ulm university universität uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Informatik
Neural Information Processing

Applying Deep Q-Networks (DQN) to the game of Tetris using high-level state spaces and different reward functions

Bachelor thesis at Ulm University.

Submitted for the degree of Bachelor of Science (BSc) in computer science.

Submitted by:

Mohamed Ashry Hassan Mahmoud

mohamedelashry7777@gmail.com

Matriculation number: 1076225

Thesis advisors:

Prof. Dr. Friedhelm Schwenker

Ulm University, Ulm

Supervisor:

Prof. Dr. Friedhelm Schwenker

Ulm, August 2020

Version August 17, 2020

Name: Mohamed Ashry Hassan Mahmoud Matriculation number: 1076225

Declaration

I hereby declare that I wrote the bachelor thesis independently and used no other aids than those cited. In each individual case, I have clearly identified the source of the passages that are taken word for word or paraphrased from other works.

Ulm, date . . . .

Mohamed Ashry Hassan Mahmoud

To my future wife, I love you.

Acknowledgements

Thank you, Allah for all the blessing and guidance you gave me throughout this hard and beautiful journey.

I have to admit my gratitude and thanks to my supervisor ***Prof. Dr. Friedhelm Schwenker*** for all his help and support during the project and the thesis. Despite the lockdown and Corona crisis, he is still giving and offering his help and support not just academically but in every other problem I have faced. Thank you so much, Prof.

There are not sufficient words that can express my love and thanks to my dear family, my ***mother, father, brothers and sisters***, that without their distant support and their prayers for me I would not finish this work or even withstand away from them.

To my friends here in Germany, ***Beltagy, Samaha, Hennawy, Rafeek and Faisal***, thank you so much for your companionship that eases my stay here in Germany.

A special thanks to my friend ***Amr Kayid*** who is always available for me and my overthinking questions in spite of his responsibilities.

Big thanks to ***Omar's cooking*** that without his delicious and easy recipes I would not eat properly.

Finally, thanks and appreciation from all of my heart to my dear friend, ***Soaad***. She always cares about me and gives me the support I need. She is always here for me. Thank you so much, dear.

Abstract

Deep reinforcement learning algorithms are used to learn complex patterns and intricate functions and deal with high dimensional state spaces. So we decided to use Deep reinforcement learning to approximate a solution to an NP-complete problem which is the game of Tetris which has a big state space.
The main purpose of the thesis is to build an RL agent who can play Tetris using Deep Q-Networks based on high-level state spaces rather than the raw data from the board. We are trying to achieve that by building a suitable Tetris environment supported by different implementations of high-level state spaces and different reward function. Also we have added the grouped action option to the environment to speed up the process of taking the decision.
We have applied some of DQN enhancements, e.g. DDQN and Dueling networks to help the agent learn better. With this approach the state space size has been reduced significantly and the learning process has been faster. The concluding results show that some of the state spaces combined with certain Neural Network architecture are leading to promising performance of the agent.

Contents

List of Figures

List of Tables

1 Introduction

Reinforcement learning (RL) is a subfield of machine learning that targets the problem of automatic learning of optimal decisions over time[1].RL is different from supervised learning and unsupervised learning, fig 1.1[1], that it is independent from the pre-collected data, that RL allows agent to take actions and based on these actions it gets observations from the environment, the agent deals with, and a reward which indicates how good or bad the taken action. So the goal of the agent is to maximize reward by taking the best action based on the current state of the environment.

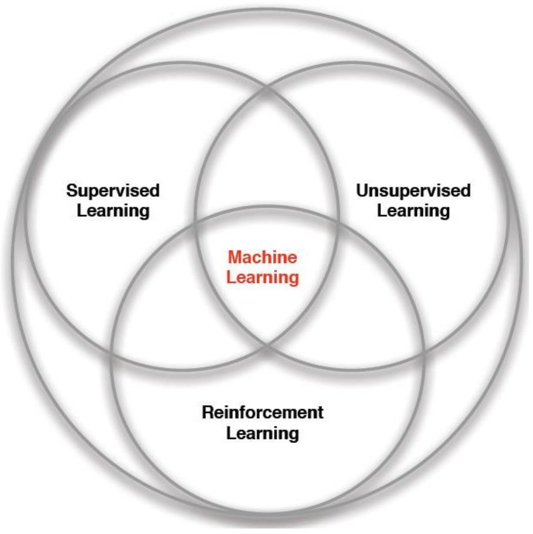

Figure 1.1: The Fields of Machine learning

RL combined with deep neural networks gives impressive results. That it would be a step to mimic human's ability to learn from its own experience.

[1]https://syncedreview.com/2017/02/24/david-silver-google-deepmind-deep-reinforcement-learning/

1.1 Motivation

Tetris is a popular computer game that was invented by the Russian mathematician Alexey Pajitnov. It it originally a board consisted of 10 columns and 20 rows, so in total it has a size of 200 cells. There are seven gaming pieces called tetriminos. Each one of them of size 4 blocks and they have the shape of (O,L,I,S,Z,T,J).

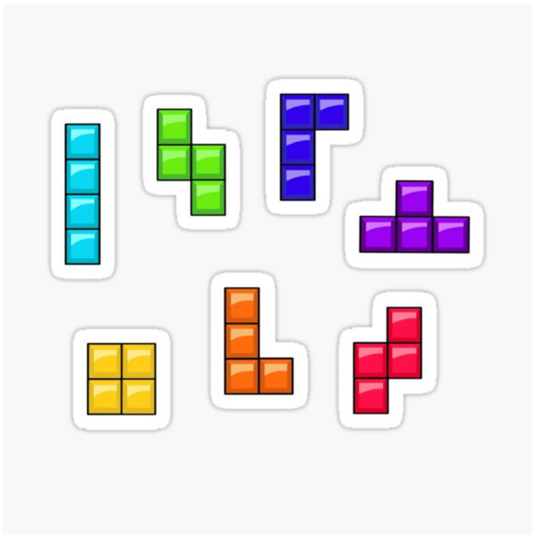
Figure 1.2: Tetriminos

During the game there would be a random falling piece from the top and the player would choose one of the ten column and the orientation of the piece before dropping it. The goal of the game is to maximize the score and that is by forming complete lines, which are then removed and all the cells above it would move down.
The cells in board are either occupied or free. So the state space would be 2^{200} * 7 since there are 7 Tetriminos. The state space is so large as it is 10^{60}. Due to its complex nature, Tetris has been proved to be NP-complete and that makes it appealing to the AI community to find approximate solutions to it.

1.2 Objective

The aim of this research project is to implement, train and test different reinforcement learning algorithms with different hyperparameters and state spaces and search for the best technique possible for implementing agent playing Tetris.

1.3 Overview and Outline

The work is organized as follows. In chapter two there would be an introduction about RL and Markov decision processes(MDPs). Also, we would introduce the use of Deep Neural Networks(DNN) to RL. Then, we would discuss the related work to the problem and its resutls. In chapter 3 we present the task description, the used frameworks, architectures and the algorithms used to train the agent. Chapter 4 contains detailed description about each implemented experiment along with its results. we would conclude the whole work, limitations that we faced during the project and future work in chapter 5.

2 Background and literature review

This chapter presents the basic concepts used throughout the work. Firstly, we would have an introduction to the reinforcement learning field and the use of Markov Decision Processes. Furthermore, we discusses Deep Reinforcement Learning and the use of Deep Neural Networks (DNN) and their impacts to the performance. Finally we discuss the related work in the Tetris game.

2.1 Reinforcement Learning

Reinforcement Learning is a machine learning approach that targets the problem of automatic learning of optimal decisions overtime[1]. It teach the **Agent** how to solve tasks optimally and that is by interacting with the **Environment** using **Actions** based on the state of the environment then the agent receives **Observations** and **Rewards** which indicates how good or bad this action for this state. **RL** is data independent which means it does not require pre-collected data. But, it generates the data immediately.

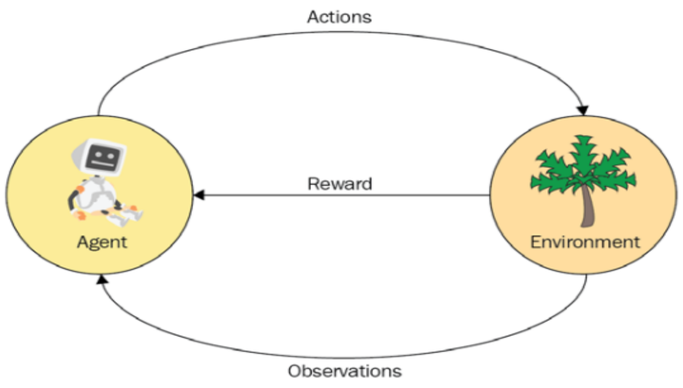

Figure 2.1: The RL entities and their communication channels[1]

2.1.1 Agent

The agent is the core of **RL** as it chooses the **Action** to interact with **Environment**, receives **Reward** and the **Observations** based on the taken action.

2.1.2 Environment

The environment is everything that agent deals with. For example the environment would be the chess board and the chess pieces. Or if the agent is a student so the school and the teachers and everything except the student are the environment.
The only communication between the agent and the environment is through the actions executed, the reward received and the observations of the next state. 2.1

2.1.3 Actions

Actions are things that agent can do with the environment. There are two types of actions, discrete actions which are finite set of actions like in the game of Tetris be like rotate left or rotate right.
Or Continuous actions which is infinite set of actions like turn the wheel with an angel between [-180,180] so the angle value can take any value in that range.

2.1.4 Observations

Observations are simply the next state of the environment after executing an action. But the observations are generally formulated that means we do not measure every atom in the state space itself, since that is impossible, but it is converted to a more sensible representation.
For example the state space in the game of Tetris can be the whole board. But that is needs gigantic amount of computation, since the state space amounts to 10^{60}, so it is converted to other feature like the column difference, number of hole, heights of columns, etc(we will get back to it in detail in the next chapter).

2.1.5 Reward

It is a scalar value we obtain from the environment after each action as an indicator of how well the agent is doing. The reward can be positive or negative, large or small. Also, the obtained reward can be once in a lifetime like in the chess game the agent would get 1 if it wins, -1 if it loses and 0 if there is a draw. Or it can be every timestamp like in the game of Tetris that indicates how well the placement of a piece is.

2.2 Markov Decision Process

Markov Decision Process **MDP** can be described through some levels. Firstly, Markov Process **MP** then Markov Reward Process **MRP** and finally Markov Decision Process **MDP**.

2.2.1 Markov Process

Markov Process is the simplest child in Markov Family[1]. To call such a system MP it has to have the **Markov Property** which indicates that any tranision from state to another only depends on the most recent state. That means the future dynamics of the system only depend on only one state not the whole history.
So, the formal definition of an MP is as follows:

- A set of all states (S).
- A transition matrix (T) that contains the probabilities from state to another.

2.2.2 Markov Reward Process

MP would be extended and added to it **Reward** value and discount factor γ. When the system state change from state *i* to state *j* it would receive a scalar value (Reward) to indicate the performance of the Agent. Discount factor γ is used as a measure of how far we look into the future. That for each timestamp *t* there is a return value G_t which is summation of the current reward and the subsequent reward multiplied by the discount factor γ.

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

By averaging the values of G_t over large number of episodes we would have **value of the state**:

$$V(s) = E[G|S_t = s]$$

So, MRP can be described as follows:

- An immediate reward function
- A discount factor $\gamma \in [0, 1]$

2.2.3 Markov Decision Process

MDP is a MRP extended with set of valid actions (A) that means the value of state *V(s)* does not depend only on the state but also on the action that leads to this state. So MDP can be defined as a tuple of five elements (S,T,R,γ,A) which are:

- A set of all states (S).
- A transition matrix (T) that contains the probabilities from state to another.
- An immediate reward function R(s_t, a_t, s_{t+1})
- A discount factor $\gamma \in [0, 1]$
- A set of valid actions (A)

2.3 Deep Reinforcement Learning

The huge success of training Deep Neural Networks(DNN) on large datasets lead to combine RL algorithms with DNN.
We would discuss one of state-of-the-art algorithms which is *Deep Q Networks* that shows great success in playing Atari games and outperformed humans in many games[2]

2.3.1 Deep Q-Network

The core of Q-learning is borrowed from supervised learning. As we try to approximate the Q(s,a) with NN using Bellman equation.

$$Q\left(s_t, a_t\right)=r_t+\gamma \max_a Q'\left(s_{t+1}, a\right) \tag{2.1}$$

Stochastic gradient descent (SGD) is used to update weights of the NN. And one of the requirements of SGD optimization is that the training data must be independent and identically distributed, so a replay buffer is used to fulfill this requirement.
DQN use MDP as when it samples experiences from the replay buffer it does not need to know the whole history just the tuple of the the state, the next state, the taken action, the reward and a boolean value indicate whether the game ended or not.

2.4 Related Work

Due to the complex nature of the game of Tetris, since its state space amounts to $2^{200} \times 7$, it represented a challenge to the AI community to solve it.
There has been many attempts to play Tetris.

2.4.1 Developing a linear evaluation function(Fixed Policy)

Developing a linear evaluation function is considered one of the most common approach to play Tetris.
The evaluation function is used to evaluate each possible placement of the Tetrimino and then select the placement of the highest value[3] The evaluation function is a linear combination of weighted features that represent the Tetris board.
One of the early attempts is done by Tsitsiklis and Van Roy. They used two simple state features which are the number of holes and the height of the highest column and they achieved a score of around 30 cleared lines[3].
One of best linear evaluation function was hand-crafted by the Pierre Dellacherie[4]. He used six simple features and tuned their weights by trial and error. These features are

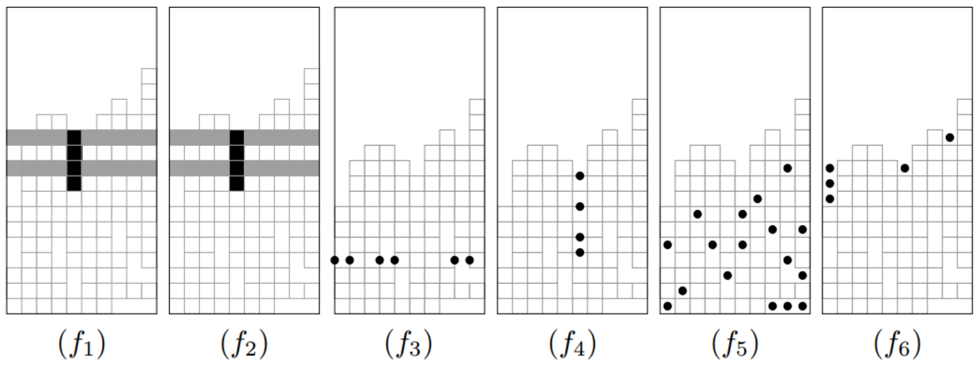

Figure 2.2: The six features of Dellacherie hand-crafted controller[5]

1. **Landing Height**: The height at which the current piece fell.
2. **Eroded pieces**: The contribution of the last piece to the cleared lines time the number of cleared lines.
3. **Row Transitions**: The number of filled cells adjacent to the empty cells summed over all rows.
4. **Column Transitions**: The same as Row Transitions but along columns.
5. **Holes**: A cell is considered to be a hole if it is empty and the cell above it is occupied.
6. **Cumulative Wells**:A well is a succession of empty cells and the cells to the left and right are occupied.
   So cumulative wells is the sum of accumulated depths of the wells.

And the weights of these features were tuned by hand as the evaluation function is

$$-4 \times Holes - CumulativeWells - RowTransition$$

$$-ColumnTransitions - LandingHeight + ErodedPieces$$

This linear evaluation function cleared an average of 660,000 lines on the full grid.

2.4.2 Genetic Algorithms

GAs were used to develop a Tetris controllers and linear evaluation functions. Thiery and Scherrer[6] used the same features as Pierre Dellacherie and added couple fea-

ture to them which are hole depth and rows with holes. They used cross-entropy algorithm to tune the weights and they achieved an average score of 35,000,000 cleared lines.
Also Boumaza[5] introduced another evolutionary algorithm to Tetris, the covariance matrix adaptation evolution strategy(CMA-ES). The resulting weights were very close to Dellacherie's and cleared around 35,000,000 lines per game.
There are many evaluation functions that have been developed. And we will introduce some of them that were used in this project.

1. **The Near perfect bot [7]** This evaluation function only consists of four features.

   a) **Aggregate Height** This tells how high the grid and it is computed by summing the heights of all columns.

   b) **Complete Lines** Since the goal of the AI agent is clearing lines so we want to maximize this value.

   c) **Holes**

   d) **Bumpiness** The bumpiness of a grid represents the variation of its column heights. It is computed by summing up the absolute differences between all two adjacent columns.

   The weights of the function using a GA algorithm. The final function weights are:

$$-.510066 \times (AggregateHeight) + 0.760666 \times (CompleteLines)$$

$$-0.35663 \times (Holes) - 0.184483 \times (Bumpiness)$$

2. **El-Tetris [8]**: This function use the same features introduced by Dellacherie. But the weights were tuned using a GA algorithm called "Particle swarm optimization". And the results were impressive, that the number of cleared lines on average reached 16,000,000 lines per game. The evaluation function is

$$-4.500158825082766 \times LandingHeight + 3.4181268101392694 \times ClearedLines$$

$$-3.2178882868487753 \times RowTransition\ -9.348695305445199 \times ColTransition$$

$$-7.899265427351652 \times Holes\ -3.3855972247263626 \times Tells$$

2.4.3 Q-Learning

Deep Q-Networks(DQNs) were used with CNN layers by Stevens and Pradhan[9]. They used the raw state as an input the to the neural network and added some other techniques like grouping actions and prioritized sweeping. The average score they achieved was around 18.0 which is much more far away the AI controllers that depend on evaluation functions to decide the placement of the pieces.
Q-learning and NN were also combined by Lundgaard and McKee[10] but they used high-level features that describe the board instead of the raw state of the board.
The used Features are

1. **The current piece**
2. **Next piece**
3. **Board Height** represents the average height of the board columns
4. **Board Level** is a boolean value that states whether or not the column heights are even.
5. **Has Single Width Valley** is a boolean value that states whether or not there is a slot that an I piece might fit into.
6. **Has Multiple Valley** is a boolean value that states whether or not there are multiple slots.
7. **Number of Holes** is reduced to ranges of 1-5, 6-10, 10 or more

Also Q-learning was used by Bdolah and Livnat[11] but, they used reduced version introduced by Melax[12]. That the height of the game is unlimited, only five reduced Tetriminoes and only six columns. They used two state spaces, contour and TTL(Top Two Levels). They tested how good the model is by knowing the max height of the columns, the lower the height the better the agent. They also used TD(0) as another RL algorithm.
Their results showed that using TTL as their state space outperformed all other strategies. Also Q-learning showed better performance than TD(0) or SARSA.

2.4.4 Temporal Difference(0)

TD(0) algorithm was applied by Thiam, Kessler and Schwenker[13] to Tetris. They managed to reduce the state space to be 7^9 as they used the contour of the board

clamped at valued (-3,3) to be their state space.
They also introduced two reward functions to describe how is the transition from the current state to the next state. For the first reward function the agent could drop 2000 pieces after training with 3.5×10^9 greedy gaming pieces but with the second reward function they managed to achieve around 400,000 pieces after training also with 3.5×10^9 greedy gaming pieces.
They used three features for the second reward function:

1. **Average Height**
2. **Holes**
3. **Quadratic Unevenness** This feature is the sum of the squared values of the differences of the neighbouring columns. It is a good indicator about how the pieces is distributed.

The weighted evaluation function are:

$$-5 \times avg_{height} - 16 \times holes - QU$$

And the reward for each time step would be:

$$r_{t+1} = evaluation_{t+1} - evaluation_t$$

Also TD(0) was used by Gross, Friedland and Schwenker but they tried to use the top four used rows as their state space[14]. But it would huge state space size as it would be $2^{40} \approx 10^{12}$ so they encoded each column to be the top used cell height. Therefore the state space would be reduced to $5^{10} \approx 10^7$. They managed to reach around 20,000 dropped pieces on average.

3 Experimental methods

This chapter contains the full details of the setup. First, a task description of the project will be explained. Second, we will discuss the software requirements for the task.

3.1 Overview

Our approach is to build an RL agent capable of playing Tetris and that is by trying many state spaces, reward functions, different NN architecture and every thing related to the agent. Also we would rebuild a Tetris environment to be able to ease and simplify the use of the environment through the project.
We have selected DQN to be our RL algorithm since it showed great success in playing Atari games[2]. But we would not whole screen pixels, due to the huge state space. Instead, we would use features extracted from the tetris board hoping that it would approximate the information about the whole board.

3.2 Task Description

Our task revolves around implementing an agent which uses DQN as RL algorithm to decide which action to take based on the state. The agent does not depend on the raw input of the board but rather we would feed the network high-level features extracted from the board for example maximum height and quadratic unevenness, we would have a deeper look on those features next chapter.

So the core of the project is to perfectly map the raw state of the board to a high-level features and engineer a reward function based on these feature to give the agent a sensible feedback about its performance.
There are many versions of Tetris. We can categorize these versions according to 2

things, the dimensionality of the board and which controller the Tetris is.

Tetris board dimension can be 1D, 2D, 3D or even 4D[4]. Regarding the controllers, there are one-piece controller which only have the knowledge of the current piece and two-piece controller which have the privilege of knowing the next piece. So we would choose the simplified version of Tetris which is 2-D and one-piece-controller version. As 1-D is a trivial problem and if 2-D is solved then the higher version of the game is also solved.
Although having the knowledge of the next piece improve the performance several orders of magnitude, but we chose the one-piece controller as in the real world any agent would not have these extra info about the environment.
Also, Tetris can be considered as a scheduling problem which can be used in real world application. So for the sake of simplicity and generality we choose the simple version of Tetris.

We have implemented many reward functions whenever we changed the high-level state space. But, we have settled down to two reward function as they proved to be great fixed policy agents. We have tuned their weights experiment after another to give the agent realistic taste of how good the performance is.

3.3 Software

In this section, I will discuss the environment and the software requirements for this project.

3.3.1 Environment Platforms

Unfortunately Tetris is not supported by OpenAI Gym, so I had to look for an environment that satisfy the requirements I need for the project.
I have found many environments for Tetris. But they are not easy APIs to use, also they are always rendering the game as channels of RGB which is quite an expensive operation that is not needed as we only care for the high-level features of the board. So we decided to rebuild an environment which is simple[15].
The environment was just 2D array of zeros and ones and has the basic actions of game (Move Left, Move Right, Rotate Left, Rotate Right, Soft drop, Hard drop, Noop)

where Soft drop is dropping the piece one cell down, Hard drop is dropping the piece to the bottom and Noop is nothing.
So We rebuild this environment to be more like Gym interface[16] But we omitted

OpenAI Gym API Interface	
Function	Description
reset(self)	Reset the environment state and returns the observation
step(self,action)	Step the environment by one time-step and returns observation , reward, done, info
render(self,mode='human')	Render one frame of the environment

Table 3.1: OpenAI Gym API Interface

some minor things for example we do not return info from the step function as we do not need it. Also at the time of implementing we were not in need for the render function so we did not give any effort then.
We added features to ease using the environment and make it for any researcher afterwards. That the user can pass these parameters while creating the environment. These features are:

- **Grouped Actions:** We implemented grouped action that instead of taking an action every time step we created an action space of size 40[1] which are a combination of all possible actions can be taken in Tetris. That the normal actions add much more computation that is not needed and barely affect the performance of the agent[2]
- **Different controllers implemented**: As we have said, we intend to make the agent learn through high-level state features. Since there is a lot of controllers introduced by many papers we implemented all most of these controllers that the users can pass what state space they want by passing the name of the controller.
  These controllers might be introduced in some papers or they are just some combined features. And the the function can return either the features itself. Or the weighted sum of these features.

[1]The number of actions depend on the size of the board. But, in this case we are using the normal dimension of the board which are width=10 and height=20
[2]Of course there would not hacks like rotating a Tetrimino at last moment to fit into a hidden corner or something like that.

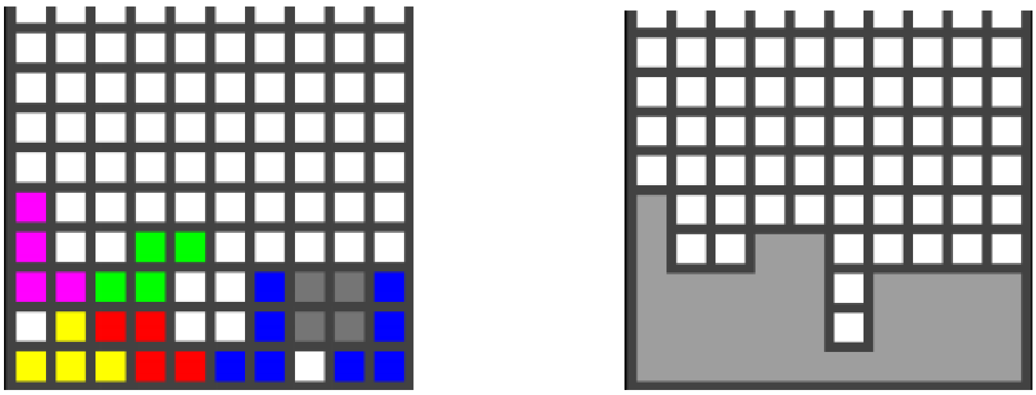
(a) Actual Board. (b) Contour.

Figure 3.1: Actual Board Vs. Contour Representation

The controllers are:

1. **Dellacherie [4]** mentioned in section 2.4.1
2. **Near the perfect bot [7]** mentioned in section 2.4.2
3. **El-Tetris [8]** mentioned in section 2.4.2
4. **Schwenker2014 [13]**The features of average height, holes and quadratic unevenness mentioned in section 2.4.4
5. **Schwenker2008[14]**The features of the encoded top four used rows that mentioned in section 2.4.4
6. **Lundgaard [10]** We used the features mentioned in section 2.4.3 but we omitted two features which are (Next Piece) and (Has Multiple Valley) and replaced (Board Level) with QU feature from [14][13]. We also ranged QU value by dividing it by 20 and then it would be clipped at 20.
7. **Melax [12][11]** Top Two Level of the board mentioned in section 2.4.3. But we added a variable to control how many top levels we want, we called this variable the *Melax Factor*.
8. **Contour [17]** This controller represents the contour of the board which is the height differences between the columns Figure (3.1b).
9. **Ashry** This controller is a mix of some features but scaled down to down-size the state space. The feature are (Aggregate height)/4, Average Height,

QU [14], Holes/2, Bumpiness/2 [7]

10. **Contour and Holes [18]** This controller combines the contour, the number of holes for each column and maximum column height.

3.3.2 Agent

We have used many methods to increase the performance of the agent.

- **Noisy Networks [1]** There is a dilemma in RL called exploit vs. explore problem. And it is usually solved by using epsilon-greedy policy which means that the agent does not take the optimal action every time. Instead, it takes random action with probability ϵ. This approach works well for simple environment with short episodes, without much non-stationarity during the game. There is a simple solution but it works well[3].
  The solution is to add noise to the weights of the fully connected layers of the network and adjust the parameters of this noise during training using backpropagation, these layers is called **Noisy Layers**.
  There are two ways to achieve this goal, but we only used the following:
  Independent Gaussain noise for every weight in a fully connected layer, a random value is sampled from the normal distribution. The parameters of the noise,μ and σ, will be trained during backpropagation.
- **Dueling DQN [1]** The core idea of this technique that instead of training the network to predict the approximate Q-values, the network is trained to predict two value V(s) and A(s,a) where A(s,a) is called Advantage.
  A(s,a) represents the delta, saying how much extra reward if specific action is taken.
  The dueling DQN process the input features through two different paths, figure 3.2, One for predicting the V(s), which we call in our experiments **value_sequential**, which is a single number, and the other for A(s,a) values, which we call in our experiments **adv_sequential**. Both of these paths originate from one common path, which we call **basic_sequential**.

[3]Noisy Networks paper

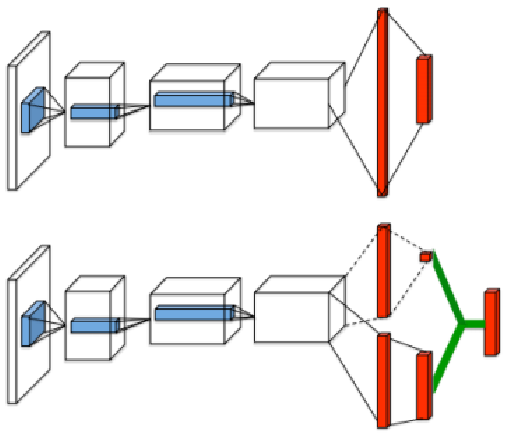

Figure 3.2: Traditional DQN(top) Vs. Dueling DQN(bottom)[1]

Q-values can be calculated using the next equation:

$$Q(s,a) = V(s) + A(s,a) - (1/N) \times \sum_{k} A(s,k)$$

The Term $(1/N) \times \sum_{k} A(s,k)$ is subtracted from the predicted values to make sure that the network will learn V(s) and A(s,a) well[4].

- **Double DQN (DDQN) [1]** This idea came from DeepMind researchers in the paper *Deep Reinforcement learning with Double Q-Learning*. They showed that the basic DQN tends to overestimate Q-values which may harm the performance and lead to suboptimal policies. The solution is to use the trained network to get the actions of the next state.

$$Q\left(s_t, a_t\right) = r_t + \gamma \max_a Q'\left(s_{t+1}, \arg\max_a Q\left(s_{t+1}, a\right)\right) \tag{3.1}$$

- **Several environments [1]** AI researchers manage to speed up neural networks training through data-parallelism. They found that using larger batch sizes would increase amounts of computation[19].
  But this may have downside as mentioned in [1][5], As the agent explores and the learn the training data changes. So the solution is to use several copies of the

[4]For more explanation, you can review the the text book chapter 8, Dueling DQN section
[5]Chapter 9

environments and on every training iteration, the replay buffer will be populated from samples from all those environments

4 Results and Discussions

This chapter include every experiment has been done throughout the project. Each experiment is divided into version. The versions of the one experiment may differ from one to another on some minor changes for example Reward function, Hyperparameters or even small modification on the Agent.
But the experiments differences would be major like changing many things in the same time for example changing state space and network architecture. Also There would be experiments whose goals is to solve side problems arises during training.

4.1 Vanilla DQN

In this experiment we are experimenting the vanilla implementation of DQN accompanied with grouped action. We are using different setups e.g. state space, reward function and hyparameters in each version.

The aim of the experiment is to test and explore the environment and form a solid idea about the agent life and then improve it.

4.1.1 Version one[1]

- **State Space**
  The state space is raw input of the grid. Which means it is simply a 2D of zeros and ones that represents the whole board including the current piece.
- **Action space**
  The action space is discrete which is grouped actions[2].

[1] Each version will contain its setup in detail

[2] In this experiment the number of actions were 44 which means there are redundant action and this issue was fixed in the next experiment

- **Reward function**
  We used the proposed evaluation function from Kessler, Thiam and Schwenker, see section 2.4.4. The reward is calculated by subtracting the evaluation of the current state from the evaluation of the previous state. We added the term $10 \times ClearedLines - Wells$ to the evaluation function to encourage the agent to take such actions that lead to clearing lines and avoid making the wells.
  There is also a death penalty of -10. That means if the game is lost the reward would be -10. The penalty has been added to make the agent learn to avoid such actions which lead to the loss.

  $$Reward(S)_t = Evaluation(S)_t - Evaluation(S)_{t-1}$$

- **Hyperparameters**
  There are some tuned hyperparameters:
  - GAMMA $=$ 0.99
  - BATCH SIZE $=$ 640
  - REPLAY SIZE $=$ 10000
  - REPLAY START SIZE $=$ 10000
  - LEARNING RATE $=$ 1e-5
  - SYNC TARGET FRAMES $=$ 10000
  - EPSILON DECAY LAST FRAME $=$ 150000
  - EPSILON START $=$ 1.0
  - EPSILON FINAL $=$ 0.01

  We used large batch size as mention in [19] would increase amounts of computation.

- **Neural Network Architecture**
  The used network is a feedforward network consisted of 6 Fully Connected (FC) layers and we used Batch Normalization and Dropout layers with dropping rate 30% between some layers. The reason we used both Batch Normalization and Dropout layers is that they would boost the training effeciency of the neural network[20]. Rectified Linear Unit (ReLU) is used as the activation function
  The final Network Architecture is :

1. **Input Layer**: FC layer of size 256.
2. **Batch Normalization Layer**
3. **ReLU**
4. **Dropout Layer**: with dropping rate of 30%
5. **Hidden Layer**: FC layer of size 200.
6. **Batch Normalization Layer**
7. **ReLU**
8. **Hidden Layer**: FC layer of size 160.
9. **Batch Normalization Layer**
10. **ReLU**
11. **Dropout Layer**: with dropping rate of 30%
12. **Hidden Layer**: FC layer of size 110.
13. **ReLU**
14. **Hidden Layer**: FC layer of size 70.
15. **ReLU**
16. **Output Layer**: FC layer of size the action number.

- **Loss function and Optimizer**
  As shown in the NN architecture we are not using activation function after the output layer since we are trying to predict the Q-values of the actions so we used Mean Square Error(MSE) as our loss function[2].

  $$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n} \tag{4.1}$$

  Also, we used Adam optimizer.

- **Results**:
  After Training the agent for playing over 3800 games. We recorded the Q-loss and the reward values.

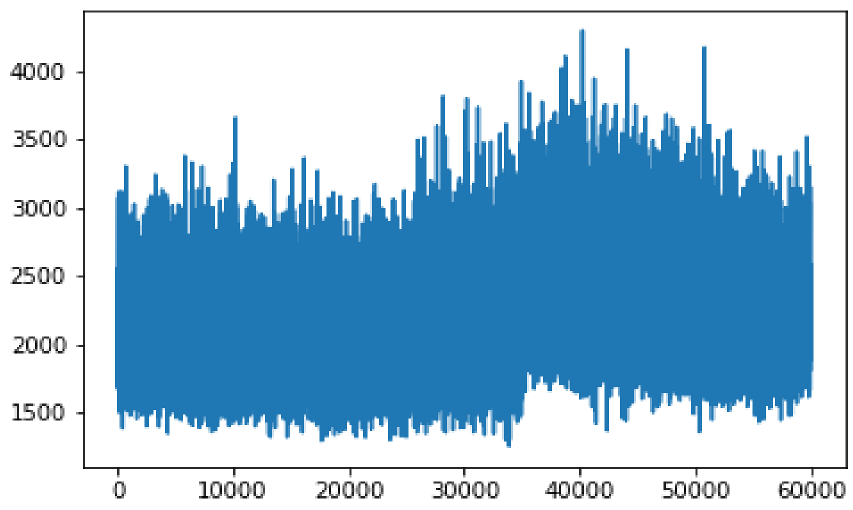

Figure 4.1: Q-losses for experiment 1 version 7
x-axis represents number of frames
y-axis represents Q-losses

As shown in figure 4.1[3] The Q-losses is starting high and by time the Q-losses tend to increase.
Also, in figure 4.2, it appears how bad the evaluation function. That the maximum value for the maximum value for the reward is -200 and the performance is decreasing.

[3]The graphs and figures of this experiment recording each value, not averaged. Therefore the graphs seems not to have specific curve or trend. This issue is fixed in the next experiments

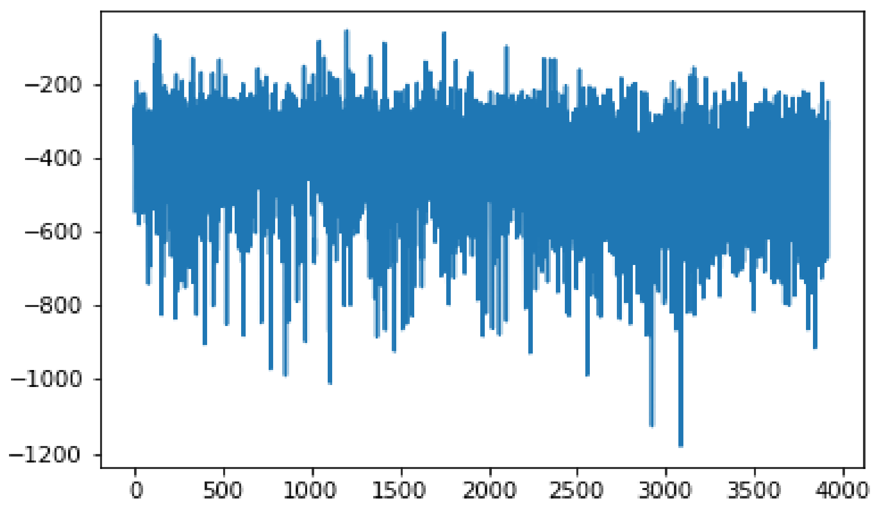

Figure 4.2: Rewards for experiment 1 version 1
x-axis represents number of games
y-axis represents rewards

4.1.2 Version two[4]

- **State Space**
  The state space is the top 5 rows of the board[5]. The idea here that some of the state would be just zeros at the beginning of the game so the more zeros the agent gets the more time it spends in the game. So the idea is to train the agent to spend more time in the game as much as possible.

- **Reward function**
  We used the same evaluation function in experiment 1 version 1 (e1.v1), but the reward is different. We added constant one to the reward and then we scaled the reward to be between [-50,50] using Sigmoid function. We followed the lead of [2], as the values of rewards were highly ranging, so we scaled them hoping

[4]If there are fields missing from a version that means that this field is the same as the previous version
[5]Not the top 5 **USED** rows

it will improve performance[6].

$$Reward(S)_t = Sigmoid(Evaluation(S)_t - Evaluation(S)_{t-1} + 1)$$

- **Results**

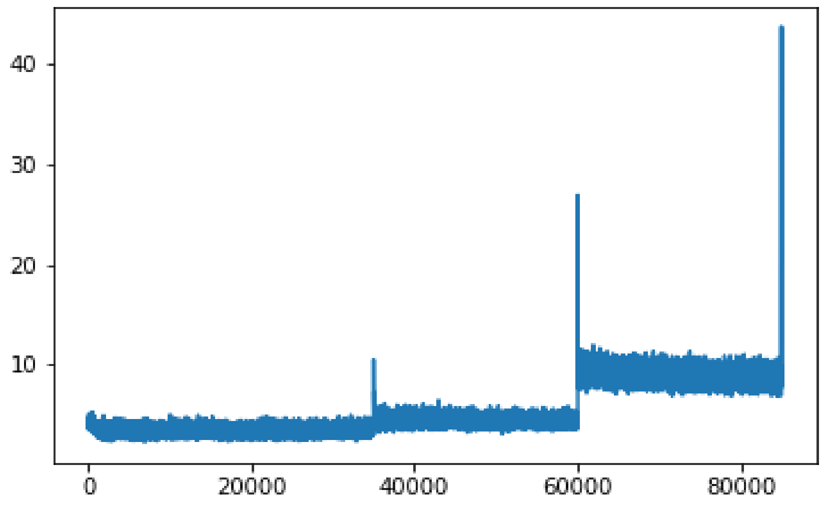

Figure 4.3: Q-losses for experiment 1 version 2
x-axis represents number of frames
y-axis represents Q-losses

As shown in figure 4.3 The Q-losses increases by time and in figure 4.4 the rewards still decreases. They are between [-10,-40] values. And that is after training the agent for 6000 games.

[6]It won't because this does not help the agent at all. As the agent won't know the difference between good and better decisions. If the agent cleared 4 lines at once it won't differ that much from clearing one line

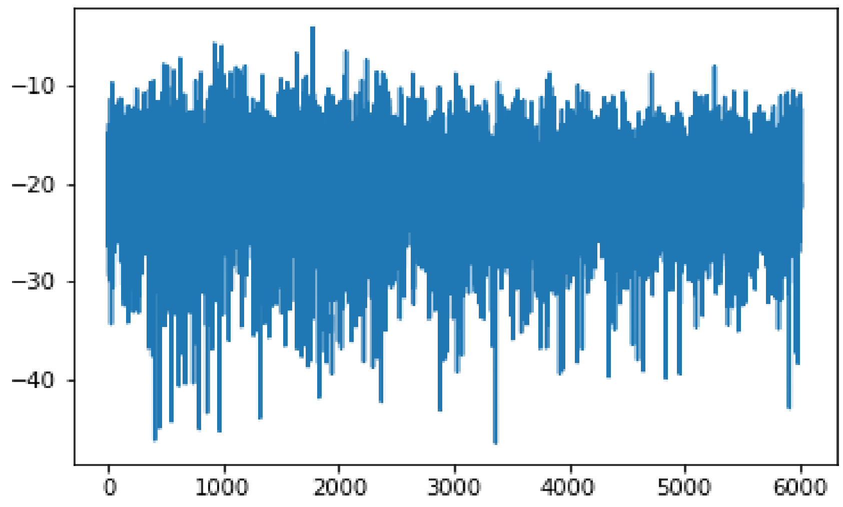

Figure 4.4: Rewards for experiment 1 version 2
x-axis represents number of games
y-axis represents rewards

4.1.3 Version three

- **State Space**
  We used the contour representation of the board[7] and the index of the current Tetromino.
- **Reward function**
  We used the same evaluation function from e1.v1 but there is not constant one nor penalty at losing.
- **Hyperparameters[8]**
  - LEARNING RATE = 1e-4
  - BATCH SIZE = 120
- **Resutls**

[7]refer to section 3.3.1 and read Implemented controllers
[8]If a hyperparameter does not exist that means it is the same as the previous version

After training the agent for over 8000 games, the results did not differ much from the previous version. As shown in figures 4.5 and 4.6

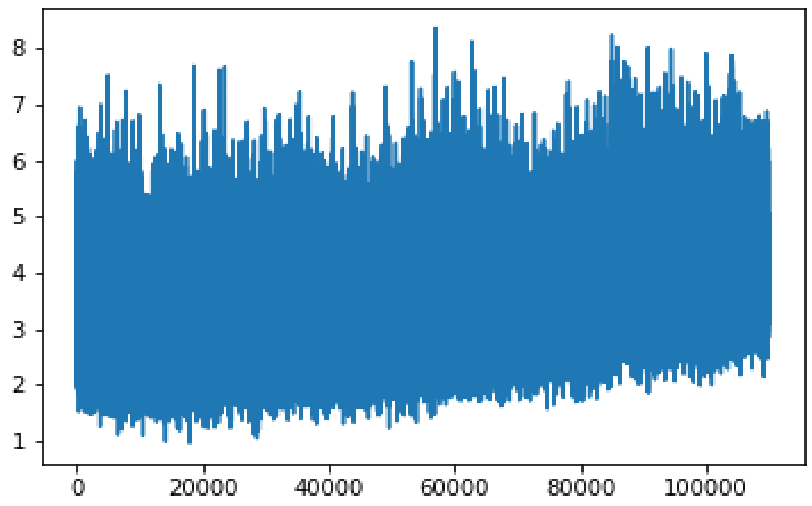

Figure 4.5: Q-losses for experiment 1 version 3
x-axis represents number of frames
y-axis represents Q-losses

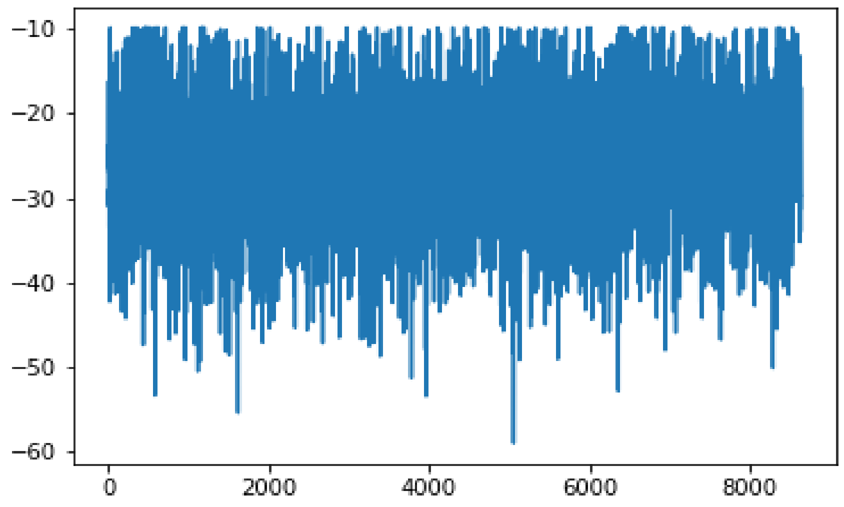

Figure 4.6: Rewards for experiment 1 version 3
x-axis represents number of games
y-axis represents rewards

4.1.4 Version four

- **Reward function**
  The evaluation function is the one proposed by Kessler, Thiam and Schwenker[9], but the reward is different.

  $$Reward(S)_t = Sigmoid(Evaluation(S)_t - Evaluation(S)_{t-1}) + 1 + 10 \times ClearedLines$$

  As the only the evaluation values are the only scaled values and it is scaled between [-15,15]. Also we used death penalty again and it is set to be -10.

- **Hyperparameters**

  – Batch Size = 256

- **Results**
  After training the agent for about 6000 games. The performance did not change

[9]see section 2.4.4

it is sill bad.

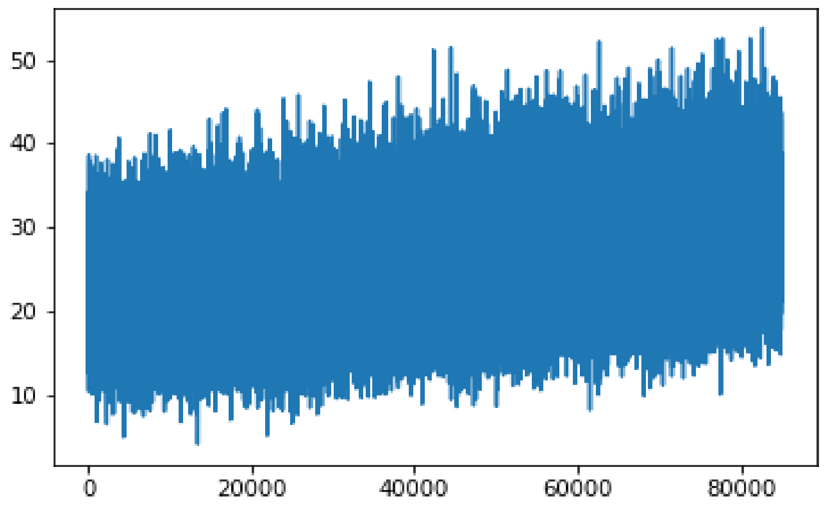

Figure 4.7: Q-losses for experiment 1 version 4
x-axis represents number of frames
y-axis represents Q-losses

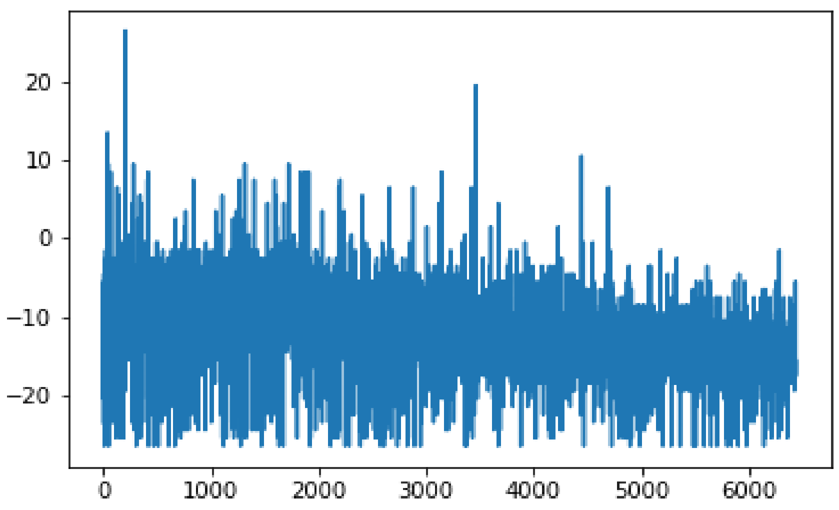

Figure 4.8: Rewards for experiment 1 version 4
x-axis represents number of games
y-axis represents rewards

4.1.5 Version five

- **Reward function**
  The only change is that the evaluation values are scaled to be between [-10,10]
- **Loss function**
  We changed the loss function from MSE loss function to be Huber loss function. Since it is less sensitive to outliers in data than MSE.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{4.2}$$

- **Results**
  After training the agent for over 8000 games. The performance is still bad as the Q-losses increases and the rewards decreases.

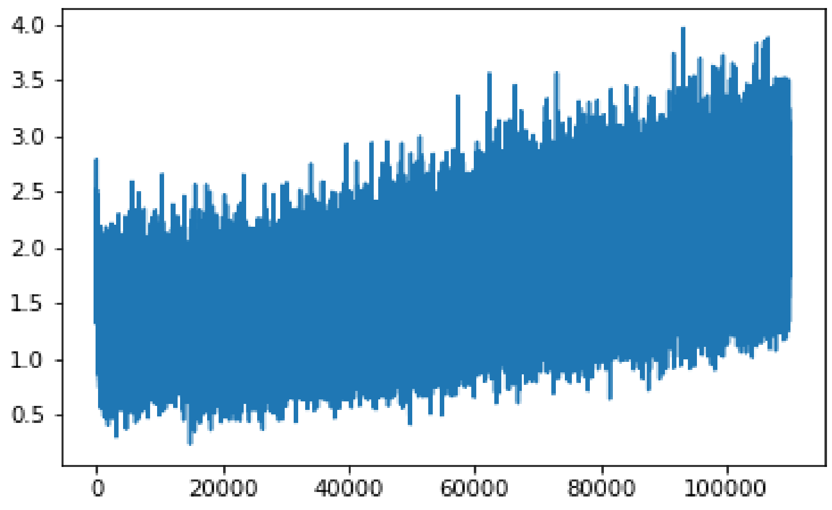

Figure 4.9: Q-losses for experiment 1 version 5
x-axis represents number of frames
y-axis represents Q-losses

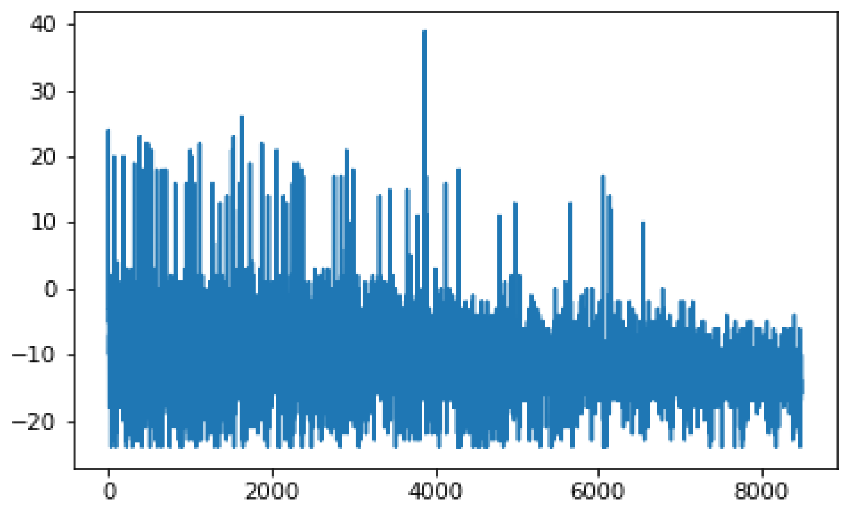

Figure 4.10: Rewards for experiment 1 version 5
x-axis represents number of games
y-axis represents rewards

4.1.6 Version six

- **Hyperparameters**
  - EPSILON DECAY LAST FRAME = 50000
- **Reward function**
  We changed penalty to be -1 instead of -10 because we noticed that most of the rewards the agent get are negative so it does not know what are the good and bad choices.
- **Results**
  After training the agent for over 10,000 games, the performance is still bad and the Q-losses still increases and the reward curve is saturating at zero.

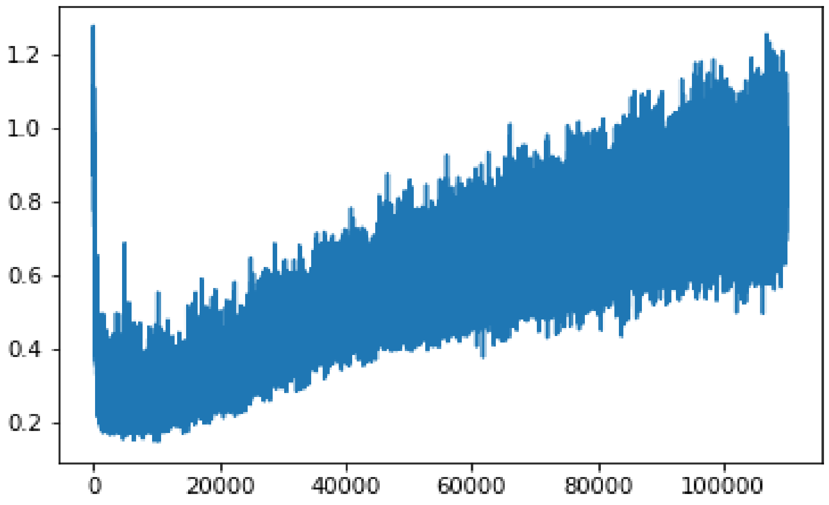

Figure 4.11: Q-losses for experiment 1 version 6
x-axis represents number of frames
y-axis represents Q-losses

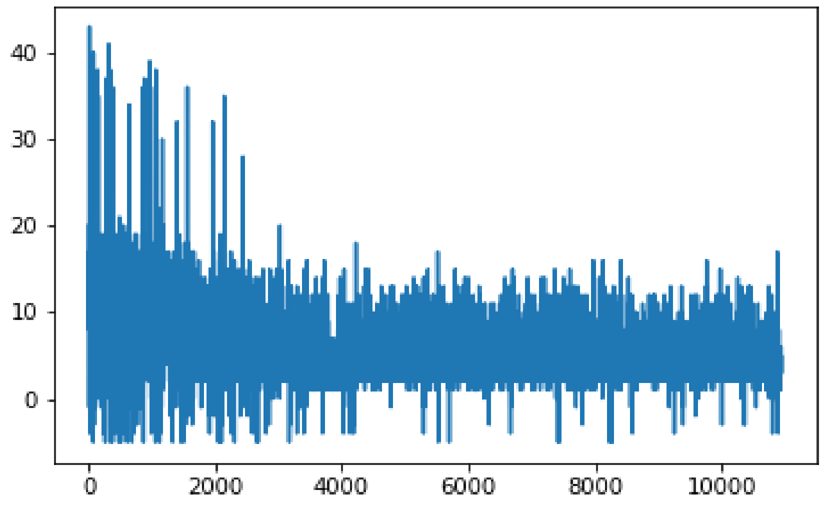

Figure 4.12: Rewards for experiment 1 version 6
x-axis represents number of games
y-axis represents rewards

4.1.7 Version seven

- **Hyperparameters**
  - EPSILON DECAY LAST FRAME $=$ 200,000
- **Results**
  The results for the first 10,000 games were disappointing that the rewards curve does not still decrease and the Q-losses increases.

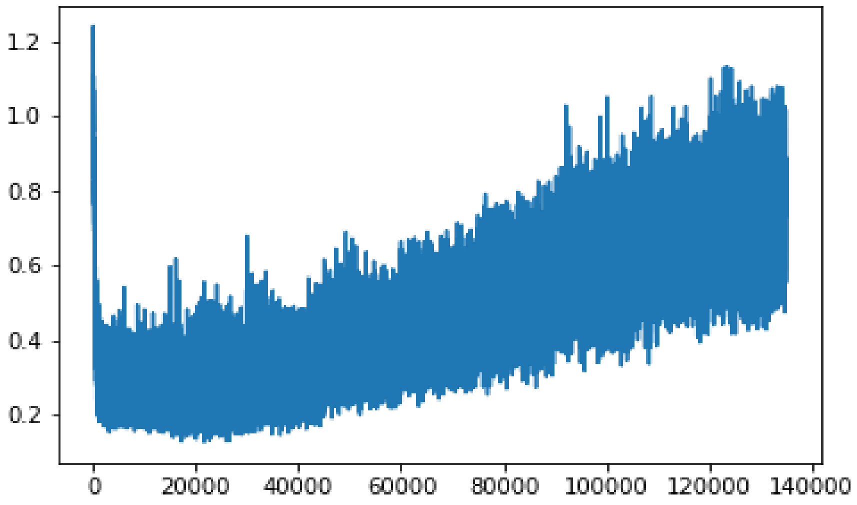

Figure 4.13: Q-losses for experiment 1 version 7
x-axis represents number of frames
y-axis represents Q-losses

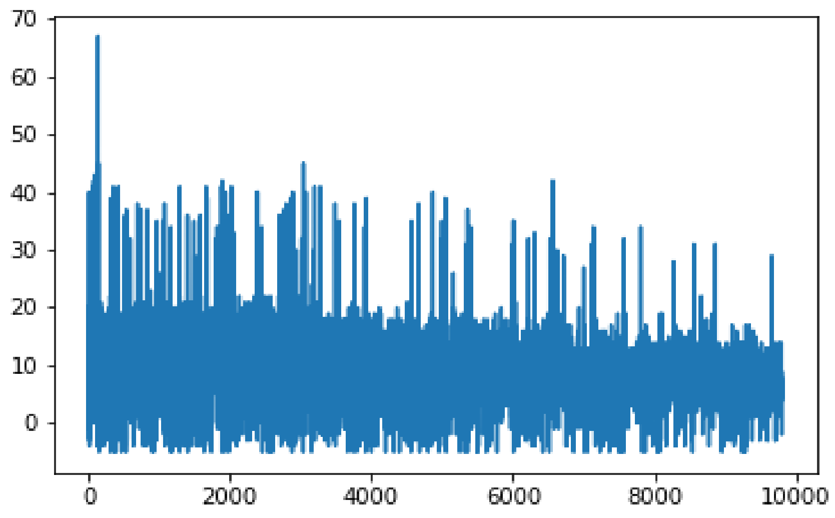

Figure 4.14: Rewards for experiment 1 version 7
x-axis represents number of games
y-axis represents rewards

4.1.8 Discussion

The results and the performance of the agent were very bad. As, In all experiments the Q-losses tend to increase and the rewards to decrease. We also tested the agent after each one of those experiments for 1000 games. It cleared zero lines in all versions except version seven as it cleared 25 lines per 1000 which is till worse than the random agent[10].

[10]The reason that the agent could clear lines in e1.v7 and zero in the other versions is that there was a bug in mapping state space which affected the performance. But, it the performance did not change that much as the agent clears 0.025 line/game

4.2 Heuristic agents[11]

In this experiment we are testing some of fixed-policy agents to compare between their performance and choose the best one of them to be a heuristic agent that is used to help the main agent to learn well.

The aim of the experiment is to implement fixed-policy agents, whether the weights are hand-crafted or developed by using GAs. That would help us in the next experiment to use these agents as heuristic agents[9].

We are testing three evaluation functions to be the fixed-policy agents. We tested each agent with the number of cleared lines and number of dropped Tetrominoes per game.
Each controllers took nearly the same amount of time but the number of games varies from one to another. As the better the performance is the more time it takes in a game so that means the less number of games in the specified period of time.

4.2.1 Schwenker2014 controller

We used the Schwenker2014 controller as a fixed-agent policy. The agent played 417 games. The max number of cleared lines is 410 line per game and the number of dropped Tetrominoes is around 1200 piece per game.

[11] You might find it useful to read again "Different controllers implemented" in section 3.3.1 before delving into that experiment

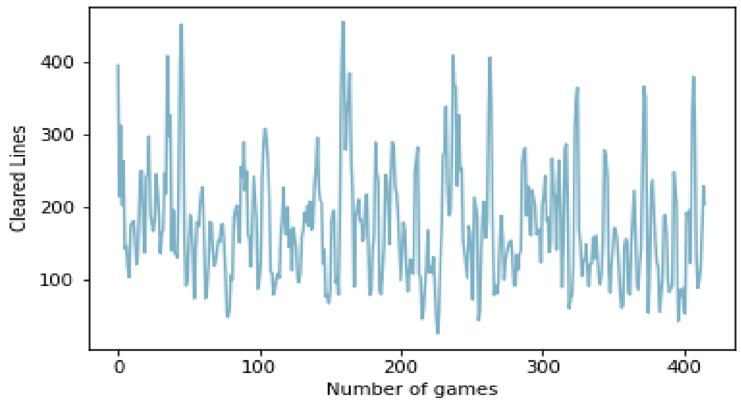

Figure 4.15: cleared lines for Schwenker2014 controller

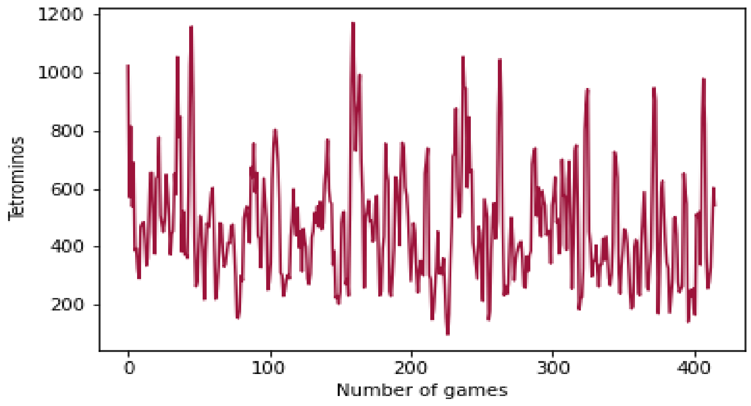

Figure 4.16: Tetrominoes for Schwenker2014 controller

4.2.2 Near controller

We used the Near controller as a fixed-agent policy. The agent played 164 games. The max number of cleared lines is 700 line per game and the number of dropped Tetrominoes is around 1600 piece per game.

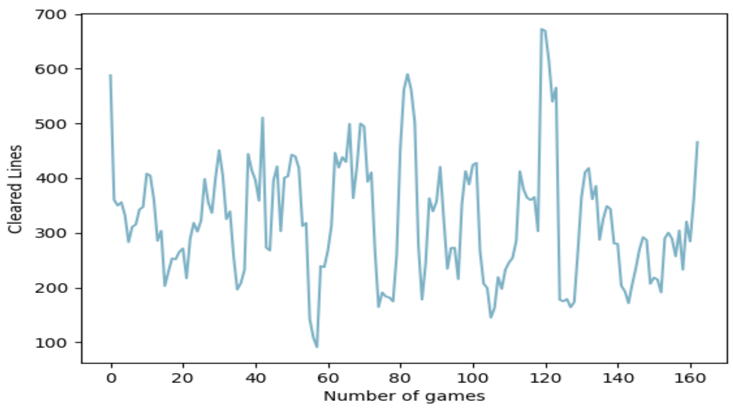

Figure 4.17: cleared lines for the Near controller

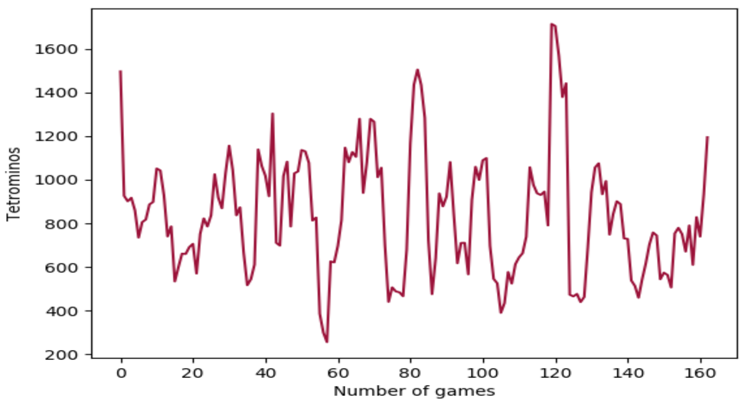

Figure 4.18: Tetrominoes for the Near controller

4.2.3 Dellacherie controller

After playing 24 games, the max number of cleared lines is 2500 line per game and the number of dropped Tetrominoes is 7000 piece per game. We tried to test the El-tetris controller but it took very long time in one game so we aborted the execution.

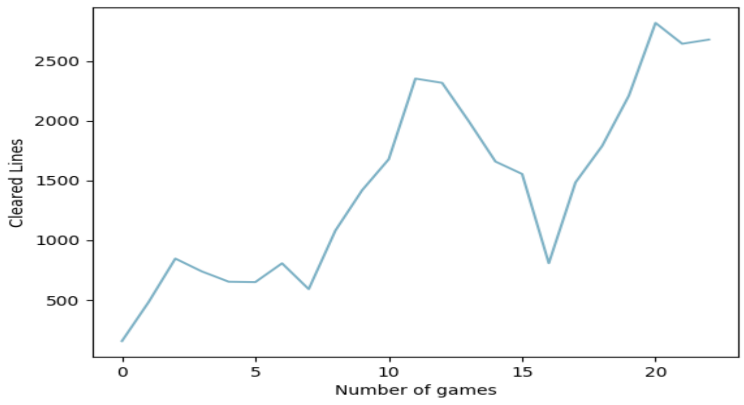

Figure 4.19: cleared lines for Dellacherie controller

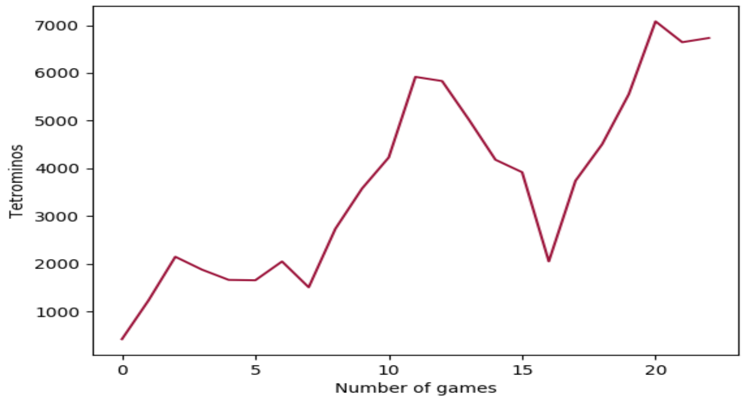

Figure 4.20: Tetrominoes for Dellacherie controller

4.2.4 Discussion

As it is shown in this experiment the performance of the fixed-policy agent, no matter what the evaluation function is, is much higher than the agent implemented in the experiment one. We would exploit this to improve the performance of the DQN agent and that will be shown in the next experiment.

4.3 Heuristic agents as an auxiliary agents

In this experiment we are using some of the heuristic agents that are mentioned in the previous experiment to help the main agent to learn from better experience rather than using just random experiences as a guidance to the agent.

The aim of the experiment is to use fixed-policy agents to help the DQN agent model to learn how to play Tetris using high-level states.

4.3.1 Version one

- **State Space**
  We used the contour representation of the board, number of holes, quadratic unevenness and the Tetromino index as the input the NN.
- **Action Space**
  We used grouped actions but we omitted the redundant actions mentioned in experiment one and now the number of actions are 40[12].
- **Reward function**
  We used the Near controller as the evaluation function.
- **Hyperparameters**

  – GAMMA = 0.99
  – BATCH SIZE = 360
  – REPLAY SIZE = 10000
  – REPLAY START SIZE = 10000
  – LEARNING RATE = 2e-6[9]
  – SYNC TARGET FRAMES = 1000[9]
  – EPSILON DECAY LAST FRAME = 1e6[9]
  – EPSILON START = 1.0

[12]Of course the number of grouped actions changes when the board width changes but in this case the board width is the normal width which is 10.

  - EPSILON FINAL = 0.01

- **Neural Network Architecture**
  The used network is a feedforward network consisted of 6 Fully Connected (FC) layers and we used Batch Normalization and Dropout layers with dropping rate 20% between all layers. Rectified Linear Unit (ReLU) is used as the activation function
  The final Network Architecture is :

  1. **Input Layer**: FC layer of size 256.
  2. **Batch Normalization Layer**
  3. **ReLU**
  4. **Dropout Layer**: with dropping rate of 20%
  5. **Hidden Layer**: FC layer of size 200.
  6. **Batch Normalization Layer**
  7. **ReLU**
  8. **Dropout Layer**: with dropping rate of 20%
  9. **Hidden Layer**: FC layer of size 160.
  10. **Batch Normalization Layer**
  11. **ReLU**
  12. **Dropout Layer**: with dropping rate of 20%
  13. **Hidden Layer**: FC layer of size 110.
  14. **Batch Normalization Layer**
  15. **ReLU**
  16. **Dropout Layer**: with dropping rate of 20%
  17. **Hidden Layer**: FC layer of size 70.
  18. **Batch Normalization Layer**
  19. **ReLU**
  20. **Dropout Layer**: with dropping rate of 20%
  21. **Output Layer**: FC layer of size the action number.

- **Epsilon-greedy policy and Heuristic agents**
  According to [9] the random actions rarely lead to a good action or even clear lines. So It is better to use an agent with a fixed policy to predict the best action. So there would be ϵ_{random} which indicate the probability of choosing a random action. ϵ_{agent} which indicates the probability of using the fixed policy agent to help Saffiya[13] to learn.

- **Results**
  After training Saffiya for about 13555 games which took much more time than the previous experiment and that is due to Epsilon Decay Rate was too small so that means in at first most of the operation will be either random or using the heuristic agents and that runs on CPU so it slowed down the process. As shown in figure 4.21 the losses increases by the time and also the rewards increases by the time, figure 4.22. Also we can see that number of cleared lines per game was about 1 and 2 lines per game, that is due to the effect of the heuristic agent, figure 4.23. But the performance Saffiya appears to decrease as the number of dropped Tetrominoes decreases by time, figure 4.24.

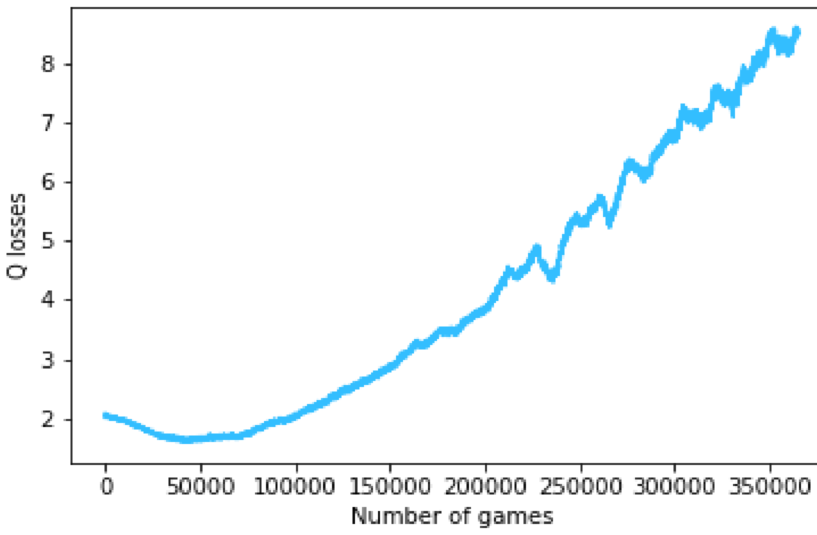

Figure 4.21: Q-losses for experiment 3 version 1

[13]We decided to call our agent Saffiya as Saffiya in arabic means pure hoping that the results of the agent will be pure without errors.

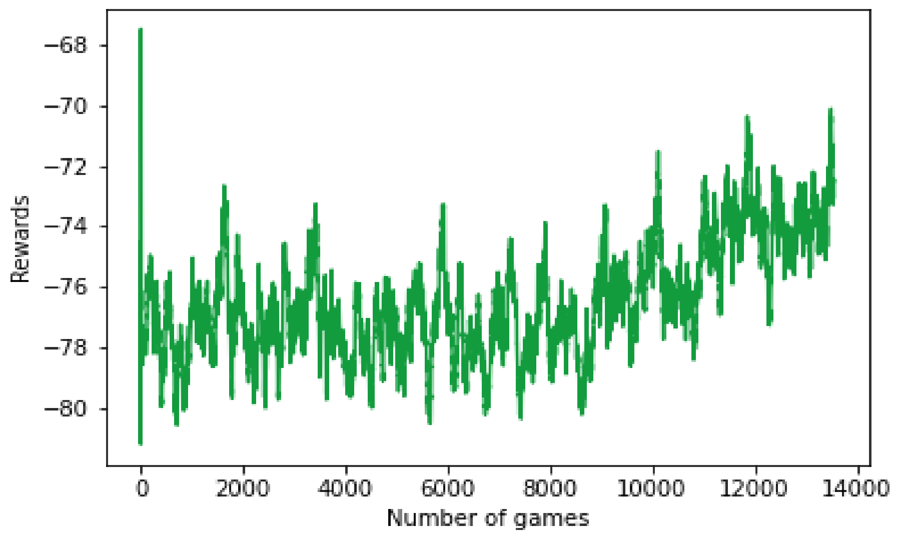

Figure 4.22: Rewards for experiment 3 version 1

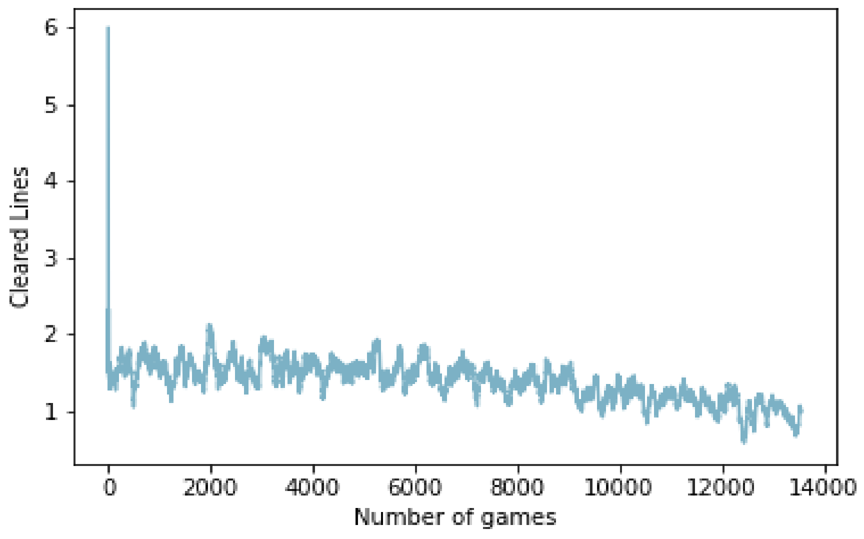

Figure 4.23: Cleared lines for experiment 3 version 1

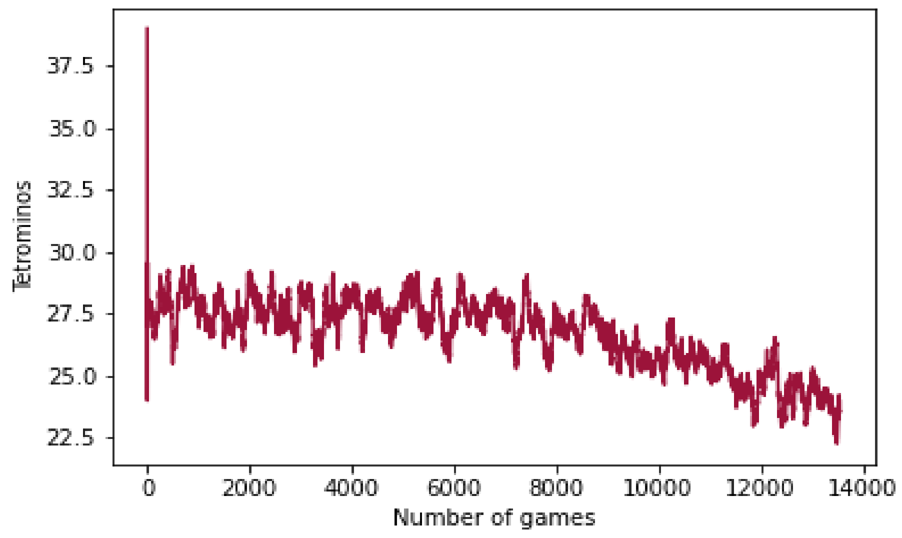

Figure 4.24: Tetrominoes for experiment 3 version 1

4.3.2 Version two

- **State Space**
  Instead of contour representation we used the column heights themselves.
- **Hyperparameters**
  - BATCH SIZE =128
  - SYNC TARGET FRAMES = 8000
  - EPSILON DECAY LAST FRAME = 5e5
- **Neural Network Architecture**
  It is the same as e3.v1 but we omitted BN layers from the model to test if there would any difference on Saffiya's performance.
- **Results**

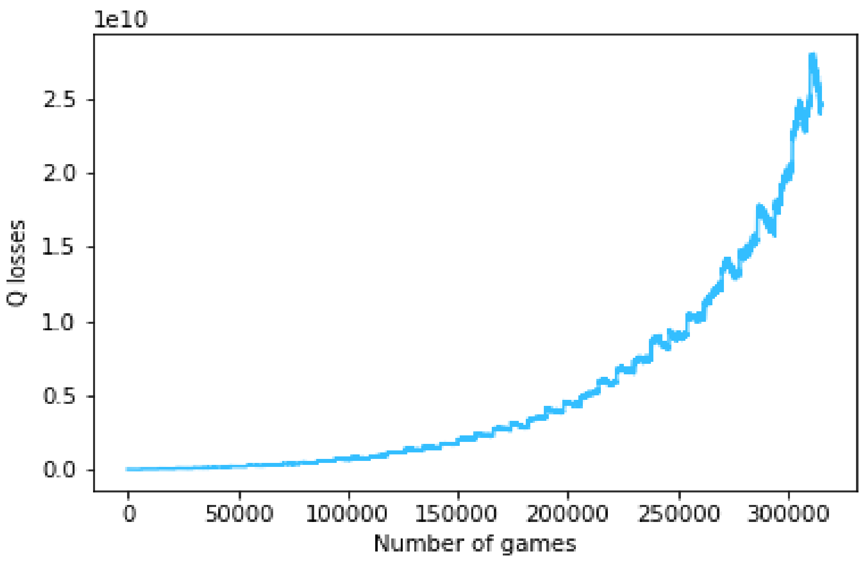

Figure 4.25: Q-losses for experiment 3 version 2

After training Saffiya for 13200 games, it is shown that omitting BN layers has actually affected the performance highly. As shown in figure 4.25 the Q-losses increase exponentially and the values are very huge (2.5e10). Nevertheless, the rewards values increase by time, figure 4.26. But, the performance of Saffiya whether in number of cleared lines or dropped Tetrominoes per game declines more.

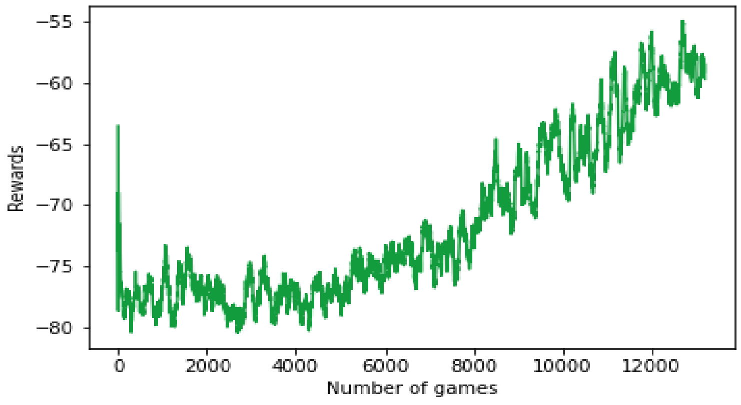

Figure 4.26: Rewards for experiment 3 version 2

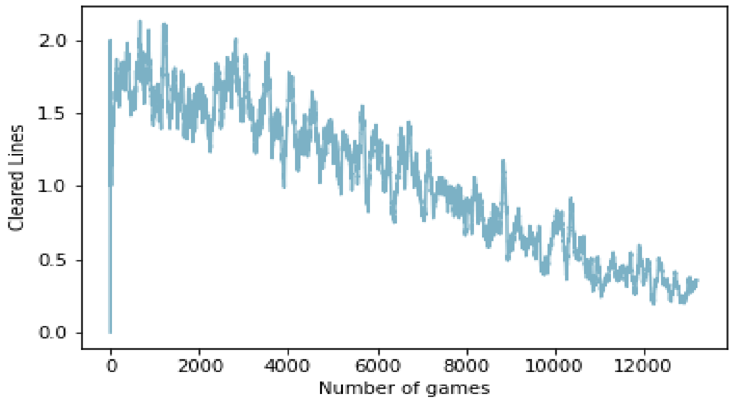

Figure 4.27: Cleared lines for experiment 3 version 2

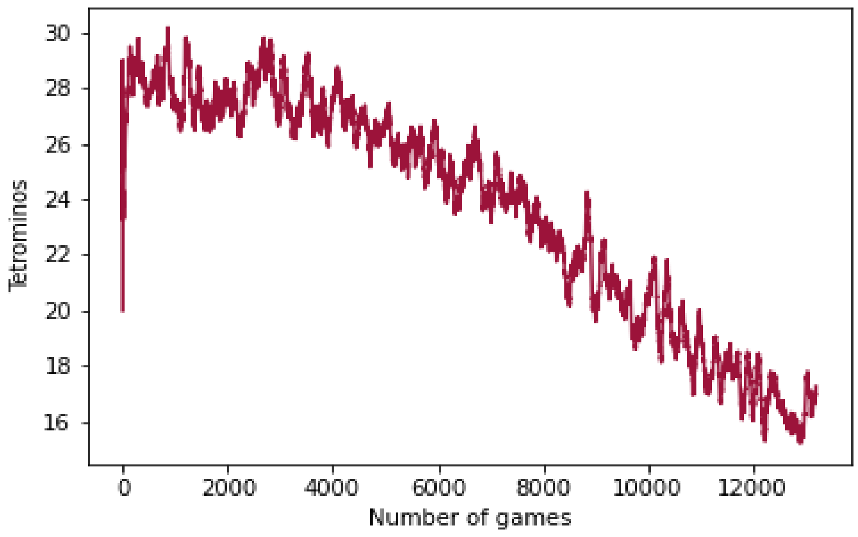

Figure 4.28: Tetrominoes for experiment 3 version 2

4.3.3 Version three

- **Hyperparameters**
  - SYNC TARGET FRAMES = 10000
  - EPSILON DECAY LAST FRAME = 1.2e5
- **Neural Network Architecture**
  We clipped the parameters of NN to be between (-1,1) as mention in [2] to improve stability of algorithm.
- **Epsilon-greedy policy and heuristic agents**
  We changed the heuristic agent to be Dellacherie controller instead of Near controller as it is shown in experiment 2 that Dellacherie controller has the best results.
- **Results**
  After training Saffiya for 16600 games, the results did not change that much from the previous version. It even worsens.

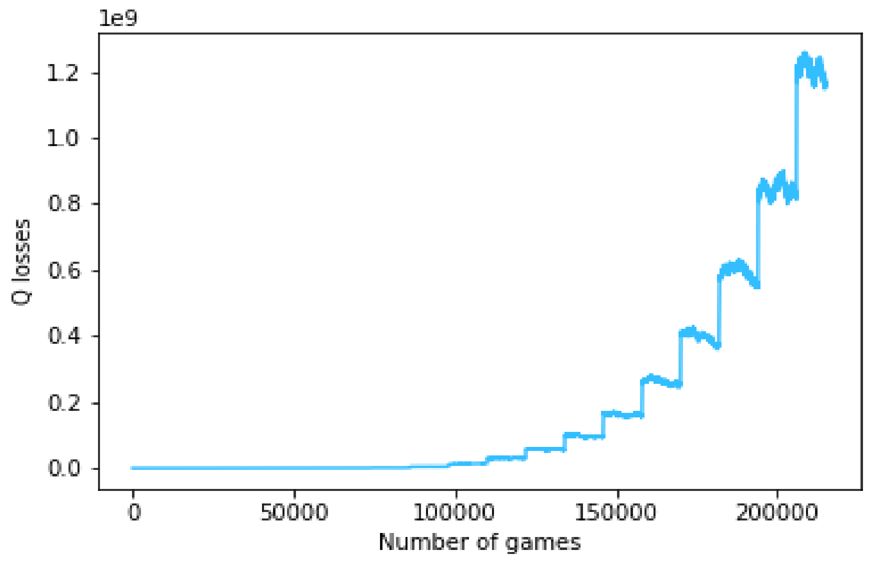

Figure 4.29: Q-losses for experiment 3 version 3

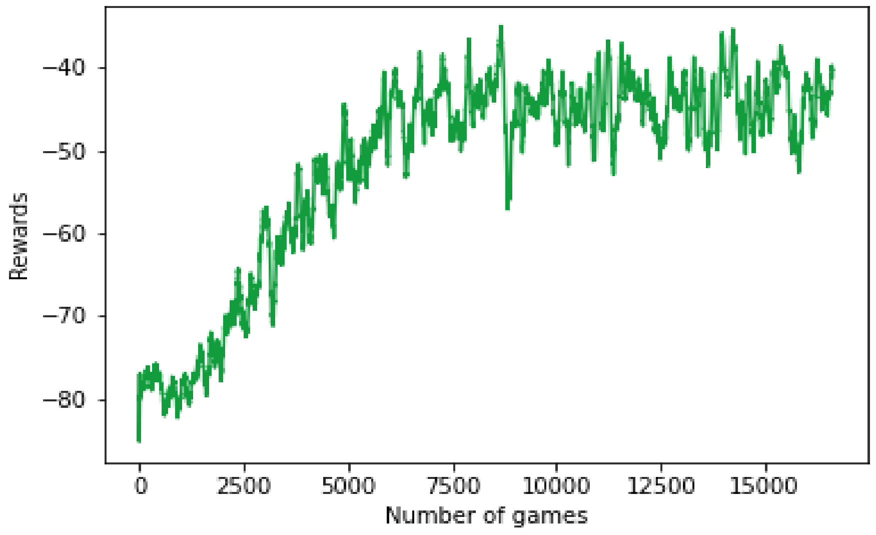

Figure 4.30: Rewards for experiment 3 version 3

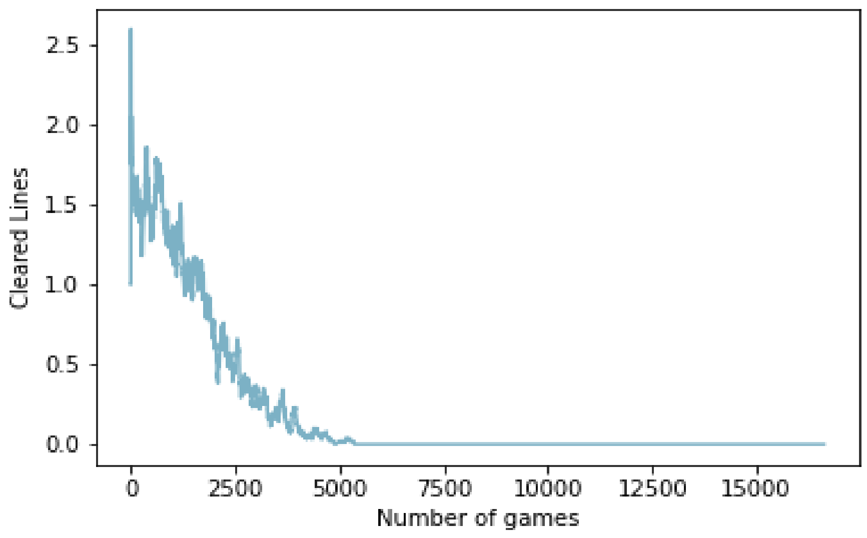

Figure 4.31: Cleared lines for experiment 3 version 3

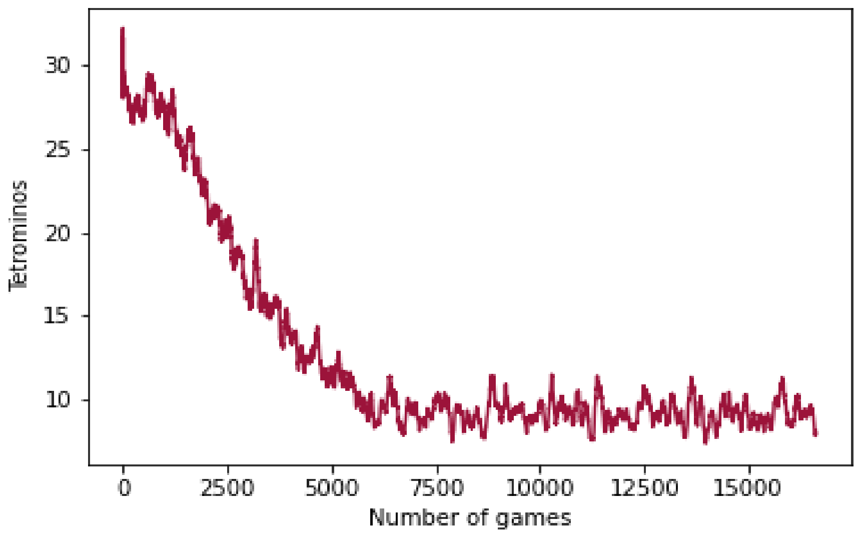

Figure 4.32: Tetrominoes for experiment 3 version 3

So, we suspected the behaviour of the NN as although of using superior Heuristic agent the performance of Saffiya declines before time, It is even worse than the random agent. So we tested Saffiya by playing 1000 games and then plotted actions frequency, figure 4.33.

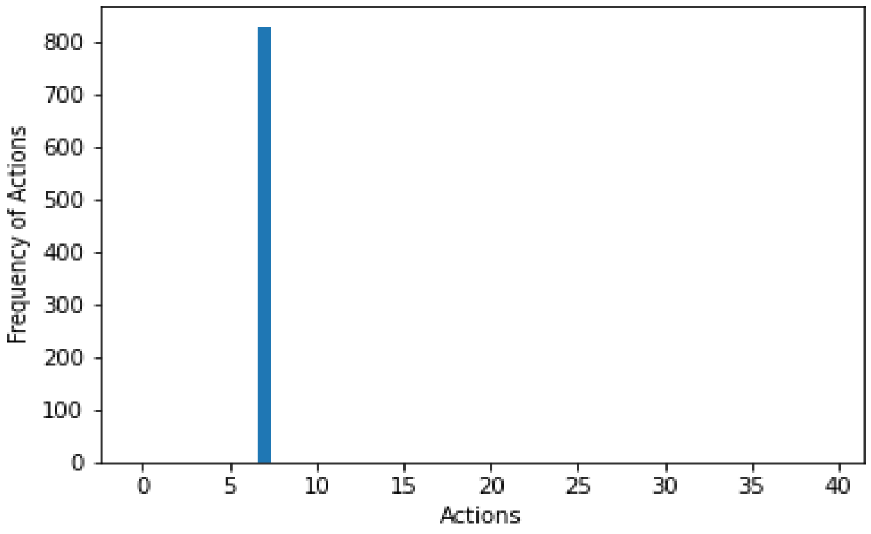

Figure 4.33: Actions frequency for experiment 3 version 3

It appears that the network saturates at specific values by time and favours just one action and it thinks it is the best. So in the next experiment we are trying to solve the problem of saturation.

4.3.4 Discussion

Although we thought that using a heurisic agent may help Saffiya learn well. But it turns out it did not help much in the performance. So after debugging we found that the network tends to saturate and favours one action every time and that is explains the bad performance of Saffiya so we need to solve the problem of saturation to help the network take (nearly) right decisions for the different states and that is what we trying to do in the next experiment.

4.4 Action saturation

In this experiment, we are testing some of the techniques to solve the neural network problem of the action saturation. The problem was not quite solved in this experiment but in the next experiment the problem was quite diminished.

The aim of the experiment is to try to solve the saturation problem of the network that the network always favours one action after long time of training and that is by applying different hacks to the model.

4.4.1 Version one

- **Reward function**
  We used Dellacherie controller to be the evaluation function.
- **Hyperparameter**
  - SYNC TARGET FRAMES = 13000
  - EPSILON DECAY LAST FRAME = 2e5
  - BETA = 0.9
- **Boltzmann distribution**
  As mentioned in [21] we used Boltzmann distribution for action selection instead of argmax. That the probability of taking certain action is computed through the following
  $$p(a \mid s)=\frac{\exp(\beta Q(a, s)}{\sum_{a^{\prime}} \exp \left(\beta Q\left(a^{\prime}, s\right)\right)} \tag{4.3}$$
  Where β is the temperature parameter and it governs Exploration-Exploitation trade-off.
- **Results**
  After training Saffiya for about 22000 games, the results was still disappointing. That the Q-loss is still increasing, Figure 4.34.

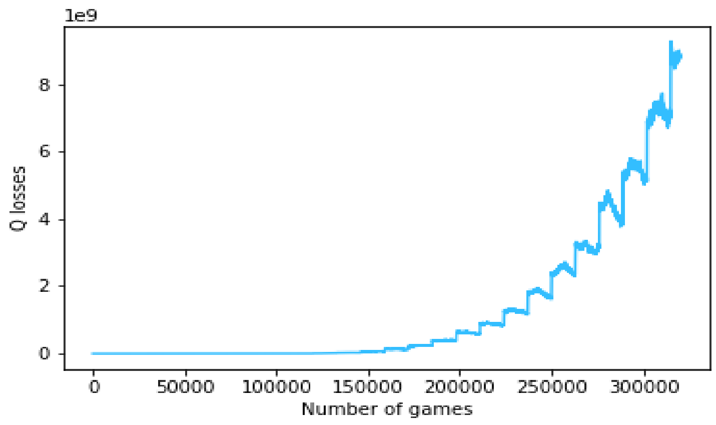

Figure 4.34: Q-losses for experiment 4 version 1

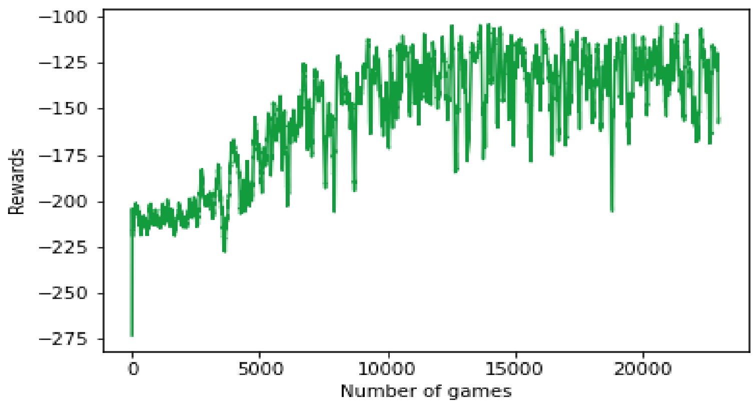

Figure 4.35: Rewards for experiment 4 version 1

Also, the rewards increases but it saturates at certain value, Figure 4.35. And Saffiya's performance in clearing lines and dropping Tetrominoes is still bad.

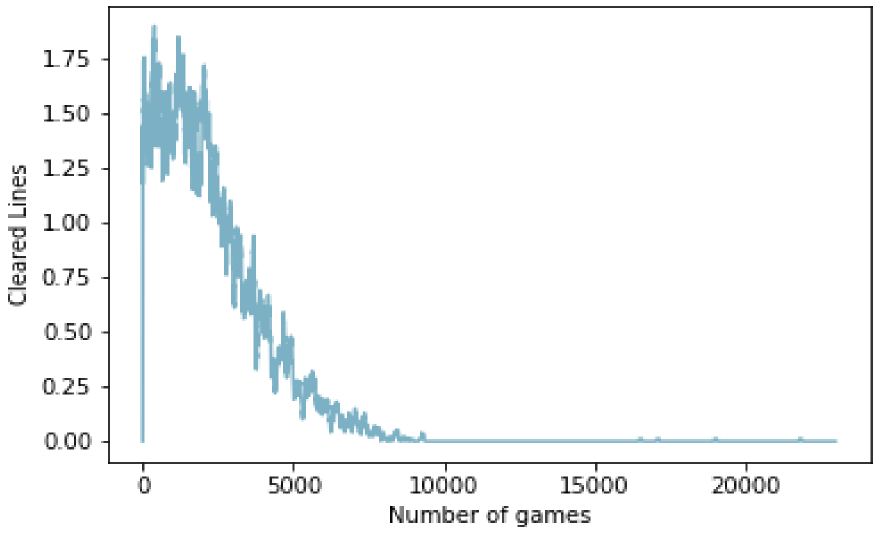

Figure 4.36: Cleared lines for experiment 4 version 1

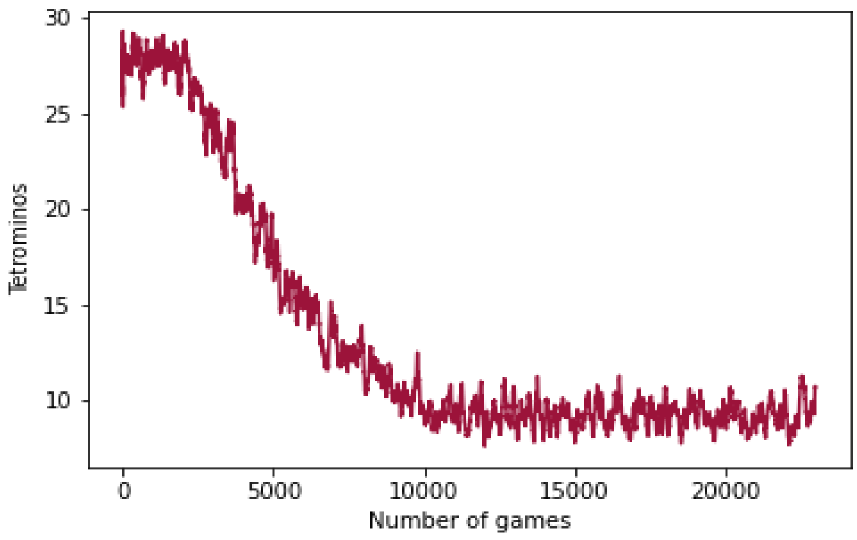

Figure 4.37: Tetrominoes for experiment 4 version 1

So we checked the actions frequency. The actions frequency, Figure 4.38, was quite distributed well during training. But after training and testing the model the NN still favours one actions no matter what the state is, Figure 4.39,

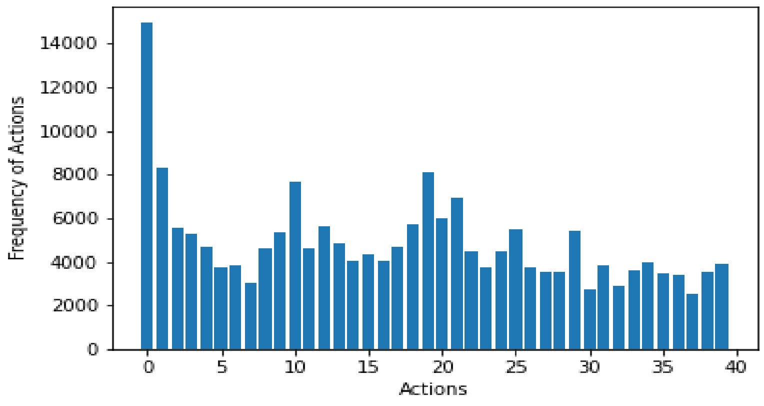

Figure 4.38: Action frequency during training for experiment 4 version 1.

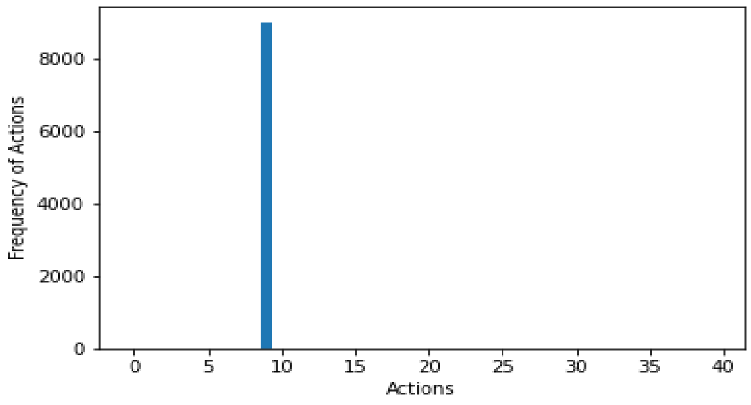

Figure 4.39: Action frequency during testing for experiment 4 version 1.

4.4.2 Version two

We removed the Boltzmann distribution for action selection.

- **Hyperparameters**
  - BATCH SIZE = 64
  - SYNC TARGET FRAMES = 5000
- **Neural Network Architecture** We thought that the network may be overfitting, so we kept the architecture as simple as possible and that is by using only two hidden layers but there are not BN nor Dropout layers.
  The final network architecture is
  - **Input Layer**: FC layer of size 220.
  - **Hidden Layer**: FC layer of size 110.
  - **Hidden Layer**: FC layer of size 70.
  - **Output Layer**: FC layer of size [Number of actions].
- **Results**
  After training Saffiya for over 5000 games, the results did not differ much from the previous. The Q-losses is increasing exponentially, Figure 4.40. On the other hand, the rewards are increasing but they are not saturating. The figures of clearing lines and dropping Tetrominoes, fig 4.42 and fig 4.49, shows that Saffiya does not improve by time.

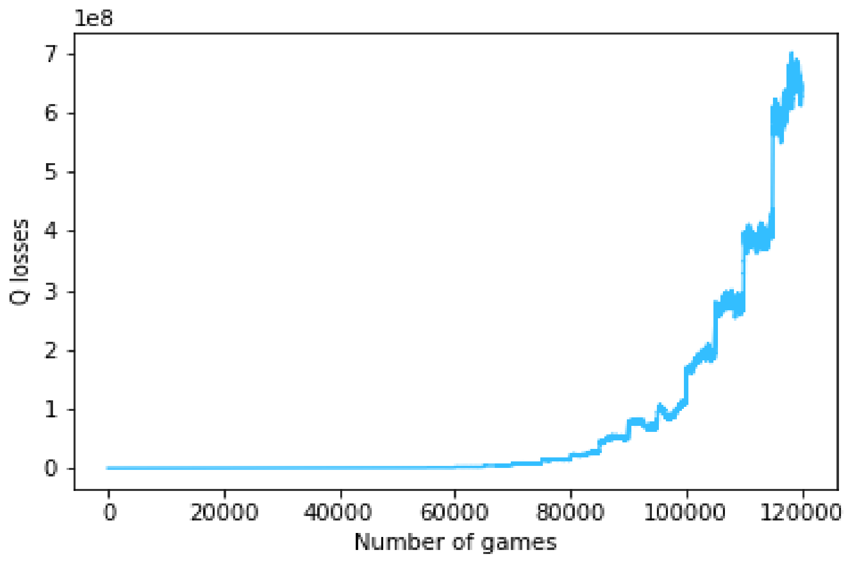

Figure 4.40: Q-losses for experiment 4 version 2

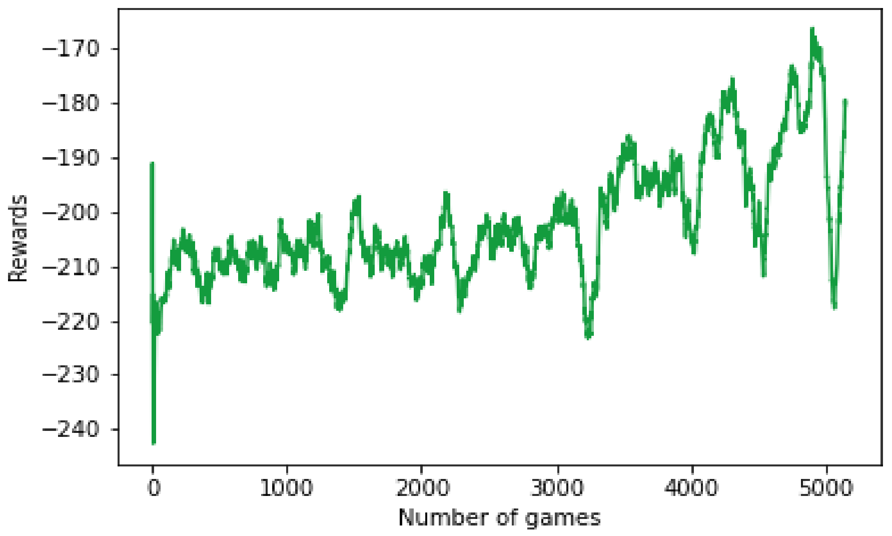

Figure 4.41: Rewards for experiment 4 version 2

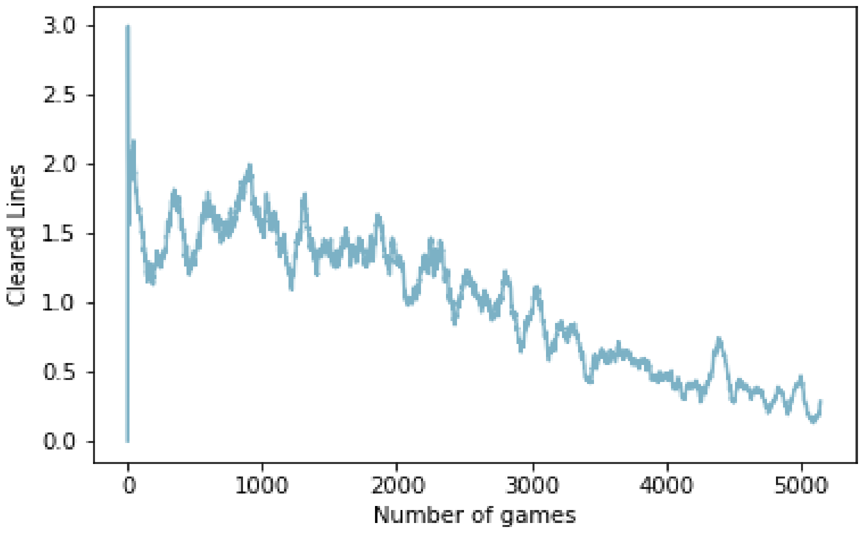

Figure 4.42: Cleared lines for experiment 4 version 2

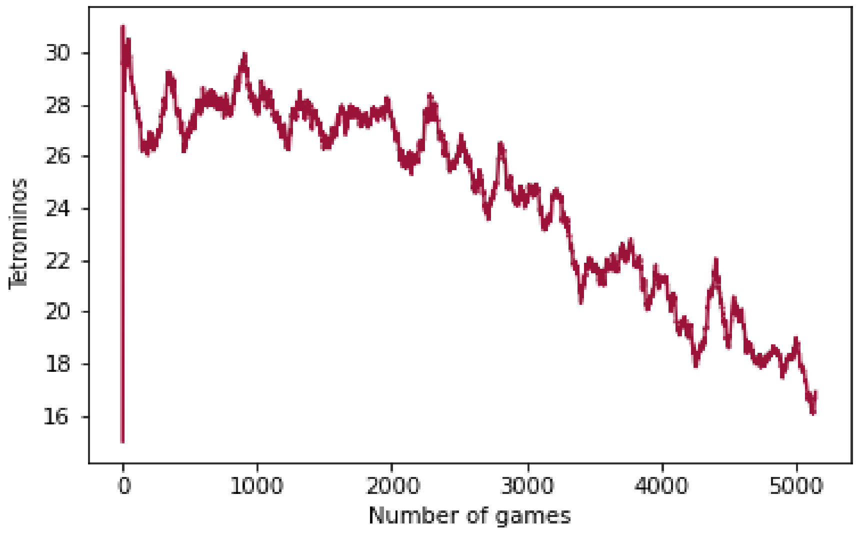

Figure 4.43: Tetrominoes for experiment 4 version 2

Although the network takes different actions during training, figure 4.44, and covers a quite wide range of actions, the network is still saturating, figure 4.44, after testing it with 1000 games.

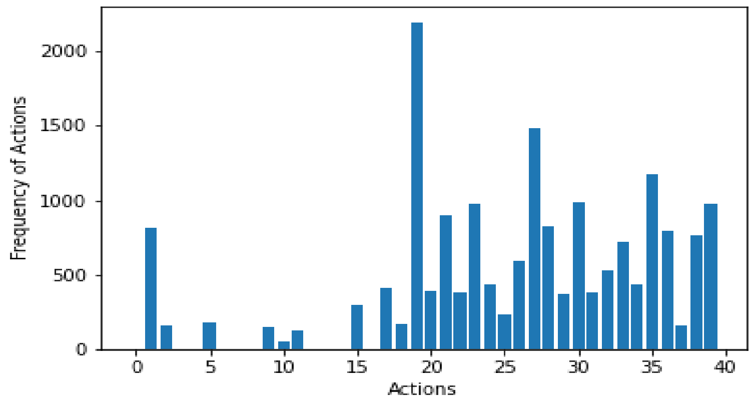

Figure 4.44: Action frequency during training for experiment 4 version 2.

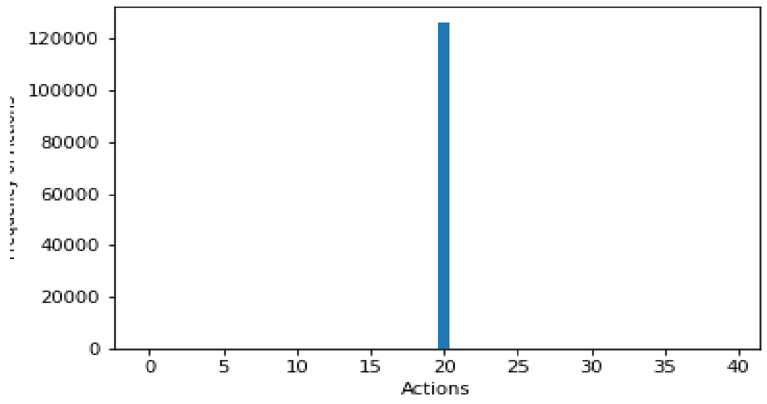

Figure 4.45: Action frequency during testing for experiment 4 version 2.

4.4.3 Version three

- **Hyperparameters**

  - LEARNING RATE = 1e-4

- **Neural Network Architecture**
  Since removing the regularization layers made the network worse. So we added back the BN and Dropout layers. We also adjusted the shape of network and the number of neurons per layer [22]. In addition, we used LeakyReLU as activation function instead of ReLU as according to [22], ReLU may suffer from suffer from the problem of dying ReLUs that during training some neurons die and stop output any value other than zero.

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{4.4}$$

  But due to the leak in LeakyReLU the neurons will always output values.

$$LeakyReLU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{4.5}$$

  So, the final network architecture would be

  - **Input Layer**: FC layer of size 47.
  - **Batch Normalization Layer**
  - **LeakyReLU**
  - **Dropout Layer**: with dropping rate of 20%
  - **Hidden Layer**: FC layer of size 71.
  - **Batch Normalization Layer**
  - **LeakyReLU**
  - **Dropout Layer**: with dropping rate of 20%
  - **Hidden Layer**: FC layer of size 40.
  - **Batch Normalization Layer**
  - **LeakyReLU**
  - **Dropout Layer**: with dropping rate of 20%
  - **Output Layer**: FC layer of size [Number of actions].

- **Results**
  This time we had trained Saffiya for about 116335 games. The Q-loss, figrue 4.46, is not so much high and the Q-losses saturate at specific value. Also, the rewards increased at first and then stopped increasing. Cleared Lines and Dropped Tetrominoes indicate how bad Saffiya is, as after long time of training, the performance worsens. And the action frequency, figure 4.50, confirm how bad network saturation is. The network only takes a few actions during training.

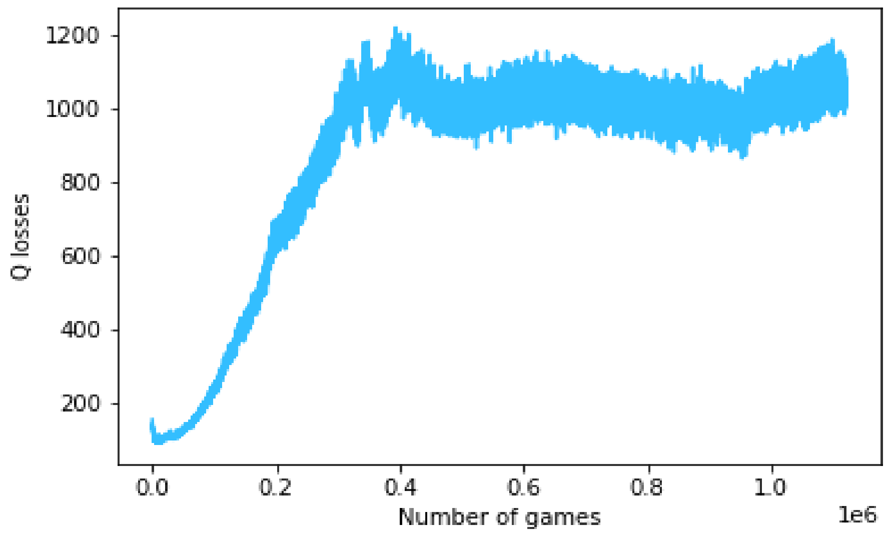

Figure 4.46: Q-losses for experiment 4 version 3.

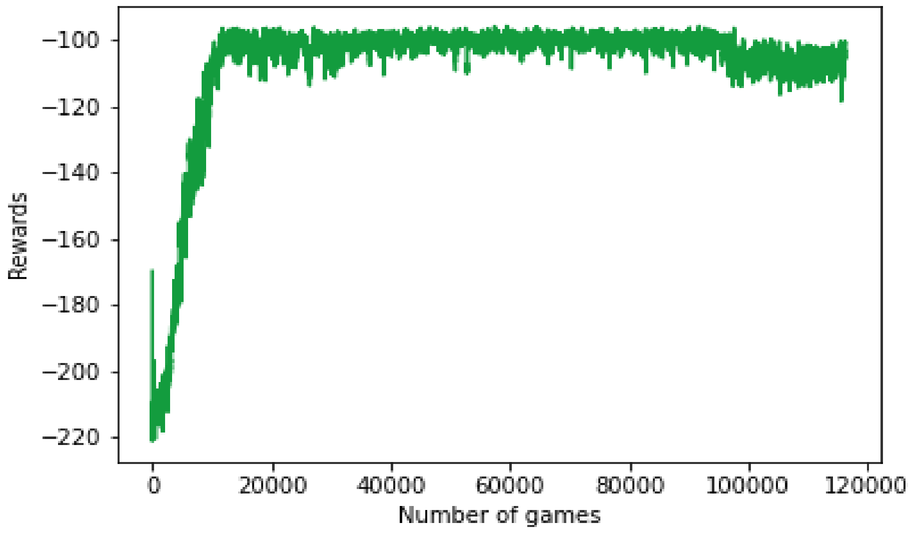

Figure 4.47: Rewards for experiment 4 version 3.

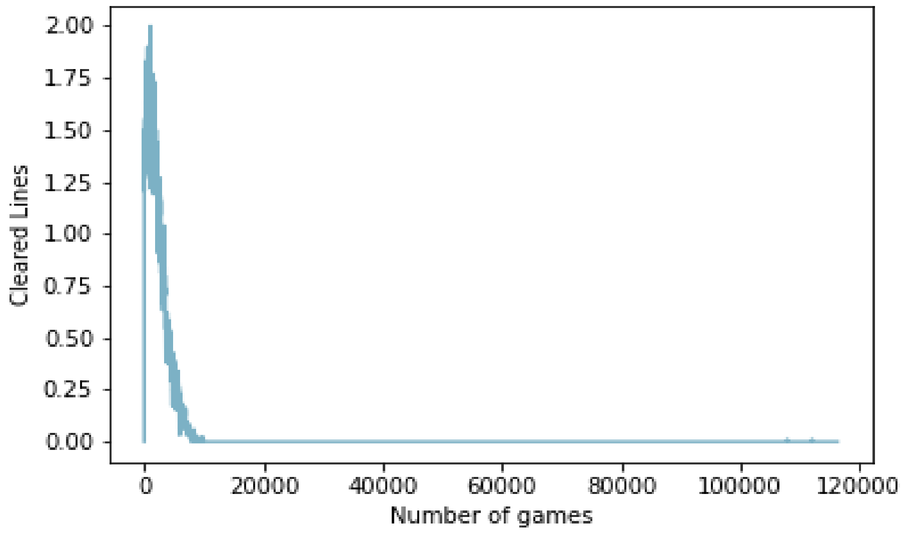

Figure 4.48: Cleared lines for experiment 4 version 3.

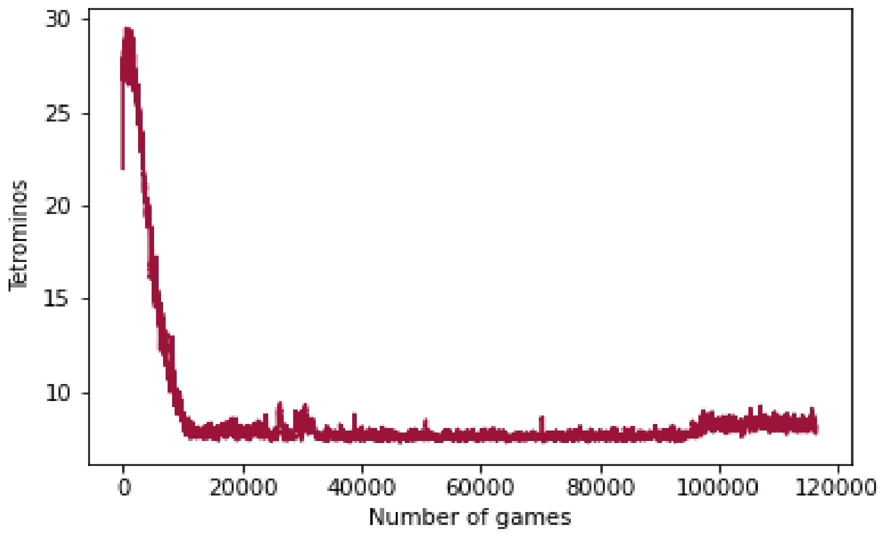

Figure 4.49: Tetrominoes for experiment 4 version 3.

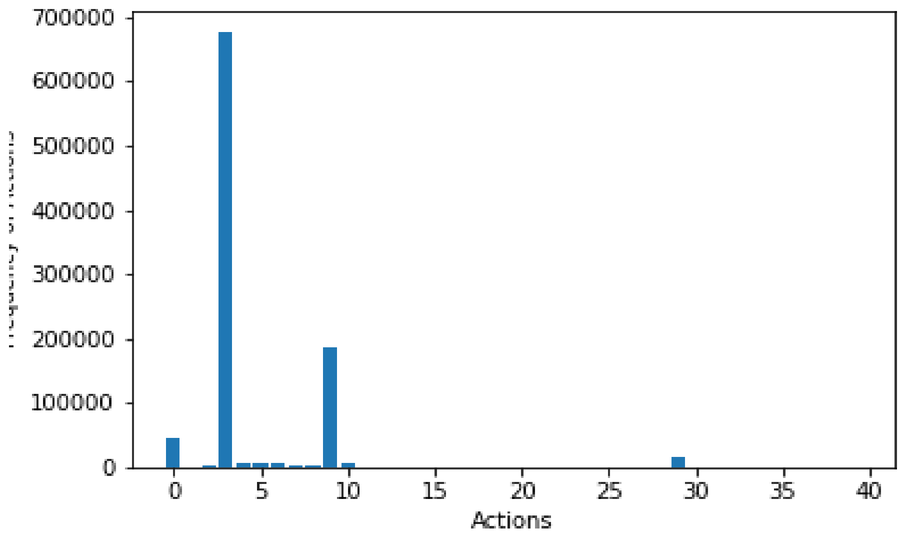

Figure 4.50: Action frequency during training for experiment 4 version 3.

4.4.4 Discussion

The normal ways of tuning hyperparameters or changing the network architecture did not change the behaviour of the NN. So we had to take substantial steps and major changes in our model to solve the saturation problem and increase Saffiya's performance in the next experiment.

4.5 Enhanced DQN[14]

In this experiment, we are adding enhancing DQN by adding DDQN, Dueling Networks and Noisy Networks techniques to the agent. We also exploited the concept of Several environments to speed up the process of learning. By using these techniques, the learning process has been faster and the agent has shown promising results. Also, the problem of action saturation seemed to be solved.

The aim of the experiment is to add major modifications to the NN and the agent to solve the problem of network saturation and improve the performance of Saffiya.

4.5.1 Version one

- **State space**
  We used Lundgaard controller features.
- **Reward function**
  Dellacherie controller but changed the factor multiplied to number of cleared lines to be 50.
- **Neural Network Architecture**
  In this experiment we merged the techniques of *Noisy Networks, Dueling Networks* and *DDQN*. As mentioned in section 3.3.2 the network will be divided into three parts **basic_sequential, adv_sequential** and **value_sequential**.
  We are still using Dropout layers and Batch Normalization layers and LeakyReLU as the activation function[15].
  - **Basic sequential**
    - **Input Layer:** NoisyLayer-47
    - **Hidden Layer:** NoisyLayer-71
  - **Advantage sequential**
    - **Hidden Layer:** NoisyLayer-87
    - **Output Layer:** NoisyLayer-[Number of Action]

[14]please review section 3.3.1 and section 3.3.2 before reading this experiment

[15]We won't write the Dropout, BN and activation functions in network architecture here as it is the same order as the previous experiments [linear -> BN -> activation_fn -> Dropout]

- **Value sequential**
  - **Hidden Layer:** NoisyLayer-87
  - **Output Layer:** NoisyLayer-1

- **Results**
  After training Saffiya for over 27000 games, the Q-losses graph shows increasing in its value ,figure 4.51. Regardless, there is increase in rewards that Saffiya gets, figure 4.52. But, the graphs of CLeared lines and dropped Tetrominoes, figure 4.53 and figure 4.54, seems odd as at first Saffiya is trained using heurisic agent so all the decisions taken are almost perfect but by time Saffiya depends on itself so the performance decline as training takes much longer time. The action saturation in network seems to be solved, figure 4.55, although the network only chose a few actions but it is normal as we are still in the beginning of training. At least it does not favour one action over the others.

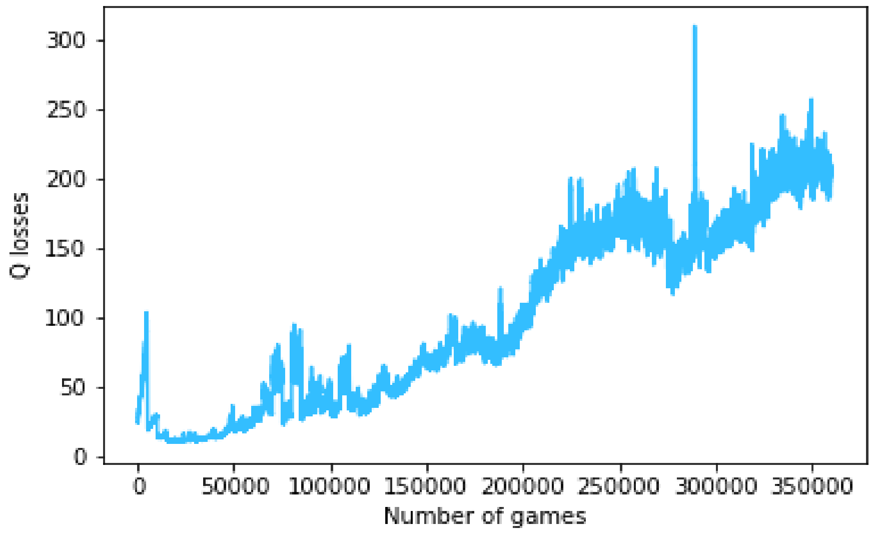

Figure 4.51: Q-losses for experiment 5 version 1

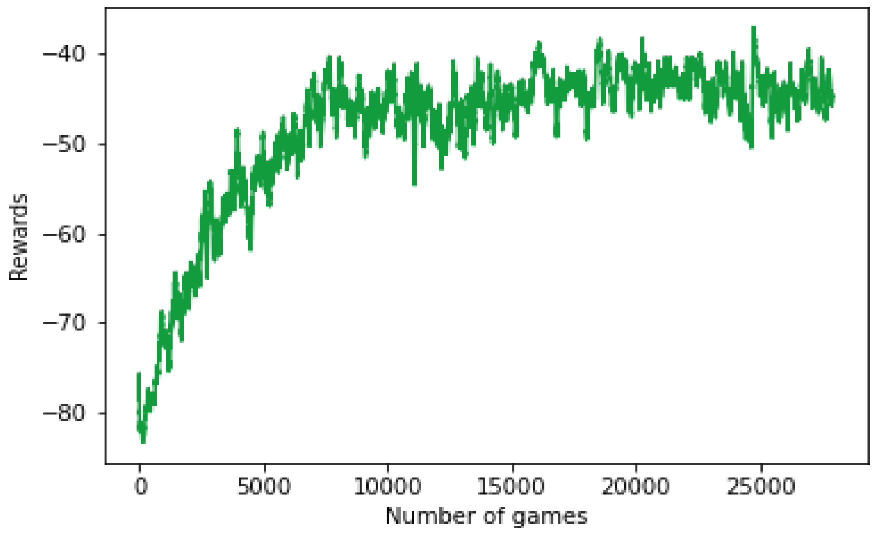

Figure 4.52: Rewards for experiment 5 version 1

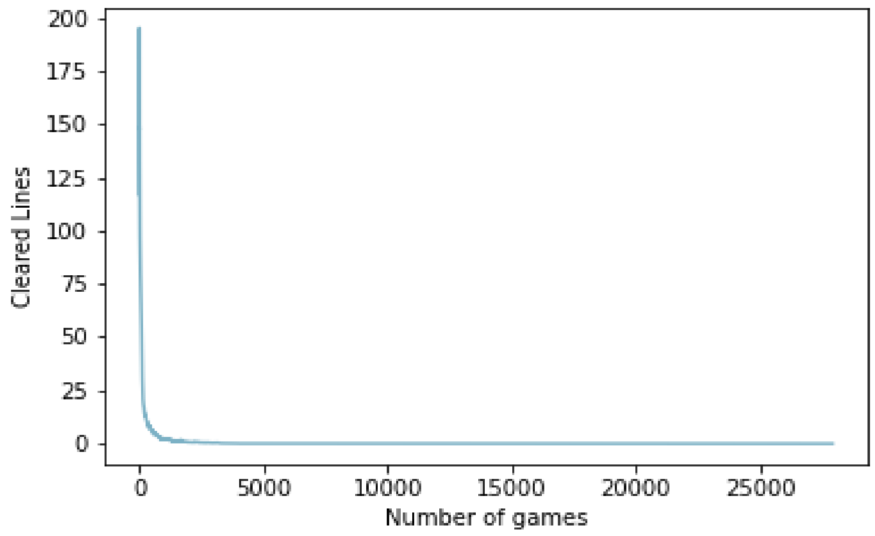

Figure 4.53: Cleared lines for experiment 5 version 1

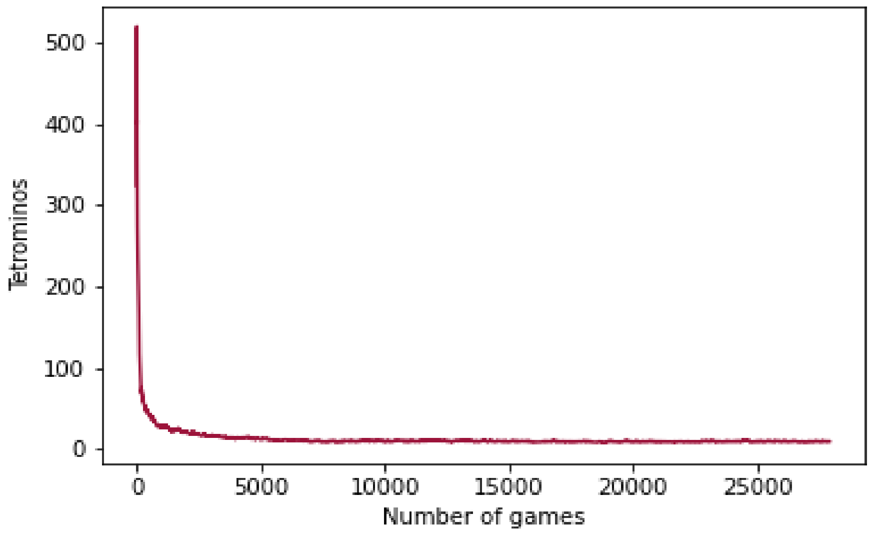

Figure 4.54: Tetrominoes for experiment 5 version 1

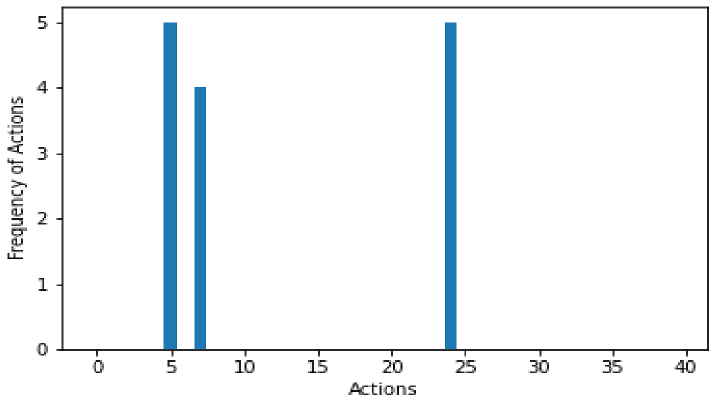

Figure 4.55: Action frequency during testing for experiment 5 version 1.

4.5.2 Version two

In this experiment we are using the technique of **Several environments**[16]

- **State space**
  We used Ashry controller featrues.
- **Reward function**
  We are still using Dellacherie, but we added the time term clipped at 5000 and then divided by 100. The reason we are clipping and then scale it because we are using a heuristic agent so at the beginning the game takes long time to end so the rewards would be very high. Also we changed the constant multiplied to the number of cleared line to be 100. The penalty for losing is set to be -100.
- **Heuristic Agent**
  We used El-tetris controller to be the heuristic agent.
- **Hyperparameters**
  - BATCH SIZE = 512
  - REPLAY SIZE = 20000
  - REPLAY START SIZE = 20000
  - LEARNING RATE = 1e-3
  - SYNC TARGET FRAMES = 5000
  - EPSILON DECAY LAST FRAME = 9e5
  - N ENVS = 2[17]
- **Loss function**
  HuberLoss function.
- **Neural Network Architecture**
  We changed the shape of the NN to have a square in its architecture[22][18]. Also, we added hidden layers.
  - **Basic sequential**

[16]see section 3.3.2
[17]This is for the technique of **Several environments**
[18]Chapter 10

    * **Input Layer:** NoisyLayer-150
    * **Hidden Layer:** NoisyLayer-100
    * **Hidden Layer:** NoisyLayer-100
  - **Advantage sequential**
    * **Hidden Layer:** NoisyLayer-70
    * **Output Layer:** NoisyLayer-[Number of Action]
  - **Value sequential**
    * **Hidden Layer:** NoisyLayer-50
    * **Output Layer:** NoisyLayer-1

- **Result**
  Saffiya has been trained about 19247 games. Due to tachnical problems the training has been disconnected many times so we will show the last batch of training which are 2543 games. The Q-losses, figure 4.56, seems to converge to a value near to the zero. Although there is a drop in the rewards curve, figure 4.57, but it has increase indication. The cleared lines and dropped Tetrominoes, figure 4.58 and figure 4.59, indicate a promising performance that they are in increase. Although there were promising results but I had to abort the execution due to limited resources and bad internet connection. Also, there was other ideas to execute.

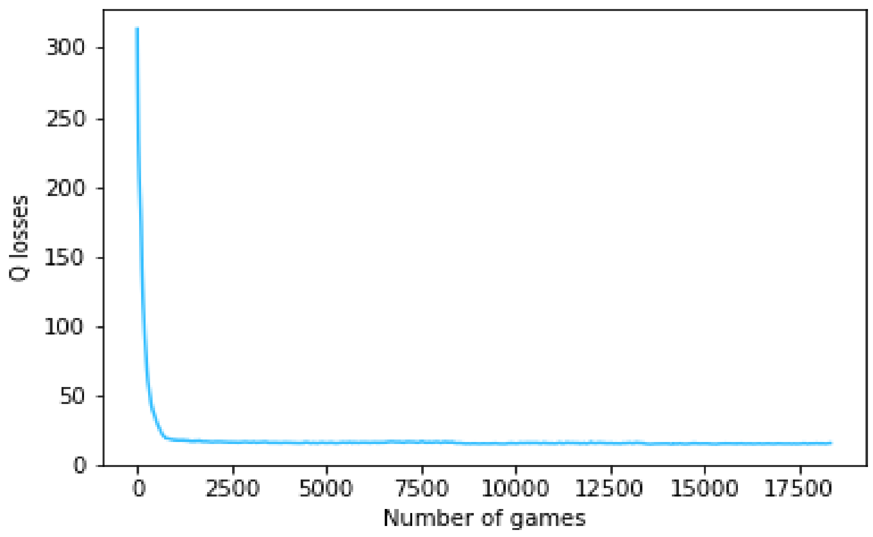

Figure 4.56: Q-losses for experiment 5 version 2

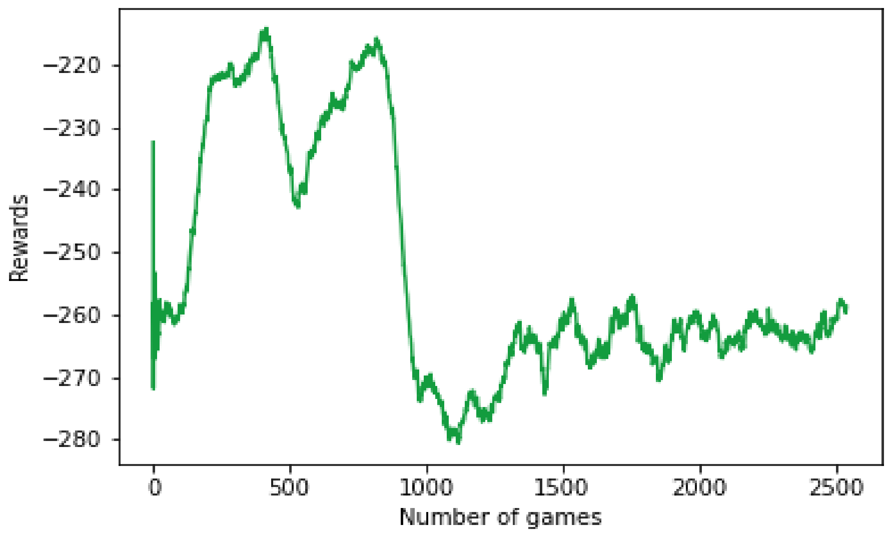

Figure 4.57: Rewards for experiment 5 version 2

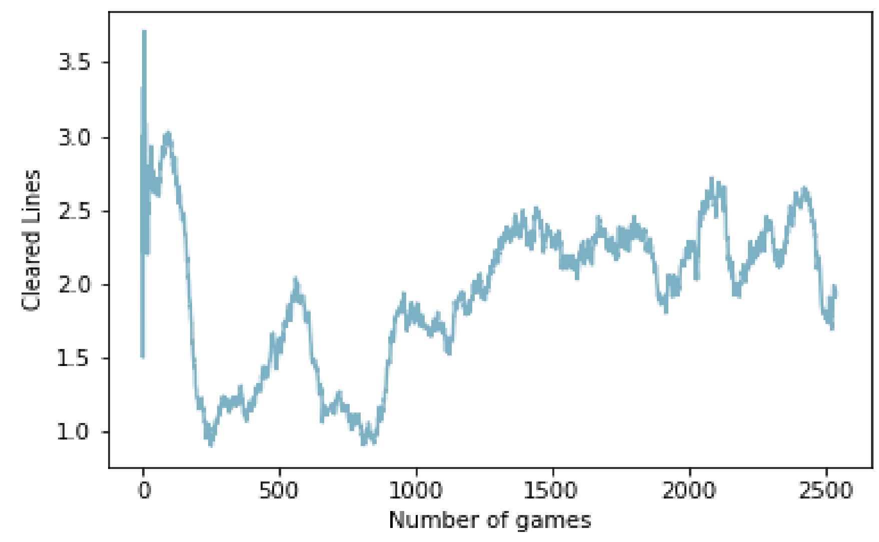

Figure 4.58: Cleared lines for experiment 5 version 2

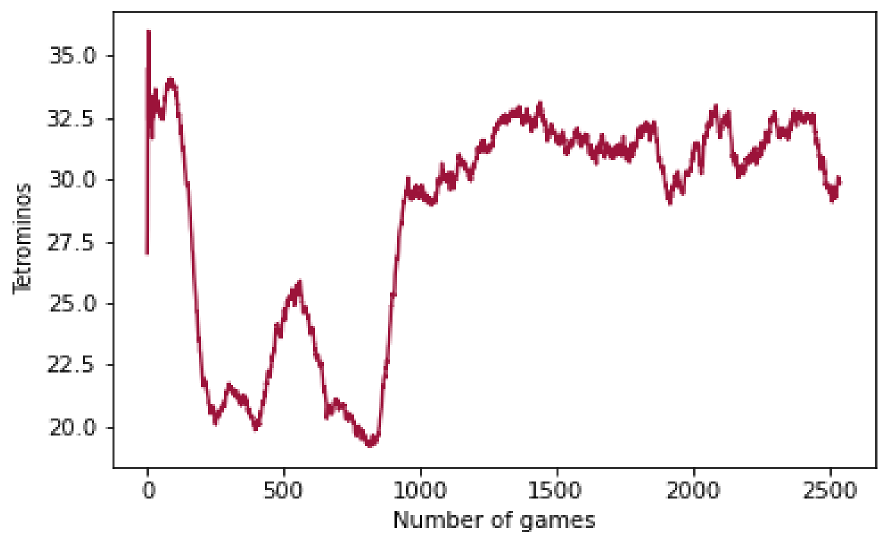

Figure 4.59: Tetrominoes for experiment 5 version 2

4.5.3 Version three

- **State space**
  We used Melax controller with melax value = 2, which is simply TTL representation of the board.
- **Reward function**
  Dividing the time term over 100 instead of 50. Also, we have changed the factors multiplied to the Dellacherie controller, that the factor multiplied to the cleared lines is 500 and all other factors are divided by 2.
- **Heuristic agent**
  We removed the heuristic agent from the model.
- **Results**
  Due to technical problems, Saffiya was trained through separated session. In all, the number of games is 48152 games. But, we will show the results of the last training session which is about 38812 games.
  As shown in figure 4.60, the Q-losses nearly converged. The rewards are in increase, figure 4.61, but there is a plateau in the curve. The max number of cleared lines is 0.1 lines per game which is too low, figure 4.62. Also the number of dropped Tetrominoes was about 24 Tetrominoes per game, figure 4.63.

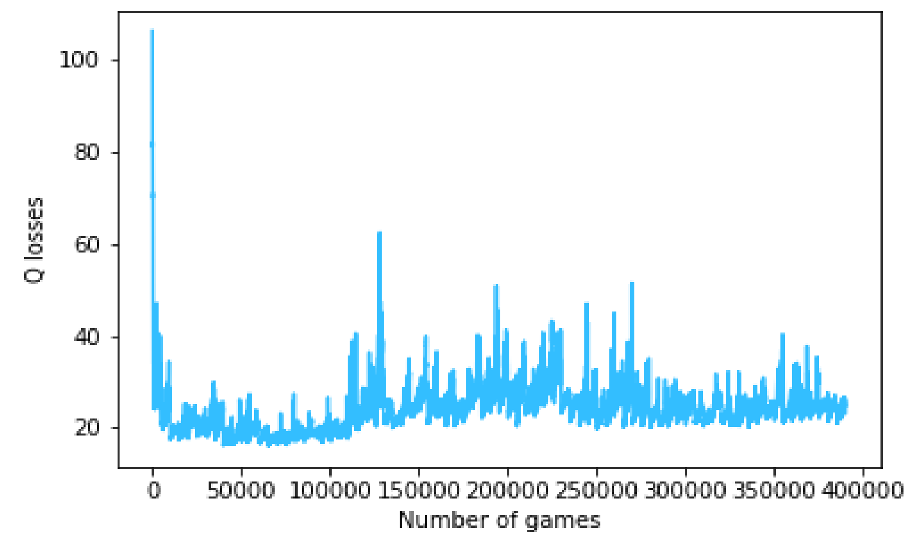

Figure 4.60: Q-losses for experiment 5 version 3

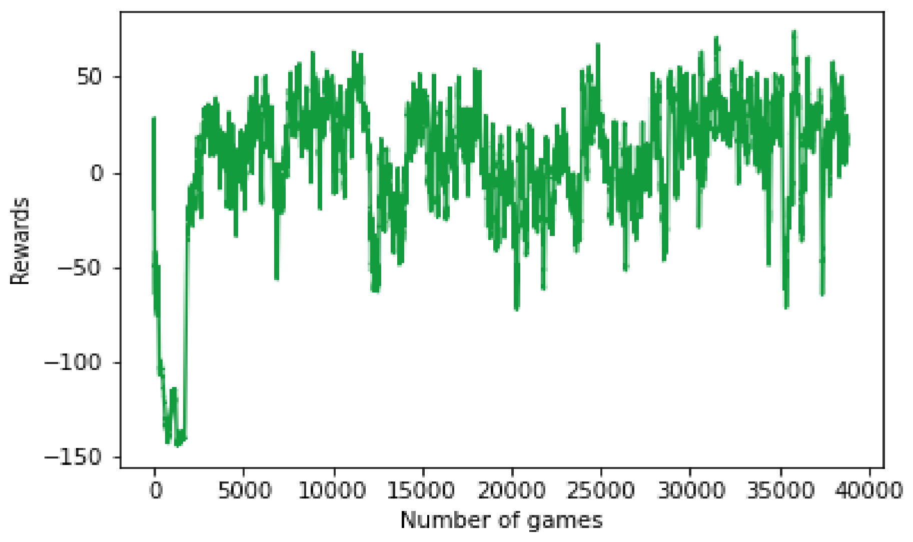

Figure 4.61: Rewards for experiment 5 version 3

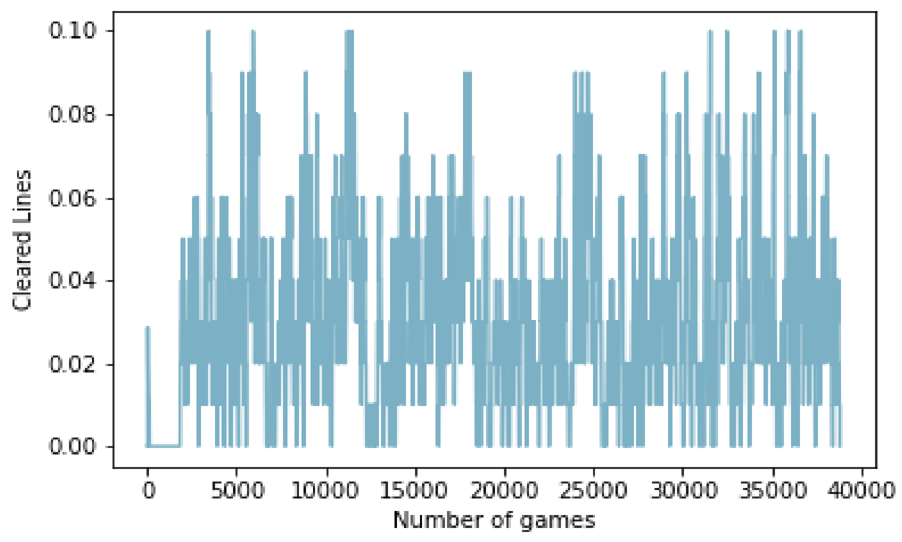

Figure 4.62: Cleared lines for experiment 5 version 3

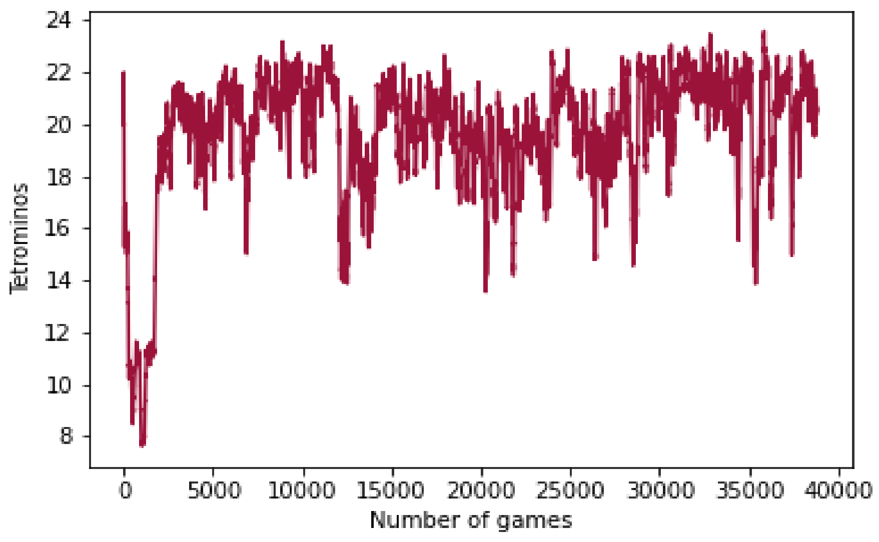

Figure 4.63: Tetrominoes for experiment 5 version 3

4.5.4 Version four[19]

- **State space**
  Contour controller.
- **Hyperparameters**
  - LEARNING RATE =1e-5
- **Results**
  After training Saffiya for 95295 games, there is a promising results. First, All of these training did not take much longer time and that is because we did not use Heuristic agent which uses heavy computations on the CPU. The Q-losses started very high but then collapsed to a very small value, we were printing the losses and it ranged from 30 to 20, also the number of cleared lines was 0.5 line per game at its peek, figure 4.66, which is quite impressive[20], because it reaches these results without guidance from a heuristic agent. But, there is still a plateau in the curves of rewards and Tetrominoes, figure 4.65 and figure 4.67.

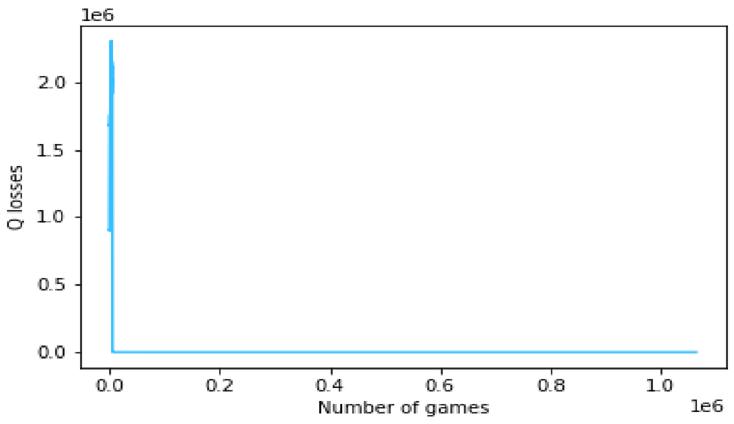

Figure 4.64: Q-losses for experiment 5 version 4

[19] All of previous experiment the mode of target network was 'train' mode, but from this experiment we changed it to be in 'eval' mode

[20] At least for me

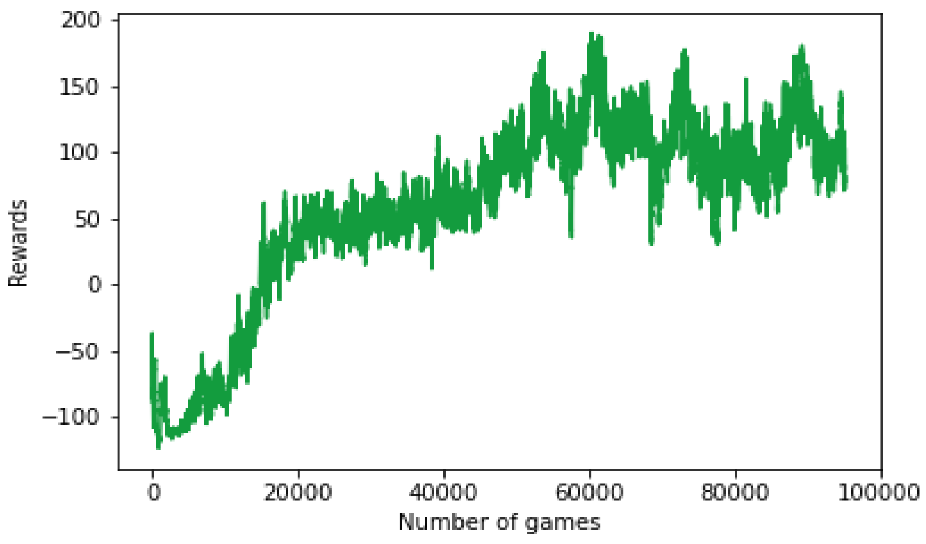

Figure 4.65: Rewards for experiment 5 version 4

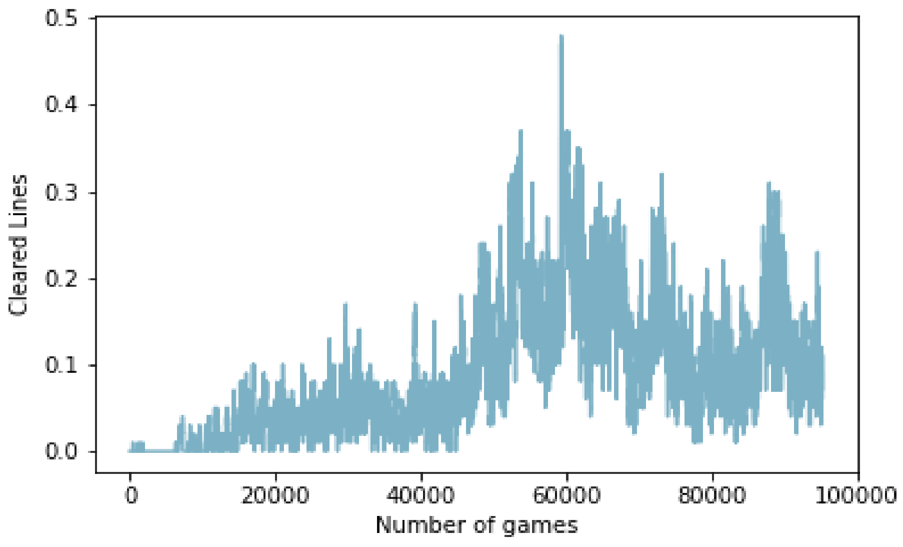

Figure 4.66: Cleared lines for experiment 5 version 4

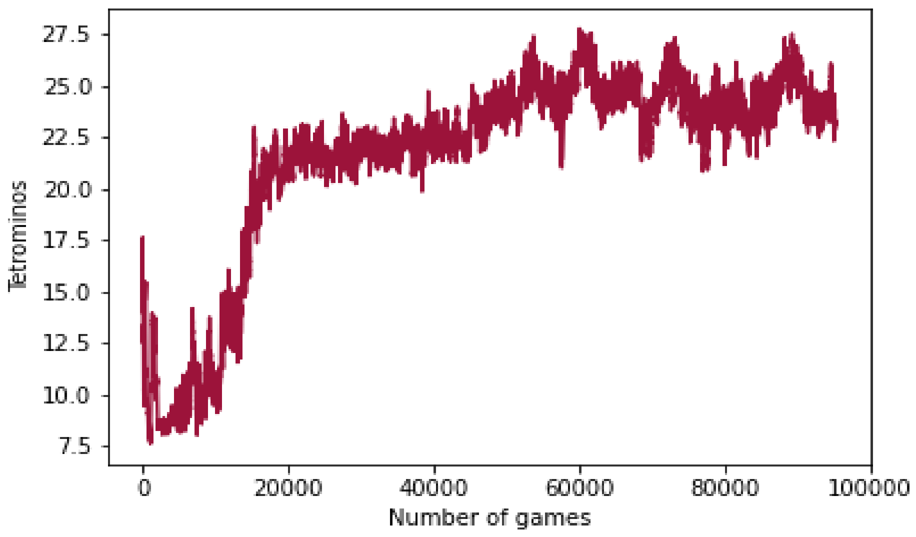

Figure 4.67: Tetrominoes for experiment 5 version 4

4.5.5 Version five

- **State space**
  We used again Melax controller, but this time Melax factor is 4 since any Tetriminoes is consisted of 4 blocks.
- **Results**
  After training Saffiya for about 124925 games, the performance of Saffiya outperformed the last version. As, the number of cleared lines reached 1.2 lines at its peek, figure 4.71, and although there was a drop but there was an indication of increase. Also, the number of Tetriminoes improved a little bit and reached over 30 pieces per game, figure 4.71.

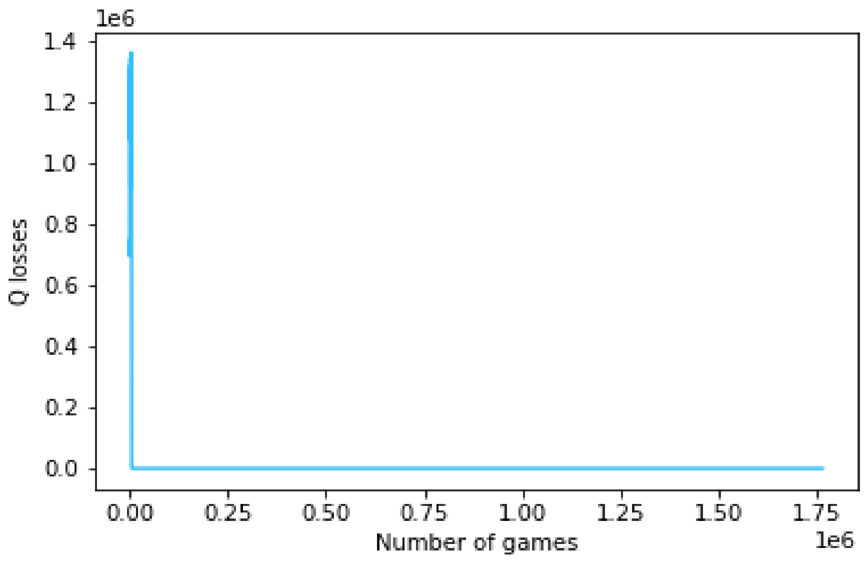

Figure 4.68: Q-losses for experiment 5 version 5

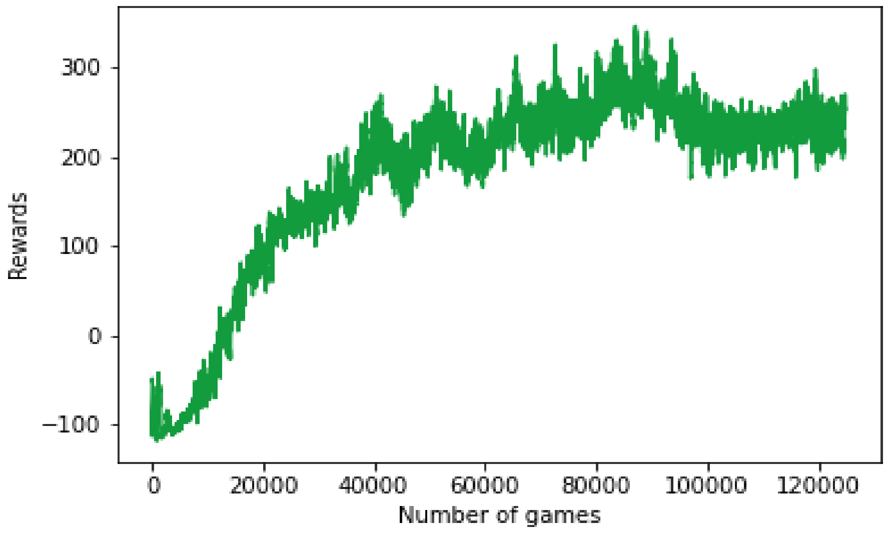

Figure 4.69: Rewards for experiment 5 version 5

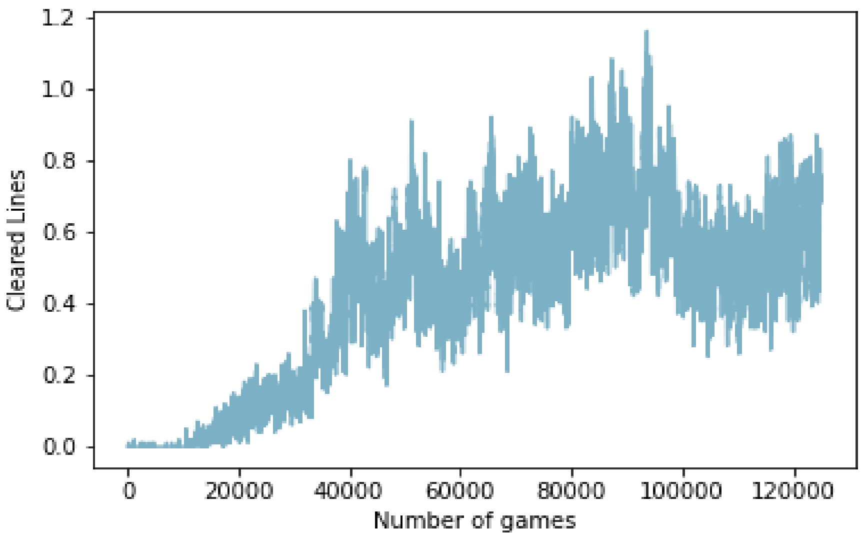

Figure 4.70: Cleared lines for experiment 5 version 5

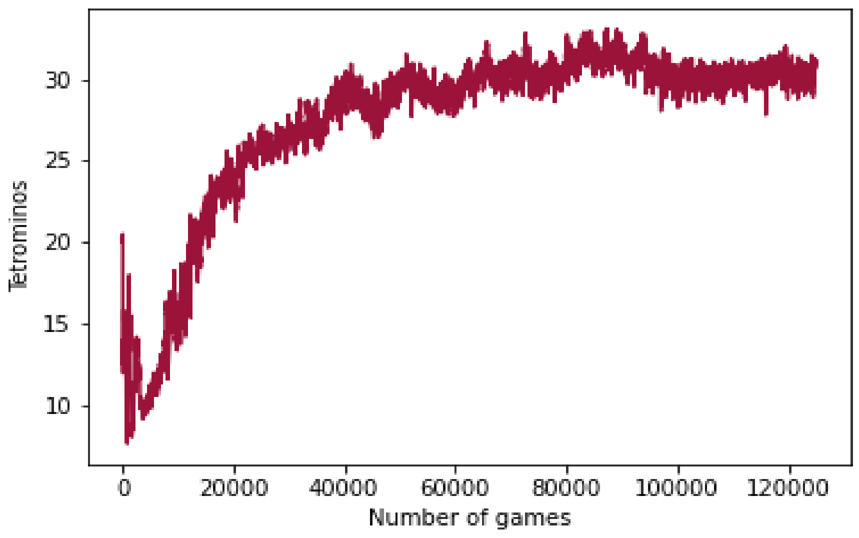

Figure 4.71: Tetrominoes for experiment 5 version 5

4.5.6 Discussion

In this experiment we had major changes to the model, there was good experiment like version 2 in which Saffiya made use of the Heuristic agent. Also, the results of version 4 and 5 were very good and they were in their ways to improve, but we had to abort them due to lack of resources and badness of the internet connection.

5 Conclusion

In this chapter, we will summarize the work that has been done throughout the project. Also, we will present the limitation that we have faced during the implementation of the project and we will propose ideas based on our experience for the future work that might help in approximating a solution for our NP problem which is Tetris.

5.1 Work Summary

In this thesis, we performed many experiments to train an RL agent to play the Tetris game using DQN algorithm. We have tried different NN architectures and different high-level states of the board to approximate the board information. We experiment the use of many high-level states and some of them together to see the effect. Also, we started with the vanilla implementation of DQN then developed it throughout the project by adding enhancements to it. For example, DDQN and Noisy Networks.
We also experiment different shapes of the NN, such as square and pyramid shapes of te NN and combined them together. We have used multiple of reward and loss functions to see their effects to the model. In addition, we implemented a Tetris environment with gym interface supported with many fixed controllers and grouped action feature for future use.

The experiments results showed huge effect from experiment to another. This was obvious from experiment three where we used heuristic agents to help the vanilla DQN agent learn. In experiment four, we encountered the problem of action saturation which we tried to overcome it by changing the NN architecture and tuning hyperparameters. Then, In experiment five we added some major modifications to the NN and the agent and its effect appeared directly. That the agent was able to clear on average in some experiments around 3-4 lines per game. Some of the high-level state showed some promising results, such as Melax and contour representation.

5.2 Limitations

There were many obstacles encountered during the implementation. One of them was the lockdown due to Corona virus which deprived me from interacting normally with my supervisor and make the of the Ulm university resources.
Also, the hardware limitation was a main issue as the only GPU I could use was the Colab GPU and it was for a limited time so the training took longer time.
In addition to that, There was some of software problems as I am using Ubuntu as my operating system and when I tried to install Nvidia drivers on it, the computer did not start and it took me a while to resolve this issue.

5.3 Future work

To extend our work, We need to use more powerful hardware for GPU training. Also, Combining different High-level state and engineer a better reward function for the agent.
Furthermore, It would useful to add more enhancements to the DQN implementation to improve the agent performance. Also, it would be interesting and promising to use model-free and model-based RL algorithms[1][1] combined together rather than DQN which is model-free RL algorithms.

[1] Chapter 22

Bibliography

[1] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. 8. Packt Publishing Ltd, 2020. ISBN: 9781838826994.

[2] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[3] Simon Algorta and Ozgur Simsek. “The Game of Tetris in Machine Learning”. In: (2019). DOI: https://arxiv.org/abs/1905.01652.

[4] Colin Fahey. *Tetris*. URL: https://www.colinfahey.com/tetris/.

[5] Amine Boumaza. “How to design good Tetris players”. In: (2013). DOI: https://hal.inria.fr/hal-00926213.

[6] Christophe Thiery and Bruno Scherrer. “Improvements on Learning Tetris with Cross Entropy”. In: *ICGA* (2009).

[7] Yiyuan Lee. *The near perfect bot*. URL: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/.

[8] Islam El-Ashi. *El-Tetris – An Improvement on Pierre Dellacherie’s Algorithm*. URL: https://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/.

[9] Matt Stevens and Sabeek Pradhan. “Playing Tetris with Deep Reinforcement Learning”. In: (2016). DOI: http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf.

[10] Nicholas Lundgaard and Brian J. Mckee. “Reinforcement Learning and Neural Networks for Tetris”. In: 2007.

[11] Yael Bdolah Dror Livnat. *Reinforcement Learning Playing Tetris*. URL: https://www.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html.

[12] Donald Carr. “Applying reinforcement learning to Tetris”. In: 2005.

[13] Patrick Thiam, Viktor Kessler, and Friedhelm Schwenker. “A Reinforcement Learning Algorithm to Train a Tetris Playing Agent”. In: *Artificial Neural Networks in Pattern Recognition*. Ed. by Neamat El Gayar, Friedhelm Schwenker, and Cheng Suen. Cham: Springer International Publishing, 2014, pp. 165–170. ISBN: 978-3-319-11656-3.

[14] Alexander Groß, Jan Friedland, and Friedhelm Schwenker. “Learning to play Tetris applying reinforcement learning methods”. In: Jan. 2008, pp. 131–136.

[15] jaybutera. *Tetris Engine*. URL: `https://github.com/jaybutera/tetrisRL/blob/master/engine.py`.

[16] Mohamed Ashry. *Tetris Engine*. URL: `https://github.com/mohamed-ashry7/tetris-gym-ai`.

[17] Bdolah and Livnat. *Contour*. URL: `https://www.tau.ac.il/~mansour/rl-course/student_proj/livnat/SmallContoursGray.gif`.

[18] Jacob Schrum. *Tetris Network*. URL: `https://people.southwestern.edu/~schrum2/TetrisNetwork.png`.

[19] Sam McCandlish et al. *An Empirical Model of Large-Batch Training*. 2018. arXiv: `1812.06162 [cs.LG]`.

[20] Guangyong Chen et al. *Rethinking the Usage of Batch Normalization and Dropout in the Training of Deep Neural Networks*. 2019. arXiv: `1905.05928 [cs.LG]`.

[21] hh32. URL: `https://datascience.stackexchange.com/questions/61262/agent-always-takes-a-same-action-in-dqn-reinforcement-learning`.

[22] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O’Reilly Media, Inc., 2017. ISBN: 1491962291.