# Super Mario Bros. with DQN and DDQN

**Mekhaill Chew**
Department of Computer Science
University of Bath
209459063
mmjc22@bath.ac.uk

**Jodie Chau**
Department of Computer Science
University of Bath
219437596
kyjc22@bath.ac.uk

**Justin Lam**
Department of Computer Science
University of Bath
219476447
khjl22@bath.ac.uk

**Ka Kam Lam**
Department of Computer Science
University of Bath
219516835
kkl58@bath.ac.uk

## 1 Problem Definition

### 1.1 Description

This project incorporates reinforcement learning methods of both Deep Q Network (DQN) and Double Deep Q Network (DDQN) to train an agent in playing the game Super Mario Bros to see if the agent can complete a level with great effectiveness. Super Mario Bros was released for the Nintendo Entertainment System (NES) in 1985. The goal of the game is to control the character Mario from the left of the level to the right so it can reach the flagpole. The player does so while avoiding enemies and obstacles within a limited time and life. Players can also collect items and beat enemies on the way to the flagpole for a higher score. The game consists of 8 worlds with 4 levels in each world [1]. The agent will be trained to complete the first world. To complete a level effectively, the method chosen will be implemented in a way that the agent learns how to complete a level as fast as possible instead of obtaining the highest score.

### 1.2 Environment

In order to train and evaluate the algorithms, the game environment was imported from the OpenAI Gym Framework using the NES python emulator [2]. The standard version is used in this project (i.e. SuperMarioBros-v0)

### 1.3 Actions

Actions are based on the original NES controller, it consists of six buttons where the player can combine the buttons to control the movements and actions of the character. The buttons are 'up', 'down', 'left', right', 'A' and 'B' [1] and it would result in $2^{64}$ possible actions. In our environment, we want to simplify the number of actions the agent can take so it can learn more efficiently. In theory, Mario can reach the flagpole by only performing actions that move to the right. However, to behave optimally, moving left is important too as Mario will have to dodge/kill enemies. Furthermore, the action of 'down' and 'up' has no practical use in the first level. Therefore, we decided to import 'SIMPLE_MOVEMENT' from the game environment. There are a total of 7 actions available in each state and the explanation is:

- NOOP: no action
- right: move right

- A: jump
- right, A: jump while moving right
- right, B: dashing to the right; throw fireball if Mario acquired the item Fire Flower
- right, A, B: jump while dashing to the right
- left: move left

### 1.4 State

From the imported environment, the observation space is 240 × 256 × 3 where 240 × 256 represents the number of pixels inside each video frame and × 3 represents the 3 RGB colour channels. To speed up the learning of our agent, the size of the frame is reduced to 84 × 84 pixels and each state is a list of 4 contiguous 84 × 84 pixels frames (i.e. a 3D array of size 4 × 84 × 84).

The initial state of the agent begins on the left of the level. The agent reaches the end state when Mario reaches the flagpole or runs out of life.

### 1.5 Transition Dynamics

As the agent moves from one screen to another screen in the environment, the agent transits into the next state by observing its current state and choosing an action. The next state also returns a reward based on the action chosen. Each transition that the agent performs is also referred as an experience. So, an experience consists of a tuple of information including the current state, current action, reward, and next state. The agent can exploit and choose an optimal action based on its most recent experience or the agent can have a probability of choosing a random action to explore what will happen in the environment. When transitioning from state to state, these experiences are gradually stored in a buffer so that the agent could also sample random experiences instead of the most recent experience.

### 1.6 Rewards

In the original game, the reward system is represented by a score. Mario could collect scores by collecting items, killing enemies, and completing the level in the smallest amount of time possible [2]. The reward function essentially shapes how we want our agent to behave. And ultimately, we intend to complete the level as fast as possible instead of going for the highest possible score. The provided reward function from the Gym environment is made up of three parts that allow us to shape the agent's behaviour as we intended. The reward function 'r' is:

$$r = v + c + d$$

, where v is the difference in the horizontal position of Mario between states. v returns a positive value, negative value, and 0 value when Mario moves right, left and stands still respectively

, where c is the difference of the game timer between frames. c returns a negative value if Mario stands still

, where d returns a negative value if Mario dies.

Basically, it rewards the agent a positive value when Mario moves right and penalises the agent when Mario stands still or dies.

## 2 Background

In this section, we discuss reinforcement learning methods that would possibly be effective within the scope of this project.

Our objective is to get Mario to the flag at the rightmost end of the level without dying. In theory, we need to observe the current state of the game, allow Mario to move from an action, check how far we are from the goal, observe the new state and make the next more. This sequence of actions, states, and rewards is known as a trajectory.[3]

If we act by considering multiple frames at once to incorporate the character's speed, we can consider this a Markov Decision Process. It means the transitions between states only depend on the latest combination of state and action without knowledge of the history.[3]

Using an estimator such as a neural network, we can then take the state and receive an action to take a policy, therefore making our goal to maximise the expected return over a trajectory:

$$J(\pi_\theta) = E_{\tau\ \pi_\theta}[R(\tau)] \tag{1}$$

, where $\pi_\theta$ - policy with parameters.

### 2.1 Vanilla Policy Gradient

To maximise the expected return we optimise the policy parameters by gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2}$$

The gradient of policy performance J is the policy gradient. After derivation, the analytical form of the policy gradient is:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\ \pi_\theta}[R(\tau)] = E_{\tau\ \pi_\theta}[\Sigma_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) \Sigma_{t'=t}^{T} \gamma^t R(s_{t'}, a_{t'}, s_{t'+1})] \tag{3}$$

This equation includes the discount factor $\gamma$ between [0,1] as a penalty to the uncertainty of future rewards. This means that we only increase the possibility of actions considering consequences. We estimate the expectation by collecting a set of trajectories where the agent produces actions according to the current policy - this is on-policy method.[3]

### 2.2 Baseline

From the above equation, if we consider a parameterised probability distribution, we can add or subtract from the policy gradient expression any baseline function. Typically, the common choice of the baseline is the on-policy value function that is approximated by neural network. This is updated concurrently with the policy. The baseline helps to reduce variance and stabilise the training process.[3]

### 2.3 Proximal Policy Optimisation

Proximal Policy Optimisation, or PPO, is a policy gradient method for reinforcement learning. The vanilla policy gradient method has poor sample efficiency because of the high variance of gradient estimation. It is due to the difficulty in credit assignment to the actions which affect the future rewards. This variance is sometimes reduced by employing the baseline method as mentioned above. However, we can use importance sampling as another way to estimate the rewards of a trajectory. In this method, the expected reward can be computed with a different policy and later refined by the importance ratio r. The importance ratio is dependent only on the policies and not the environment.[3] The equation for the importance ratio:

$$r_\tau(\theta) = \frac{P(\tau|\theta)}{P(\tau|\theta_{old})} \tag{4}$$

Another reason for low sample efficiencies of the policy gradient is because it is unable to reuse old policies. Therefore, it requires to collect new trajectories of samples using the new policy to calculate the next gradient update. To mitigate this, there is a surrogate objective function which guarantees a policy improvement as long as the new and old policies are similar. [3]

# 3 Method

## 3.1 Q-Learning

An approach to solving reinforcement learning problems is to approximate an optimal action-value function. This gives us the maximum expected return given state s and some action a.[3][4][6] The optimal action-value function:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}_{\tau\ \pi_\theta}[R(\tau)|s_0 = s, a_0 = a] \tag{5}$$

The above optimal action-value function obeys the Bellman equation. This can tell us if the optimal value Q for the state s is known for all possible actions a.[3] Then the optimal strategy is to select the action a maximizing the reward plus the value of the state you enter next:

$$Q^*(s,a) = \mathbb{E}_{s'\ P}[r(s,a) + \gamma \max_{a'} Q^*(s^{'}, a^{'})] \tag{6}$$

Q-Learning has been a long standing algorithm for estimation of optimal Q values for all state-action pairs. It assumes we have a large look-up table for all such pairs. The agent observes the current state s at time step t and then selects and performs an action with the highest Q value for the current state. However sometimes it may pick a random action, called the $\epsilon-$greedy approach. It then observes the next state $s^{'}$ and the reward $r^{'}$ at time step $t+1$.[3][5][6] The Q values are then updated according to the formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r^{'}_{t+a} + \gamma \max_{a \epsilon A} Q^{'}(s^{'}_{t+1}, a) - Q(s_t, a_t)) \tag{7}$$

where, $\gamma \in [0,1]$ is the discount factor that will determine the trade-off between short and long-term rewards

Memorization of all Q values for all state-action pairs is not practical, therefore we use neural networks to approximate Q values.[6] We assume that all sequences in the environment terminate in a finite number of time steps. From there we can model Super Mario as a finite MDP, in which each sequence is a distinct state.[4][5] This allows us to use standard Reinforcement Learning methods for MDP by using the complete sequence as the state representation at time $t_n$.

## 3.2 DQN

Deep Q-network is a multi-layered convolutional neural network. Given the state s and network parameters $\theta$ it outputs a vector of action values. It is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n is the dimension of the state space and m is the dimension of the action space.[4] The foundation of the Deep Q-network algorithm is built on two innovative approaches; experience replay and periodically updated target network.[3]

Experience replay addresses the previously stated problem that rewards are often time-delayed. It uses a single replay memory of fixed size where the N last (s,a,r,$s^{'}$) tuples are stored. These samples are then randomly drawn from the replay memory during training - this significantly increases sample efficiency and helps break correlations in data and learn from all past policies.[3][5]

The periodically updated network keeps separate identical instances of the neural network where the weights are static and synced with the leading network. Essentially, fixing the parameters in the target network and only updating every $\tau$ time-steps so that $\theta^-_t \leftarrow \theta_t$ at designated intervals. This network is used to estimate the target Q values during training. This then reduces the impact of fluctuations and stabilizes the training process.[3][4][6] Our objective function is:

$$y^{DQN}_t = r_{t+1} + \gamma max_a Q(s_{t+1}, a; \theta_t) \tag{8}$$

We then clip the rewards at a threshold that we determine, [-1;1]. This prevents the Q-values from growing too large and also keeps the gradients well conditioned.

### 3.3 Double Deep Q-Learning (DDQN)

In both traditional and deep Q-learning algorithms, the max operator uses the same values to choose and evaluate an action. This leads to greater estimation error and overconfidence.[4] Hado van Hasselt introduced a new way to estimate Q values. By assigning experiences randomly to update one of two value functions. This results in two sets of weights, $\theta$ and $\theta^{'}$; one is used to determine the greedy policy while the other determines its value.[5] This approach fixes the overestimation problem because one of the estimators might see samples that overestimate action a1, while the other sees samples that overestimate action a2.[3] The target network in the Deep Q-network model provides a second value function without having to create another network. This then allows for it to estimate its value with the target number.[4] The target then becomes:

$$y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t), \theta_t^-) \tag{9}$$

## 4 Results

In order to compare how DDQN and how a deeper neural network can benefit the result, 3 different networks with different strategies were trained. The first network (shallower_ddqn) is trained in structure1 in DDQN. The second network (deeper_ddqn) is trained in structure2 in DDQN. The third one (deeper_dqn) is trained in structure2 in DDQN. Each of the networks was trained and finished in 10,000 episodes. NVidia RTX 3060 was used to train these networks.

- Exploration vs. Exploitation
  During the training, an exploratory decaying epsilon-greedy approach was used. The agent has a probability of (1- epsilon) to select the optimized action and probability epsilon to uniformly select an action from the valid actions for exploration. The epsilon is decreased to 0.99 of its latest value in each episode and can only be decreased to 0.02, which is set as the minimum exploration rate. (The epsilon decayed with a factor of 0.99 after each episode until reaching the minimum value of 0.02.)

- Training
  In both DQN and DDQN, a target network and an online network were used. The online network is the network updated for each episode. The weights of the target network are only copied from the online network every 5000 steps. In order to minimize the error between the target Q-value, Huber loss was used to calculate the loss and Adam optimization is used to update the network weights. In DQN, the action selected for calculating the Q-value in the next state comes from previous chosen actions. This is different to DDQN, in which the selected action comes from the prediction of the online network. A constant discount rate (gamma) of 0.99 was used throughout all training sessions for maximizing long-term rewards. The learning rate (alpha) was fixed to 0.00025.

The training results are shown as above. Illustrated in Fig 1, the learning curve of shallow_ddqn network converged to a score of around 1000 while the learning curve of deeper_ddqn converged to a much higher score of around 1400. The initial improvement of the learning curve of deeper_ddqn was faster than shallower_ddqn.

As shown in Fig 2, with similar reward improvement rate, the learning curve of the deeper-ddqn network started to converge at around episode 9000; however, that of the deeper-dqn network was still climbing even at episode 10000.

After the training, the trained models were tested by 50 plays with 0.05 exploration rate. The models were tested in different episodes. The average rewards of models trained in different episodes are shown in Table 1. The deeper_ddqn model achieved the best average reward of 1671.94.

The best performance model, deeper_ddqn, was also tested in the World 1-2. However the average reward of 50 plays is only 129.7. It is much lower than the result played in World 1-1, which is 1671.94.

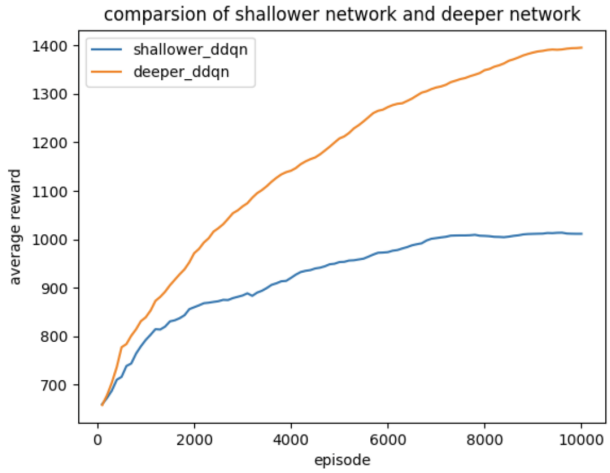


Figure 1: Comparison of Shallow vs Deeper network

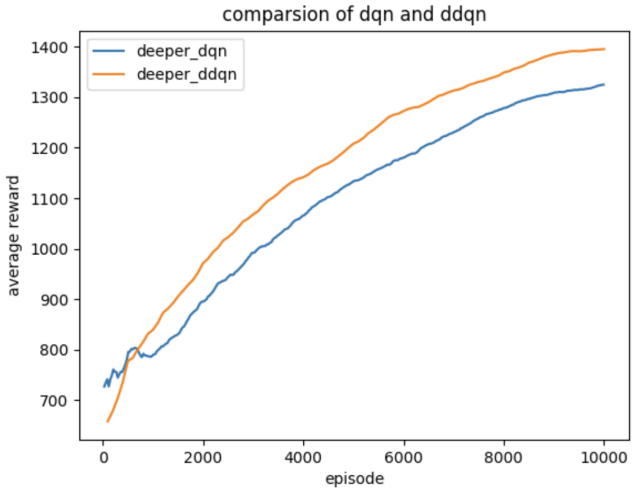


Figure 2: Comparison of DQN vs DDQN

Table 1: Comparison of Scores over various Episodes and Methods

| Episodes | Shallower_DDQN | Deeper_DQN | Deeper_DDQN |
|---|---|---|---|
| 300 | 868.06 | 861.34 | 361.20 |
| 3000 | 786.48.06 | 1197.48 | 909.94 |
| 10000 | 1027.86 | 1271.68 | 1671.94 |

## 5 Discussion

As we can observe, a deeper network typically performs better. Although it takes more time to train in a deeper layer for one epoch, it will outperform a shallower network. The results showed that the agent trained with the deeper_DDQN performed better than the agent with shallower networks. The state of the environment of these networks were 84 x 84 greyscaled and down-scaled images of the original game frames which involved many different levels with various features. The deeper networks can learn more intermediate features and patterns at different levels of abstraction that shallower ones. This suggests that using deeper networks may be beneficial in these training runs.

Moreover, there are more learnable parameters in a deeper network. This enables higher degrees of freedom in fitting and thus better approximation to the function mapping input states to target Q values. However, a deeper network cost more time due to it being more computationally heavy in each episode during training and with a higher chance of over-fitting.

We notice that DDQN agent achieved better performance than DQN using the same set of two networks. As mentioned by Van Hasselt et al. in 2016, DQN is susceptible to overestimation of target Q values, which eventually leads to suboptimal learnt policies. DDQN could reduce the overestimation by decoupling findings of the action for maximum discounted q value and calculation of target q value with such action [7].

The images used in World 1-1 are totally different from the images in other levels. This also includes the type of enemies involved (eg. new enemies like Piranha Plant), and the shade used in the background (eg. World 1-2). As the model was only trained from the pixels in World 1-1 and the environment of this level was deterministic, the models were unable to properly fit the function for input state frames from other levels. The lack of generalisation may be due to an insufficient number of layers and may be solved by using networks deeper than deeper_ddqn.

## 6 Future Work

As we have performed this project only on World 1-1, in order to let the agent play on other levels we can use transfer learning instead of training the whole network from scratch. The first two layers of the network trained in level 1 can be frozen. Further training will only train the other layers. The reason for freezing the first two layers is because the first two layers have learned some general features which can be used on the other levels. The agent can therefore use these features to learn the other levels in a much faster way without learning from scratch.

## 7 Personal Experience

Overall, we found this project interesting. We have chosen to tackle the game Super Mario Bros because we thought when the agent advances into different levels of the game, the complexity of the levels also increases, and it would be interesting to see how well the algorithm amongst different levels is. However, as mentioned in the discussion section, we soon realised the problem after implementing the algorithm. Nonetheless, we would love to experiment with more algorithms and complete more levels if given enough time. At the end, although this project is challenging, we are content with the performance of our agent.

## References

[1] Kauten, C., 2021. GitHub - Kautenja/gym-super-mario-bros: An OpenAI Gym interface to Super Mario Bros. & Super Mario Bros. 2 (Lost Levels) on The NES. [online] GitHub. Available at: <https://github.com/Kautenja/gym-super-mario-bros> [Accessed 2 May 2022].

[2] MarioWiki. 2022. Super Mario Bros.. [online] Available at: <https://www.mariowiki.com/Super_Mario_Bros.> [Accessed 2 May 2022].

[3] Schneider, N., 2021. A Simple Guide To Reinforcement Learning With The Super Mario Bros. Environment. [online] Medium. Available at: <https://medium.com/geekculture/a-simple-guide-to-reinforcement-learning-with-the-super-mario-bros-environment-495a13974a54> [Accessed 1 May 2022].

[4] Barkar, D., 2019. AI Agent for NES Super Mario Brothers. Stanford University, [online] Available at: <http://cs230.stanford.edu/projects_winter_2019/reports/15808588.pdf> [Accessed 2 May 2022].

[5] Klein, S., 2016. Deep Q-Learning to Play Mario. Stanford University, [online] Available at: <https://cs229.stanford.edu/proj2016/report/klein-autonomousmariowithdeepreinforcementlearning-report.pdf> [Accessed 2 May 2022].

[6] Grebenisan, A., 2020. Building a Deep Q-Network to Play Super Mario Bros | Paperspace Blog. [online] Paperspace Blog. Available at: <https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/> [Accessed 3 May 2022].

[7] Hasselt, H., Guez, A. and Silver, D., 2016. Deep reinforcement learning with double Q-Learning. AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pp.2094–2100. [online] Available at: <https://dl.acm.org/doi/10.5555/3016100.3016191>

# Appendices

## Appendix A: Problem Domain

In this paper, we study and use the reinforcement learning methods Deep Q Network (DQN) and Double Deep Q Network (DDQN) to train an agent in the OpenAI Gym Framework using the Nintendo Entertainment System (NES) python emulator. We used the SuperMarioBros-v0 version to train the agent. The goal of the game is to control the character, Mario, to reach the flagpole whilst avoiding enemies, fall pits and other obstacles.

For the state, from the imported environment, the observation space is 240 x 256 x 3 where 240 x 256 represents the number of pixels inside each video frame and the 3 represents the 3 RGB colour channels. To speed up the learning of our agent, the size of the frame is reduced to 84 x 84 pixels and each state is a list of 4 contiguous 84 x 84 pixel frames. The initial state of the agent, therefore, begins on the left of the level and reaches the end state when Mario reaches the flagpole or dies.

The actions are based on the original NES controller. It consists of six buttons with the ability to combine buttons to control the movement of the character. The available movements are 'up', 'down', 'left', right', 'A' and 'B'. In our environment, we want to simplify the number of actions the agent can take so it can learn more efficiently. In theory, Mario can reach the flagpole by only performing actions that move to the right. However, to behave optimally, moving left is important too as Mario will have to dodge/kill enemies. Furthermore, the action of 'down' and 'up' has no practical use in the first level. Therefore, we decided to import 'SIMPLE_MOVEMENT' from the game environment. There are a total of 7 actions available in each state and the explanation is:

- NOOP: no action
- right: move right
- A: jump
- right, A: jump while moving right
- right, B: dashing to the right; throw fireball if Mario acquired the item Fire Flower
- right, A, B: jump while dashing to the right
- left: move left

In the original game, the reward system is represented by a score. Mario could collect scores by collecting items, killing enemies, and completing the level in the smallest amount of time possible [2]. The reward function essentially shapes how we want our agent to behave. And ultimately, we intend to complete the level as fast as possible instead of going for the highest possible score. The provided reward function from the Gym environment is made up of three parts that allow us to shape the agent's behaviour as we intended. The reward function 'r' is:

$$r = v + c + d$$

, where v is the difference in the horizontal position of Mario between states. v returns a positive value, negative value, and 0 value when Mario moves right, left and stands still respectively.

, where c is the difference of the game timer between frames. c returns a negative value if Mario stands still.

, where d returns a negative value if Mario dies.

Basically, it rewards the agent a positive value when Mario moves right and penalises the agent when Mario stands still or dies.

In terms of transition dynamics, as the agent moves from one screen to another screen in the environment, the agent transits into the next state by observing its current state and choosing an action. The next state also returns a reward based on the action chosen. Each transition that the agent performs is also referred as an experience. So, an experience consists of a tuple of information including the current state, current action, reward, and next state. The agent can exploit and choose an optimal action based on its most recent experience or the agent can have a probability of choosing a random action to explore what will happen in the environment. When transitioning from state to state, these experiences are gradually stored in a buffer so that the agent could also sample random experiences instead of the most recent experience.

## Appendix B: Experimental Details to Replicate Results

This programme runs with Python 3.8 and Tensorflow 2.8.0. The OpenAI Gym environment for Super Mario Bros is from `https://github.com/Kautenja/gym-super-mario-bros` and is the environment that we used in this programme. It was made by Christian Kauten.

For installation of Tensorflow 2.8.0, please refer to: `https://www.tensorflow.org/install/pip#ubuntu`

For training with Nvdia GPUs, please refer to: `https://www.tensorflow.org/install/gpu`

The following dependencies and PIP packages are required for this programme:

1. apt-get install ffmpeg libsm6 libxext6 -y
2. pip3 install gym
3. pip3 install gym-super-mario-bros
4. pip3 install opencv-python

To run the program, please run python3 main.py for training the agent. To plot the reward graphs please run python3 plot_reward_epoch.py

### Appendix C: Convolutional Neural Network (CNN)

The CNN models used processed in-game frames as the input. The frames were converted from original 240 x 256 8-bit coloured to 84 x 84 grayscale images. Four consecutive frames were grouped as the input. These can help the model know how to process the changes on the screen. The latter one reduced VRAM usages in training with the GPU because only one frame was stored in case of common frames.

Two different neural networks architectures were used. One had more layers than the other and used different sizes of convolutional filters and different step sizes.

The first architectures, which is called as structure1, is described below:

1. Input: Four frames in grayscale with 84x84 pixels resolution
2. Hidden layer: Convolves 32 8x8 filters of stride 4 with the input and applies a Rectified Linear Unit(ReLU)
3. Hidden layer: Convolves 64 4x4 filters of stride 2 with the input and applies a Rectified Linear Unit(ReLU)
4. Hidden layer: Convolves 64 3x3 filters of stride 1 with the input and applies a Rectified Linear Unit(ReLU)
5. Hidden layer: Fully connected layer with 512 units and applies a Rectified Linear Unit(ReLU)
6. Output: Fully connected linear layer which outputs Q-values of each valid action (total 7 actions)

The other architectures, which is called as structure2, is described below:

1. Input: Four frames in grayscale with 84x84 pixels resolution
2. Hidden layer: Convolves 32 3x3 filters of stride 2 with the input and applies a Rectified Linear Unit(ReLU)
3. Hidden layer: Convolves 64 3x3 filters of stride 1 with the input and applies a Rectified Linear Unit(ReLU)
4. Hidden layer: Convolves 128 3x3 filters of stride 2 with the input and applies a Rectified Linear Unit(ReLU)
5. Hidden layer: Convolves 64 3x3 filters of stride 1 with the input and applies a Rectified Linear Unit(ReLU)
6. Hidden layer: Fully connected layer with 512 units and applies a Rectified Linear Unit(ReLU)
7. Output: Fully connected linear layer which outputs Q-values of each valid action (total 7 actions)

### Appendix D: Video of agent performance

`https://drive.google.com/file/d/1T43c4kj_yIeonEKaUGmuHcsxwCbs4rsD/view?usp=sharing`

### Appendix E: Video of presentation of the project

`https://drive.google.com/file/d/11qE-EAaG1r_cT89PiWRPE2wAiZBIfUhB/view?usp=sharing`