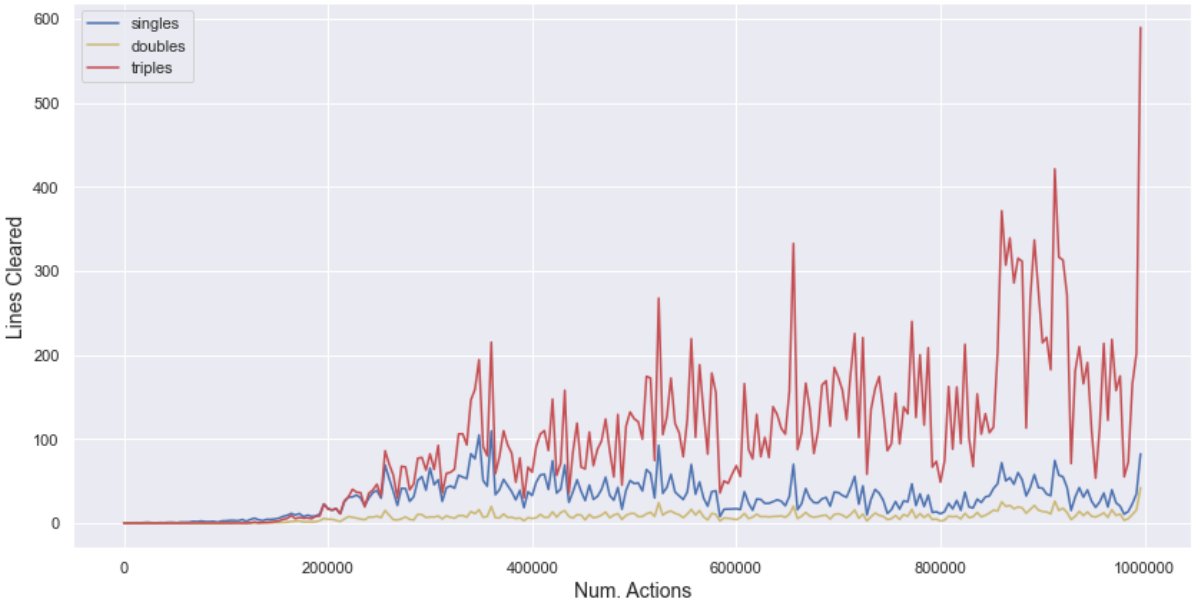


Figure 6.17: PPO's Performance for the glimpse representation (top) and the type of lines cleared (bottom)

Once again, the PPO demonstrates great understanding of the game's tactics, achieving the highest number of cleared lines so far with an average of over 600 lines per episode. PPO can outperform the DQN agent at the 300 thousand actions threshold where it is able to clear over 250 lines per episode. The most impressive feature of the PPO Glimpse is the fact that after just 200 thousand actions, the agent clears more triple lines that single ones. By the one million threshold point, the agent perfects this tactic. In fact, the agent performs the same way as the PPO 12-feat, keeping column 3 empty to drop the vertical I piece. This can be demonstrated in feature 6.19.

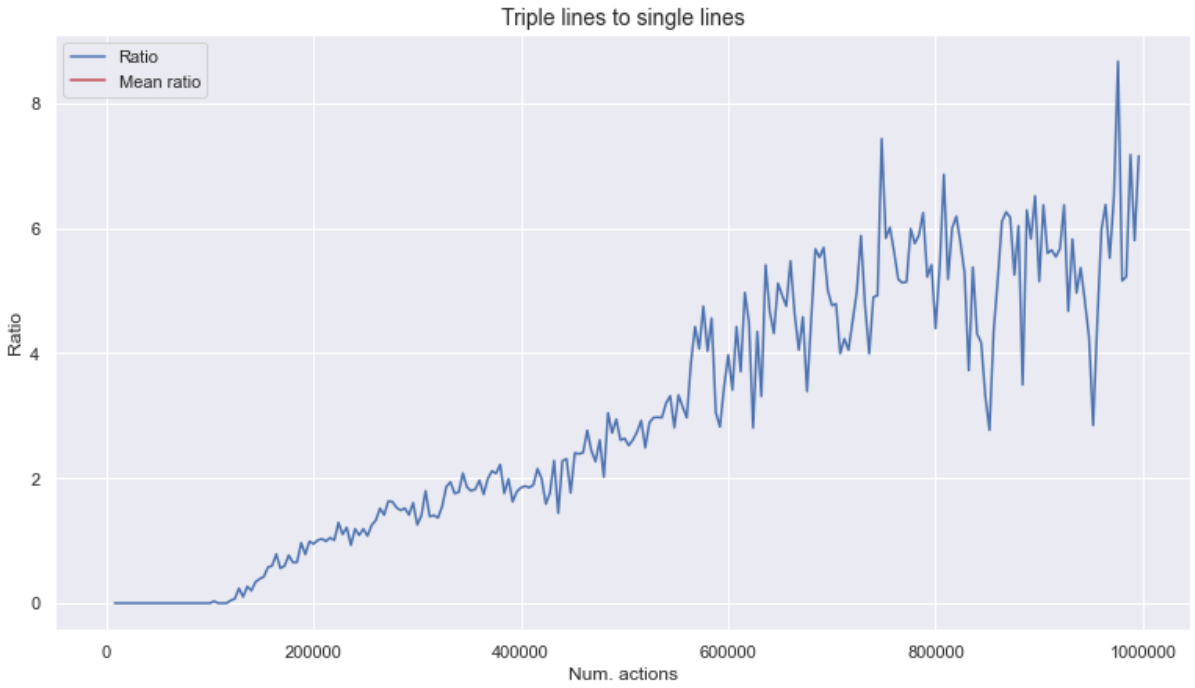


Figure 6.18: Ratio of triple lines to single lines bottom of the PPO Glimpse

The agent can clear 6 times more triple lines than single lines by stacking the blocks showing a deep understanding on how to stack blocks on top of each other. Then it can clear in quick succession multiple triple lines. Figure 6.19 demonstrates this effect.

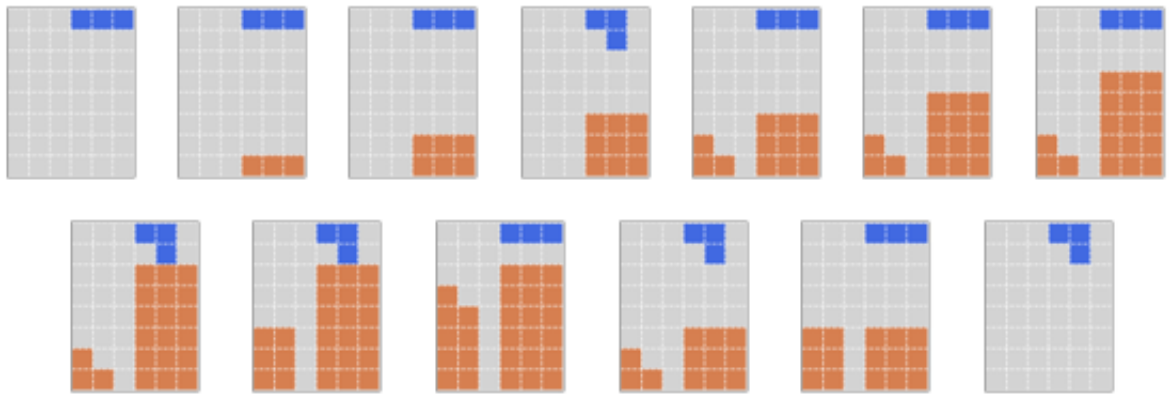


Figure 6.19: PPO Glimpse clearing triple lines in quick succession

6.2.4 Arising inconsistencies

For result's consistency all algorithms were run from scratch for a second time. In most of the cases the results were duplicated. However, the result's of PPO Glimpse were very difference the second time. The second run showed a very different behavior.

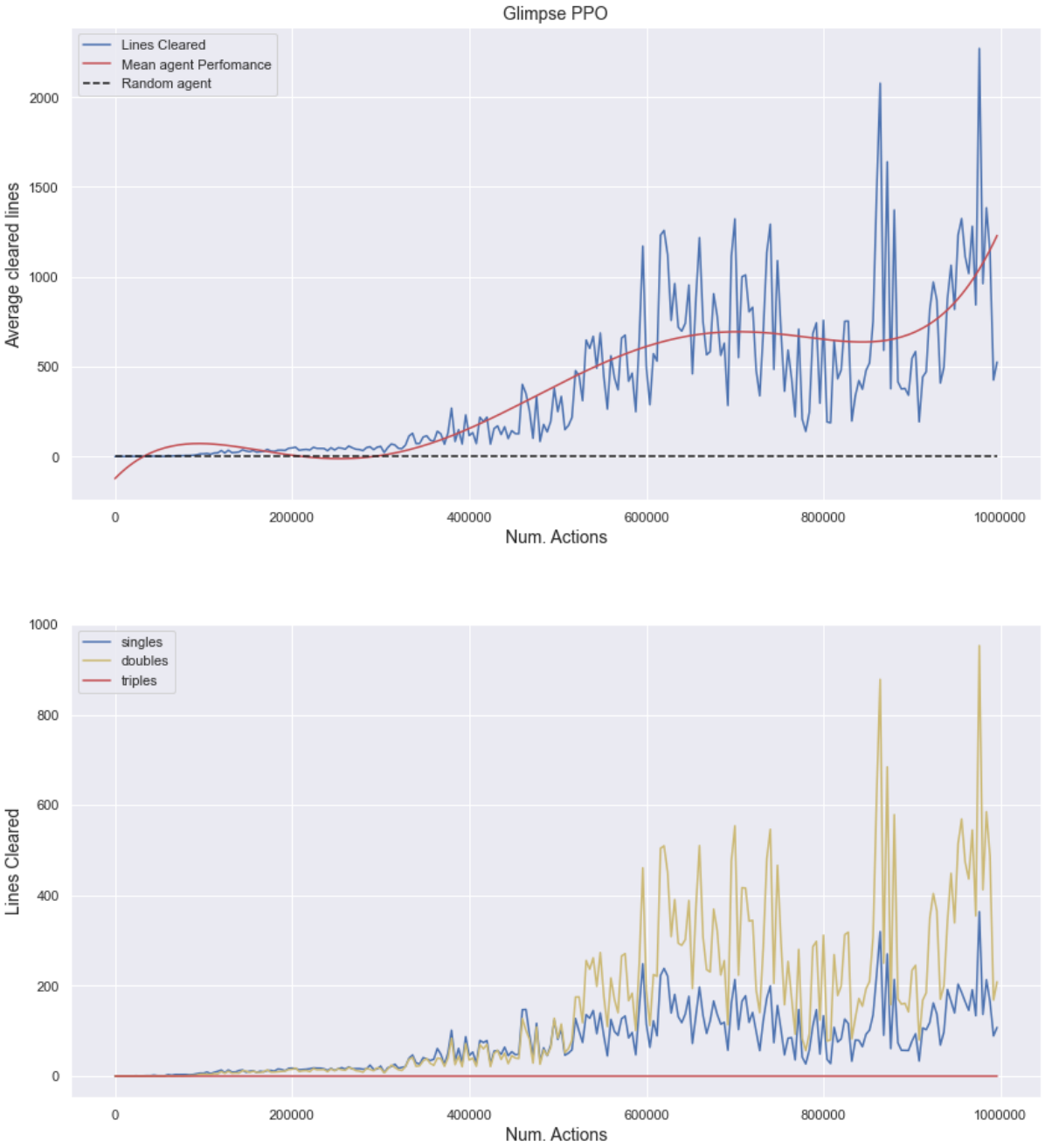


Figure 6.20: PPO's Performance for the glimpse representation of the second run (top) and the type of lines cleared (bottom)

Even though the hyper-parameters, the agent, and the state representation were exactly the same, the second run was able to clear almost 2000 lines per episode by using a different approach. The agent never encountered a triple line, so it never learned how to wait and gain a superior reward. Instead, it was able to understand how to clear double lines and developed an early preference towards them. Even though the average number of lines cleared it's higher, the first run achieved a higher reward due to the exponential factor of the reward function. This raises some bias concerns regarding the PPO agent where the weights of Neural Net converge fast not allowing the agent to explore the environment properly. Furthermore, it showcases that PPO performances can sometimes be up to luck. In this case, the agent was unlucky enough to not encounter a triple line, so it never learnt the optimal strategy.

6.3 Reward Shaping

As discussed previously, the aim of the project is to perfect the game of Tetris by teaching the agent to clear simultaneously as many lines are possible. Even though the reward function used so far, focuses on rewarding the agent for clearing many simultaneous lines. However, some agents like the DQN have hard time learning the optimal policy, choosing a more stable approach by clearing single lines. Therefore another reward function designed to help the agents was created based on the weights discussed in section 4.2.2.

$$\vec{p} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}, \vec{x} = \begin{pmatrix} AggregateHeight \\ CompleteLines \\ Holes \\ Bumpiness \end{pmatrix}$$

$$a = -0.51$$
$$b = 0.76$$
$$c = -0.35$$
$$d = -0.18$$

$$Reward = \vec{p} \cdot \vec{x}$$

The reward function does not reward the agent for clearing multiple lines, but instead it rewards it for building a proper structured geometry, with no holes, low bumpiness and in general low height. This way the agents can learn to play the game longer, instead of playing the game better.

However, after some training, it was clear that this reward function produced bad performance. Figure 6.23 demonstrates the results.

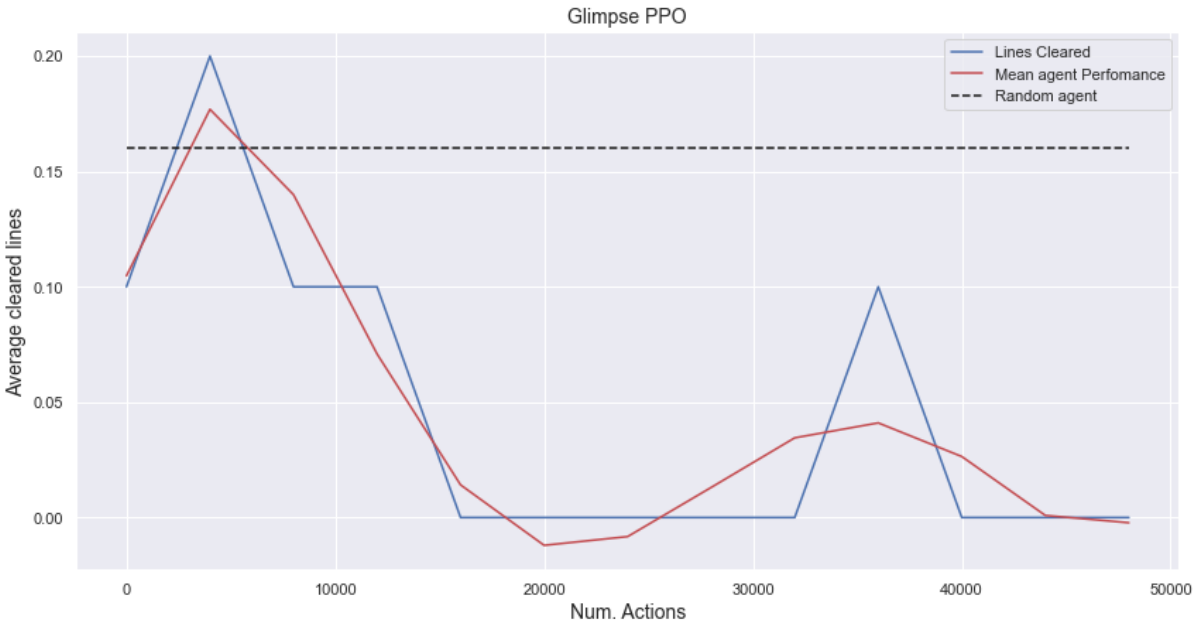


Figure 6.21: PPO's Performance on the 8x6 board for the glimpse representation and the structuring function

Even though the PPO Glimpse version performed the best so far, it quickly decided to start losing as fast as possible to avoid from getting negative regards. It is the first time an agent performs worst than a random agent. This is the same phenomenon that occurred in section 6.1. The DQN was not used, to avoid computational costs. Since the main idea is to help the agent clear lines, the weight of the LinesCleared was increased from 0.76 to 10. This was done to show the agent the emphasis of clearing lines. Therefore the reward function transformed to:

$$\vec{p} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}, \vec{x} = \begin{pmatrix} AggregateHeight \\ CompleteLines \\ Holes \\ Bumpiness \end{pmatrix}$$

$$a = -0.51$$

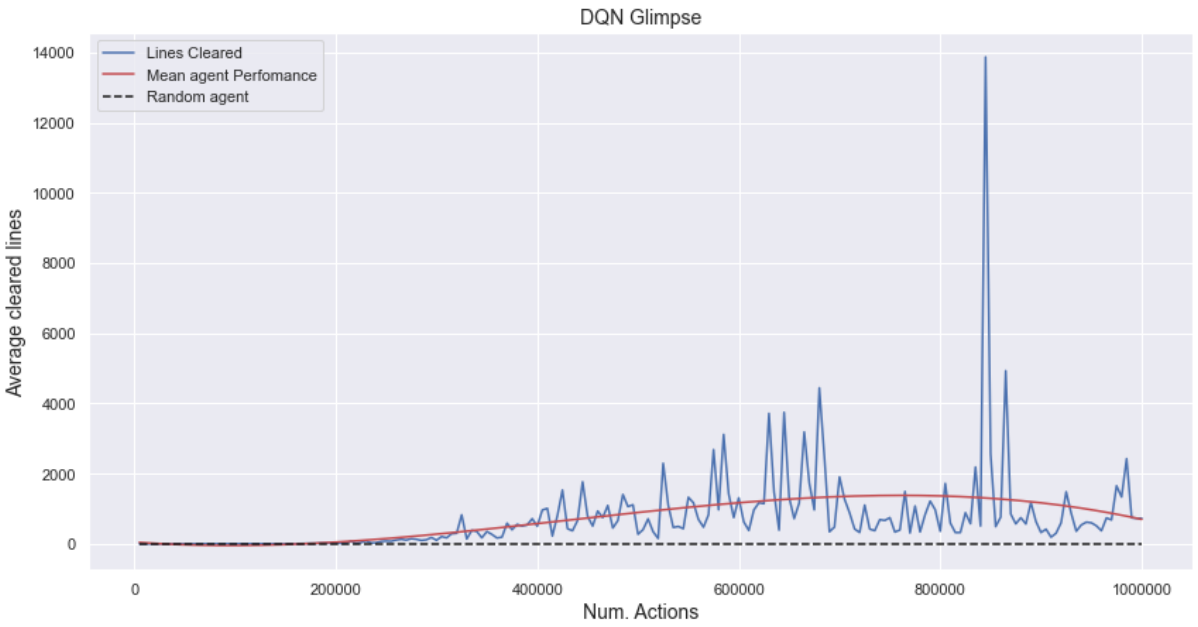
$$b = 10$$

$$c = -0.35$$

$$d = -0.18$$

$$Reward = \vec{p} \cdot \vec{x}$$

The weights of the remaining features stayed the same. This will be referred as the Structuring Reward Function from now on. The following results were produced:



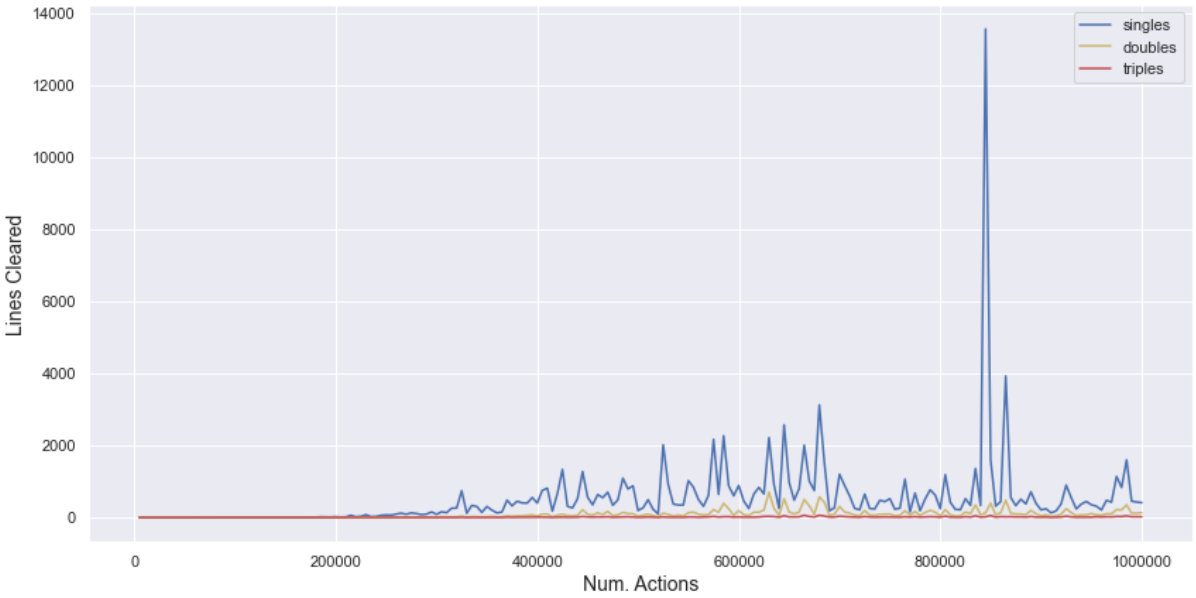
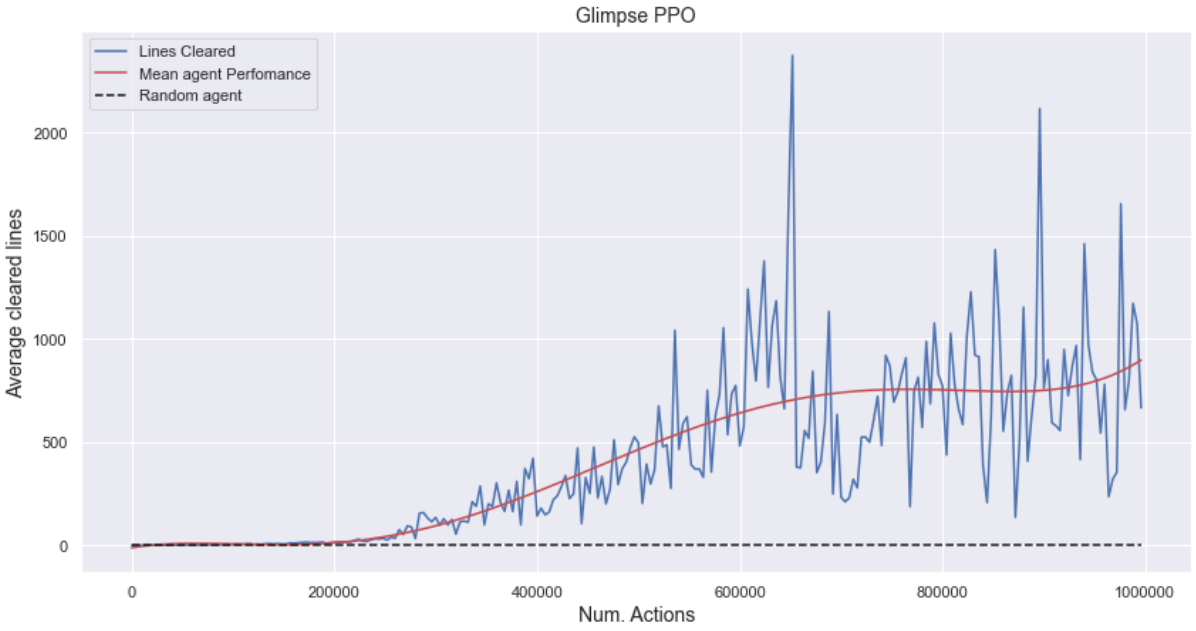


Figure 6.22: DQN's Performance on the 8x6 board for the structuring function (top) and the type of lines cleared (bottom)

As expected, the agent is not able to clear multiple lines simultaneously. However, the state representation along with the new reward function mark the highest results of cleared lines so far, clearing around 2000 lines. Around 845000 thousand actions in, the agent averaged at almost 14 thousand cleared lines, surpassing previous high scores by 7 times making it a very strong contender for attempting the full board. Unfortunately, performance dips once again after that but it still shows a very high level of understanding.



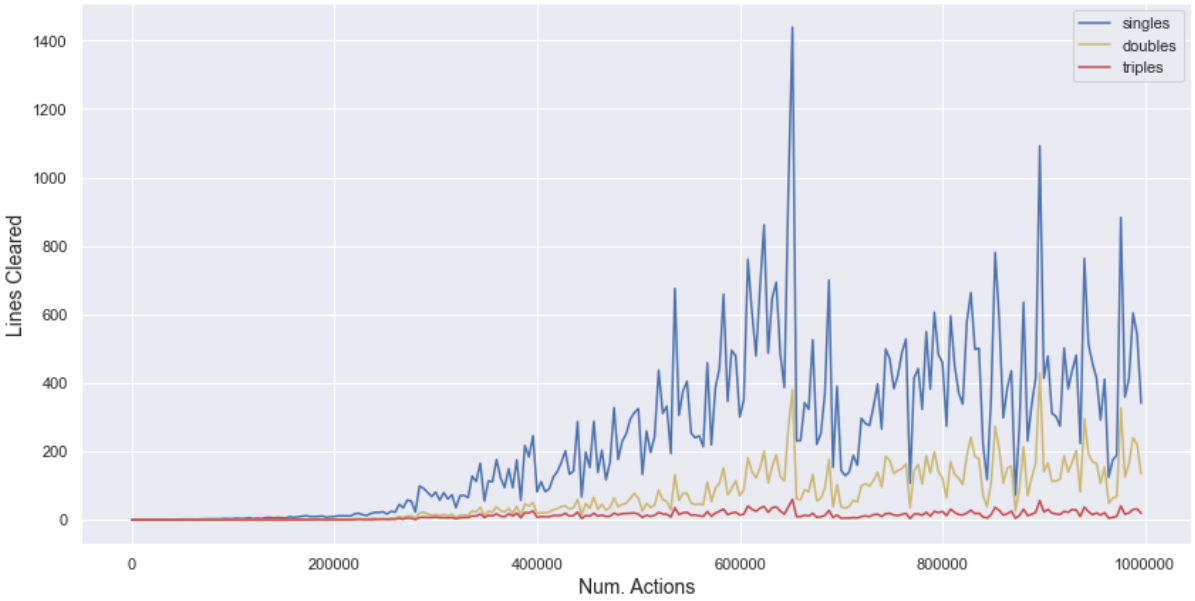
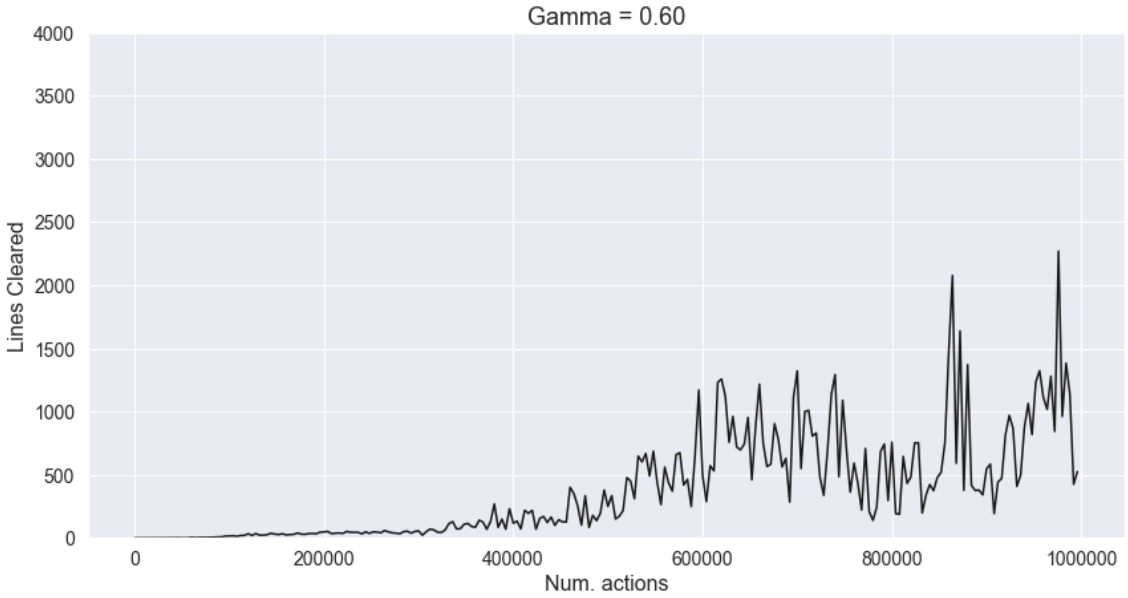


Figure 6.23: PPO's Performance on the 8x6 board for the structuring reward function (top) and the type of lines cleared (bottom)

Even though PPO does not clear as many lines as its DQN counter part, it is still able to surpass most previous versions of PPO in clearing lines. The agent does not play the perfect game anymore. It clears mostly single lines and occasionally double lines. However it is able to last longer and play the game even more, clearing more lines at the end. A very good contender for the full board.

6.4 The effect of Discount rate

As mentioned in section 3.2.1 Gamma, the discount factor, quantifies how much weight it's given to future rewards. The lower the gamma, the agent will tend to consider only instant rewards. The 8x6 Glimpse version of the PPO agent along with the Perfecting function will be used to tune the gamma appropriately, as it showed great potential for understanding future rewards and the training is fast. The follow results were recorded:


Gamma = 0.60
Lines Cleared
Num. actions
0
500
1000
1500
2000
2500
3000
3500
4000
0
200000
400000
600000
800000
1000000

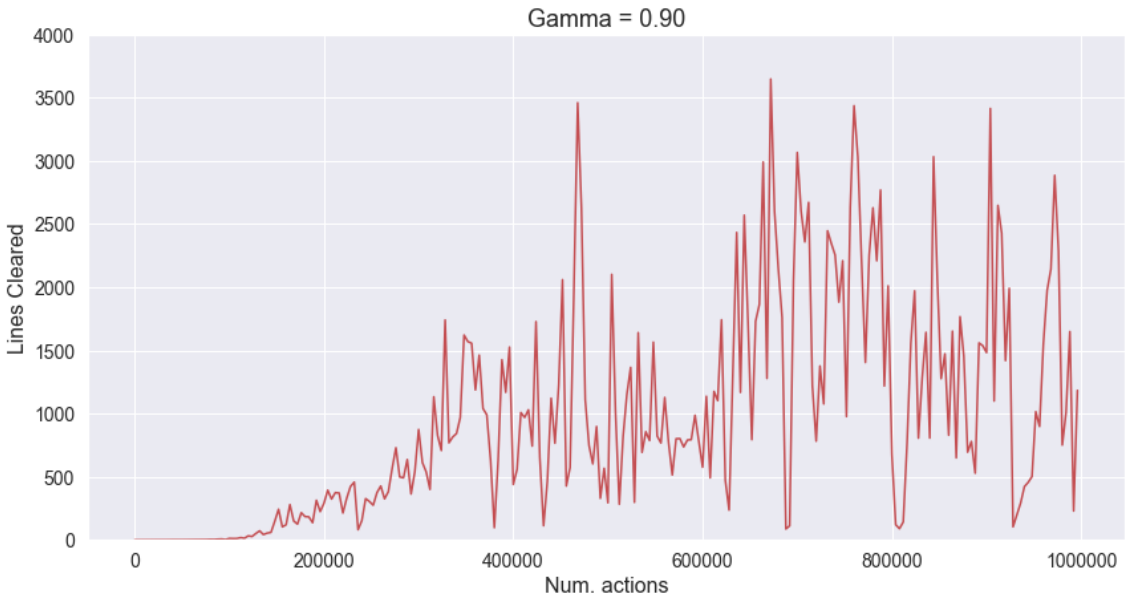
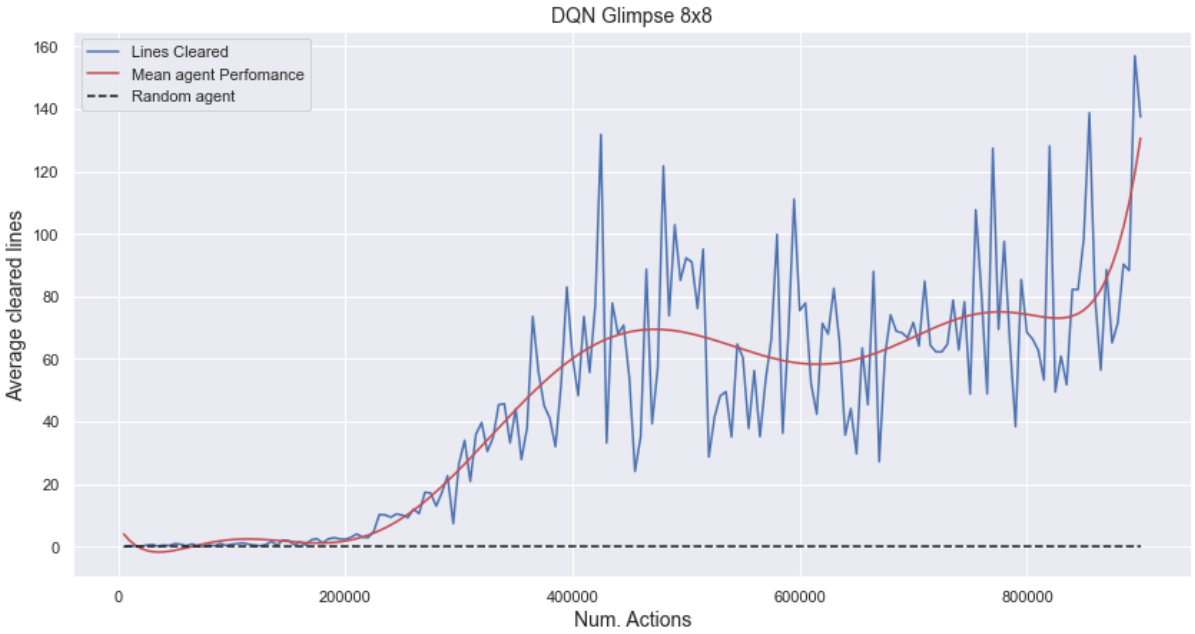

Gamma = 0.90
Lines Cleared
Num. actions
0
500
1000
1500
2000
2500
3000
3500
4000
0
200000
400000
600000
800000
1000000

Figure 6.24: The Performance of the PPO agent using four different gammas. From the smallest-0.60 (top) to the highest-0.99 (bottom)

Each agent was trained for one million actions. With a gamma of 0.6 the agent learns very slowly, and it takes a longer period to understand how to clear some lines. When the gamma is modified to be above 0.9 all of the agents learn to pick up the strategy around the 200 thousand threshold. The best results that produce the maximum number of lines cleared arise when gamma is 0,90. However the spiky nature of the graph suggest that the learning process is very unstable, going all the way from clearing an average of 3500 lines to clearing almost 0 in just a few thousand actions. The next best performing agent uses a gamma of 0.99 which emphasizes on future results, which is the agent discussed in section 6.2.3. In fact, it is the only agent being able to clear any triple lines as shown by figure 6.17. The version with gamma 0.95 is inferior to both 0.90 and 0.99 as it shows no improvement. Therefore, the PPO version using the glimpse representation will use both 0.90 and 0.99 in the full board environment.

6.5 Tetris 8x8

Another factor that might be affecting the quality of the results so far is the board geometry. Both boards tested so far, the 4x4 and the 8x6, have an interesting geometry pattern, which helps the agents reach better findings. The shapes' size on each board can complete a whole line with just two actions. This allows for easy placement of the pieces. Looking back at the Board representation, the agents performed very well, even without knowing which pieces are coming clearing mostly single lines. This explains why the agent's performed so well. To investigate this, a different environment was set up with the same geometrical shapes used in the 8x6 board, but with 8x8 dimensions. To an average human player, this is the equivalent of setting the mode to 'easy' as the game is the same, with a wider board to play around. Using the Glimpse Feature representation, the following results were recorded:



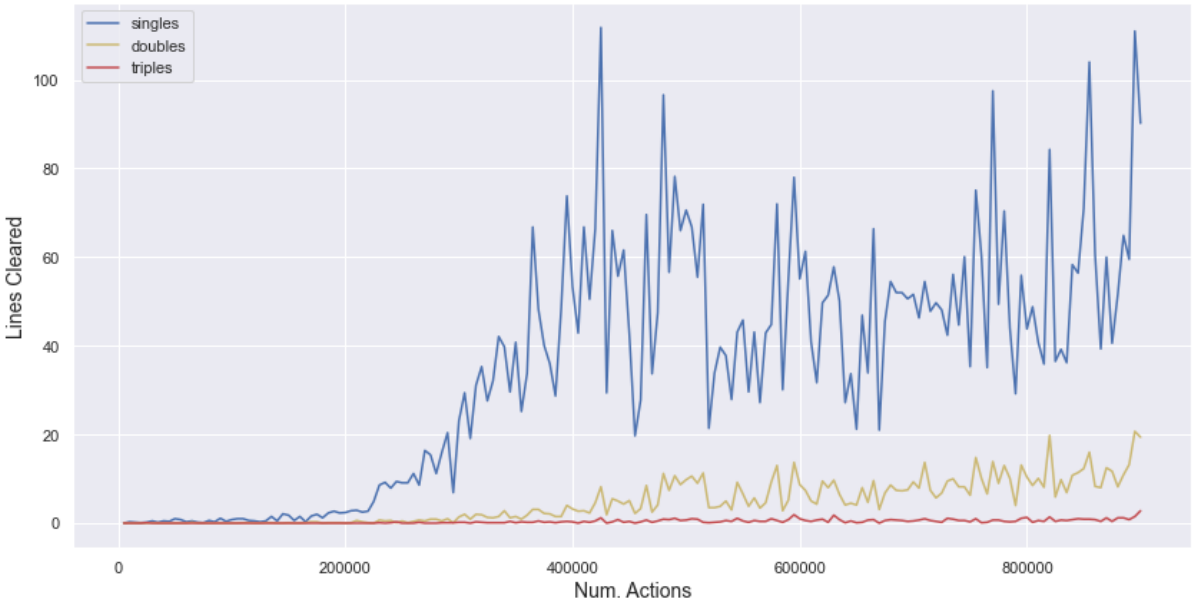
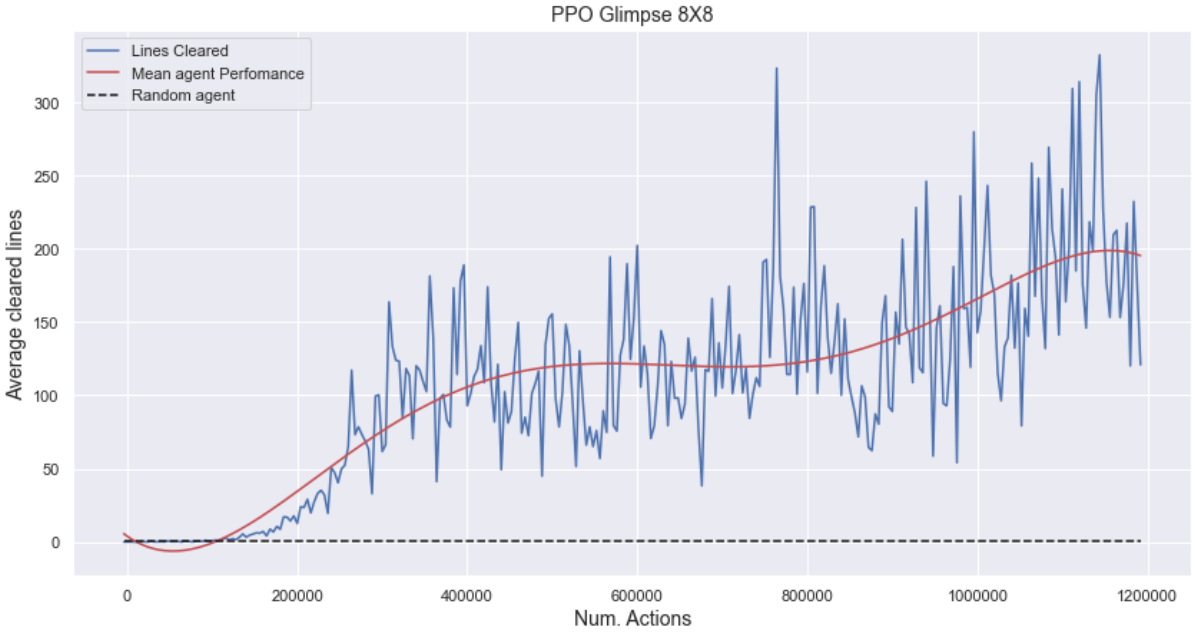


Figure 6.25: DQN's Performance on the 8x8 board for the glimpse representation (top) and the type of lines cleared (bottom)

The DQN's results dropped by from clearing 300 lines, peaking now at around 120 lines. The continuing endurance trend of the DQN continues, learning to clear mostly single lines once again.



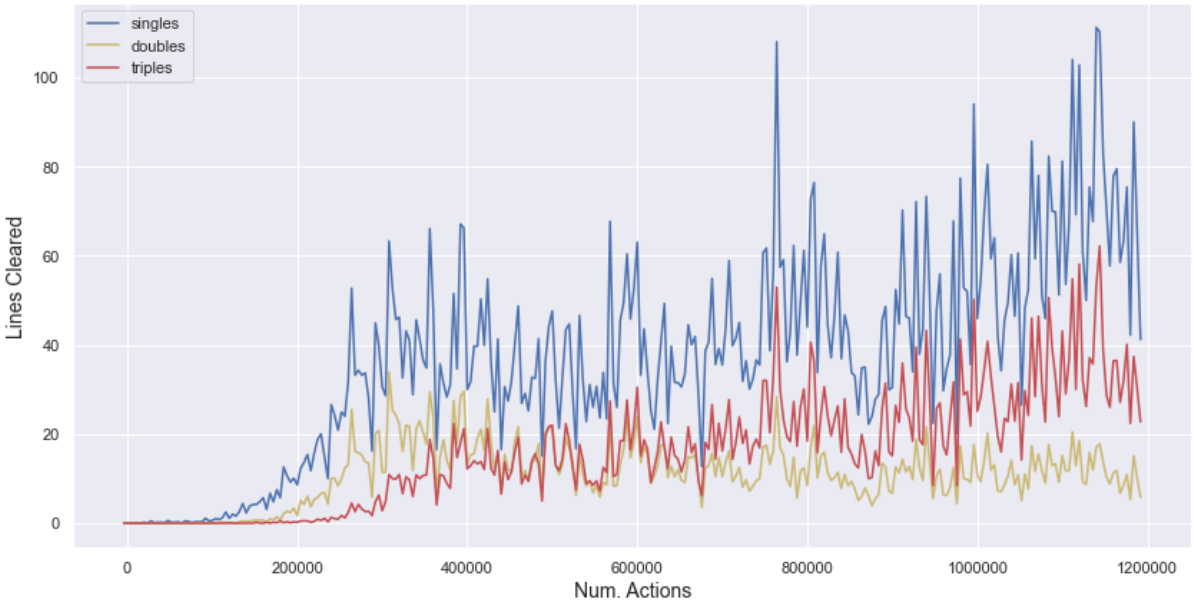


Figure 6.26: PPO's Performance on the 8x8 board for the glimpse representation (top) and the type of lines cleared (bottom)

The PPO's results dropped by from clearing 600 lines, peaking now at around 200 lines. Both agents saw a drop in performance, clearing around a third of the lines that they used to clear. However, PPO is no longer finding the optimal policy. PPO is now clearly mostly single lines, instead of triples. This is a major drop in the quality of the agent and further showcases its instability to adapt to a more complex environment. On the other hand, DQN's result continue to be consistent. The drop can be justified as the search space of the environment has now been multiplied by a factor of 2^{16} due the 16 more blocks introduced to the board. This also speaks to the complexity of the actual board, where the action and the search space are very wide.

6.6 Overall Testing Results and Expectations

State Representation	Agent	Environment	Reward Function	Lines Cleared
4-feat	DQN	8x6	Perfecting	1.5
4-feat	PPO	8x6	Perfecting	3.5
12-feat	DQN	8x6	Perfecting	4.5
12-feat	PPO	8x6	Perfecting	500
Board	DQN	8x6	Perfecting	30
Board	PPO	8x6	Perfecting	35
Glimpse	DQN	8x6	Perfecting	300
Glimpse	PPO	8x6	Perfecting	600
Glimpse	PPO 2^{nd}	8x6	Perfecting	1850
Glimpse	DQN	8x8	Perfecting	125
Glimpse	PPO	8x8	Perfecting	200
Glimpse	DQN	8x6	Structuring	2000
Glimpse	PPO	8x6	Structruing	1000

Table 6.1: Testing Results

Overall, using the Perfecting reward function, the PPO agent was able to match or surpass the performance of the corresponding DQN agent but showed signs of unstableness while it doesn't respond well to an increase in complexity. The DQN performance was stable and recursive over every state showing that it is unable to learn how to clear more double and triple lines than single lines. The PPO versions of the perfecting reward function are expected to perform well on the full board and have the highest expectations of clearing simultaneously many lines on the big board.

On the other hand, using the structuring reward function, the DQN was able to demonstrate great understanding, clearing thousands of lines. The PPO performed well but not to the extent of its corresponding DQN. Using the structuring reward function, the agent should be able to clear more lines on the big board, while the DQN is expected to clear the most lines out of every agent.

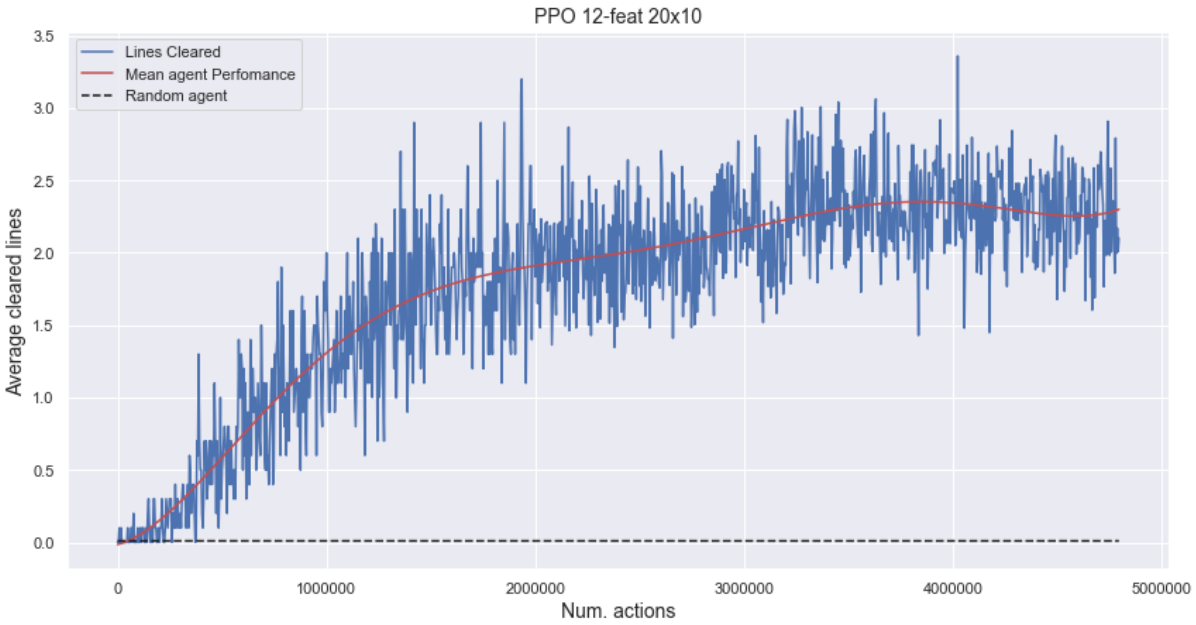
Chapter 7

Results and Discussion

7.1 Perfecting Function

7.1.1 PPO 12-feat

The 12-feat was tested first and the results are shown below. The results below are averaged over 10 agents using an epsilon (randomness) of zero. For reference, a random agent on the 8x6 board clears on average 0.01 lines per game.



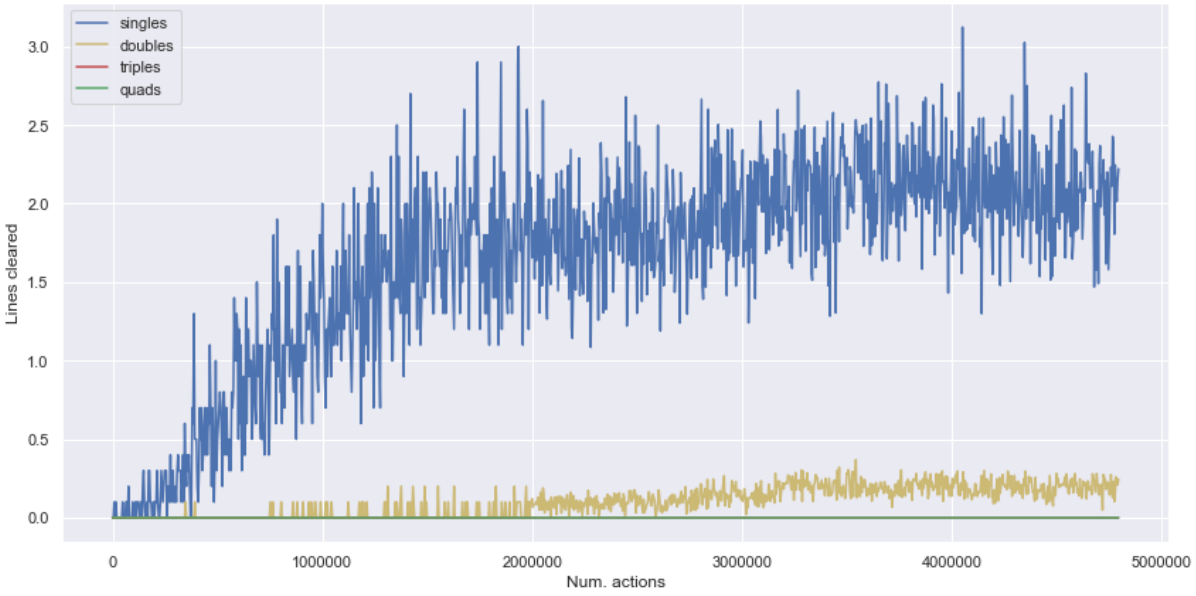
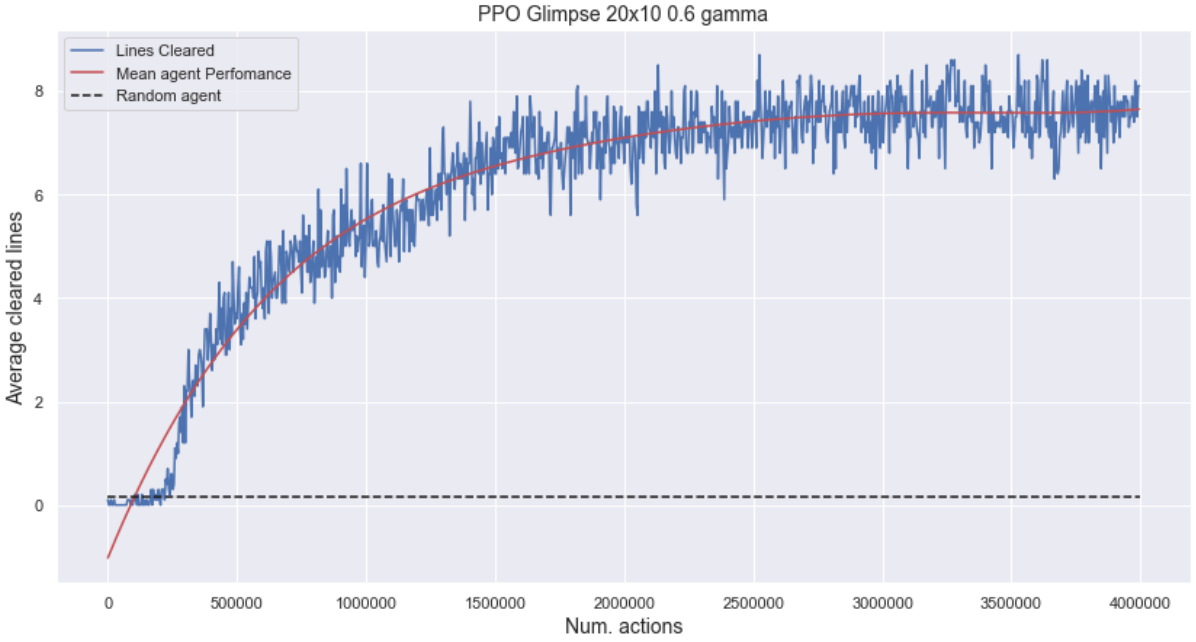


Figure 7.1: PPO's Performance on the 20x10 board for the 12-feat representation (top) and the type of lines cleared (bottom)

The agent was let to run for around 5 million actions with a discount rate of 0.99, the same as the previous successful 12-feat representation. Even though it is no match for its previous performance, the results converged around the 2 million threshold, clearing on average more than 2 single lines, with minimal evidence of being able to clear som double lines as well.

7.1.2 PPO Glimpse: Discount rate = 0.90



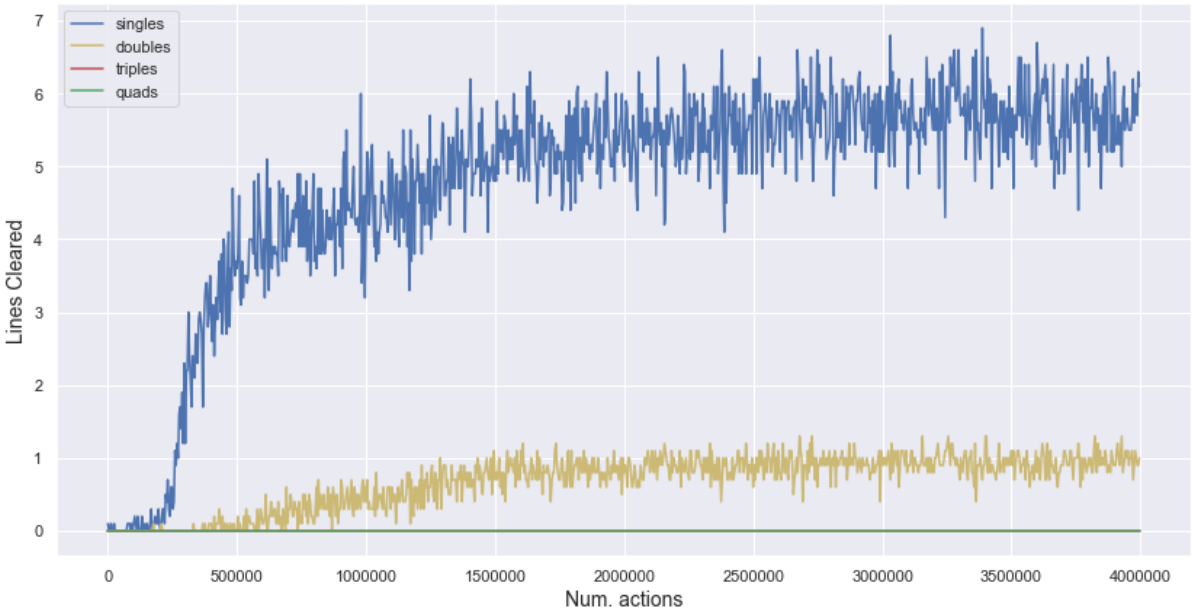


Figure 7.2: PPO's Performance with a discount rate of 0.90 on the 20x10 board for the glimpse representation (top) and the type of lines cleared (bottom)

The PPO Glimpse version having a discount rate of 0.90 cleared on average almost 8 lines, surpassing the 12-feat by a factor of 4 and it converged around the 2 million threshold. Mostly singles lines are cleared and around a double line per episode. The agent behaves similarly to the its previous successful versions on the 8x6 board by leaving a 'pitch' in the middle of the board, filling blocks around it, and then trying to complete the lines. This can be demonstrated in figure 7.3, where the grid is blue, and the agent clears a double line. However, the agent does not know to properly place the pieces at the edges, leaving holes in the grid. However, this sometimes backfires at the agency. The agents drops some blocks down the pitch where there are holes at the side of the board. This can be demonstrated in figure 7.3, where the grid is red. This causes an imbalance in the already bad structure and causes it to lose much faster.

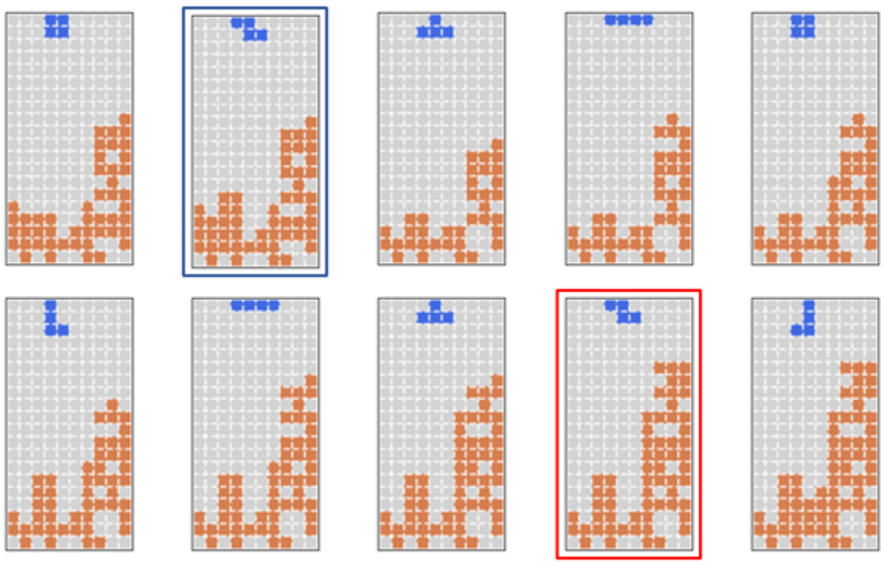
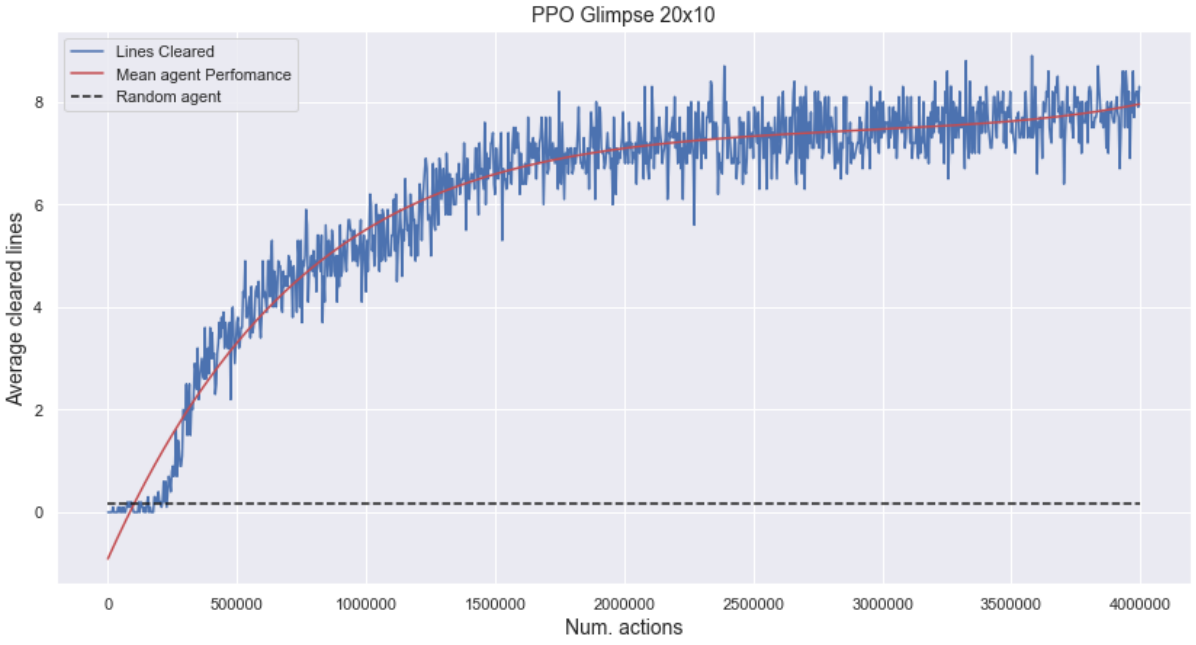


Figure 7.3: PPO's strategy over the course of 10 moves. (Glimpse-Perfecting-Gamma-0.9)

7.1.3 PPO Glimpse: Discount rate = 0.99



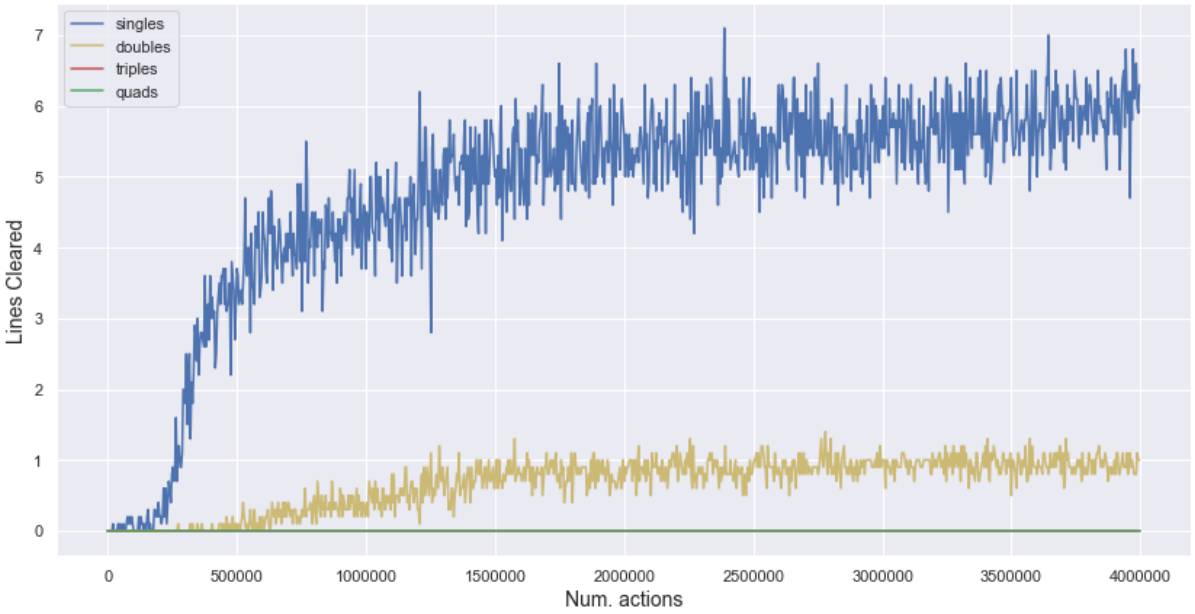
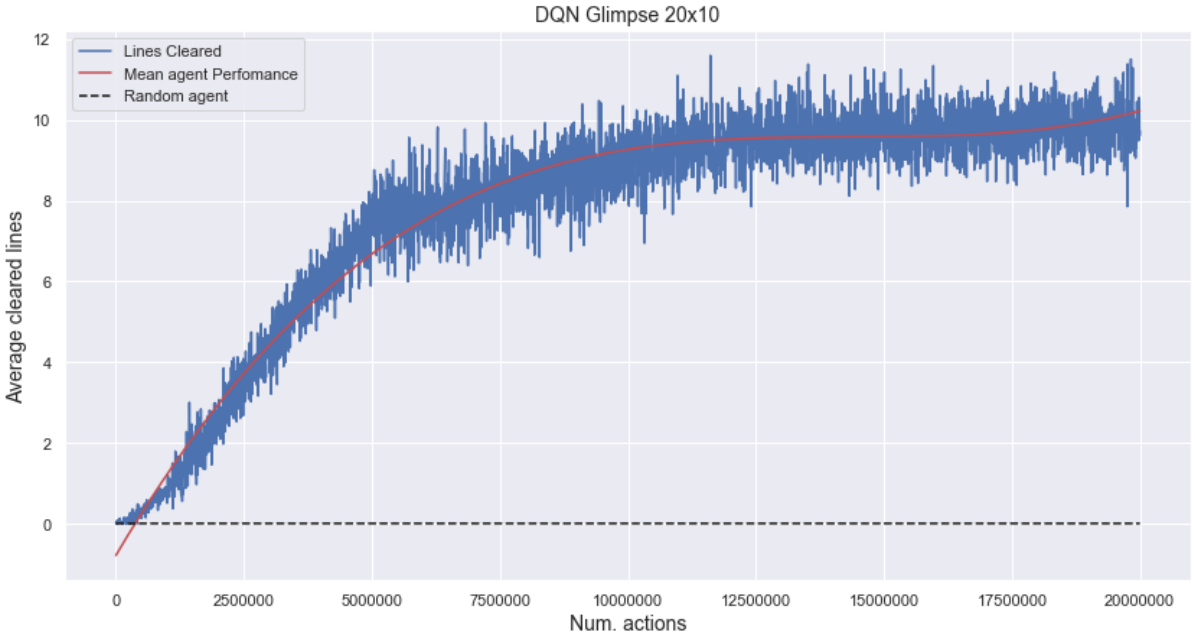


Figure 7.4: PPO's Performance with a discount rate of 0.99 on the 20x10 board for the glimpse representation (top) and the type of lines cleared (bottom)

7.1.4 DQN Glimpse



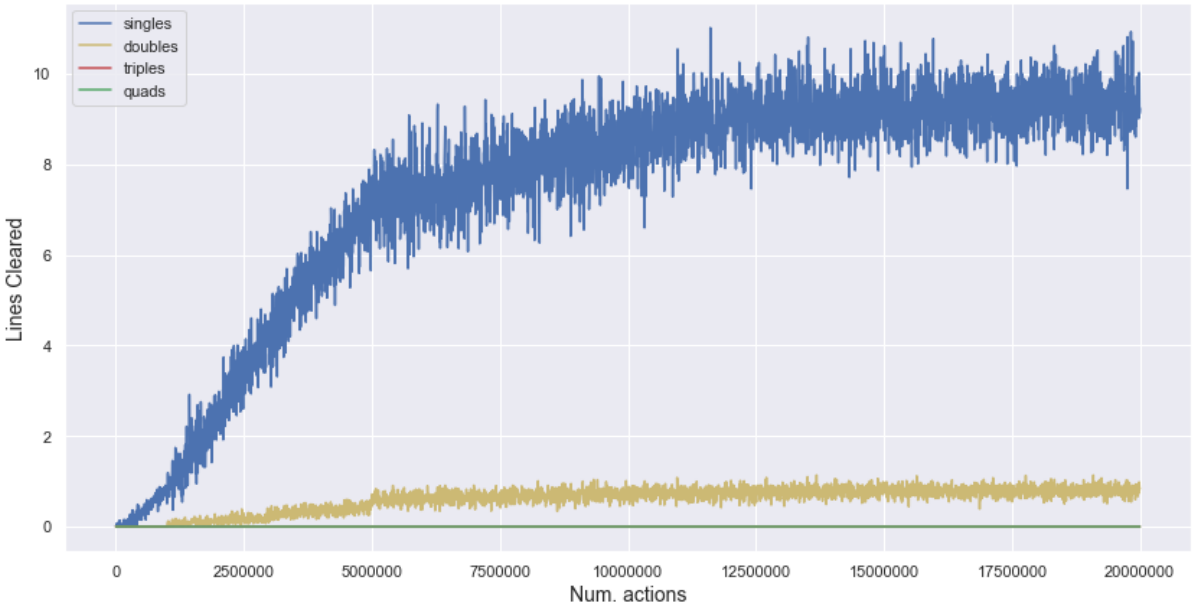


Figure 7.5: DQN's Performance on the 20x10 board for the glimpse representation (top) and the type of lines cleared (bottom)

Even though thought the PPO with the 12-feat representation was able to hugely over perform the best DQN agent at the 8x6 board, the endurance strategy of the DQN has surpassed it. The DQN agent was run for 20 million actions, which was more than 3 full weeks of training. Convergence occurs around the 12 million threshold clearing more than 10 lines in average, with around 95% of them being single lines. Once again, no sign of triple or quads line being cleared.

Overall the agents using the perfecting reward function try to fill up the board with blocks before losing, maximizing episode length as much as possible.

7.2 Structuring reward

7.2.1 PPO Glimpse

The PPO agent was trained used the Glimpse representation, a discount rate of 0.99 and the structuring reward function. However the results are very disappointing:

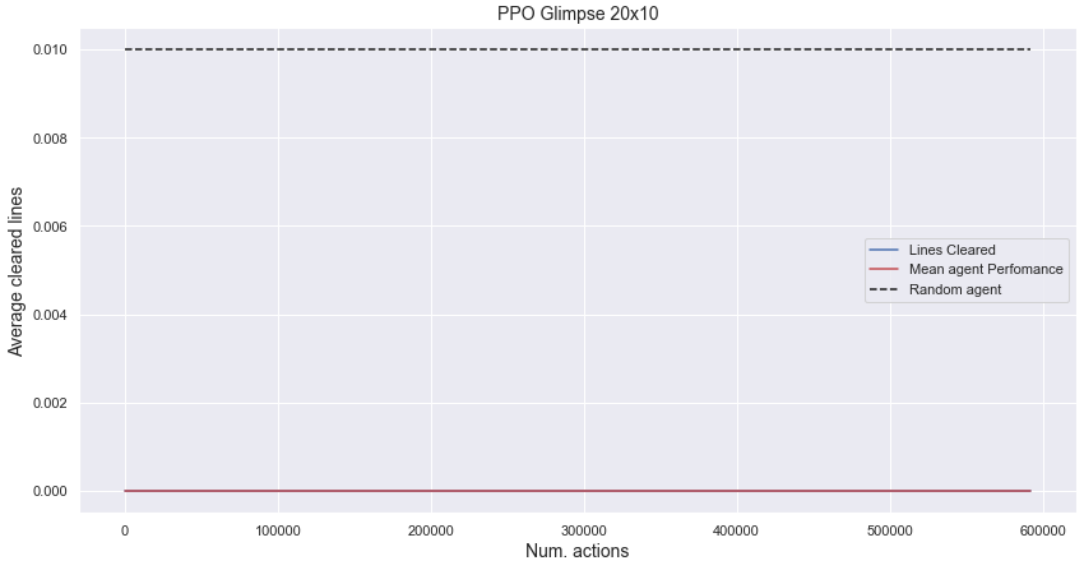


Figure 7.6: PPO's Performance on the 20x10 board for the glimpse representation using the structuring reward function

After 600 thousands actions, the PPO agent showed no results. The agent was never able to clear a line which in return developed a bias towards losing as fast as possible, in order to avoid getting more negative rewards. The agent did not learn how to play the game. In fact, the agent learn to stack up the blocks and it is demonstrated in figure 7.7

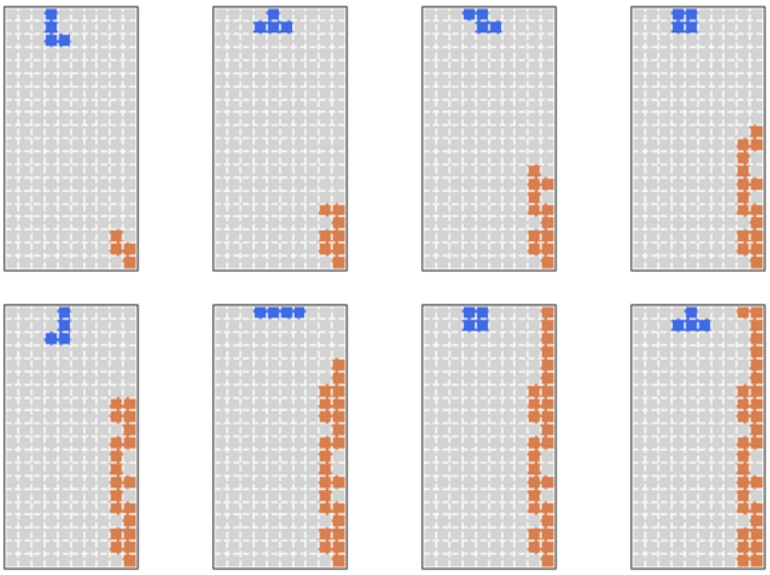


Figure 7.7: PPO's strategy using the structuring reward function. It loses in just 8 moves. (Glimpse-Structuring-Gamma-0.99)

7.2.2 DQN Glimpse

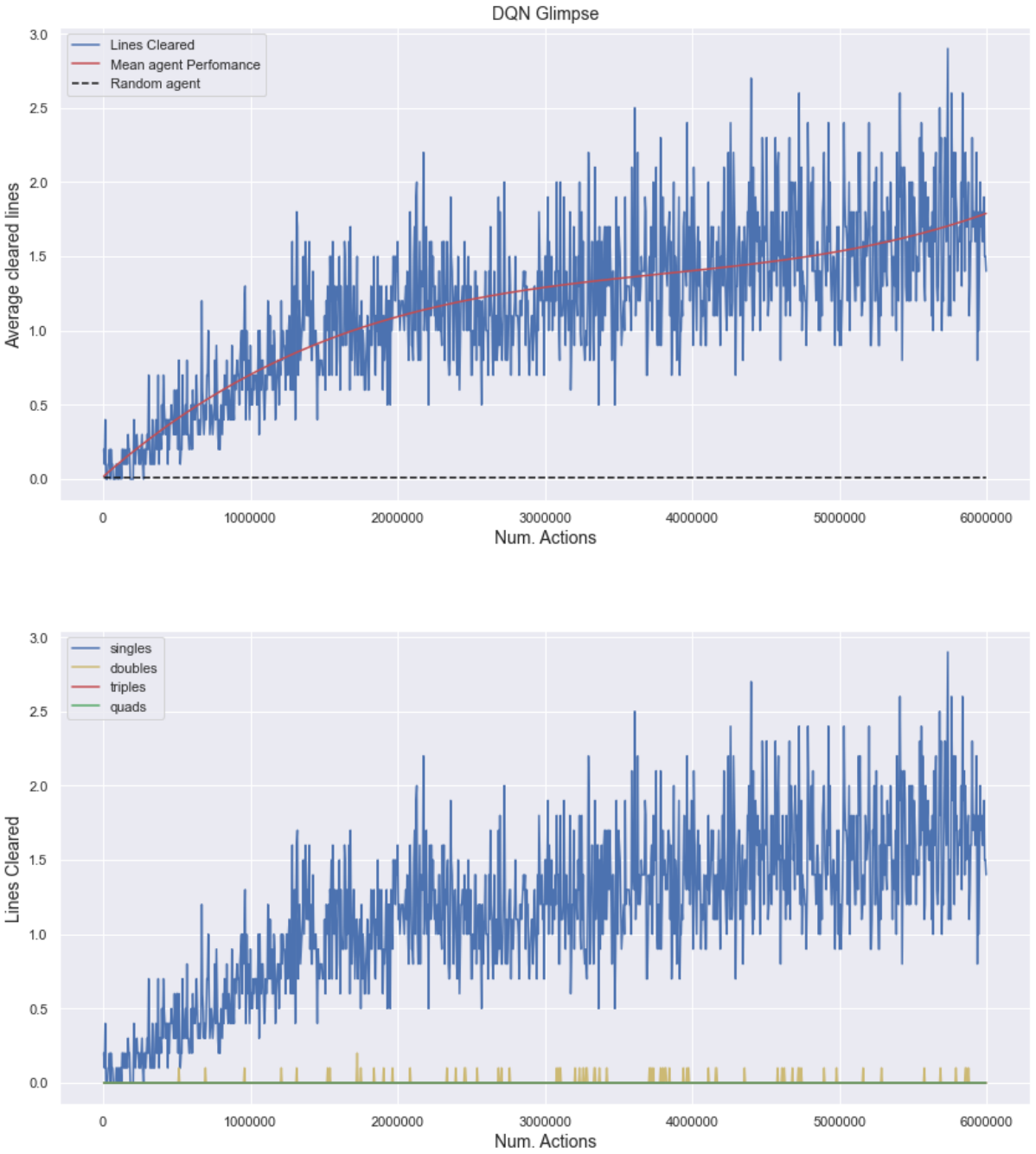


Figure 7.8: DQN's Performance on the 20x10 board for the glimpse representation (top) and the type of lines cleared (bottom)

The agent performs worse than expected cleaning on average 2 lines after a total of 6 million lines. Even though the agent didn't converge, due to tight time schedule, the training had to stop. The dimensions of the board cause the structuring function methods to fail. The function punishes the agent for building high columns and as soon as the agent passes a high of 12 blocks, the agents always gets a negative reward, no matter if lines are cleared. Therefore the agents cannot really explore the environment and prefer to lose instantly when the height exceeds 12. The 8x6 board did not face this issue due to its dimensions.

7.3 Discussion

The table below 7.1 showcases the final results of the agent.

State Representation	Agent	Reward Function	Discount Rate	Lines Cleared per game
12-feat	PPO	Perfecting	0.99	2
Glimpse	PPO	Perfecting	0.90	8
Glimpse	PPO	Perfecting	0.99	8
Glimpse	DQN	Perfecting	0.99	12
Glimpse	PPO	Structuring	0.99	0
Glimpse	DQN	Structuring	0.99	2

Table 7.1: Final Results

Overall, the agents did not perform as good as they were expected to do so. As tested in section 6.5, where the board dimensions were slightly changed, the agents did not respond well to the increase in complexity. This caused major drops in performance in every single agent, excluding the special case of PPO Structuring where the agent completely failed. Introducing 5 more shapes than the testing environment with more sophisticated geometry did not help either.

As mentioned in section 6.2.4, PPO is very volatile. This is demonstrated again in the case of PPO Structuring case in the full board. In order for the agent to realise that clearing a line will produce a a higher reward he must first clear lines, and the chances of clearing a line randomly on the full board are very low. Therefore the agents learns to maximize the reward by losing fast. The PPO agent using the perfecting function was able to clear some lines, but nothing compared to it's previous tested approaches. In general, the PPO agents under performed and they were unable to find a working policy. On the other hand, the DQN agents over performed the PPO agents in both reward functions used. Even though the agents don't clear much more lines, their endurance policy produces results.

The perfomance produced by the project cannot be compared to the results of genetic algorithms discussed in section 4.2.2. The GAs were built specifically around the game of Tetris, focusing a lot on the feature and property representations. Furthermore model-based methods discussed such as Nuno (2019), show a deeper understanding of the game and are much more robust than the model-free approaches. The model-free approach of the project demonstrates the complications and barriers that similar methods encounter.

The performance by the DQN agent match the performance of Stevens and Pradhan, 2016. Their DQN agents cleared less than 20 lines in this given time space. Even though the delay between action and reward is minimized by dropping the pieces immediately, the agents still have trouble understanding the concept of clearing lines to get a higher reward. Despite the fact that both reward actions are promising and showed some excellent results on the small 8x6 board, there are some properties that does not allow an optimal convergence. The high complexity of the board does not allow the perfecting reward function to properly build a board structure, causing imbalances and causing the agent to eventually lose. On the other hand, the structuring function punishes the agent for building high columns and as soon as the agent passes a high of 12 blocks, the agents prefers to loses instead of keep on playing and getting negative rewards.

Given enough time, more state representations, better suited reward functions and a more complicated deep architecture can be tested. Using the reward function proposed by Yiyuan (2013) and not modifying it for the need of the small board might have hindered the results. The original weights used were optimal for the full 20x10 grid. This can be an explanation of why the agents unperformed using the structuring reward function.

It is possible that if the training continued, the agents could find a better approach resulting in a better performance and stable runs. By using bigger buffer sizes, the agent could learn more as the sample variety would be wider. Furthermore, some of the agents did not fully converge but instead they stopped early, due to tight schedule plans and limited computational resources. Through the use of some different agents the training time should be reduced. A Prioritised experience replay can be used, discussed in section 3.5.1, instead of the vanilla replay buffer. By selecting the states which had the more error loss, the agent learns more efficiently and it can speed up the process. Asynchronous methods like A3C are much faster than the methods as used due to agents working in parallel. As every agent is independent and it interacts with a copy of it's own environment and all the agents work together, the experience available is more diverse. This also can speed up the process.

Chapter 8

Conclusions and Future work

The deep Learning methods used are able to perfect the game at a lower complexity, using a smaller board and a lower number of shapes, but fail to do so in the full board. The algorithms need a lot of computation and a lot of data but sometimes it leads to over-estimation of parametric weight degrading the quality of the results. The PPO agent produces excellent results in the small board, learning to play the game optimally, but does not respond well to the increased complexity. The DQN agent does not learn how to clear multiple lines but it shows a deeper understanding in building a good structure of the board and playing the game as long as possible. Algorta and Şimşek (2019) described Tetris as a hard game, even to approximate, and have good reasons to do so.

State representations including the future shapes, the agents were able exponentially improve their performance. Even though some agents were able to perform very well using feature representation on the small board, eg. PPO using the 12-feat, agents using visual input outperformed them by at least a factor of 2 in the complete board. This demonstrates the power of feature representation that Neural Network possess.

Implementing the improvement suggestions in section 7.3 should show an increase in performance of the agents, or at least better convergence. Given unlimited resources and unlimited training, the game of Tetris could be solved perfectly by model-free Deep Reinforcement Learning, but unfortunately that is not the case.

Bibliography

Algorta, S. and Şimşek, , 2019. The game of tetris in machine learning. ArXiv preprint arXiv:1905.01652.

Atkeson, C. and Santamaria, J., 1997. A comparison of direct and model-based reinforcement learning. *Proceedings of international conference on robotics and automation*. IEEE, vol. 4, p.3557–3564.

Baxter, J. and Bartlett, P., 2000. May. direct gradient-based reinforcement learning. *2000 ieee international symposium on circuits and systems (iscas*. IEEE, vol. 3, p.271–274.

Bellman, R., 1952. On the theory of dynamic programming. *Proceedings of the national academy of sciences of the united states of america*, 38(8), p.716.

Bertsekas, D. and Tsitsiklis, J., 1996. *Neuro-dynamic programming*. Athena Scientific.

Bubeck, S. and Cesa-Bianchi, N., 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. ArXiv preprint arXiv:1204.5721.

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

De Asis, K., Hernandez-Garcia, J., Holland, G. and Sutton, R., 2018. Multi-step reinforcement learning: A unifying algorithm. *Proceedings of the aaai conference on artificial intelligence*, 32(1)).

Deisenroth, M. and Rasmussen, C., 2011. Pilco: A model-based and data-efficient approach to policy search. *Proceedings of the 28th international conference on machine learning (icml-11*. p.465–472.

Demaine, E., Hohenberger, S. and Liben-Nowell, D., 2003. Tetris is hard, even to approximate. *International computing and combinatorics conference*. Berlin, Heidelberg: Springer, p.351–363.

Earling, A., 1996. *The tetris effect: do computer games fry your brain*. Philadelphia Citypaper.

Fahey, C., 2003. Tetris ai [Online]. Available from: `https://www.colinfahey.com/tetris/tetris.html`.

Faultstick, B., 2014. June, cira tetris game sets guinness world record [Online]. Available from: `https://drexel.edu/now/archive/2014/June/Cira-Tetris-Guinness/`.

Fleming, W. and Rishel, R., 1975. Deterministic and stochastic optimal control. Of. Applications of mathematics.

Fortunato, M., Azar, M., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O. and Blundell, C., 2017. Noisy networks for exploration. arXiv preprint arXiv:1706.10295.

Frear, D., 2009. [Online]. Tetris Review. URL. Available from: `https://www.nintendolife.com/reviews/2009/07/3d_tetris_retro`.

Gabillon, V., Ghavamzadeh, M. and Scherrer, B., 2013. Approximate dynamic programming finally performs well in the game of tetris. *Neural information processing systems (nips*.

Hansen, N. and Ostermeier, A., 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2), p.159–195.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2018. April. rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the aaai conference on artificial intelligence*, 32(1)).

Lagoudakis, M., Parr, R. and Littman, M., 2002. Least-squares methods in reinforcement learning for control. *Hellenic conference on artificial intelligence*. Berlin, Heidelberg: Springer, p.249–260.

Liu, H. and Liu, L., n.d. Learn to play tetris with deep reinforcement learning.

McCulloch, W. and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), p.115–133.

Metropolis, N. and Ulam, S., 1949. The monte carlo method. *Journal of the american statistical association*, 44(247), p.335–341.

Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. *International conference on machine learning*. PMLR, p.1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. ArXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540), p.529–533.

Nuno, F., 2019. A deep reinforcment learning bot [Online]. [Online]. Available from: `https://github.com/nuno-faria/tetris-ai`.

Olson, M., 2020. October, tetris effect: Connected brings some competitive edge back to the chillest tetris url [Online]. Available from: `https://www.usgamer.net/articles/tetris-effect-connected-preview-coop-mode-classic-score-attack-ctwc`.

OpenAI., 2020. Proximal policy optimizations [Online]. [Online]. Available from: `https://spinningup.openai.com/en/latest/algorithms/ppo.html`.

Pollack, J. and Blair, A., 1997. Why did td-gammon work? *Advances in neural information processing systems*, 9(9), p.10–16.

Robbins, H. and Monro, S., 1951. A stochastic approximation method. *The annals of mathematical statistics*, p.400–407.

Rummery, G. and Niranjan, M., 1994. *On-line q-learning using connectionist systems*, vol. 37. Cambridge, UK: University of Cambridge, Department of Engineering.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2015. Prioritized experience replay. ArXiv preprint arXiv:1511.05952.

Schmidhuber, J., 2015. Deep learning in neural networks: An overview. *Neural networks*, 61, p.85–117.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015. Trust region policy optimization. *International conference on machine learning*. PMLR, p.1889–1897.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. ArXiv preprint arXiv:1707.06347.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. and Lillicrap, T., 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ArXiv preprint arXiv:1712.01815.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017b. Mastering the game of go without human knowledge. *nature*, 550(7676), p.354–359.

Stevens, M. and Pradhan, S., 2016. Playing tetris with deep reinforcement learning.

Sutton, R. and Barto, A., 1998. *Introduction to reinforcement learning*, vol. 135. Cambridge: MIT press.

Sutton, R. and Barto, A., 2018. *Reinforcement learning: An introduction*. The MIT Press.

Sutton, R., McAllester, D., Singh, S. and Mansour, Y., 1999. Policy gradient methods for reinforcement learning with function approximation. *Neural information processing systems 12*. p.1057–1063.

Szita, I. and Lörincz, A., 2006. Learning tetris using the noisy cross-entropy method. *Neural computation*, 18(12), p.2936–2941.

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

Thiery, C. and Scherrer, B., 2009. Building controllers for tetris. *Icga journal*, 32(1), p.3–11.

Tsitsiklis, J. and Van Roy, B., 1996. Feature-based methods for large scale dynamic programming. *Machine learning*, 22(1), p.59–94.

Turing, A., 2009. Computing machinery and intelligence. *Parsing the turing test*. Dordrecht: Springer, p.23–65.

Van Hasselt, H., Guez, A. and Silver, D., 2016. Deep reinforcement learning with double q-learning. *Proceedings of the aaai conference on artificial intelligence*, 30(1)).

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N., 2016. Dueling network architectures for deep reinforcement learning. *International conference on machine learning*. PMLR, p.1995–2003.

Watkins, C. and Dayan, P., 1992. Q-learning. *Machine learning*, 8(3-4), p.279–292.

Williams, R., 1987. A class of gradient-estimation algorithms for reinforcement learning in neural networks. *Proceedings of the international conference on neural networks*. p.–601.

Yiyuan, L., 2013. Tetris ai – the (near) perfect bot. *Code my road*, (1-4).

Appendix A

Environment

A.1 Environment raw Code

Listing A.1: Environment Code

```
import random
import cv2
import numpy as np
from PIL import Image
from time import sleep
import matplotlib.pyplot as plt

# Tetris game class
class Tetris:

    '''Tetris game class'''
    moves = []
    for i in range(10):
        for j in range(0,360,90):
            moves.append((i,j))

    action_map = {i:move for i,move in enumerate(moves)}
    # BOARD
    MAP_EMPTY = 0
    MAP_BLOCK = 1
    MAP_PLAYER = 2
    BOARD_WIDTH = 10
    BOARD_HEIGHT = 20

    TETROMINOS = {
        0: { # I
            0: [(0,0), (1,0), (2,0), (3,0)],
            90: [(0,0), (0,1), (0,2), (0,3)],
            180: [(3,0), (2,0), (1,0), (0,0)],
            270: [(0,3), (0,2), (0,1), (0,0)],
        },
```

```
        1: { # T
            0: [(1,0), (0,1), (1,1), (2,1)],
            90: [(0,1), (1,2), (1,1), (1,0)],
            180: [(1,2), (2,1), (1,1), (0,1)],
            270: [(1,1), (0,0), (0,1), (0,2)],
        },
        2: { # L
            0: [(0,0), (0,1), (0,2), (1,2)],
            90: [(0,1), (1,1), (2,1), (2,0)],
            180: [(1,2), (1,1), (1,0), (0,0)],
            270: [(2,1), (1,1), (0,1), (0,2)],
        },
        3: { # J
            0: [(1,0), (1,1), (1,2), (0,2)],
            90: [(0,1), (1,1), (2,1), (2,2)],
            180: [(0,2), (0,1), (0,0), (1,0)],
            270: [(2,1), (1,1), (0,1), (0,0)],
        },
        4: { # Z
            0: [(0,0), (1,0), (1,1), (2,1)],
            90: [(0,2), (0,1), (1,1), (1,0)],
            180: [(2,1), (1,1), (1,0), (0,0)],
            270: [(1,0), (1,1), (0,1), (0,2)],
        },
        5: { # S
            0: [(2,0), (1,0), (1,1), (0,1)],
            90: [(0,0), (0,1), (1,1), (1,2)],
            180: [(0,1), (1,1), (1,0), (2,0)],
            270: [(1,2), (1,1), (0,1), (0,0)],
        },
        6: { # O
            0: [(0,0), (1,0), (0,1), (1,1)],
            90: [(0,0), (1,0), (0,1), (1,1)],
            180: [(0,0), (1,0), (0,1), (1,1)],
            270: [(0,0), (1,0), (0,1), (1,1)],
        }
    }

    COLORS = {
        0: (255, 255, 255),
        1: (255, 0, 0),
        2: (0, 0, 255),
        3: (0,0,0),
        4: (0,128,0),
    }


    def __init__(self, mode = 'normal'):
        self.mode = mode
        if mode in ['normal','feat']: self.testing =
            Tetris.BOARD_WIDTH
```

```
        else: self.testing = Tetris.BOARD_WIDTH +6
        self.reset()


    def reset(self):
        '''Resets the game, returning the current state'''
        self.board = [[0] * Tetris.BOARD_WIDTH for _ in
            range(Tetris.BOARD_HEIGHT)]
        self.game_over = False
        self.bag = list(range(len(Tetris.TETROMINOS)))
        random.shuffle(self.bag)
        self.next_piece = self.bag.pop()
        self._new_round()
        self.score = 0
        if self.mode == 'glimpse':
            return np.array(self.glimpse())
        elif self.mode == 'feat':
            return self._get_board_props(self.board)
        else:
            return np.array(self.board)


    def _get_rotated_piece(self):
        '''Returns the current piece, including rotation'''
        return
            Tetris.TETROMINOS[self.current_piece][self.current_rotation]


    def _get_complete_board(self):
        '''Returns the complete board, including the current piece'''
        piece = self._get_rotated_piece()
        piece = [np.add(x, self.current_pos) for x in piece]
        board = [x[:] for x in self.board]
        for x, y in piece:
            board[y][x] = Tetris.MAP_PLAYER
        return board



    def _new_round(self):
        '''Starts a new round (new piece)'''
        # Generate new bag with the pieces
        if len(self.bag) == 0:
            self.bag = list(range(len(Tetris.TETROMINOS)))
            random.shuffle(self.bag)

        self.current_piece = self.next_piece
        self.next_piece = self.bag.pop()
        self.current_pos = [3, 0]
        self.current_rotation = 0
```

```
        if self._check_collision(self._get_rotated_piece(),
            self.current_pos):
            self.game_over = True


    def _check_collision(self, piece, pos):
        '''Check if there is a collision between the current piece
            and the board'''
        for x, y in piece:
            x += pos[0]
            y += pos[1]
            if x < 0 or x >= Tetris.BOARD_WIDTH
                or y < 0 or y >= Tetris.BOARD_HEIGHT
                                    or self.board[y][x] ==
                Tetris.MAP_BLOCK:
                return True
        return False


    def _rotate(self, angle):
        '''Change the current rotation'''
        r = self.current_rotation + angle

        if r == 360:
            r = 0
        if r < 0:
            r += 360
        elif r > 360:
            r -= 360

        self.current_rotation = r


    def _add_piece_to_board(self, piece, pos):
        '''Place a piece in the board, returning the resulting
            board'''
        board = [x[:] for x in self.board]
        for x, y in piece:
            board[y + pos[1]][x + pos[0]] = Tetris.MAP_BLOCK
        return board


    def _clear_lines(self, board):
        '''Clears completed lines in a board'''
        # Check if lines can be cleared
        lines_to_clear = [index for index, row in enumerate(board)
            if sum(row) == Tetris.BOARD_WIDTH]
        if lines_to_clear:
            board = [row for index, row in enumerate(board) if index
                not in lines_to_clear]
            # Add new lines at the top
```

```
            for _ in lines_to_clear:
                board.insert(0, [0 for _ in
                    range(Tetris.BOARD_WIDTH)])
        return len(lines_to_clear), board


    def _number_of_holes(self, board):
        '''Number of holes in the board (empty sqquare with at least
            one block above it)'''
        holes = 0

        for col in zip(*board):
            i = 0
            while i < Tetris.BOARD_HEIGHT and col[i] !=
                Tetris.MAP_BLOCK:
                i += 1
            holes += len([x for x in col[i+1:] if x ==
                Tetris.MAP_EMPTY])

        return holes


    def _bumpiness(self, board):
        '''Sum of the differences of heights between pair of
            columns'''
        total_bumpiness = 0
        max_bumpiness = 0
        min_ys = []

        for col in zip(*board):
            i = 0
            while i < Tetris.BOARD_HEIGHT and col[i] !=
                Tetris.MAP_BLOCK:
                i += 1
            min_ys.append(i)

        for i in range(len(min_ys) - 1):
            bumpiness = abs(min_ys[i] - min_ys[i+1])
            max_bumpiness = max(bumpiness, max_bumpiness)
            total_bumpiness += abs(min_ys[i] - min_ys[i+1])

        return total_bumpiness, max_bumpiness


    def _height(self, board):
        '''Sum and maximum height of the board'''
        sum_height = 0
        max_height = 0
        min_height = Tetris.BOARD_HEIGHT

        for col in zip(*board):
```

```
            i = 0
            while i < Tetris.BOARD_HEIGHT and col[i] ==
                Tetris.MAP_EMPTY:
                i += 1
            height = Tetris.BOARD_HEIGHT - i
            sum_height += height
            if height > max_height:
                max_height = height
            elif height < min_height:
                min_height = height

        return sum_height, max_height, min_height


    def _get_board_props(self, board):
        '''Get properties of the board'''
        lines, board = self._clear_lines(board)
        holes = self._number_of_holes(board)
        total_bumpiness, max_bumpiness = self._bumpiness(board)
        sum_height, max_height, min_height = self._height(board)
        h1, h2 , h3, h4 , h5 , h6, h7, h8 ,h9 , h10 = np.sum(board,
            axis = 0)
        return np.array([lines, holes, total_bumpiness,
            sum_height,h1, h2 , h3, h4 , h5 , h6, h7, h8 ,h9 , h10,
            self.current_piece, self.next_piece ])



    def play(self, move, render=False, render_delay=None):
        '''Makes a play given a position and a rotation, returning
            the reward and if the game is over'''
        action = Tetris.action_map[move]
        x = action[0]
        rotation =action[1]


        if self.current_piece == 0:
            if rotation in [0,180]:
                rotation = 0
                if x > 6:
                    x =6
        elif self.current_piece in [1]:
            if rotation in [90,270]:
                if x>8:
                    x=8
            else:
                if x>7:
                    x=7
        elif self.current_piece in [2]:
            if rotation in [0,90,270]:
                if x>7:
```

```
                x = 7
        else:
            if x>8:
                x=8
    elif self.current_piece in [3]:
        if rotation in [90,270]:
            if x>7:
                x = 7
        else:
            if x>8:
                x=8
    elif self.current_piece in [4]:
        if rotation in [0,180]:
            if x>7:
                x=7
        else:
            if x>8:
                x=8
    elif self.current_piece in [5]:
        if rotation in [0,180]:
            if x>7:
                x=7
        else:
            if x>8:
                x=8
    else:
        if x>8:
            x=8

    self.current_pos = [x, 0]
    self.current_rotation = rotation
    # Drop piece
    if self._check_collision(self._get_rotated_piece(),
       self.current_pos):
        self.game_over = True
    while not self._check_collision(self._get_rotated_piece(),
       self.current_pos):

        if render:
            self.render()
            if render_delay:
                sleep(render_delay)
        self.current_pos[1] += 1
    self.current_pos[1] -= 1

    # Update board and calculate score
    self.board =
       self._add_piece_to_board(self._get_rotated_piece(),
       self.current_pos)
    self.lines_cleared, self.board =
       self._clear_lines(self.board)
```

```
        reward = 0

        self._new_round()
        if self.game_over:
            reward = -10
        elif self.lines_cleared == 1:
            reward = 1
        elif self.lines_cleared == 2:
            reward = 4
        elif self.lines_cleared == 3:
            reward = 16
        elif self.lines_cleared == 4:
            reward = 64

        if self.mode == 'glimpse':
            return np.array(self.glimpse()),reward, self.game_over
        elif self.mode == 'feat':
            return self._get_board_props(self.board),reward,
                self.game_over
        else:
            return np.array(self.board),reward, self.game_over



    def glimpse(self):
        new_board = [[3]+[0]*5 for i in range(20)]

        piece = [(x+8,y+2) for x,y in
            Tetris.TETROMINOS[self.next_piece][0]]
        for x,y in piece:
            new_board[x][y] = 4
        random_bo = self._get_complete_board()
        result = [random_bo[i] + new_board[i] for i in range(20)]
        return result

    def render(self):
        '''Renders the current board'''
        if self.mode == 'glimpse':
            new_board = [[3]+[0]*5 for i in range(20)]

            piece = [(x+8,y+2) for x,y in
                Tetris.TETROMINOS[self.next_piece][0]]
            for x,y in piece:
                new_board[x][y] = 4
            random_bo = self._get_complete_board()
            result = [random_bo[i] + new_board[i] for i in range(20)]
        else:
            result = self._get_complete_board()

        img = [Tetris.COLORS[p] for row in result for p in row]
```

```
img = np.array(img).reshape(Tetris.BOARD_HEIGHT,
    self.testing, 3).astype(np.uint8)
img = img[..., ::-1] # Convert RRG to BGR (used by cv2)
img = Image.fromarray(img, 'RGB')
img = np.array(img)
img=cv2.resize(img,(self.testing *25, Tetris.BOARD_HEIGHT *
    25),interpolation = cv2.INTER_AREA)
cv2.imshow('image', np.array(img))
cv2.waitKey(1)
```

A.2 Environment Functionality

Function	Comment
reset()	Resets the board and the variables of the class. Takes no argument and returns back the current state of the board based on its mode
get rotated piece	Rotates the piece on the board. Only for internal use within the class
get complete board	Attaches the next piece on the board for rendering. Only for internal use within the class
new round	When the current piece falls, it replaces it with the new piece and picks the next piece coming. Only for internal use within the class
check collision	Checks whether the placed or falling piece collides with the board. Only for internal use withing the class
add piece board	Add the next piece to the board. Only for internal use withing the class
clear lines	Clears the complete lines of the board and drops the rest of blocks down. Only for internal use within the class
number of holes	Returns back the number of holes in the board (empty squareds with at least one block above it). Used only in the feat representation. Only for internal use within the class
bumpiness	Returns the sum of the differences of heights between the pair of columns. Used only in the feat representation. Only for internal use within the class
height	Returns the sum and the maximum height of the board. Used only in the feat representation. Only for internal use within the class
get board props	Returns the properties of the board. Used only in the feat representation. Only for internal use within the class
play	Takes as input an action given by the player or agent and transforms it into a location and rotation. Takes optional arguments whether the environment must be rendered or not and the render delay. Returns back the current state based on the feature, the corresponding reward and whether or not the game is finished
glimpse	Attaches the next piece also on the board. Used in the glimpse representation and rendering. Only for internal use within the class
render	Renders the current board

Table A.1: Envirnoment Functionality

Appendix B

DQN

B.1 DQN raw code

```
class DQN_Agent():
    def __init__(self,epsilon =0.9, gamma = 0.6,update_after_actions
       = 1,
                 epsilon_greedy_frames = 250000,
                    epsilon_random_frames =12500,
                 epsilon_min = 0.1, max_memory_length = 100000,
                 update_network = 5000, batch_size
                    =32,num_actions=40):

        #initialization of the parameters of the agent
        self.num_actions = num_actions
        self.actions = [i for i in range(num_actions)]
        self.max_memory_length = max_memory_length
        self.loss_function = tensorflow.keras.losses.Huber()
        self.optimizer =
           tensorflow.keras.optimizers.Adam(learning_rate = 0.00025,
           clipnorm = 1)
        self.epsilon_random_frames = epsilon_random_frames
        self.gamma = gamma
        self.batch_size = batch_size
        self.epsilon_min = epsilon_min
        self.epsilon_greedy_frames = epsilon_greedy_frames
        self.epsilon = epsilon
        self.epsilon_interval = 1- self.epsilon_min
        self.update_after_actions = update_after_actions
        self.buffer_actions = deque(maxlen = max_memory_length)
        self.buffer_state = deque(maxlen = max_memory_length)
        self.buffer_nextstate = deque(maxlen = max_memory_length)
        self.buffer_done = deque(maxlen = max_memory_length)
        self.buffer_rewards = deque(maxlen = max_memory_length)
        self.frames = 0
        self.update_network = update_network
        self.model = self.create_model()
        self.target_model = self.create_model()
```

```
        self.target_model.set_weights(self.model.get_weights())


    def
       update_replay_buffer(self,action,state,state_next,done,reward):
        #filling the replay memory with the recent state, next state
            ,action,reward , and done information
        self.buffer_actions.append(action)
        self.buffer_state.append(state)
        self.buffer_nextstate.append(state_next)
        self.buffer_done.append(done)
        self.buffer_rewards.append(reward)

    def update_weights(self):
        #update the weights of the target model
        self.target_model.set_weights(self.model.get_weights())

    def choose_action(self,state):
      #choosing an action based on an epsilon greedy policy
        self.frames += 1
        if np.random.uniform(0,1) < self.epsilon or self.frames
           <self.epsilon_random_frames:
            action = np.random.choice(self.actions)
        else:
            state_tensor = tf.convert_to_tensor(state)
            state_tensor = tf.expand_dims(state_tensor, 0)
            # outputs the Q value for each action
            Q_val = self.model(state_tensor, training=False)
            # Take best action
            action = tf.argmax(Q_val[0]).numpy()

        #epsilon decay
        self.epsilon -= self.epsilon_interval
           /self.epsilon_greedy_frames
        # epsilon capped at minimum epsilon (0.1)
        self.epsilon = max(self.epsilon,self.epsilon_min)

        return action

    def learn(self):
      #based on the hyperparameters , the agent samples states from
         the replay memory to update its weights
        if self.frames % self.update_after_actions==0 and
           len(self.buffer_actions) > self.batch_size:
            indices =
               np.random.choice(range(len(self.buffer_actions)),
               size = self.batch_size)
            state_sample = np.array([self.buffer_state[i] for i in
               indices])
            state_next_sample = np.array([self.buffer_nextstate[i]
               for i in indices])
```

```
            rewards_sample = [self.buffer_rewards[i] for i in
                indices]
            action_sample = [self.buffer_actions[i] for i in indices]
            done_sample =
                tf.convert_to_tensor([float(self.buffer_done[i]) for
                i in indices])


            future_rewards =
                self.target_model.predict(state_next_sample)
            updated_q_values = rewards_sample + self.gamma
                *tf.reduce_max(future_rewards, axis =1)

            updated_q_values = updated_q_values * (1 - done_sample)
                - done_sample

            masks = tf.one_hot(action_sample, self.num_actions)

            with tf.GradientTape() as tape:
                # Train the model on the states and updated Q-values
                q_values = self.model(state_sample)

                # Apply the masks to the Q-values to get the Q-value
                    for action taken
                q_action = tf.reduce_sum(tf.multiply(q_values,
                    masks), axis=1)
                # Calculate loss between new Q-value and old Q-value
                loss = self.loss_function(updated_q_values, q_action)

            # Backpropagation
            grads = tape.gradient(loss,
                self.model.trainable_variables)
            self.optimizer.apply_gradients(zip(grads,
                self.model.trainable_variables))

        #update the weights of the target network each 10000 frames
        if self.frames % self.update_network == 0:
            self.update_weights()

    def create_model(self):
        # Network defined by the Deepmind paper
        inputs = layers.Input(shape=(20, 16,))

        # Convolutions on the frames on the screen
        layer1 = layers.Conv1D(32, 3, strides=3, activation="relu",
        kernel_initializer=tf.keras.initializers.VarianceScaling
        (scale=2))(inputs)
        layer2 = layers.Conv1D(64, 2, strides=2, activation="relu",
        kernel_initializer=tf.keras.initializers.VarianceScaling
        (scale=2))(layer1)
```

```
        layer4 = layers.Flatten()(layer2)

        layer5 = layers.Dense(512, activation="relu",
        kernel_initializer=tf.keras.initializers.VarianceScaling
        (scale=2))(layer4)
        action = layers.Dense(self.num_actions, activation="linear",
        kernel_initializer=tf.keras.initializers.VarianceScaling
        (scale=2))(layer5)

        return keras.Model(inputs=inputs, outputs=action)

    def save_model(self,episode_num):
        self.model.save('models/model_frames_'+ str(self.frames)+
            '.h5')
        print(f'Model saved succesfully at episode {episode_num}',
            flush = True)

    def save_buffer(self,i):
      #save the current buffer in case of programm crashes
        with open('buffer/history.txt','w') as b:
            b.write(f'Latest Frames Saved : {self.frames} Episode :
                {i}\n')
        pickle.dump( self.buffer_rewards,
            open('buffer/buffer_rewards.p', "wb" ))
        pickle.dump( self.buffer_state,
            open('buffer/buffer_state.p', "wb" ))
        pickle.dump( self.buffer_nextstate,
            open('buffer/buffer_nextstate.p', "wb" ))
        pickle.dump( self.buffer_actions,
            open('buffer/buffer_actions.p', "wb" ))
        pickle.dump( self.buffer_done, open('buffer/buffer_done.p',
            "wb" ))

    def load_model_buffer(self, frames, epsilon):
        self.model.load_weights('models/model_frames_' + str(frames)
            + '.h5')
        self.update_weights()
        self.epsilon = epsilon
        self.frames = frames
        with open('buffer/buffer_rewards.p', 'rb') as f:
            self.buffer_rewards = pickle.load(f)
        with open('buffer/buffer_state.p', 'rb') as f:
            self.buffer_state = pickle.load(f)
        with open('buffer/buffer_nextstate.p', 'rb') as f:
            self.buffer_nextstate = pickle.load(f)
        with open('buffer/buffer_actions.p', 'rb') as f:
            self.buffer_actions = pickle.load(f)
        with open('buffer/buffer_done.p', 'rb') as f:
            self.buffer_done = pickle.load(f)
        print('----------Loaded model and buffer
            Succesfully----------')
```

B.2 DQN Functionality

Function	Comment
Initialisation of the agent	Takes as an input all the possible parameters needed for the DQN agent and set's the process
Update Replay buffer	Updates the buffer with the respective data. It is an external function and it is expected to be called at a set frequency by the user.
Update weights	Updates the parametric weights of the target model based on the main model based on a hyper-parameter set by the user. Only for internal use within the class
Choose action	This function takes as an input the current state of the board and it returns the action chosen by the agent, always based on epsilon. It is also responsible for decreasing the value of epsilon slowly to reach the minimum
Learn	It's where the magic happens. After a set number of frames, the agent samples a batch of 32 actions from the buffer and updates the weights of the main model according to the 'target' values given by the target network
Create model	Creates the NN based on a fixed architecture. Sadly the agent does not accept any NN. In order to change the architecture one must change this function from the class by scratch. Only for internal use within the class
Save model	The current model weights are saved in order to be used later and reproduce the results during the different phases of the agent. It is an external function and it is expected to be called at a set frequency by the user.
Load model and buffer	Used as a checkpoint. Takes as an input the frames and the epsilon when the agent was stopped for the last time. Then it updates the buffer and the current model weights so that the agent can continue. Only used in case more training needs to be done when the agent was stopped.

Table B.1: DQN Functionality

Appendix C

PPO

C.1 PPO raw code

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


from Tetris import Tetris
import tensorflow as tf
import numpy as np
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import scipy.signal
import time
import cv2
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
import pickle


#Hyperparameters
steps_per_epoch = 4000
epochs = 300
gamma = 0.99
clip_ratio = 0.2
policy_learning_rate = 3e-4
value_function_learning_rate = 1e-3
train_policy_iterations = 80
train_value_iterations = 80
lam = 0.97
target_kl = 0.01
hidden_sizes = (64,64)
```

```
render = False

def discounted_cumulative_sums(x, discount):
    # Discounted cumulative sums of vectors for computing
       rewards-to-go and advantage estimates
    return scipy.signal.lfilter([1], [1, float(-discount)], x[::-1],
       axis=0)[::-1]


class Buffer:
    # Buffer for storing trajectories
    def __init__(self, observation_dimensions, size, gamma=0.99,
       lam=0.95):
        # Buffer initialization
        self.observation_buffer = np.zeros(
            (size, observation_dimensions), dtype=np.float32
        )
        self.action_buffer = np.zeros(size, dtype=np.int32)
        self.advantage_buffer = np.zeros(size, dtype=np.float32)
        self.reward_buffer = np.zeros(size, dtype=np.float32)
        self.return_buffer = np.zeros(size, dtype=np.float32)
        self.value_buffer = np.zeros(size, dtype=np.float32)
        self.logprobability_buffer = np.zeros(size, dtype=np.float32)
        self.gamma, self.lam = gamma, lam
        self.pointer, self.trajectory_start_index = 0, 0

    def store(self, observation, action, reward, value,
       logprobability):
        # Append one step of agent-environment interaction
        self.observation_buffer[self.pointer] = observation
        self.action_buffer[self.pointer] = action
        self.reward_buffer[self.pointer] = reward
        self.value_buffer[self.pointer] = value
        self.logprobability_buffer[self.pointer] = logprobability
        self.pointer += 1

    def finish_trajectory(self, last_value=0):
        # Finish the trajectory by computing advantage estimates and
           rewards-to-go
        path_slice = slice(self.trajectory_start_index, self.pointer)
        rewards = np.append(self.reward_buffer[path_slice],
           last_value)
        values = np.append(self.value_buffer[path_slice], last_value)

        deltas = rewards[:-1] + self.gamma * values[1:] - values[:-1]

        self.advantage_buffer[path_slice] =
           discounted_cumulative_sums(
            deltas, self.gamma * self.lam
        )
        self.return_buffer[path_slice] = discounted_cumulative_sums(
```

```
            rewards, self.gamma
        )[:-1]

        self.trajectory_start_index = self.pointer

    def get(self):
        # Get all data of the buffer and normalize the advantages
        self.pointer, self.trajectory_start_index = 0, 0
        advantage_mean, advantage_std = (
            np.mean(self.advantage_buffer),
            np.std(self.advantage_buffer),
        )
        self.advantage_buffer = (self.advantage_buffer -
           advantage_mean) / advantage_std
        return (
            self.observation_buffer,
            self.action_buffer,
            self.advantage_buffer,
            self.return_buffer,
            self.logprobability_buffer,
        )


def mlp(x, sizes, activation=tf.tanh, output_activation=None):
    # Build a feedforward neural network
    for size in sizes[:-1]:
        x = layers.Dense(units=size, activation=activation)(x)
    return layers.Dense(units=sizes[-1],
       activation=output_activation)(x)


def logprobabilities(logits, a):
    # Compute the log-probabilities of taking actions a by using the
       logits (i.e. the output of the actor)
    logprobabilities_all = tf.nn.log_softmax(logits)
    logprobability = tf.reduce_sum(
        tf.one_hot(a, num_actions) * logprobabilities_all, axis=1
    )
    return logprobability


# Sample action from actor
@tf.function
def sample_action(observation):
    logits = actor(observation)
    action = tf.squeeze(tf.random.categorical(logits, 1), axis=1)
    return logits, action


# Train the policy by maxizing the PPO-Clip objective
@tf.function
```

```
def train_policy(
    observation_buffer, action_buffer, logprobability_buffer,
       advantage_buffer
):

    with tf.GradientTape() as tape:  # Record operations for
       automatic differentiation.
        ratio = tf.exp(
            logprobabilities(actor(observation_buffer),
               action_buffer)
            - logprobability_buffer
        )
        min_advantage = tf.where(
            advantage_buffer > 0,
            (1 + clip_ratio) * advantage_buffer,
            (1 - clip_ratio) * advantage_buffer,
        )

        policy_loss = -tf.reduce_mean(
            tf.minimum(ratio * advantage_buffer, min_advantage)
        )
    policy_grads = tape.gradient(policy_loss,
       actor.trainable_variables)
    policy_optimizer.apply_gradients(zip(policy_grads,
       actor.trainable_variables))

    kl = tf.reduce_mean(
        logprobability_buffer
        - logprobabilities(actor(observation_buffer), action_buffer)
    )
    kl = tf.reduce_sum(kl)
    return kl


# Train the value function by regression on mean-squared error
@tf.function
def train_value_function(observation_buffer, return_buffer):
    with tf.GradientTape() as tape:  # Record operations for
       automatic differentiation.
        value_loss = tf.reduce_mean((return_buffer -
           critic(observation_buffer)) ** 2)
    value_grads = tape.gradient(value_loss,
       critic.trainable_variables)
    value_optimizer.apply_gradients(zip(value_grads,
       critic.trainable_variables))

def models(observation_dimensions, num_actions):
    observation_input = keras.Input(shape=(observation_dimensions,),
       dtype=tf.float32)
    logits = mlp(observation_input, list(hidden_sizes) +
       [num_actions], tf.tanh, None)
```

```
    actor = keras.Model(inputs=observation_input, outputs=logits)
    value = tf.squeeze(
        mlp(observation_input, list(hidden_sizes) + [1], tf.tanh,
            None), axis=1
    )
    critic = keras.Model(inputs=observation_input, outputs=value)
    return actor, critic


env = Tetris()
observation_dimensions = 48
num_actions = 24

# Initialize the buffer
buffer = Buffer(observation_dimensions, steps_per_epoch)

# Initialize the actor and the critic as keras models



actor, critc = models(observation_dimensions, num_actions)

policy_optimizer =
    keras.optimizers.Adam(learning_rate=policy_learning_rate)
value_optimizer =
    keras.optimizers.Adam(learning_rate=value_function_learning_rate)

observation, episode_return, episode_length = env.reset(), 0, 0

def train():
    total_actions = 0
    for epoch in range(epochs):
        # Initialize the sum of the returns, lengths and number of
            episodes for each epoch
        sum_return = 0
        sum_length = 0
        num_episodes = 0

        # Iterate over the steps of each epoch
        for t in range(steps_per_epoch):
            if render:
                env.render()

            # Get the logits, action, and take one step in the
                environment
            observation = observation.reshape(1, -1)
            logits, action = sample_action(observation)
            observation_new, reward, done=
                env.play(action[0].numpy())
            total_actions += 1
            episode_return += reward
```

```
            episode_length += 1

            # Get the value and log-probability of the action
            value_t = critic(observation)
            logprobability_t = logprobabilities(logits, action)

            # Store obs, act, rew, v_t, logp_pi_t
            buffer.store(observation, action, reward, value_t,
                logprobability_t)

            # Update the observation
            observation = observation_new

            # Finish trajectory if reached to a terminal state
            terminal = done
            if terminal or (t == steps_per_epoch - 1):
                last_value = 0 if done else
                    critic(observation.reshape(1, -1))
                buffer.finish_trajectory(last_value)
                sum_return += episode_return
                sum_length += episode_length
                num_episodes += 1
                observation, episode_return, episode_length =
                    env.reset(), 0, 0

        # Get values from the buffer
        (
            observation_buffer,
            action_buffer,
            advantage_buffer,
            return_buffer,
            logprobability_buffer,
        ) = buffer.get()

        # Update the policy and implement early stopping using KL
            divergence
        for _ in range(train_policy_iterations):
            kl = train_policy(
                observation_buffer, action_buffer,
                    logprobability_buffer, advantage_buffer
            )
            if kl > 1.5 * target_kl:
                # Early Stopping
                break

        # Update the value function
        for _ in range(train_value_iterations):
            train_value_function(observation_buffer, return_buffer)

        # Print mean return and length for each epoch
        print(
```

```
            f"Epoch:{epoch+1}.Mean␣Return:{sum_return/num_episodes}.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣Mean_Length:{sum_length/num_episodes}")
        actor.save('models/model_frames_'+str(epoch)+'.h5')
        t = time.localtime()
        current_time = time.strftime('%H:%M:%S', t)
        with open('checkpoint.txt', 'a') as checkpoint:
            checkpoint.write(
        f'Checkpoint::Epoch{epoch},Actions={total_actions},reward:{sum_return␣
           /␣num_episodes},
␣␣␣␣␣␣␣␣␣␣time:{current_time}\n')
```

C.2 PPO Functionality

Function	Comment
Discount Cumulative sums	Discounted cumulative sums of vectors for computing rewards-to-go and advantage estimates
mlp	Build a feed forward neural network
Buffer Class	Buffer for storing trajectories. It can append one step of agent-environment interaction, Finish the trajectory by computing advantage estimates and rewards-to-go and get all data of the buffer and normalize the advantages
Sample action	Sample action from actor
train policy	Train the policy by maximizing the PPO-Clip objective
train value function	Train the value function by regression on mean-squared error
models	Creates the actor and critic needed for PPO based on the set hyper parameters
Train	trains the agent with the given hyper parameters

Table C.1: PPO Functionality