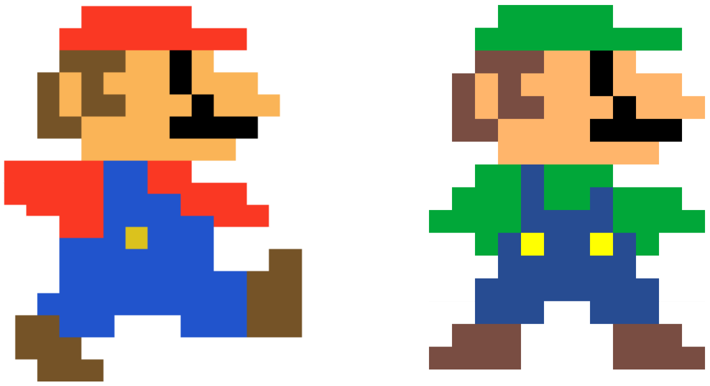
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.021 Artificial Intelligence

Final Report



IT IS ME, MARIO!

Prepared by:
Ong Xiang Qian (1002646)
Mong Chang Hsi (1003085)
Chok Hao Ze (1003071)
Ian Lim Li Ern (1002678)

2021

Table of Content

1 INTRODUCTION **2**
1.1 Task Description 2
1.2 Definitions 2
1.3 Dataset 3
2 INSTRUCTIONS TO RUN THE CODE **4**
2.1 Source Code 4
2.2 Setting Up the Environment 4
2.2.1 Prerequisites 4
2.3 Running the application 5
2.3.1 Training the Model 5
2.3.2 Evaluating the Model 5
2.3.3 GUI 5
3 EXPLORATORY DATA ANALYSIS **6**
3.1 Exploration 6
3.2 Preprocessing 7
3.2.1 Gray Scale Operation 7
3.2.2 Normalization 7
3.2.3 SkipFrames 7
3.2.4 FrameStack 7
4 MODEL **8**
4.1.1 Deep Q-Network (State-of-the-Art) 8
4.1.2 Double Q-learning 10
4.1.3 Double Deep Q Network (DDQN) 11
4.1.4 TD Estimate and TD Target 11
4.1.5 Updating the Model 12
4.1.6 Model architecture 12
4.2 Experiments 13
4.2.1 Parameter 1: Memory 15
4.2.2 Parameter 2: Exploration Decay Rate 18
4.2.3 Parameter 3: Syncing Steps 21
4.2.4 Parameter 4: Learning Steps 24
5 RESULTS & EVALUATION **26**
6 CONCLUSION **27**
Bibliography **27**

1 INTRODUCTION

1.1 Task Description

Super Mario Bros is an arcade game that was released in 1985 by Nintendo and was a hit with gamers worldwide. The aim of the game is to control the character, Mario, and save Princess Toadstool from the antagonist, Browser.

This involves controlling Mario to perform simple actions such as moving left, right and jumping, over a series of stages. Along the way, there are adversarial enemies such as walking mushrooms (Goomba) which will end the game if Mario gets in contact with them. The only way to destroy them is by getting Mario to jump directly on top of them. The further the distance covered by Mario, the closer the goal the player will reach. On top of that, there are coins that can be collected by the player which will increase the final score when the stage is cleared.

For this project, the task is to train a reinforced model that is capable of controlling Mario to reach as far as possible in the stage.

1.2 Definitions

Environment: The agent's world in which it lives in and interacts with. The agent is able to interact with the environment by performing actions but cannot alter the environment's rules and dynamics. Our super mario environment is made up of mushrooms, tubes, and other components.

Action α: The way an agent interacts with and responds to the environment. The *action-space* is the set of all possible actions the agent can perform. Our super mario agent has the predefined actions of jumping on the spot, jumping rightwards and moving rightwards.

State s: Describes the current situation of the agent and its environment. The *state-space* is a set of all possible permutations of the agent's and environment's situations. In our super mario game, the state-space includes all possible situations of mario and its environment at any point in time. This will be presented in the form pixels with RGB channels.

Reward r: Reward functions are feedback from the Environment to the Agent to teach it how to behave in a particular state. It helps the Agent learn by optimizing its future action. Return is a cumulation of rewards over multiple time steps. An agent's goal is to maximize its cumulative reward. In our super Mario game, Mario aims to maximize the collected coins and distance travelled in a particular stage.

Optimal Action-Value function Q*(s, α): Gives the expected return if you start in state s, take an arbitrary action α, and then for each future time step take the action that maximizes returns. *Q* can be said to stand for the "quality" of the action in a state.

1.3 Dataset

Given the nature of reinforcement training, the dataset is produced simultaneously during the training period itself. For this task, the dataset will be provided in the form of images taken from the game live. These images describe the current situation that is happening within the game, which will be seen by the players before they make another move.

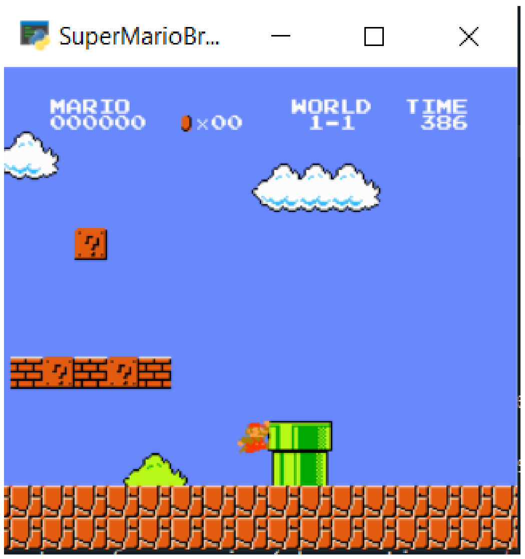


Figure 1. Example of Mario Image on Emulator

Input:

- **Frame(s) of situation**

Output:

- **Next Action (e.g. Move left, Move Right, Jump)**

For this task, we will be using OpenAI's GYM library and Mario Library. This library creates wrappers for games such as Super Mario Bros which can be used to perform an action on the environment, extract the scores and obtain relevant information (e.g. ending of the game) that is required for training the reinforced model. The Mario library provides an NES emulator that allows the game to be loaded using Python.

2 INSTRUCTIONS TO RUN THE CODE

2.1 Source Code

The source code for this project can be found in the GitHub Repository: https://github.com/ianlimle/ItsMeMario

The link for the trained weights can be found in Google Drive: https://drive.google.com/drive/folders/11e0kPqND14o1LITcmo-3-iLtkOPJUxt6?usp=sharing

2.2 Setting Up the Environment

2.2.1 Prerequisites

1. Install conda
2. Install dependencies with ***environment.yml***

```
$ conda env create -f environment.yml
```

Check the new environment ***mario-env*** is created successfully.

3. Activate ***mario-env*** environment

```
$ conda activate mario-env
```

4. If your shell is not properly configured to use ***conda activate***, it may be a source line in your bash scripts that has to explicitly reference your conda installation path. You can reference your conda installation path with the following:

```
$ CONDA_PREFIX=$(conda info --base)
$ source $CONDA_PREFIX/etc/profile.d/conda.sh
$ conda activate mario-env
```

2.3 Running the application

2.3.1 Training the Model

To start the training process for the Mario agent,

```
$ python train.py
```

This starts the learning and logs key training metrics to ***checkpoints*** . A copy of ***MarioNet*** will be saved.

2.3.2 Evaluating the Model

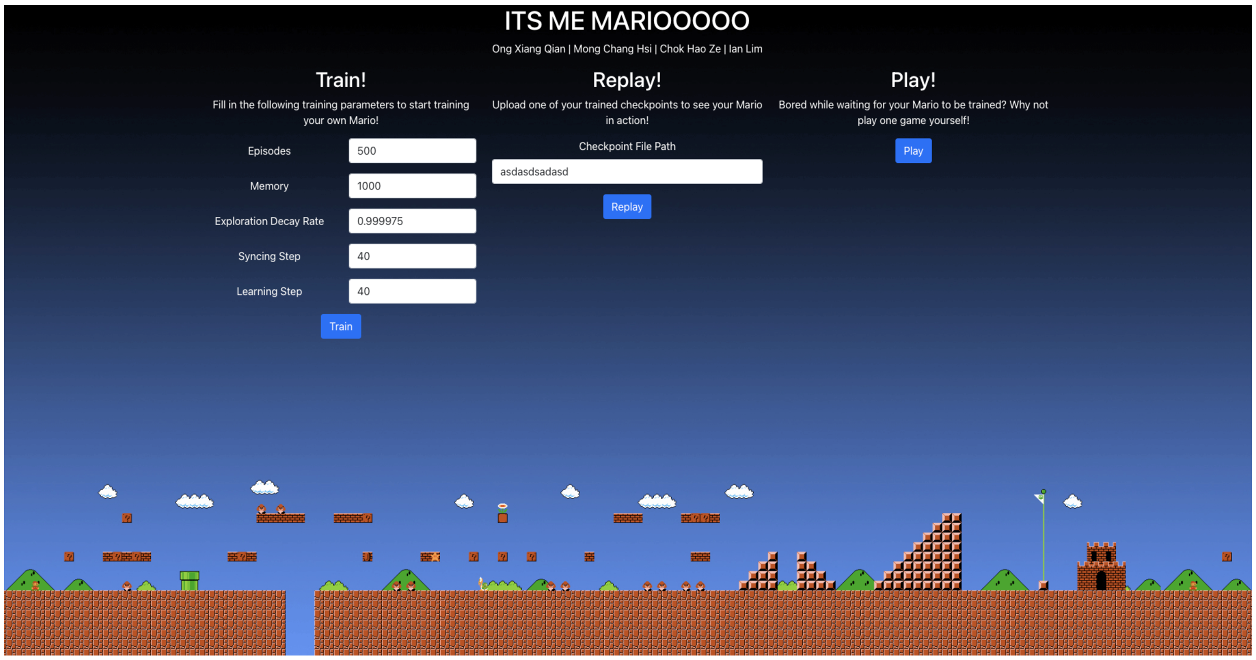
To evaluate a trained Mario agent,

```
$ python evaluate.py
```

This renders the game environment and visualizes the Mario agent playing the game in a window. Performance metrics will be logged to a new folder under ***checkpoints***. Change the ***load_dir***, e.g. ***checkpoints/2021-08-06T22-00-00***, in ***Mario.load()*** to check a specific timestamp.

2.3.3 GUI

Alternatively, the entire service can be run on a GUI web dashboard that is developed in ReactJS and compiled into HTML, CSS and Javascript.



ITS ME MARIOOOOO

Ong Xiang Qian | Mong Chang Hsi | Chok Hao Ze | Ian Lim

Train!

Fill in the following training parameters to start training your own Mario!

Episodes: 500

Memory: 1000

Exploration Decay Rate: 0.999975

Syncing Step: 40

Learning Step: 40

Replay!

Upload one of your trained checkpoints to see your Mario in action!

Checkpoint File Path: asdasdsadasd

Play!

Bored while waiting for your Mario to be trained? Why not play one game yourself!

Figure 2. GUI to train, test and play

The GUI enables the user to be able to train the model and tune the hyperparameters manually. The hyperparameter values will be fed to the model and a subprocess call will be triggered to run the training.

On top of that, users can also upload the model checkpoint to test the model and watch the trained model progress in the game.

Lastly, the GUI allows the user to be able to play the Mario game itself without any model involved. This provides the enjoyment and a sanity check to see if the model is any better than the user.

To run the GUI service, simply follow the steps below. You would also need NPM to be installed on your machine.

```
# move into the application directory

$ npm install
$ npm run start
```

3 EXPLORATORY DATA ANALYSIS

3.1 Exploration

The data provided by the emulator comes in the form of RGB images of 240 x 256 pixels. This state is provided upon every action rendered from the environment, along with the score and distance travelled from the previous move. Since images are involved, Convolutional Neural Networks will be used in the process.

3.2 Preprocessing

3.2.1 Gray Scale Operation

Each state that the environment can be in is represented by a (3, 240, 256) size array. However, this is often more information that the agent requires as Mario's actions do not depend on the colour of the pipes or the sky. Grayscaling images for training is a standard operation as it reduces the number of channels from three (RGB) to one. This reduces the size of the state representation without losing useful information, allowing the model to focus on extracting the relevant features, thus reducing unnecessary computation. The size of the original image is (3, 240, 256), after grayscaling, the size of each state becomes (1, 240, 256).

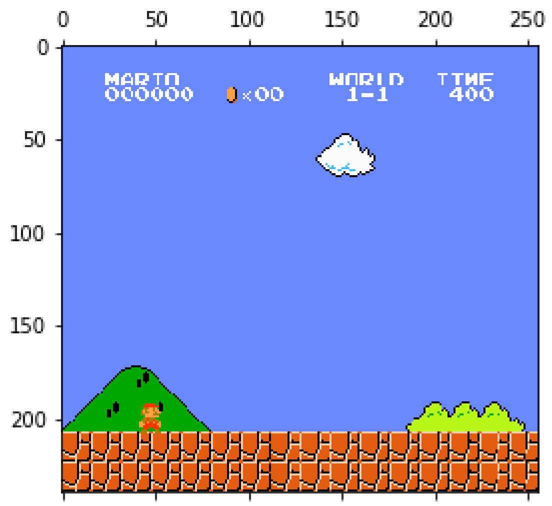


Figure 3: RGB Mario Image

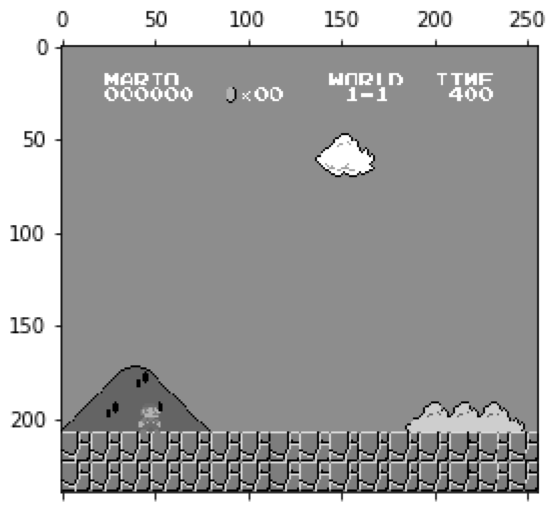


Figure 4. Grayscale Mario Image

3.2.2 Normalization

Normalization reduces the raw pixel intensities into a range from zero to one. This controls the values and ensures that the gradients on the model do not go exponentially high due to the weights and biases that are added into the calculation. We do this by adding a resizing and rescaling layer to downsample each state into a square image of size (1, 84, 84).

3.2.3 SkipFrames

Because consecutive frames don't vary much, we can choose to skip n-intermediate frames without losing much information. The n-th frame aggregates rewards accumulated over each skipped frame.

3.2.4 FrameStack

Augmenting data by stacking up the previous frames helps to improve the model's feature detection by providing historical records. This adds a small number of temporal attributes in the input data to the model and allows the model to pick up features that are related to temporal changes. This way, we can identify if Mario was landing or jumping based on the direction of his movement in the previous several frames.

This works by taking k frames sequentially before the current frame and appending all of them together into a multi-channel input. In our task, we choose to set k = 4.

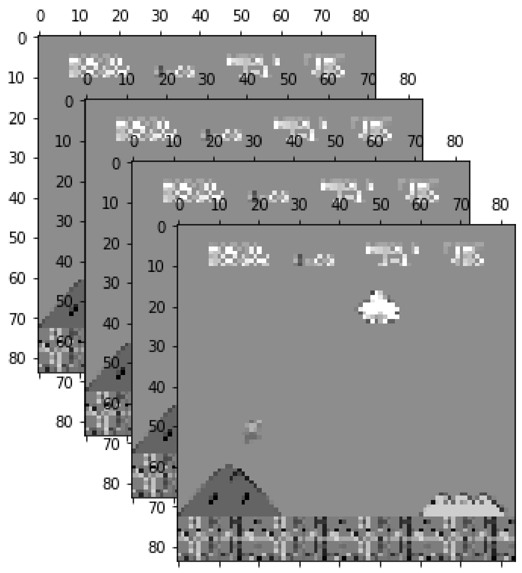
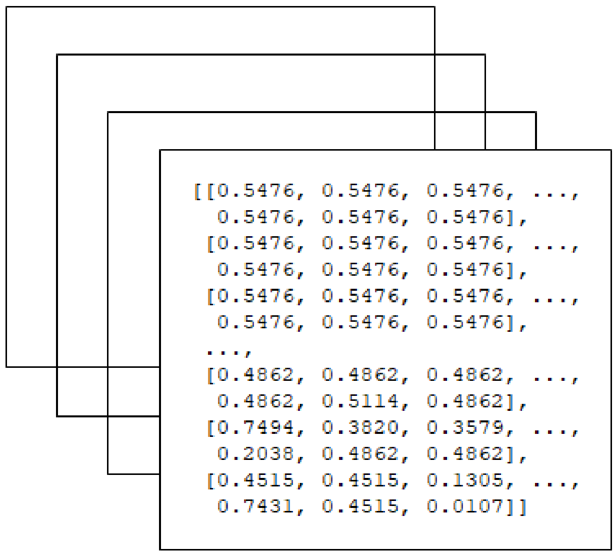


Figure 5: Final state



```
[[0.5476, 0.5476, 0.5476, ...,
  0.5476, 0.5476, 0.5476],
 [0.5476, 0.5476, 0.5476, ...,
  0.5476, 0.5476, 0.5476],
 [0.5476, 0.5476, 0.5476, ...,
  0.5476, 0.5476, 0.5476],
 ...,
 [0.4862, 0.4862, 0.4862, ...,
  0.4862, 0.5114, 0.4862],
 [0.7494, 0.3820, 0.3579, ...,
  0.2038, 0.4862, 0.4862],
 [0.4515, 0.4515, 0.1305, ...,
  0.7431, 0.4515, 0.0107]]
```

Figure 6: Visualization of stacked arrays

The final state consists of 4 grey-scaled consecutive frames stacked together as shown above in Figure 4. Each time Mario makes an action, the environment responds with a state of this structure. The structure is represented by a 3D array of size (4, 84, 84).

4 MODEL

4.1.1 Deep Q-Network (State-of-the-Art)

Deep Q-Network (DQN) is a state of the art, value-based reinforcement learning algorithm that builds on a neural network. The state in the environment comes in pixels and the agent would perform discrete actions. It initializes a neural network model, performs an action and updates the neural network weights using Bellman Equation in a repeated format. On top of that, the learning takes place with a defined exploration and exploitation rate, which determines the type of strategy the model is going to perform next.

DQN uses techniques such as “Experience Replay” and “Replay Memory” during training, which helps to train the network, breaking correlation between consecutive steps and avoiding inefficient learning. (Mehta, 2019)

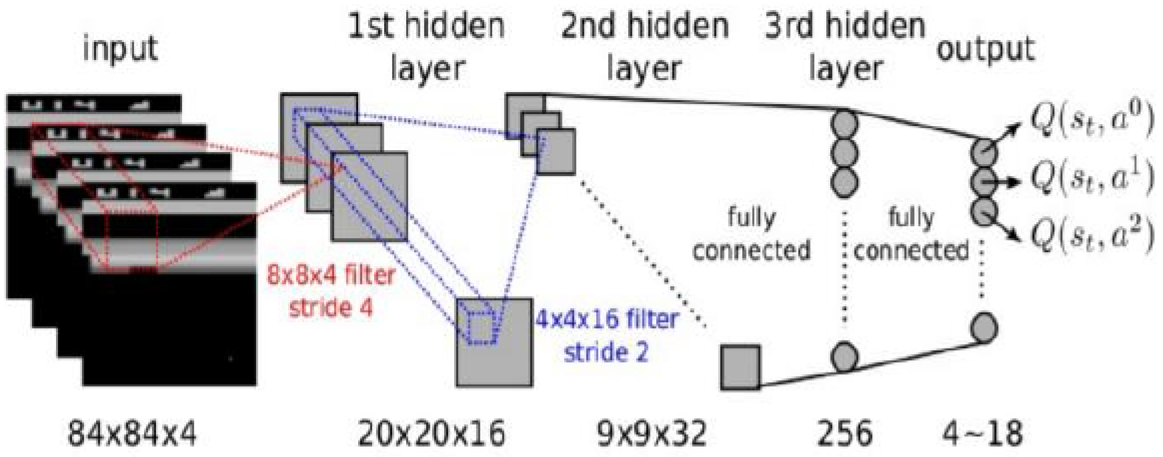


Figure 7: Deep Q Networks with a stack of picture frames and output as Q-values (Mehta, 2019)

It uses two networks, namely Policy Network and Target Network. The policy network is fed by random samples for training and output Q-values. The loss is propagated and minimized, with the new Q-value being equal to the weighted sum of the old Q-value. To prevent inefficiency during training by not using the same Q-values, there is a Target Network used for getting target values. After a set amount of steps, the Target Network is synced with the Policy Network as seen in Figure 8.

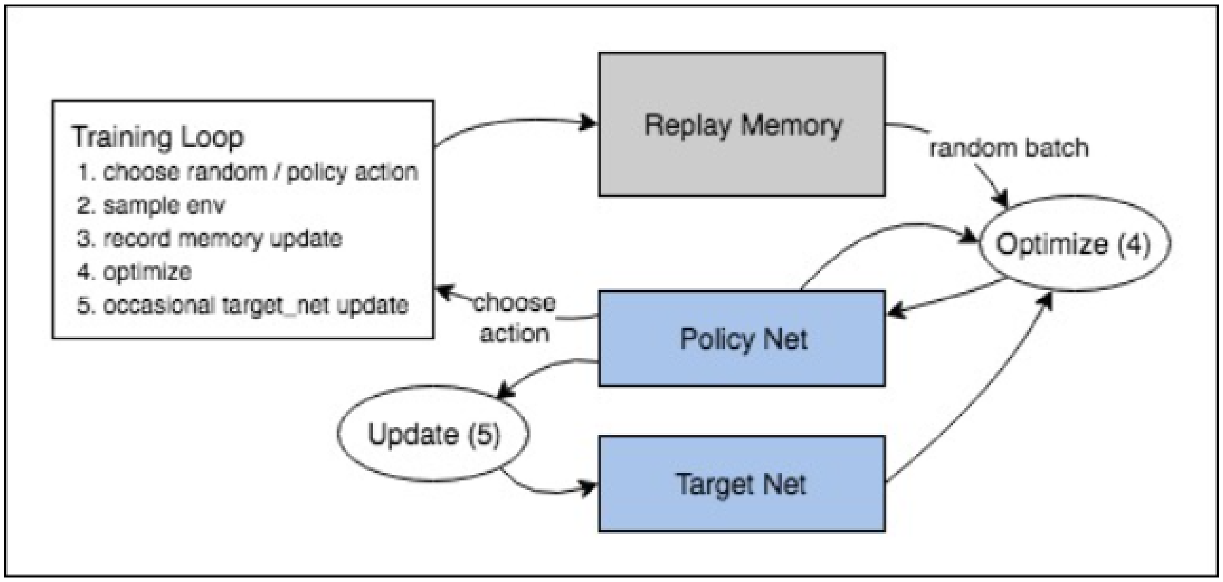


Figure 8: Visual of DQN training, taken from pytorch.org

LOSS FUNCTION

$$Loss = E\,[R_{t+1} + \gamma max_{a'}\, Q_*(s', a')] - E\,[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}]$$

Q-VALUE

$$Q^{new}(s,a) = (1\text{-}\alpha)\; Q(s,a) + \alpha\; (R_{t+1} + \gamma max_{a'}\, Q_*(s', a'))$$

TARGET

$$Target = r + \gamma(1 - done)\; max\, \{Q_*(s', :)\}$$

PREDICTION

$$\text{Prediction} = \text{Q(s,a)}$$

ERROR

$$Error = (Target - Prediction)^2$$

4.1.2 Double Q-learning

Double Q-learning is a reinforcement learning algorithm that counteracts overestimation problems with the popular Q-learning algorithm. The max operator in DQN and the standard Q-learning uses the same values to select and evaluate action. As this increases the probability of selecting values that are overestimated, it causes the resulting predictions to have over-optimistic values. To resolve the tendency of getting over-optimistic values, one has to disassociate the selection from the evaluation, which forms the main motivation for the development of the Double Q-learning algorithm.

There are two sets of weights, θ and θ’, in the standard Double Q-learning algorithm. They are obtained when each of the two value functions is learned by having randomly assigned experiences for them to be updated. For each update, two sets of weights are used to determine the greedy policy and its value.

In the standard Q-learning algorithm, the target Y_t^Q is defined as:

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t)$$

Disassociating the selection and evaluation in the standard Q-learning algorithm above, the target could be re-written as:

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

Double Q-learning error could be re-written as:

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

Even though the selection of action in the argmax is still due to the weights θ_t, the second set of weights θ_t' is used to moderate the value of the greedy policy. θ_t' can be updated equally by alternating the roles of θ and θ' (Hasselt et al., 2015, 2).

4.1.3 Double Deep Q Network (DDQN)

Mario uses the DDQN algorithm, which uses two ConvNets (Qonline and Qtarget) that independently approximate the primal action-value function. The DDQN algorithm is formulated with reference to both Double Q-learning and DQN. Its update is similar to that of DQN's update which is given by:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

Compared to Double Q-learning, the second network's weights θ_t' are replaced with θ_t^-, which are the weights of the target network to produce a more accurate evaluation of the greedy policy. The target network's update remains the same from DQN, retaining a periodic replica of the online network. DDQN strives to tweak the DQN towards Double Q-learning to get its benefits while retaining the majority of the DQN algorithm for an accurate comparison with minimal computational complexity (Hasselt et al., 2015, 4).

4.1.4 TD Estimate and TD Target

To help our model learn, we calculate the TD Estimate and the TD Target. TD estimate calculates the predicted optimal Q* for a given state s while TD Target calculates an aggregation of the current reward and the predicted Q* in the next state s'

$$TD_e = Q^*_{online}(s, a)$$

$$TD_t = r + \gamma Q^*_{target}(s', a')$$, where $$a' = argmax_a Q_{online}(s', a)$$

We use action a to maximize Q_{online} in the next state s' since we do not know what the next action a' will be.

4.1.5 Updating the Model

As our agent learns, TD_t and TD_e are computed. The loss is back-propagated down Q_{online} so that the parameters θ_{online} could be updated. On the other hand, instead of being updated during the process of backpropagation, θ_{target} is referenced from θ_{online} from time to time.

$$\theta_{online} \leftarrow \theta_{online} + \alpha\nabla(TD_e - TD_t)$$

$$\theta_{target} \leftarrow \theta_{online}$$

4.1.6 Model architecture

The two networks - Q_{online} and Q_{online} have the same architecture as follows:

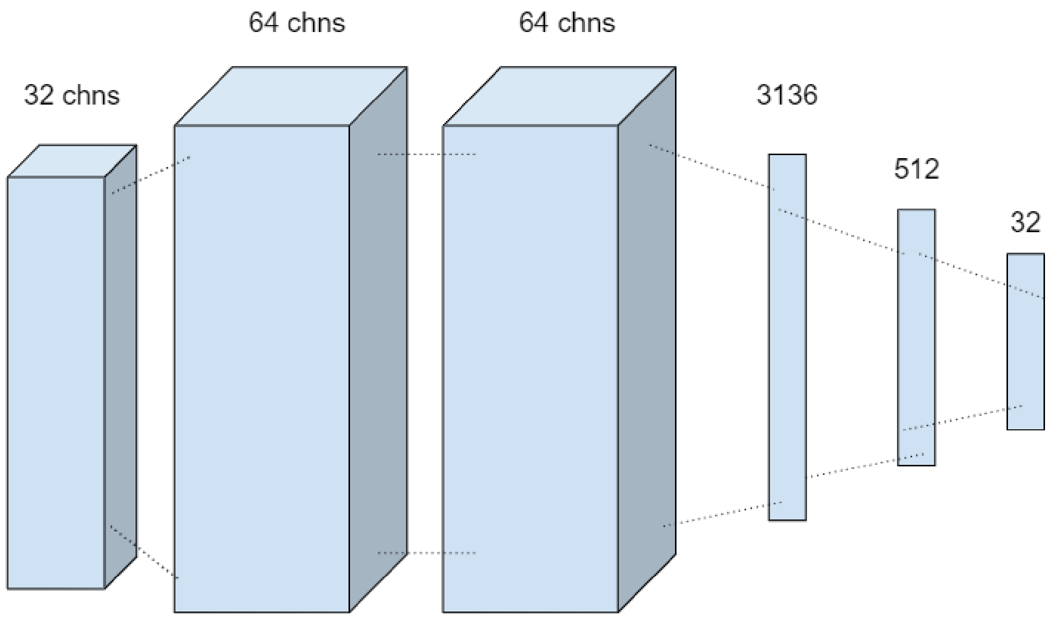
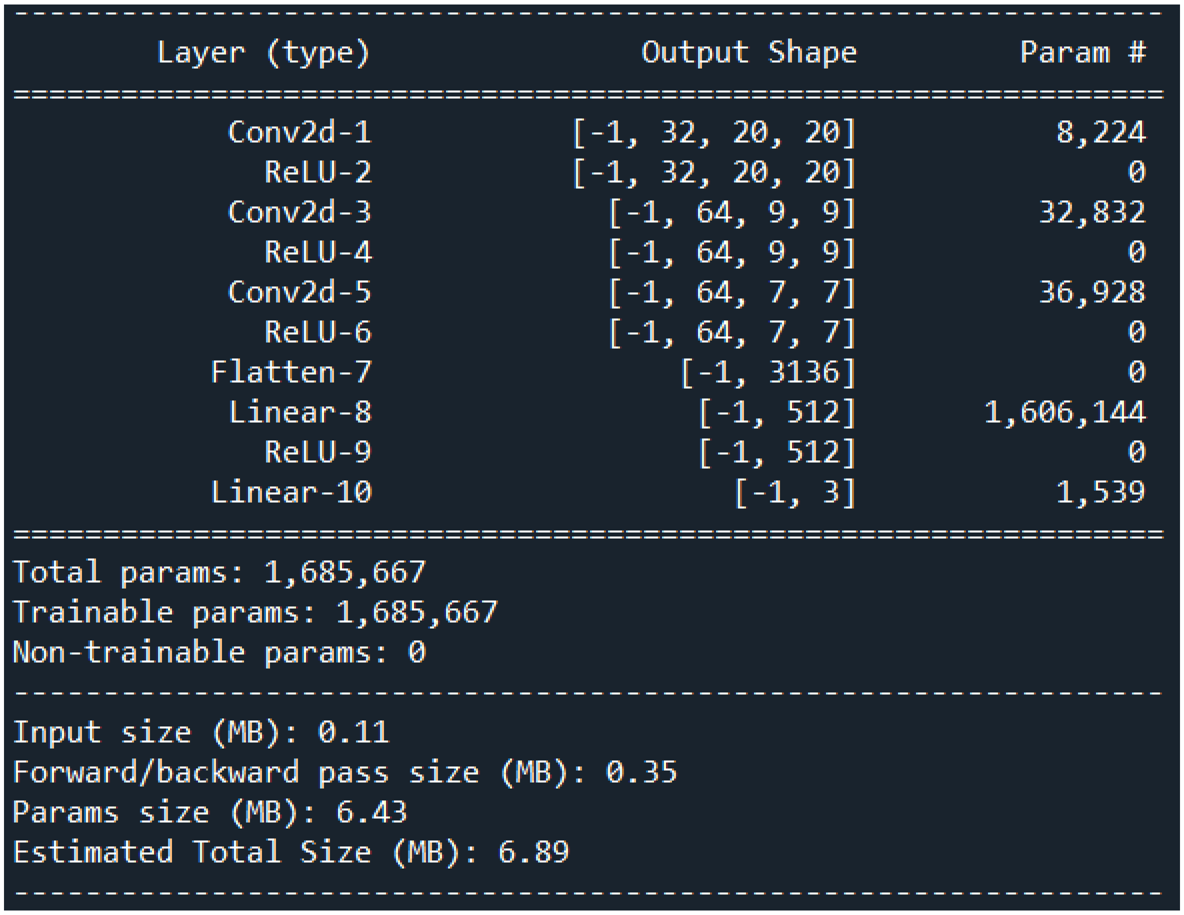


Figure 9. Convolutional Neural Network Visualisation


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 20, 20]           8,224
              ReLU-2           [-1, 32, 20, 20]               0
            Conv2d-3             [-1, 64, 9, 9]          32,832
              ReLU-4             [-1, 64, 9, 9]               0
            Conv2d-5             [-1, 64, 7, 7]          36,928
              ReLU-6             [-1, 64, 7, 7]               0
           Flatten-7                 [-1, 3136]               0
            Linear-8                  [-1, 512]       1,606,144
              ReLU-9                  [-1, 512]               0
           Linear-10                    [-1, 3]           1,539
================================================================
Total params: 1,685,667
Trainable params: 1,685,667
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.35
Params size (MB): 6.43
Estimated Total Size (MB): 6.89
----------------------------------------------------------------
```
Figure 10. Pytorch model layers

The models adopt the Adam optimizer algorithm as proposed in Adam: A Method for Stochastic Optimization. The loss function is the Smooth L1 loss implemented under the PyTorch library which creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise. We choose this loss as it is known to be less sensitive to outliers than the Mean Squared Error loss and in some cases prevent exploding gradients.

4.2 Experiments

In this experiment, we will be looking at a number of hyperparameters to adjust. Below, we have listed the name of the hyperparameters as well as their role in the entire DDQN. We will be tuning one hyperparameter at a time while keeping all the other hyperparameters to the default parameters. This ensures that we are able to compare any concrete differences and be able to explain the reason behind the difference.

The experiments will be performed on **SuperMarioBros-1-1-v0**, found in the gym_super_maro_bros library. The reward score that is returned by the environment per action is a sum of instantaneous velocity, the difference in the game clock between the current and previous frame as well as the death penalty. (Kauten, 2020).

Each episode ends when the agent, Mario, has failed the level.

Hyperparameters	**Role**	**Values**

Memory	The replay memory dataset keeps up to **k** amount of past values. These values involve the state, next state, action and reward. A random sample amount of memory is used for training.	k= 100
		k= 1000 (default)
		k= 10000

Exploration Decay Rate	The percentage **k** in which the model reduces the exploration rate. Reducing the exploration rate increases the exploitation rate.	k= 0.9975
		k= 0.99975
		k= 0.999975 (default)

Syncing Steps	The number of steps **k** taken before the Policy Network and Target Network syncs. Each step refers to a frame.	**k= 100 (default)**
		k= 1000
		k= 10000

Learning Steps	The number of steps **k** taken before the Policy Network trains from the memory.
	Each step refers to a frame.
		k= 4 (default)
		k= 40
		k= 400

Other notable default parameters will be as follow:

Hyperparameters	**Role**	**Values**
Joypad Controls	The list of allowable actions to be performed by the agent	**[‘right’, ‘right, A’, ‘A’]** (move right, right jump and jump)
Batch size	The number of frames used during every training step	**32**
Minimum exploration rate	The lowest percentage for exploration rate	**0.1**
Starting exploration rate	The starting percentage for exploration rate	**1.0**
Gamma	The discount rate	**0.9**
Burning	The number of initial frames	**33**

	that will be accumulated first before the first training	
Learning rate	The value which determines how fast the neural network would converge to a minima	**0.00025**

4.2.1 Parameter 1: Memory

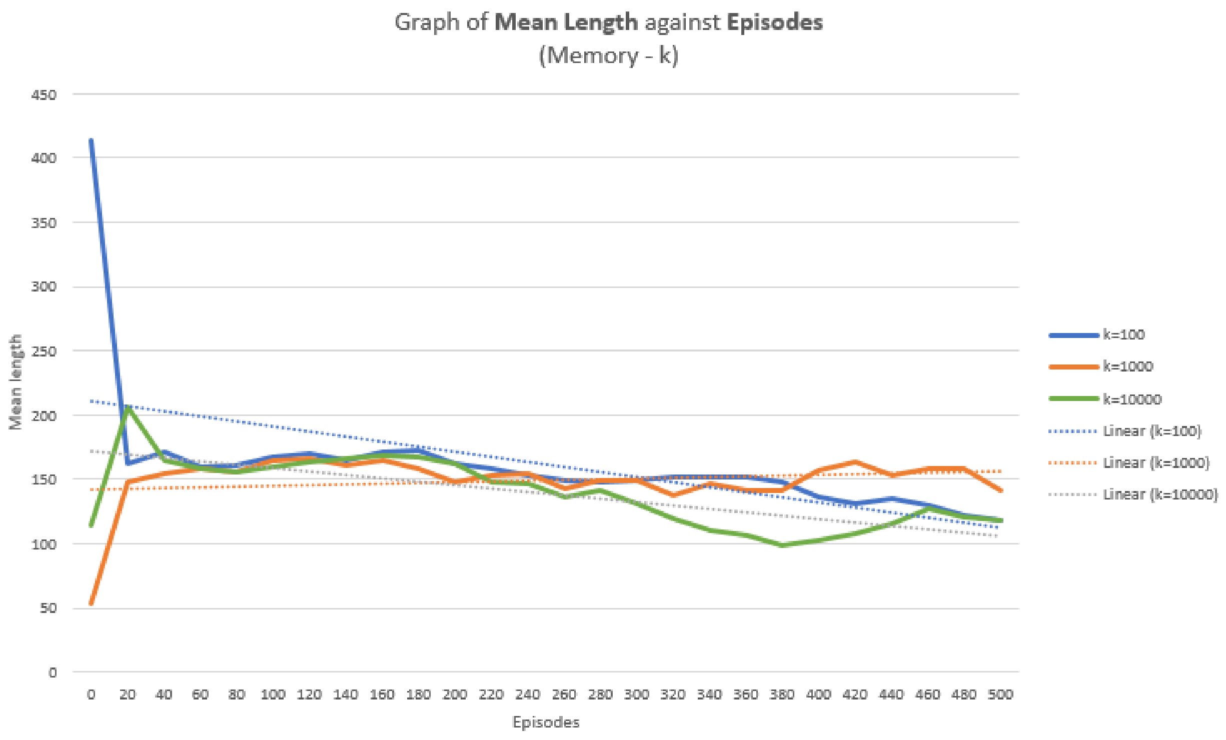


Figure 11.1: Graph of **Mean length** against **Episodes** for Memory of **k=100, k=1000 and k=10000**

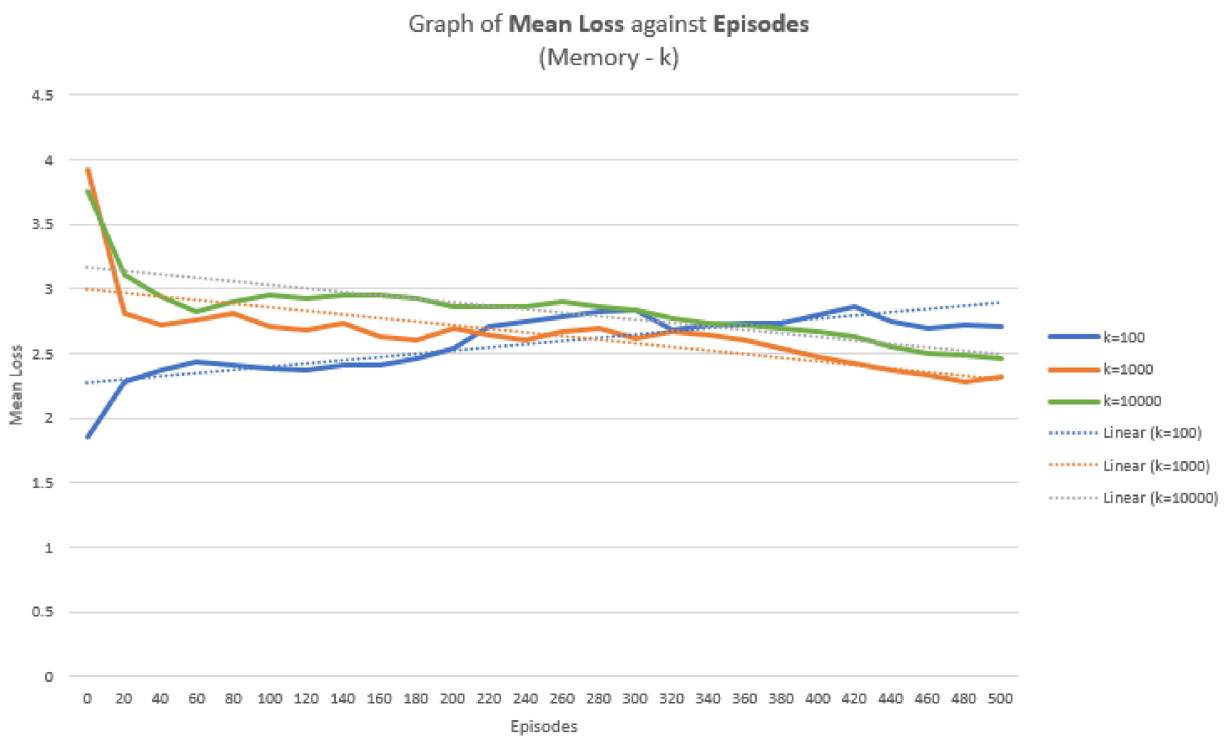


Figure 11.2: Graph of **Mean Loss** against **Episodes** for Memory of **k=100, k=1000 and k=10000**

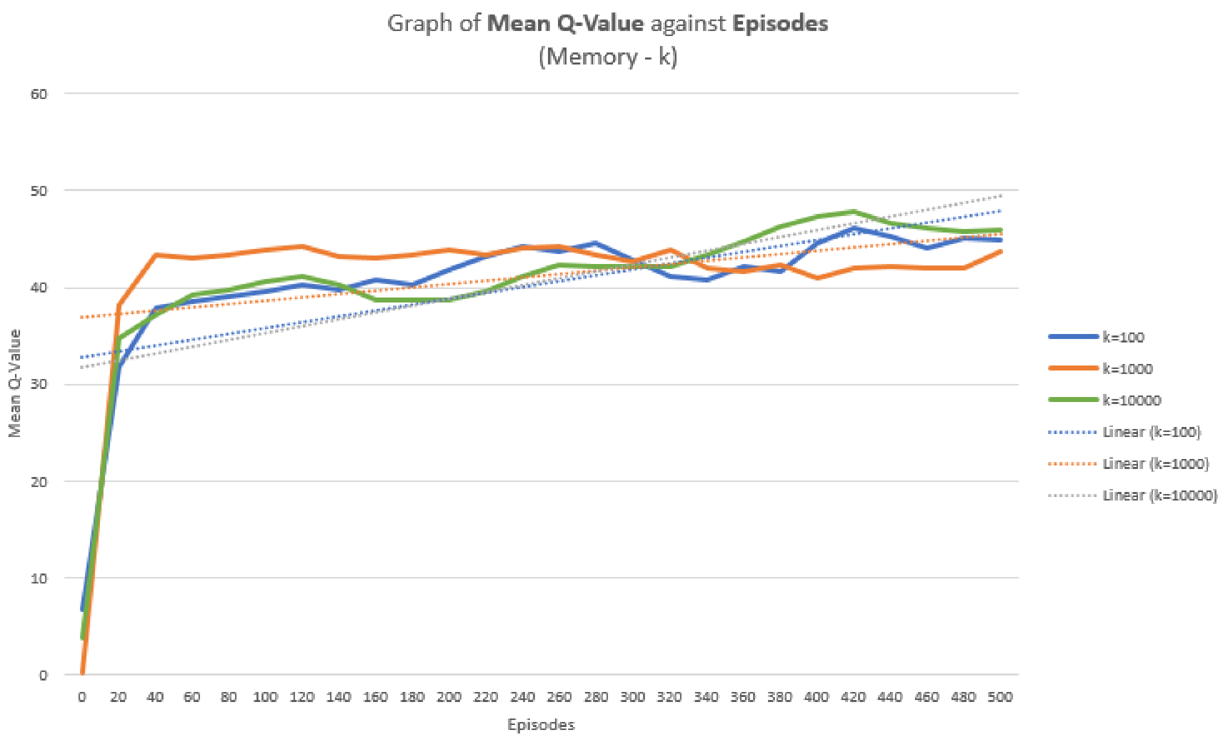


Figure 11.3: Graph of **Mean Q-Value** against **Episodes** for Memory of **k=100, k=1000 and k=10000**

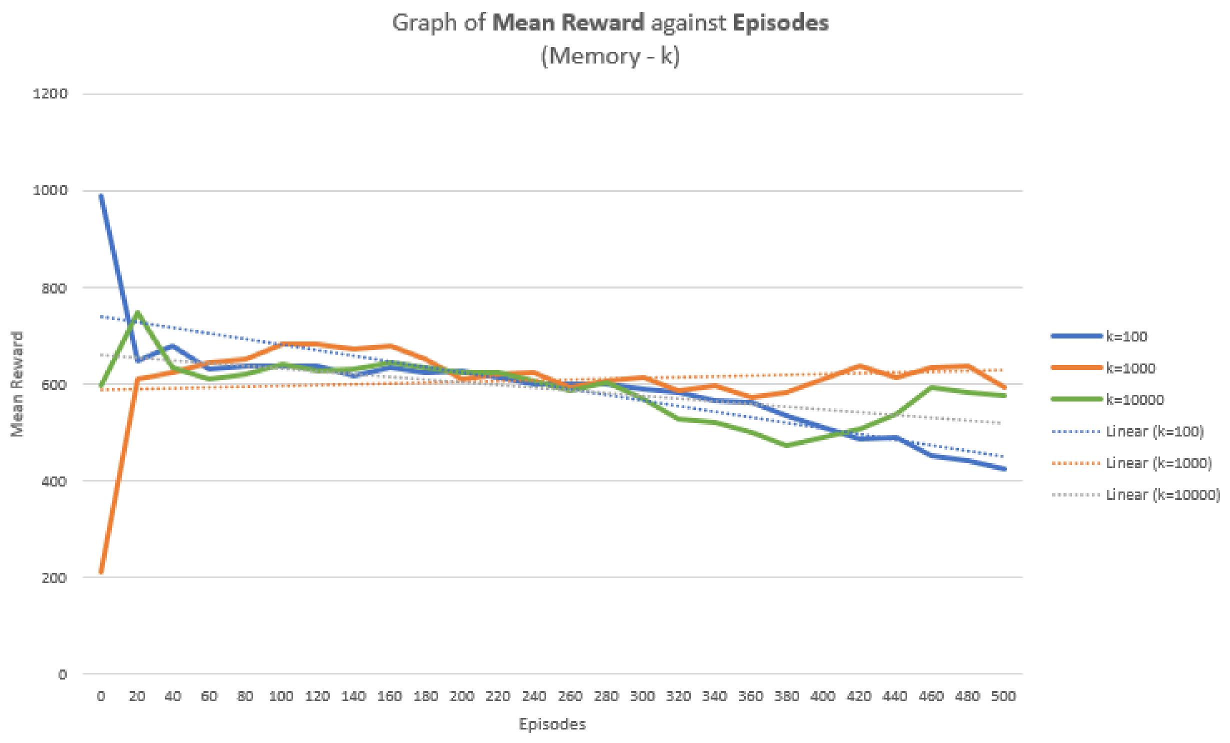


Figure 11.4: Graph of **Mean Reward** against **Episodes** for Memory of **k=100, k=1000 and k=10000**

The above graphs are trained on the memory hyperparameter, which limits the number of actions and states stored within the memory replay. From the results obtained, we can see that in the ideal case the memory hyperparameter is k=1000.

In comparison, when k=100 or k=10000, the mean length and mean reward drops lower than k=1000 as the number of episodes increases. On the other hand, for k=1000, the mean length and mean reward increase.

This could be as a result of possible underfitting for k=100 and overfitting for k=10000. For k=100, the model might not have been able to learn the best moves for each state due to the limited number of training data from memory and thereby fail to achieve a higher mean length.

As for k=10000, it might have too much data from the memory, with lots of repeated data stored in the memory brought over by the previous episodes. This could cause the model to learn the same data points without variation of data for the later part of the stage. As such, it could cause the model to overfit the early parts of the stages.

From this experiment, we find that k=1000 is a fit that nicely learns both the different parts of the stages.

4.2.2 Parameter 2: Exploration Decay Rate

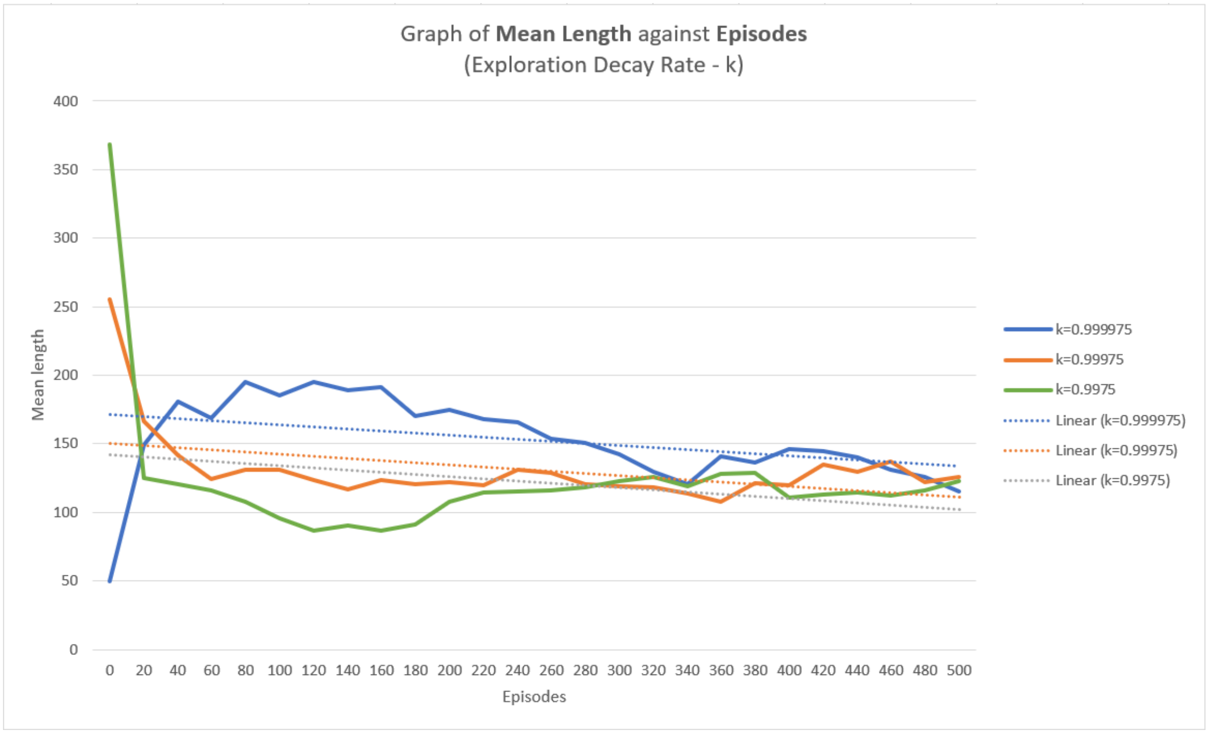


Figure 12.1: Graph of **Mean Length** against **Episodes** for exploration decay rates of **k=0.999975, k=0.99975 and k=0.9975**

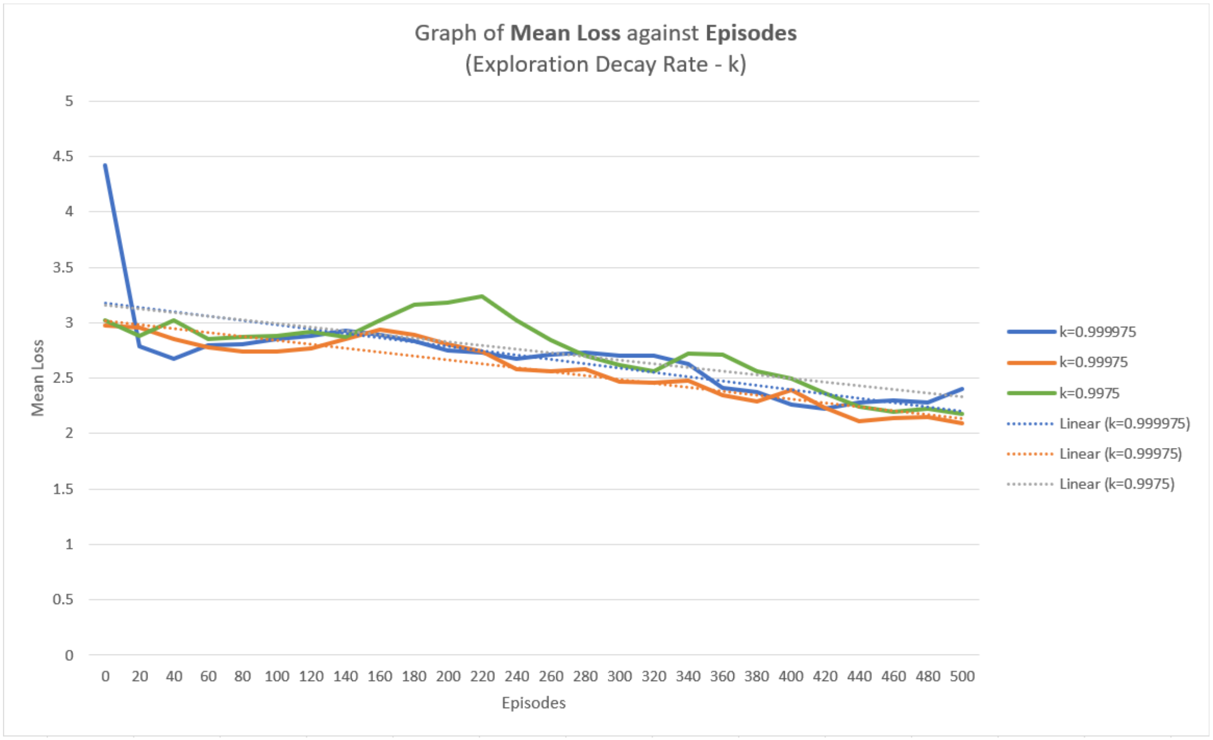


Figure 12.2: Graph of **Mean Loss** against **Episodes** for exploration decay rates of **k=0.999975, k=0.99975 and k=0.9975**

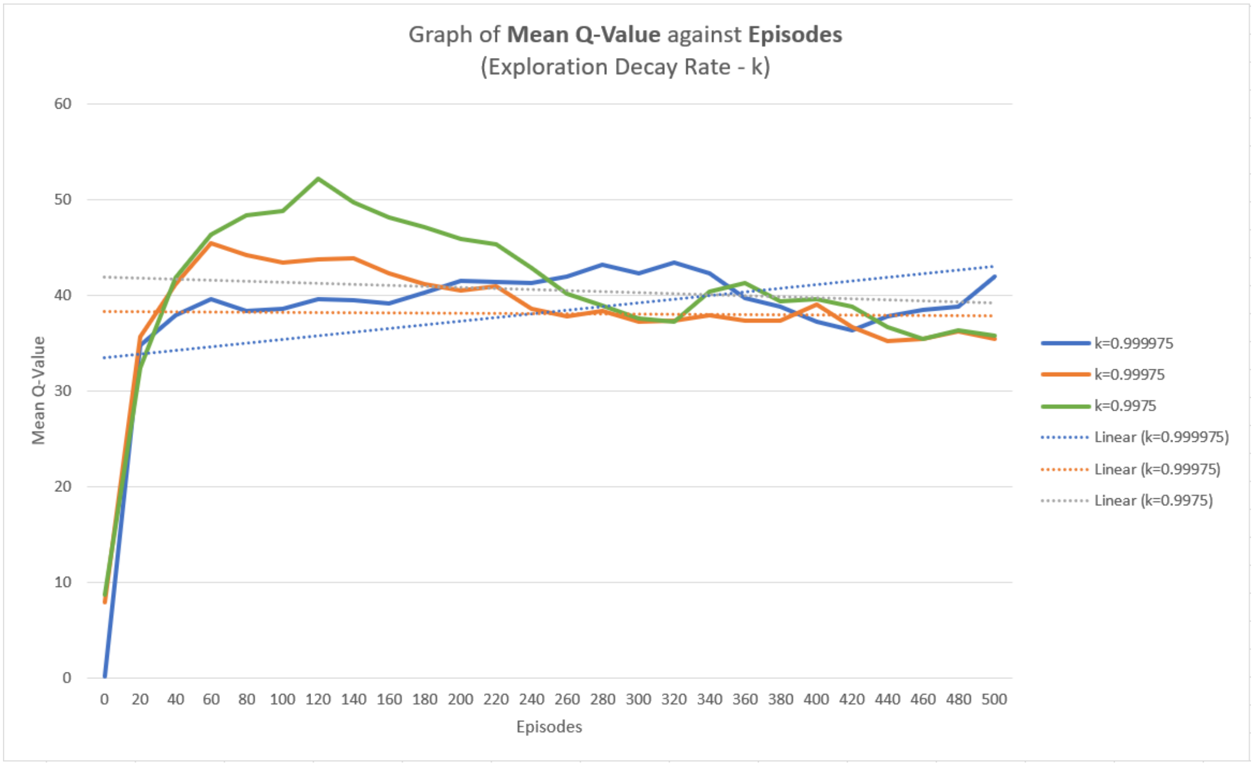


Figure 12.3: Graph of **Mean Q-Value** against **Episodes** for exploration decay rates of **k=0.999975, k=0.99975 and k=0.9975**

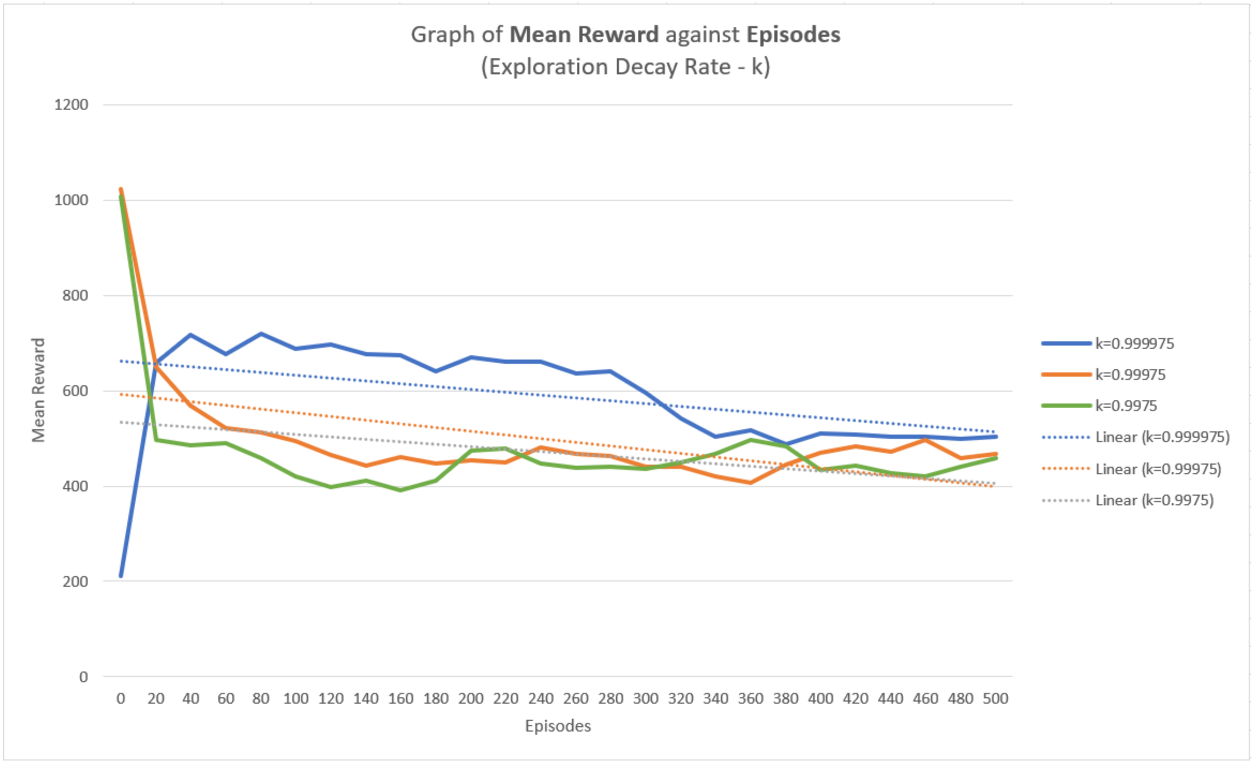


Figure 12.4: Graph of **Mean Reward** against **Episodes** for exploration decay rates of **k=0.999975, k=0.99975 and k=0.9975**

From the graphs above, we can see that the mean length and mean reward are slowly decreasing with each episode. Since it happens for all values of k, it could mean that the other default hyperparameters might not be the best or the overall model is not learning well. The total number of episodes performed could be insufficient to get an accurate overall picture.

However even with the limitations, based on the mean Q-value graph we can see that k=0.999975 has a better learning rate as the mean Q-value increases as the number of episodes increases. For k=0.99975 and k=0.9975, the mean Q-value has a large increase before it ultimately drops.

This could be due to the rapid decline of the exploration rate on the model. Since the model might not have sufficient training on the later part of the stages, it has a low probability of making the right action during exploitation and thus it impacts the mean Q-value.

On top of that, we can also be assured that the model is still training as the mean loss for all the k values are decreasing as the number of episodes increases. This signifies that the model is training though it does not tell if the model is overfitting.

4.2.3 Parameter 3: Syncing Steps

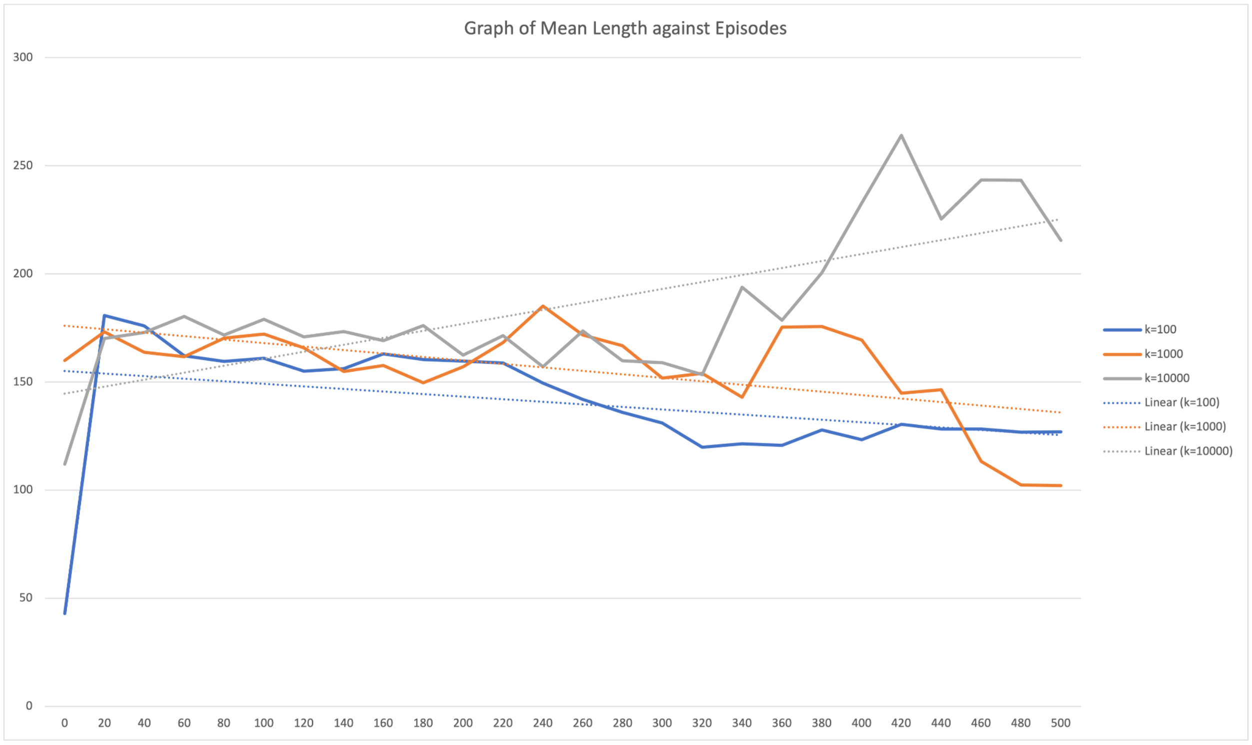


Figure 13.1: Graph of **Mean Length** against **Episodes** for syncing steps of **k=100, k=1000 and k=10000**

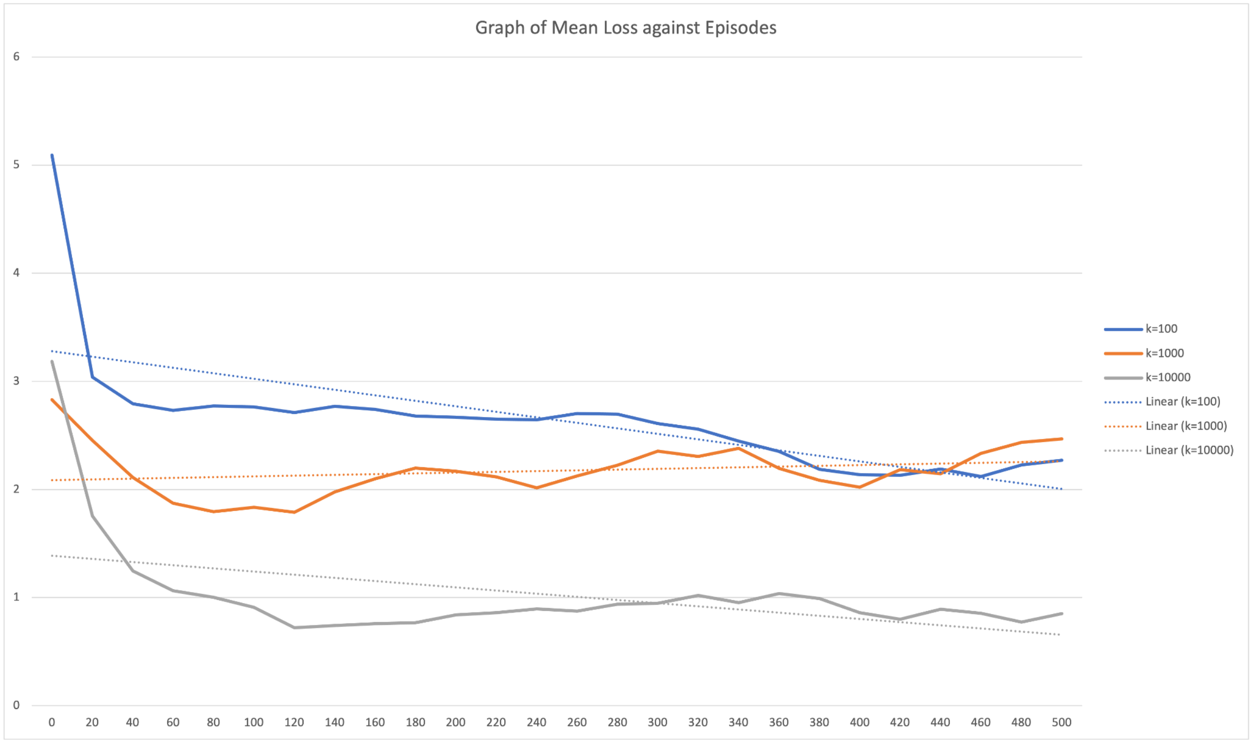


Figure 13.2: Graph of **Mean Loss** against **Episodes** for syncing steps of **k=100, k=1000 and k=10000**

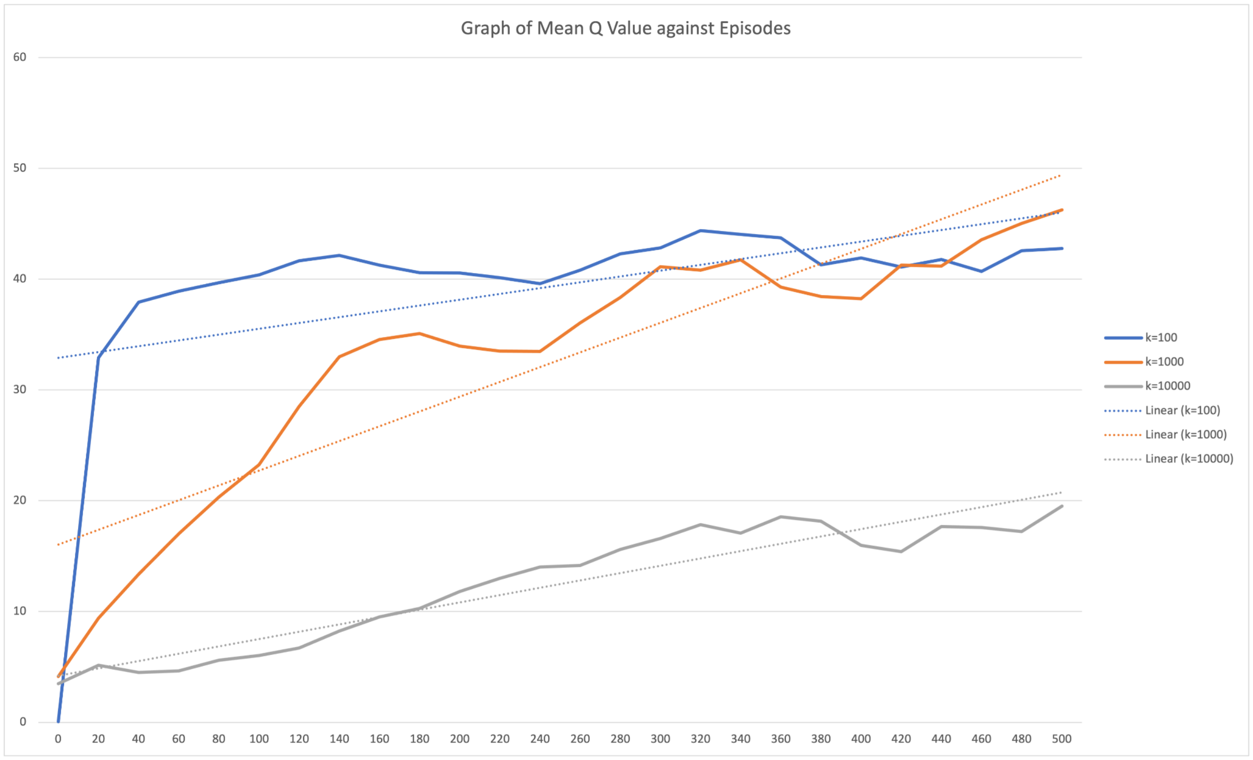


Figure 13.3: Graph of **Mean Q Value** against **Episodes** for syncing steps of **k=100, k=1000 and k=10000**

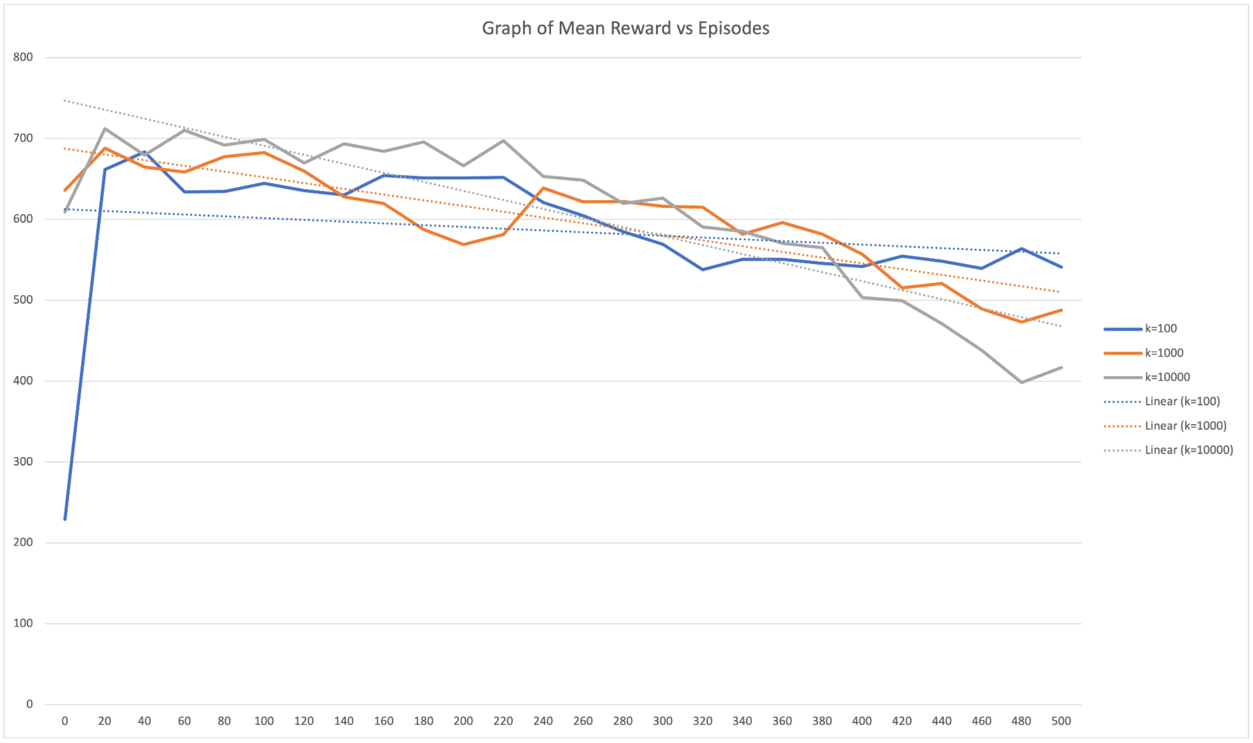


Figure 13.4: Graph of **Mean Reward** against **Episodes** for syncing steps of **k=100, k=1000 and k=10000**

In this experiment, we change the hyperparameter of the syncing step to determine the number of steps the agent will take before syncing the policy network and the target network. From the mean length graph and mean loss graph, we can see that the mean length for k=10000 has a very huge jump as compared to the other 2 values. The loss graph is also way lower for k=10000 as the number of episodes increases.

This can be explained as the delayed sync creates a more stable learning process for the entire model. Having stable learning allows the agent to make a better prediction action and thus enable the agent to cover more distance.

However, it is interesting to note that the mean Q-value of k=10000 is lower than the other k values, even though all of the k values have a positive gradient.

4.2.4 Parameter 4: Learning Steps

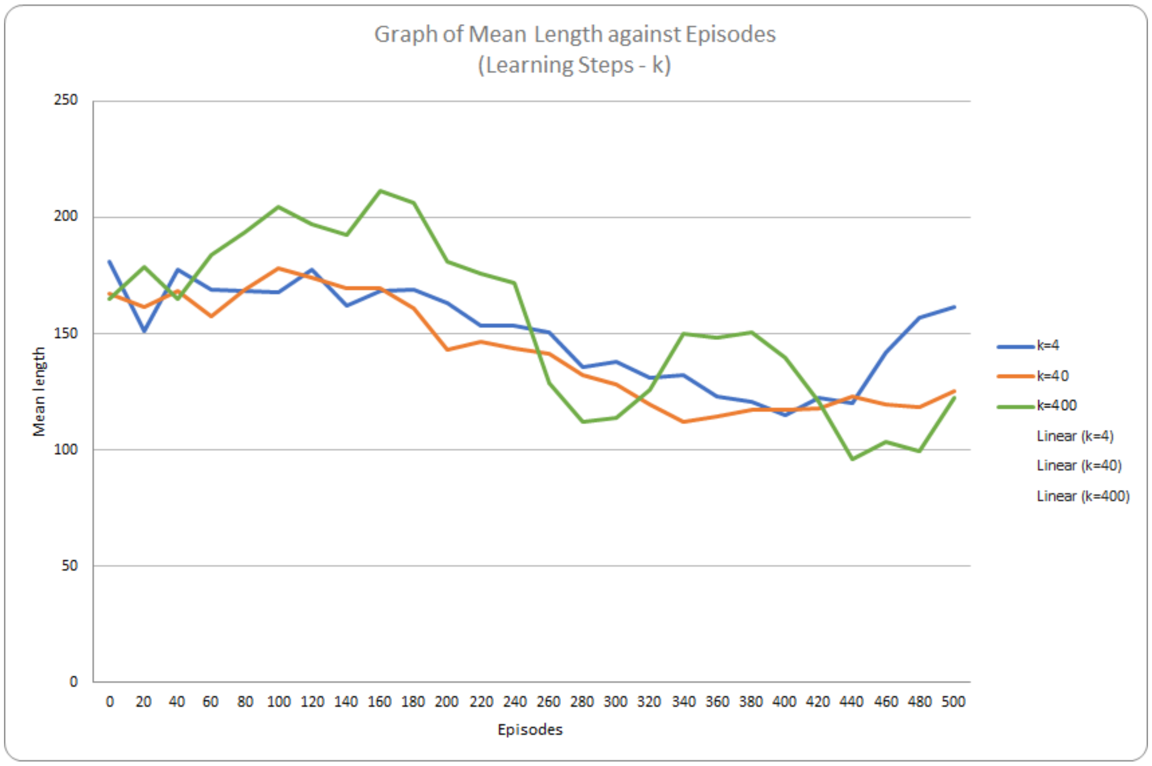


Figure 14.1: Graph of **Mean Length** against **Episodes** for learning steps of **k=4, k=40 and k=400**

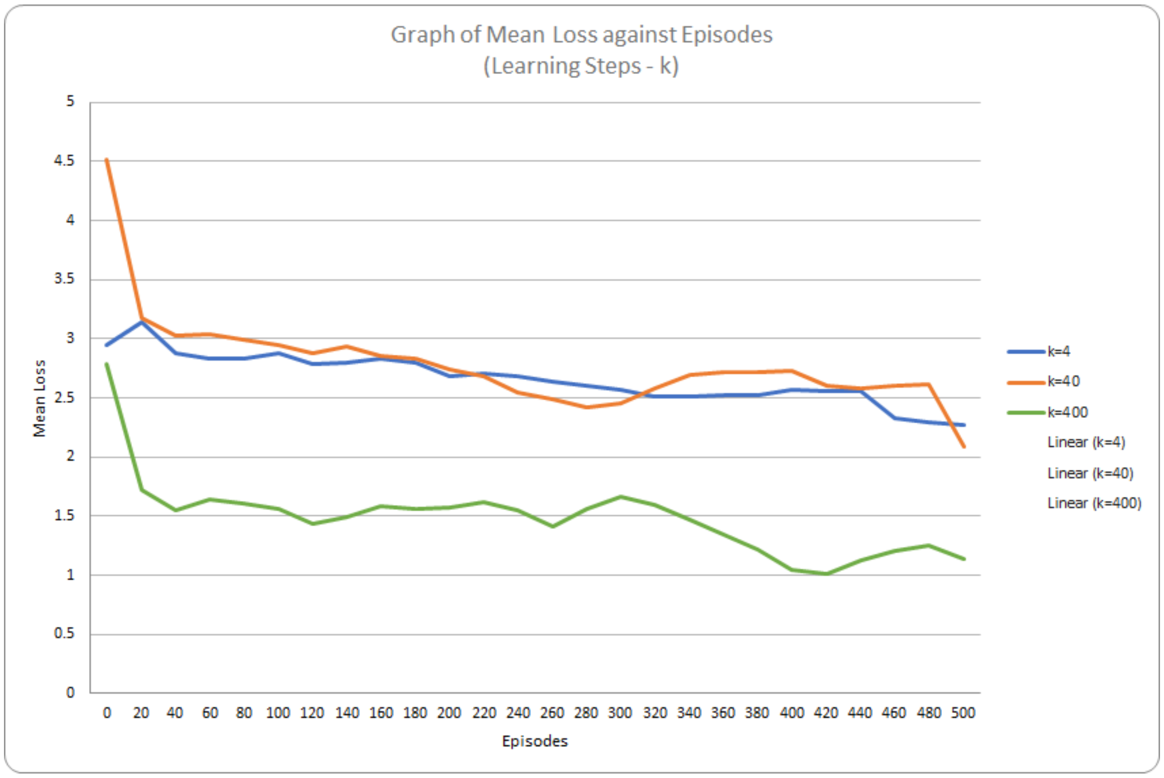


Figure 14.2: Graph of **Mean Loss** against **Episodes** for learning steps of **k=4, k=40 and k=400**

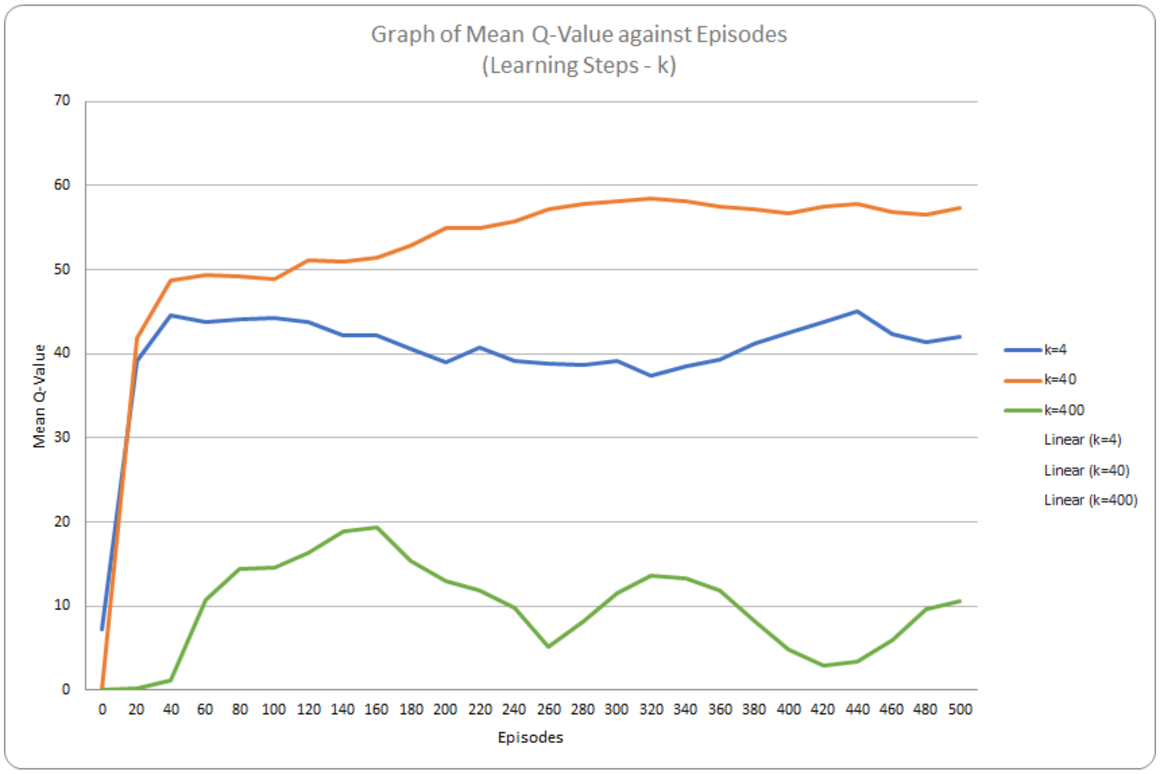


Figure 14.3: Graph of **Mean Q-Value** against **Episodes** for learning steps of **k=4, k=40 and k=400**

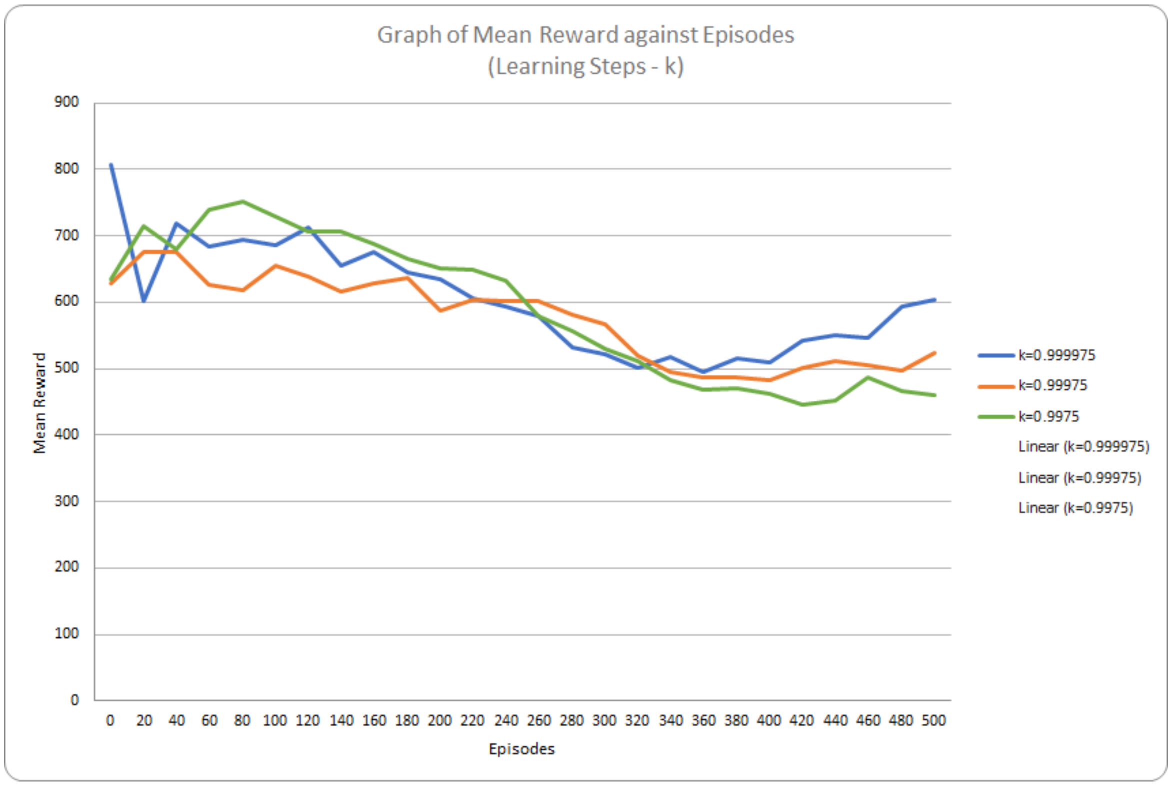


Figure 14.4: Graph of **Mean Reward** against **Episodes** for learning steps of **k=4, k=40 and k=400**

For this experiment, we change the hyperparameter for the learning step which captures the number of frames before the policy network does a training step.

What is interesting when experimenting with this hyperparameter is the mean Q-value produced by the different k values. From Figure 4.3, we can see that there is a clear distinction and stable range that each k values produce. The higher Q-values are achieved by k=40. This could possibly be attributed to both overfitting and underfitting as well.

For k=4, the model is learning at a rapid rate of once every four steps are taken. This can cause the model overfit by learning too much from the same sample of data, resulting in lower mean Q-values. As for k=400, the model might be underfitting as it only learns once every four hundred steps taken. As such, given the number of fixed episodes, the model might not be able to make the best-predicted action, resulting in the same low mean Q-values.

5 RESULTS & EVALUATION

The overall DDQN works in terms of being able to perform reinforcement learning in this game environment. It is able to take in the state of the game in pixel format and output the next actions that should be taken.

Based on the experimentations, we can see that fine-tuning the hyperparameters is an important step to take in training a reinforcement learning model as the small differences in the parameters does result in a significant change in the results.

The differences could result in the entire model being under fitted or overfit, just like in a classic deep learning training process. However, given the nature of the problem with dynamic runtime datasets, reinforcement learning proves to be much more difficult to tune.

We find that the following hyperparameters perform better when all the rest of the other hyperparameters remain the default.

- Memory - k=1000
- Exploration Decay Rate - k=0.999975
- Syncing Step - k=10000
- Learning Step - k=40

However, it is to be noted that these values might not give the best results when used together. For that to happen, grid search has to take place.

On top of that, it is merely a hypothesis that the chosen hyperparameters affect the model to a higher extent as compared to the other untouched hyperparameters. To fully prove the hypothesis, experiments would have to be done on the other default valued hyperparameters as well. However, due to the lack of time and resources, our team is unable to test out all the hyperparameters and chose the four hyperparameters based on initial research.

6 CONCLUSION

Moving forward, it is possible to try out other forms of reinforcement learning as well that works on pixel inputs by utilizing a Convolutional Neural Network (CNN). Reinforcement models such as A2C/A3C and Deep Deterministic Policy Gradient Algorithm. With some adjustments in terms of the Convolutional Neural Network, both models should be able to perform reinforcement learning on the game and might potentially provide better outcomes.

Changes to the CNN can be tweaked as well for better feature extraction. Modifications such as transformer, attention-based layer, could be added to the CNN for possibly better performance.

To improve the training time, better computation resources such as GPU could be used to have multiple training and even perform grid-search on the hyperparameters to find the best combination.

Bibliography

Hasselt, H. v., Guez, A., & Silver, D. (2015, December). Google DeepMind. *Deep Reinforcement Learning with Double Q-learning*, 2. arXiv:1509.06461v3

Kauten, C. (2020, June 12). *gym-super-mario-bros*. gym-super-mario-bros. https://pypi.org/project/gym-super-mario-bros/

Mehta, D. (2019, December). State-of-the-Art Reinforcement Learning Algorithms. *International Journal of Engineering Research & Technology*, *8*(12). https://www.researchgate.net/publication/338396174_State-of-the-Art_Reinforcement_Learning_Algorithms