Training a deep reinforcement learning agent to play Super Mario Bros.

Charlie Mason
Department of Computer Science
University of Bath
Bath, BA2 7AY
cm956@bath.ac.uk

Arron Mcdermott
Department of Computer Science
University of Bath
Bath, BA2 7AY
am3292@bath.ac.uk

Bartlomiej Bilski
Department of Computer Science
University of Bath
Bath, BA2 7AY
bmb49@bath.ac.uk

Mark Weston-Arnold
Department of Computer Science
University of Bath
Bath, BA2 7AY
mdwa20@bath.ac.uk

I-Hsiu Chiang
Department of Computer Science
University of Bath
Bath, BA2 7AY
ihc37@bath.ac.uk

Tharadevi Rameshkumar
Department of Computer Science
University of Bath
Bath, BA2 7AY
tr719@bath.ac.uk

1 Problem Definition

For this group assignment, the challenge we have decided to solve is training a reinforcement learning (RL) agent to play the 1985 Nintendo Super Mario Bros game. This agent will learn to complete the first stage in the Mario game by learning from the game's raw pixel data in an effort to reach the flag at the end of the stage as quickly as possible. For this problem, we will be using the OpenAI Gym environment for Super Mario Bros. & Super Mario Bros. 2 (Lost Levels) on the Nintendo Entertainment System (NES) using the nes-py emulator (Kauten, 2018).

States: A state in the Mario OpenAI Gym environment is a 256x240 pixel RGB image that is equivalent to a single frame of the game.

Actions: In this environment, the agent has access to the entire NES action space of 256 potential actions, corresponding to all possible combinations of the 8 buttons on an NES controller. However, the OpenAI Gym environment allows us to constrain these actions to a smaller subset, reducing agent training time.

Transition dynamics: Super Mario Bros is a 2D platforming game, its environment is deterministic, with highly complex pixel data. The agent (the Mario character) performs actions to move across the level in the horizontal direction to reach a flagpole that marks the end of the level. The game levels contain a number of objects and obstacles that can either impede or support the agent's progress, full details of which can be found in Appendix A.

Reward function: For this project, our main focus was on making the agent finish the level as fast as possible. To achieve this, the reward function was setup such that positive rewards are obtained by the agent when it moves right (in the direction of the flagpole), whilst negative rewards were obtained at regular time intervals and when the agent dies. Appendix B contains a more detailed description of how the reward function is calculated. Due to the time constraints of this project, we did not consider

the additional challenge of making the agent obtain the highest in-game score possible, though this would be achievable through modification of the environment's reward function.

2 Background

2.1 Deep Q-learning

Deep Q-learning (DQN) is an off-policy method created by DeepMind as a variation of the tabular Q-learning algorithm, which uses a deep multi-layered Convolutional Neural Network (CNN) to approximate a state-value function (Mnih et al., 2013). This was the first deep learning model to use high-dimensional sensory input, the in-game score, to learn control policies and output an approximated Q-value for Atari 2600 computer games. They demonstrated their proposed model to play 6 deterministic Atari 2600 games, in which their results showed that their model outperformed all previous approaches and was even comparable to human players.

DQN is a suitable choice of reinforcement learning algorithm to apply to Super Mario Bros, as implementing the traditional tabular Q-learning algorithm with a Q-table would result in an immensely vast state-action space, which would be unfeasible to store in a Q-table since a Q-table for the Mario world would have to represent all of its possible permutations. As a result, implementing a tabular reinforcement learning method to solve this problem would be an ineffective approach. Therefore, we would have to rely on function approximation methods such as DQN, to approximate the Q-table.

2.2 Double Deep Q-learning

One of the flaws of DQN is that its max operator causes overestimation of action-values estimates, often resulting in less-optimal policies. This is due to the max operator using the same values to both choose and evaluate an action, and this overestimation has been observed in the Atari 2600 domain (van Hasselt et al., 2016). As such, advancements in reinforcement learning resulted in an improvement to the original DQN algorithm called Double Deep Q-learning (DDQN), which has been proposed to reduce this overestimation to yield more accurate value estimates, leading to improved model performance. DDQN accomplishes this with the use of two neural networks, one is used to determine the greedy policy, and the other is used to determine its value.

2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy and policy gradient method developed by OpenAI in 2017, which was intended to improve upon the Trust Region Policy Optimization (TRPO) algorithm (Schulman et al., 2017), due to its high complexity and high computational requirements. As a result, PPO is proposed to be more data efficient, robust and easy to implement and tune while only using first-order optimisation.

PPO is able to reduce its complexity over TRPO by adopting the "clipping" of the Surrogate objective function to a range, to try and limit how much the policy can deviate from the previous policy between updates. This is performed to ensure that updates do not cause the policy irreversible harm, making convergence to the optimal policy easier and improving training stability. Due to the surrogate function's clipping, several epochs of stochastic gradient ascent can now be applied on the sample data without causing destructive policy updates, resulting in an increase in sampling efficiency.

The authors tested their proposed PPO algorithm by training a PPO agent to play 49 different Atari games. The results of their agent significantly outperformed algorithms such as A2C and ACER.

2.4 Convolutional Neural Network

A CNN is a class of deep neural networks commonly applied to image classification and analysis. CNNs were first developed and used around the 1980s. At the time, they were only able to recognise handwritten digits and were used in the postal sectors to read zip codes, pin codes, etc. (Mandal, 2021). Due to the amount of data available, it was not until 2012 that Alex Krizhevsky revived the field with AlexNet, a deep learning model that uses multi-layered neural networks (Krizhevsky et al., 2012).

A CNN has three main types of layers; a convolutional layer, a pooling layer and a fully-connected (FC) layer. The convolutional layer is the main part of a CNN. It requires components such as the input data, a filter, and a feature map. It also consists of a feature detector known as a kernel or a filter, which moves across the receptive fields of a given image to check if a feature is present (IBM Cloud Education, 2021). Pooling happens right after the convolutional layer, where it takes groups of pixels and performs an aggregation over the pixels. By choosing max pooling, the aggregations take and extract the maximum value of the pixels in the group (Zafra, 2020). Not only does this intensify the information, it also reduces the dimensionality and size of the pixel. Lastly, the FC layer performs a classification task based on the features extracted through the previous layers and leverages an activation function to classify inputs appropriately, outputting a signal with a probability from 0 to 1 (IBM Cloud Education, 2021).

3 Method

The Gym environment allows us to have access to a number of wrapper classes that helps us to transform and constrain the environment to produce an environment that is easier to use and more convenient with less complexity. The wrappers that we used for our implementation are as follows:

1. **MaxAndSkip** - This wrapper from the stable_baselines3 library repeats the same action for a given number of frames and returns a max pooling of the last two frames. Every fourth frame is skipped in our implementation, as the difference between two consecutive frames is usually minimal and would not provide the neural network with much additional information that the previous frame did not already provide.
2. **Grayscale** - This wrapper converts the state representation from three RGB channels to a single grayscale channel. This reduces the number of pixels that need to be processed by the neural network by a factor of three, reducing the training time.
3. **Resize** - Reduces the scale of each frame to 84x84 pixels to further reduce the training time of the convolutional neural network, whilst retaining enough detail from the original image for the network to learn effectively. Converting the original 256x240 pixel image to an 84x84 image reduces the number of pixels to be processed by a factor of approximately 8.7.
4. **FrameStack** - Stacks the last four frames together. Stacked frames provide temporal information to the neural network that can help it to determine the direction in which objects are moving within the game world and subsequently provide more accurate estimates during training.

After these four wrappers are applied, the resulting observation is normalised by converting from pixel values (integers between 0 and 255) to float values within the unit range (0.0 to 1.0) before being fed into the neural network.

The neural networks used in our implementation of the DDQN algorithm take in the input states from the Mario environment and runs it through three convolutional layers each proceeded by a Re-LU activation function, followed by a flatten layer and two dense layers with a Re-LU activation function in between. Re-LU stands for rectified linear, an activation function that either produces a positive output if the input is positive or an output of zero if the input is negative (Brownlee, 2020). The dense layer receives the previous layer's outputs and merges them into a single output. (Dumane, 2020). The flatten layer is the conversion of previous input into a one-dimensional array. The last layer is a FC linear transformation layer, which takes the output from the previous layer and produces seven outputs, each of which corresponds to a single action from the environment's action space. We have chosen to use the "SIMPLE_MOVEMENT" action space in our implementation, which comprises of just seven of the most useful button combinations needed to play the Super Mario Bros. game.

Figure 1 shows the DDQN pseudocode that we used as a reference for our own implementation.

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

Equation 1 shows the Huber loss function used during the gradient descent step of our DDQN algorithm to determine how the weights of the neural network should be adjusted to minimise loss.

Algorithm: Deep Q-Learning with Experience Replay and Fixed Target Network
Initialise replay memory D to capacity N
Initialise action-value network $\hat{q}_1$ with arbitrary weights $\boldsymbol{\theta}_1$
Initialise target action-value network $\hat{q}_2$ with weights $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

For episode $= 1, M$ **do**
  Initialise initial state S_1
  For $t = 1, T$ **do**
    With probability ϵ select random action A_t
    With probability $1 - \epsilon$ select action $A_t = \text{argmax}_a \hat{q}_1(S_t, a, \boldsymbol{\theta}_1)$
    Execute action A_t, observe reward R_t and next state S_{t+1}
    Store transition (S_t, A_t, R_t, S_{t+1}) in D
    Sample random minibatch of transitions (S_j, A_j, R_j, S_{j+1}) from D
    Set $y_j = \begin{cases} R_j + 0, & \text{if } S_{j+1} \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2(S_{j+1}, a', \boldsymbol{\theta}_2), & \text{otherwise} \end{cases}$
    Perform gradient descent step $\nabla_{\boldsymbol{\theta}_1} L_\delta \left(y_j, \hat{q}_1(S_j, A_j, \boldsymbol{\theta}_1) \right)$
    Every C steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$
  End For
End For

Figure 1: Pseudocode for DDQN (Evans, 2022)

The Huber loss function was chosen over the mean squared error (MSE) loss function because Huber loss is less sensitive to outliers. This means that large errors will not have a significantly greater impact on the network weights than smaller errors, as would be the case with MSE (Seif, 2020).

4 Results

When an agent following a policy that randomly selects actions is made to play the game, the agent will either immediately die to the first enemy (Goomba) at the start of the level or get stuck at the second pipe as it cannot select an action containing the 'jump' input enough times in a row to gain sufficient height to clear the pipe. Therefore, it is extremely unlikely that the random agent will ever complete the level (although not impossible).

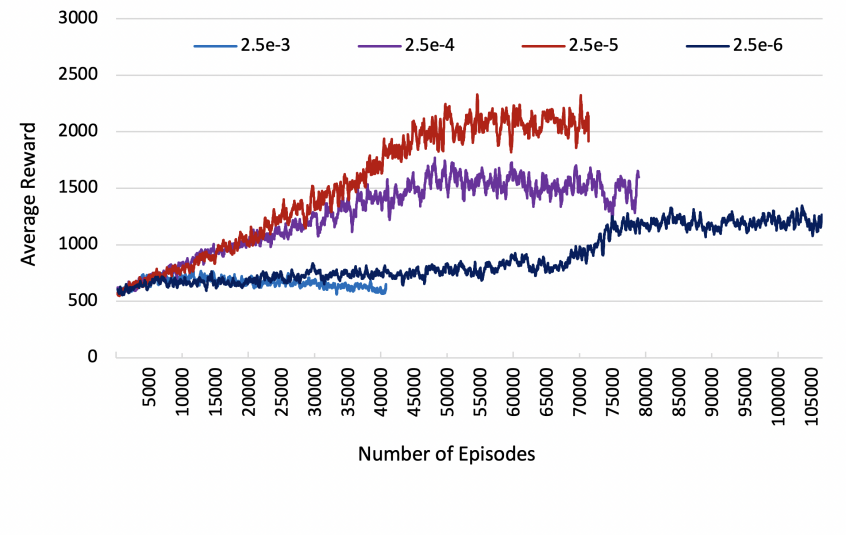

Figure 2: Average reward for 2.5e-3, 2.5e-4, 2.5e-5 and 2.5e-6 learning rates for DDQN

Figure 2 shows the average episodic rewards of our DDQN agent trained using 4 different learning rates. Firstly, the agent does not learn suitably over time when the learning rate parameter is $2.5e-3$. The average reward does not improve with the number of episodes, and the agent is likely to only achieve random successes. Decreasing the learning rate by a factor of ten significantly improves the agent's performance. With a learning rate parameter of $2.5e-4$, the average reward improves over time, increasing linearly to around 50,000 episodes and then flattening out at a peak reward of around 1600. Decreasing the learning rate parameter by another factor of ten improves performance further and the agent is able to reach a greater average reward of around 2000. Decreasing the learning rate by another order of magnitude does not give the same benefits; the learning rate of $2.5e-6$ flattened out around 1200. Therefore, the optimal learning rate is expected to lie between $2.5e-4$ and $2.5e-6$.

Studying the average loss over time elucidates the differences between the agents. With the $2.5e-3$ learning rate, the average loss in Appendix D does not decrease over time and has a very large spike in the first 5000 episodes. For a learning rate of $2.5e-4$, the average loss is shown to increase up to around 15,000 episodes, then decreases over time, plateauing to 3. The performance benefits of the smaller $2.5e-5$ learning rate parameter are revealed by the smaller average loss reached (1.8). The smallest learning rate performed worse than $2.5e-4$ and $2.5e-5$, stabilising above 3.

An agent with the best performing learning rate value was also trained in the absence of a target network, using the online network for both action selection and evaluation. Whilst we had expected that this would result in lower performance, Appendix E shows that the training results of both the DQN and DDQN agents were in fact very similar.

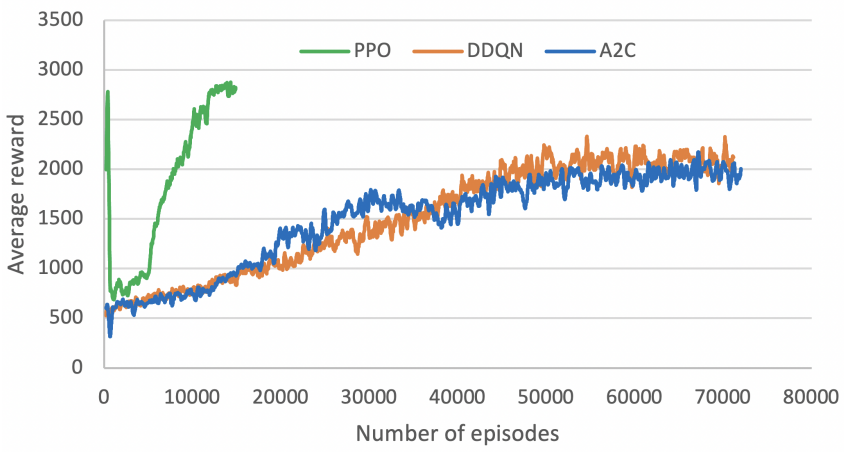

Figure 3: Comparison of DDQN with PPO and A2C, using $2.5e-5$ learning rate parameter and 0.9 discount factor.

Figure 3 shows a comparison of our DDQN algorithm against PPO and A2C (actor to critic) implementations. PPO is shown to perform significantly better than DDQN, reaching a higher average reward than DDQN in a much shorter time of only 15,000 episodes (compared to DDQN's plateau around 50,000 episodes). PPO achieves the greatest average reward (around 2900), compared to the 2200 reward attained by DDQN. The A2C algorithm performs similarly to DDQN in regards to the required number of training episodes. It learns faster up to 30,000 episodes, but then drops in performance and plateaus below DDQN's attained reward.

We next compare the performance of our trained DDQN model against the performance of the 6 members in our group. Firstly, we allowed each member in the group to practice playing the first Mario level for approximately 10 minutes to get familiar with the environment and controls. Reflecting the goal of our Mario problem, each member was then tasked with completing the first Mario level as fast as possible over 10 epochs and their results were recorded. The fastest time a member achieved was 23.9 seconds, while the group's average time to complete the level was 30.33 seconds. Our DDQN agent was able to complete the same Mario level in 19.2 seconds after being trained for ten million steps. We observe that the DDQN agent outperformed the fastest group member, having been trained a much larger number of epochs than our members and discovering

sub-optimal paths throughout the level to reduce their time. Due to the project's time constraints, we could not train the members of our group an exceedingly large of episodes to better compare against the DDQN model. Additionally, none of our members had practised nor observed speed running paths through the Mario level before attempting this task.

5 Discussion

5.1 DDQN learning rate and comparison with vanilla DQN

Previous studies have shown how DQN can be used in Atari environments by estimating the Q values (Mnih et al., 2013). Similarly with DDQN, however, with the improvement of not overestimating action-value estimates. From the results on the average rewards of the DDQN in Figure 2, we discover that changing the learning rate has a large impact on the performance of the DDQN if the learning rate changes by a factor of 10. We conclude that for this particular environment a learning rate between $2.5e-4$ and $2.5e-6$ is most optimal for our DDQN implementation. For the average reward, the DDQN was able to reach a maximum of 2000 at 50000 episodes with a learning rate of $2.5e-5$. At a learning rate of $2.5e-4$, the average reward also plateaus at approximately the same episode as $2.5e-5$ but with a much lower average reward of 1500. For the other learning rates such as $2.5e-3$ and $2.5e-6$, the agent was not able to learn properly. The agent with a learning rate of $2.5e-3$ begins to drop in reward at 30000 episodes, which shows that it is not learning at all. The rate that the agent with a learning rate of $2.5e-6$ was considered to be too slow, as it did not have a significant increase in average reward before episode 70000. It is observed that learning rates above $2.5e-4$ are considered to be too large for training, as they continue to overshoot the minimum gradient for the loss function. If the learning rate is set below $2.5e-6$, the learning rate is thought to be too small to have a substantial effect on the loss function, as seen in Figure 4. This has a direct effect on the average reward it earns and the agent would take too long to produce a satisfactory result. Therefore, $2.5e-5$ is the best learning rate out of all learning rates plotted.

Although the reasoning for why the DQN agent performed so similarly to the DDQN agent is somewhat unclear, it could perhaps be attributed to the particular environment and relatively small learning rates being used. In our DDQN implementation, the target network is synced with the online network at intervals of ten thousand steps, but because the learning rate is small, any changes to the online network's weights during this interval may be minimal. Therefore, increasing the sync interval in our DDQN implementation may make the expected performance difference between DQN and DDQN agents more observable. There is also some suspicion of the algorithm itself, as the target network's best-estimated action may not be the best action at the beginning of the game, and implementation of the target network that solves overestimation in DQN has little to no effect on the actual problem.

5.2 On-policy and off-policy algorithm comparison

To compare different algorithms, both Advantage actor-critic (A2C) and PPO algorithms are taken from the Stable Baselines3 package, using the same hyperparameter values from our DDQN implementation. The reason for choosing these algorithms for comparison is because DDQN learns its environment by taking the best action from its replay memory and target network, which the target network is an estimation/prediction from its best action taken from the first network. A2C behaves similarly to DDQN, except it contains an actor that performs an action and a critic that evaluates the actor by passing a state value function. PPO also uses a similar idea as a base for updating its policy but in a more refined manner, which will be discussed shortly. Also, another difference between the two comparison algorithms and DDQN is that the comparison algorithms are on-policy algorithms and DDQN is off-policy. Therefore, it would be interesting to see whether an on-policy or off-policy algorithm performs best in the Super Mario Bros. environment.

In Figure 3, a comparison of the PPO, A2C and DDQN algorithms is displayed with respect to the average rewards against the number of episodes. It is observed that after a certain amount of episodes passed, the PPO algorithm begins to significantly outperform both the DDQN and A2C algorithms, such that it takes considerably less time to train. The PPO algorithm was able to reach the maximum reward from the Mario environment before it stopped training. PPO significantly outperforms the other algorithms due to the effect of its advantage function. The advantage function is a measure of whether taking a particular action is a 'good' or 'bad' decision given a certain state so that the

action distribution for a particular state in the actor does not move too much and is not affected by exploding gradients. The clip ratio in the PPO algorithm is set to a value of 0.2 (in a range of 0.1 to 0.3) so that it does not move too far away from the old policy. Initially, A2C was performing similarly to DDQN and was able to surpass the performance of DDQN at around episode 20000. However, due to the algorithm's inherent structure of the A2C algorithm, it was not able to perform very well from episode 30,000 onward. This is because A2C does not have clipping implemented in its algorithm; hence some of its trajectories can significantly influence an actor's action, causing the agent to learn nothing from its environment. This can be seen between episodes 30,000 and 40,000 where the average reward begins to drop before it starts to learn effectively again at episode 40,000. One of the surprising points is that the learning curves of the A2C and DDQN agents are very similar.

6 Future Work

A potential improvement that could be made to our solution would be to remove the constraints on the action space to allow the full range of button combinations to be used by the agent, which may allow it to take more appropriate actions in a given state. For example, allowing the agent to use the 'down' button would allow Mario to utilise pipes to warp to different areas of the level. As seen from video recordings of speed-runners, warping through a pipe in World 1 Level 1-1 (the level that our agent was trained on) can reduce the time to complete the level and could allow the agent to be able compete with human speed runners for comparison.

We could also remove the frame-skipping wrapper (MaxAndSkip) from the environment, which would allow Mario to make more granular movements by performing a potentially different action every frame, rather than the same action for four consecutive frames as is currently implemented. This would significantly increase the training time, but may lead to improvements to the average reward seen by a fully trained agent.

Due to time limitations in this project, we were only able to train our agent on the first level of Super Mario Bros. Some levels have game mechanics and enemies that are not present in the first level of the game which our trained agent has not experienced before, and it would be interesting to see how well our implementation would perform on these levels, some of which are significantly more challenging. Additionally, We could suitably adjust the agent's reward function such that it aims to achieve the maximum in-game score rather than fastest level completion time, which would encourage the agent to collect coins and defeat enemies.

Dueling DQN is another variant of the DQN algorithm that we could implement in the future to potentially improve performance (Wang et al., 2016). The Dueling DQN differs in the neural network architecture, which replaces the single stream proceeding the convolutional layers with two streams; a value stream for estimating the value of a given state and the advantage stream to calculate the advantage of taking an action. These two streams then combine using a convolutional encoder and a custom aggregating layer to produce an estimate of the state-action value function Q. This modification to the network architecture will allow the Dueling DQN to generalise across actions.

7 Personal Experience

A few of our team members had never used neural networks in practice before and had only a foundational knowledge of their concepts. This project allowed them to gain practical experience with frameworks such as OpenAI Gym and PyTorch, although they did feel that there was quite a steep learning curve for newcomers to these tools. When writing the code, getting the state representation into the right format to be fed into the neural network took a lot of time to get working correctly, although it was ultimately very rewarding to see that our trained agent performed well in the Mario environment. Also, because of the complexity of our chosen environment, training our agents took a long time, which made the code difficult to debug and resulted in a number of days spent training an agent that ultimately showed no learning.

As some members of our team were on different courses with different optional module choices, it was difficult to coordinate meetings due to conflicting schedules, but we were still able to meet up frequently enough to ensure that the project continued to progress relatively smoothly.

References

Brownlee, J. (2020), 'A gentle introduction to the rectified linear unit (relu)'.
URL: *https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/*

Dumane, G. (2020), 'Introduction to convolutional neural network (cnn) using tensorflow'.
URL: *https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83*

Evans, J. (2022), 'Deep reinforcement learning - part 2: Dqn deep-dive', University of Bath. Unpublished.

Feng, Y., Subramanian, S., Wang, H. and Guo, S. (n.d.), 'Train a mario-playing rl agent'.
URL: *https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html*

IBM Cloud Education (2021), 'Convolutional neural networks'.
URL: *https://www.ibm.com/cloud/learn/convolutional-neural-networks*

Kauten, C. (2018), 'Super Mario Bros for OpenAI Gym', GitHub.
URL: *https://github.com/Kautenja/gym-super-mario-bros*

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012), 'Imagenet classification with deep convolutional neural networks', *Advances in neural information processing systems* **25**.

Mandal, M. (2021), 'Introduction to convolutional neural networks (cnn)'.
URL: *https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/*

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

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

Seif, G. (2020), 'Understanding the 3 most common loss functions for machine learning regression'.
URL: *https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3*

van Hasselt, H., Guez, A. and Silver, D. (2016), 'Deep reinforcement learning with double q-learning', *Proceedings of the AAAI Conference on Artificial Intelligence* **30**(1).
URL: *https://ojs.aaai.org/index.php/AAAI/article/view/10295*

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N. (2016), Dueling network architectures for deep reinforcement learning, *in* 'International conference on machine learning', PMLR, pp. 1995–2003.

Zafra, M. F. (2020), 'Understanding convolutions and pooling in neural networks: a simple explanation'.
URL: *https://towardsdatascience.com/understanding-convolutions-and-pooling-in-neural-networks-a-simple-explanation-885a2d78f211*

Appendices

Appendix A: Environment Transition Dynamics

- **Pipes**: An obstacle which the agent must jump over. Pipes can also be used by agents to "warp" between different areas of the level, thereby allowing the agent to complete the level faster or with a higher score in some cases.
- **Brick blocks**: An obstacle or platform that can be destroyed if the agent jumps up and collides with the bottom of the block.
- **Question blocks**: Can generate coins or power ups that can give the agent extra lives and abilities.
- **Pits**: Holes in the level which an agent must jump over. Falling into a pit will result in the agent dying and losing a life.
- **Enemies**: There are multiple enemies, such as Goombas, Koopa Troopas and Piranha Plants. The agent will die if it collides with an enemy whilst not under the effect of a power up.
- **Coins**: The agent can collect these to increase the in-game score.
- **Flagpole**: The destination that the agent will need to reach to complete the level. This pole is located at the far right of the level.

Appendix B: Reward Function

- **r = v + c + d** - The equation for the output reward after a state transition.
- **v** - The agent's instantaneous velocity (velocity is positive when the agent is moving right)
- **c** - The penalty for not moving, calculated via the difference in the game clock between frames. When the game clock is decremented between two frames, a penalty of -1 is applied to the reward function.
- **d** - The death penalty of an agent in a state. d = 0 if alive and d = -15 when dead.

Appendix C: Experimental Details and Hyperparameters

The Zip file accompanying this report contains all of the source code that can be used to replicate the results of the experiments presented in this report. To run the code, a number of dependencies will need to be installed including OpenAI Gym, the Super Mario Bros Gym environment (Kauten, 2018), PyTorch, stable_baselines3 and numpy.

Additionally, the implementation of our agent has been written to be trained on an Nvidia GPU to benefit from its CUDA technology. To run the code on a machine without an Nvidia GPU and instead train on the CPU, it may be neccessary to make minor modifications such as removing ".cuda()" from all tensors when they are created.

Within each of the source files, the various hyperparameters have kept the values that were used during our own experiments, and so should produce repeatable results. The hyperparameters themselves were chosen by first observing the values used by other DQN/DDQN implementations for the same Super Mario Bros environment that were found online (Feng et al., n.d.), whilst adjusting them where necessary such as in cases of memory requirements being exceeded (the size of the replay buffer in our DDQN implementation initially exceeded the GPUs available memory and so had to be reduced). This was to ensure that we didn't lose a significant portion of the project time to training agents that would ultimately not demonstrate any learning. We initially started training agents with a fixed value of epsilon, but later introduced a decay to the value of epsilon that started at a value of 1.0 and eventually decayed to a value of 0.1 after approximately nine million training episodes. Changes to the learning rate were also experimented with, as shown in Figure 2, in an attempt to converge on a more suitable value for the learning rate that resulted in more efficient learning and improved agent performance. The potential effects of changing other hyperparameter values could not be investigated during this project due to the significant amount of time that it takes to sufficiently train each agent.

Appendix D: DDQN loss graph

Appendix E: DQN vs DDQN

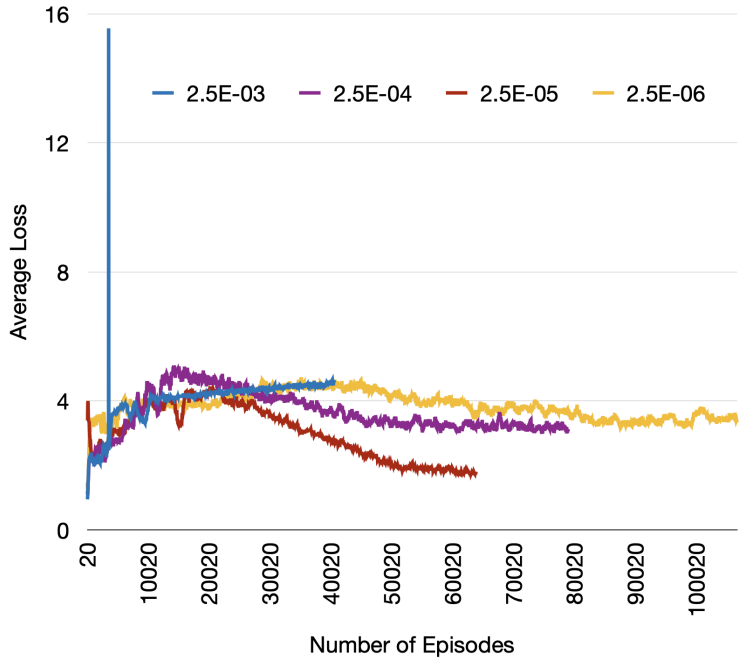

Figure 4: Average loss for 0.0025, 0.00025, 0.000025 and 0.0000025 learning rates.

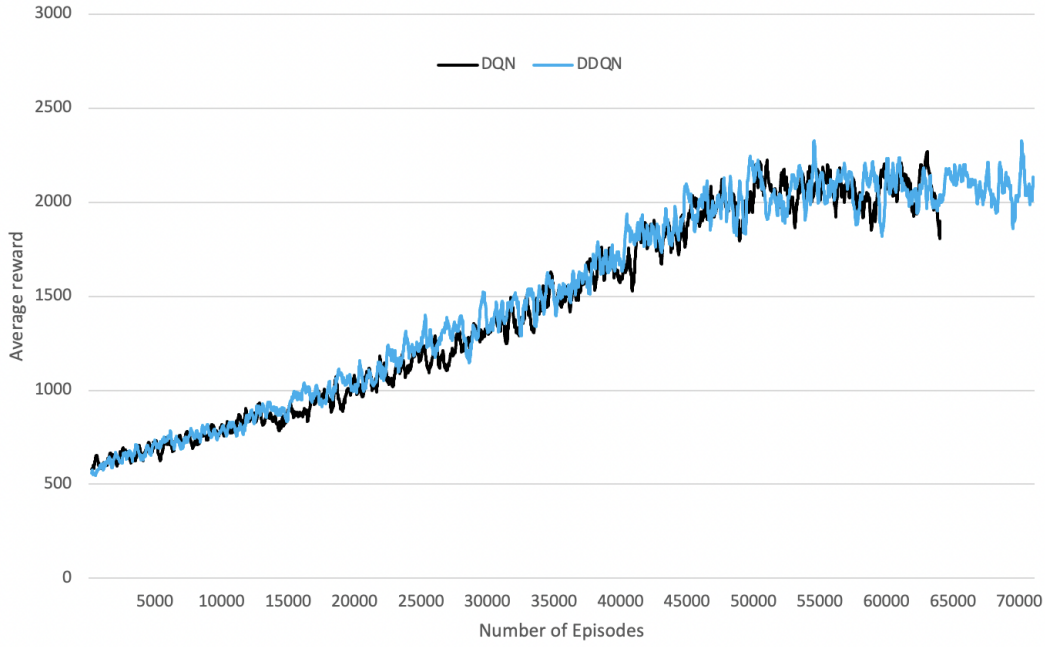

Figure 5: Comparison of DQN agent vs DDQN agent, both trained with the same parameter values