CS4287 Neural Computing
Assignment 3: Deep Q Networks (DQNs) Atari

Dominion Gbadamosi: 20243561

1. Why use Reinforcement Learning?

Reinforcement learning (RL) is the machine learning paradigm of choice for training an Atari game for a number of reasons.

In relation to Atari games, RL's capability of handling sequential decision making is crucial (Mnih et al., 2013). The agent must make a series of decisions and the actions it selects influence the game environment. Through the assignment of rewards and penalties, the agent optimises its decision making process, as it aims to maximise the reward. This sequential learning process is fundamental to successfully play Atari games. RL algorithms such as Q-learning, are incredibly efficient in adapting to these environments (Sutton & Barto, 2018).

A key advantage RL has over other machine learning paradigms such as supervised and unsupervised learning is that it learns by trial and error rather than being fed inputs and outputs to learn from. In Atari games, where there is no explicit data input or output, RL allows agents to learn from their interactions with the environment (Watkins & Dayan 1992). RL algorithms also handle infrequent rewards adeptly, which is a common thing in Atari games. The algorithms can learn to associate actions with delayed or infrequent rewards (Mnih et al. 2015).

Having a balance between exploration and exploitation is crucial. RL algorithms such as the epsilon-greedy approach incorporate mechanisms to explore the environment and exploit strategies that have proven to be successful (Sutton & Barto, 2018). The flexibility of RL allows it to be highly adaptable to a variety of different tasks which makes it suitable for training agents to play different Atari games (Sutton & Barto, 2018).

In summary, reinforcement learning stands out as the machine learning paradigm of choice due to its ability to handle sequential decision making, trial and error learning, infrequent rewards, the exploration-exploitation tradeoff, and its flexibility. These factors and many more make it a powerful tool that can be used to master many complex gaming environments.

2. The Gym Environment

Chosen Atari Environment: Breakout-v4

Game Description
Breakout is an Atari game where you move a paddle and hit a ball at a brick wall on top of the screen. The goal of the game is to destroy the brick wall.

Gym
This Atari environment was provided by Gym, which is an API created by Open AI for reinforcement learning, containing a diverse collection of reference environments such as, Atari, board games, 2D/3D physical simulations etc. It is also possible to design your own custom reinforcement learning environments (Gym, 2022).

The following formation was taken from the official Gym documentation in relation to the Breakout environment (https://www.gymlibrary.dev/environments/atari/breakout/):

Creating the Environment
In order to create an instance of the Breakout-v4 environment the Gym library provides a *make* function.

```
# Create the Breakout environment
env = gym.make('Breakout-v4', render_mode='rgb_array')
```

This line creates an instance of the Breakout-v4 environment and specifies the render mode to be an RGB array. After the environment is created, it can take actions, receive observations and obtain rewards, which is essential for training and evaluating the RL agent.

Action Space
Breakout-v4 has an action space of four discrete actions. In the documentation, the mapping and meaning of these actions are provided.

Action 0: **NOOP** (No Operation): The agent will take no action.
Action 1: **FIRE**: This initiates the game or fires the ball.
Action 2: **RIGHT**: Moves the paddle to the right.
Action 3: **LEFT**: Moves the paddle to the left.

Num	Action
0	NOOP
1	FIRE
2	RIGHT
3	LEFT

Figure 1. Breakout-v4 Action Space

When working with this environment, the agent can choose one of these actions at each time step, with the goal being for the agent to learn a policy that maximises the cumulative rewards over multiple episodes (instances of the game) by taking the appropriate actions necessary to break the bricks and achieve a high score.

Observation Space

The observation space refers to the space of all possible observations that the agent can receive from the environment. Observations represent the state of the environment and are given to the agent, to make decisions about the actions to take.

Observation Space	(210, 160, 3)
Observation High	255
Observation Low	0

```
Box([0 ... 0], [255 ... 255], (128,), uint8)
```

Figure 2. Observation space of the Breakout Environment

The Breakout environment's observation space is defined as a 3D box, meaning it's an array with three dimensions. The specific definition of this array is: Box(0, 255, (210, 160, 3), uint8)

Broken down this means:

- The minimum value of the observation space is 0 pixels
- The maximum value of the observation space is 255 pixels
- The shape of the space is (210, 160, 3), representing a 3D array with dimensions of 210 pixels in height, 160 pixels in width, and 3 colour channels (RGB)
- The data type of the elements in the observation space is uint8 (unsigned 8-bit integers), meaning each pixel value can range from 0 to 255

The agents can use this information from the observation space to learn and make decisions about the state of the game, like the position of the paddle, the position of the ball and the layout of the bricks on the screen.

3. Implementation

(a). Capture and Preprocessing of the data

Capture

After the environment is created, it is reset to its initial state using the *reset* method provided by Gym, and the state variable captures the initial observation of the Breakout environment.

```
# Reset the environment
state = env.reset()
```

Figure 3. Reset the environment to its initial state

The Action and Observation spaces are then printed out, just to confirm that the environment is behaving as expected. The expected output for the Action and Observation spaces are Discrete(4) and Box(0, 255, (210, 160, 3), uint8) respectively, as covered in the last section.

```
# Action and observation spaces
print("Action space:", env.action_space)
print("Observation space:", env.observation_space)
```

Figure 4. Code to print the action and observation spaces

Here the max number of steps per episode (a full game of breakout) and the reward range are printed out.

```
# Maximum episode steps
print("Max episode steps:", env.spec.max_episode_steps)

# Reward range
print("Reward range:", env.reward_range)
```

Figure 5. Code to print the Max Steps per episode + the range of rewards

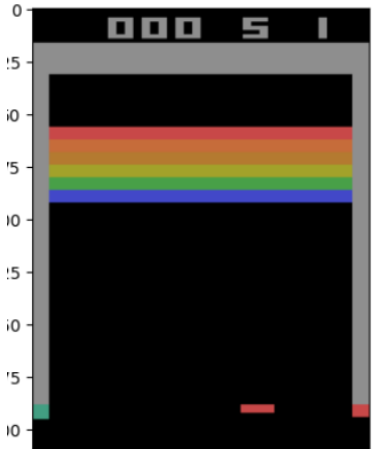
The Breakout environment is displayed after the earlier reset. This is a visualisation using the Matplotlib library to show that the environment is being captured correctly.

```
# Display the Breakout environment after the reset
plt.imshow(env.render())
plt.show()
```

Figure 6. Code to plot the a frame of the Breakout Environment

The results of the previous code are in the figures below:

```
Action space: Discrete(4)
Observation space: Box(0, 255, (210, 160, 3), uint8)
Max episode steps: None
Reward range: (-inf, inf)
```



Figures 7 & 8 Results of Print Statements, Rendered Breakout Environment

The Action and Observation spaces were captured correctly, displaying the correct values for the Breakout environment.
The value 'None' for the maximum episode steps implies that there is no limit on the number of steps per episode which makes sense, as in Breakout the episode will only terminate if the player runs out of lives. So in each case the episode will continue until that condition is met.
The reward range means that there is no upper or lower bound on the reward values, so the rewards can take any real value.
The plot of the frame captured of the environment is included in the output also, which shows that the game environment has been successfully created.

Preprocessing

The *preprocess_state* function defines the steps taken for the state data captured from the environment to be preprocessed into a format suitable for training a neural network.

```python
def preprocess_state(self, state):
    # Extract the state from the tuple if needed
    if isinstance(state, tuple):
        state = state[0]

    # Ensure that the state is in the correct format (RGB image)
    if len(state.shape) == 3 and state.shape[2] == 3:
        # Convert RGB to grayscale
        state = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)

    # Resize the state
    state = cv2.resize(state, (84, 84))

    # Normalize pixel values to the range [0, 1]
    state = state / 255.0

    return state
```

Figure 9. Defining the preprocess_state method

The steps include:

- Extraction of the data from a tuple if it needs to be. (Sometimes the state is stored within

tuples, and extracting the state ensures that the processing steps are being done on the state itself and not the tuples).

- Making sure the state is in the correct format (RGB image) as maintaining consistency in the observation space is necessary when dealing with an environment that will be providing a number of different observations.
- Converting that state from RGB into grayscale. (Converting the image to grayscale means that less computational resources will be used than when dealing with RGB images and important visual information is still retained. This makes it easier for the neural network to learn and reduces training time).
- Resizing the state to (84, 84) pixels. (Neural networks often require specific input sizes, so resizing the state to a smaller dimension reduces the computational cost and memory requirements. Additionally, it helps the network to generalise more efficiently.
- Normalising the range of pixel values from [0, 255] to [0, 1]. This helps gradient based optimisation algorithms (e.g. Adam, RMSprop) to converge more effectively.

(b). The Network Structure

```
def build_model(input_shape, num_actions):

    tf.random.set_seed(42) #set a specific seed value

    # Neural network architecture for the dueling DQN
    input_layer = Input(shape=input_shape)
    x = Flatten()(input_layer)
    x = Dense(512, activation='relu')(x)

    # Value stream
    val_stream = Dense(256, activation='relu')(x)
    value = Dense(1)(val_stream)

    # Advantage stream
    adv_stream = Dense(256, activation='relu')(x)
    advantage = Dense(num_actions)(adv_stream)

    # Combine value and advantage streams
    mean_advantage = concatenate([value, advantage])

    # Duelling layer
    dueling_layer = Dense(num_actions + 1, activation='linear')(mean_advantage)

    model = Model(inputs=input_layer, outputs=dueling_layer)
    model.compile(optimizer=Adam(learning_rate), loss='mse')

    return model
```

Figure 10. Code for the Dueling DQN Network Structure

The network structure is that of a Duelling Deep Q Network (DQN). The Duelling DQN introduces a modification to the traditional DQN by separating the estimation of the state value and the advantages. A traditional DQN outputs the Q-values for each action without separating value and advantage. The separation of the state value and the advantages in the Duelling

DQN is motivated by the goal of improving the efficiency of learning in RL tasks. It can contribute to more effective learning by a number of means, such as:

Enhancing Exploration:
The duelling architecture can enhance exploration by providing meaningful feedback to the learning algorithm. The network can learn to identify states that can maximise the rewards using the value stream (which also helps it to generalise better across different states), and which actions are most advantageous in the states (by using the advantage stream). This separation allows for the creation of more efficient/effective exploration strategies.

Faster Convergence:
Separating the state value and advantages can facilitate convergence during training. The network can divert its focus to learning the state values, which are less varying as they don't change as frequently, while separately learning the advantages associated with each action. Having a targeted approach like this can speed up the convergence process by a great amount.

Reduced Variance in Learning:
Traditional DQNs directly approximate the action values for each state, and in situations where the advantages of different actions are highly variable (like in Atari), this can lead to variance in the learning signal. By separating the state value and advantages, Duelling DQNs can reduce this variance in the learning process, allowing it to be more stable.

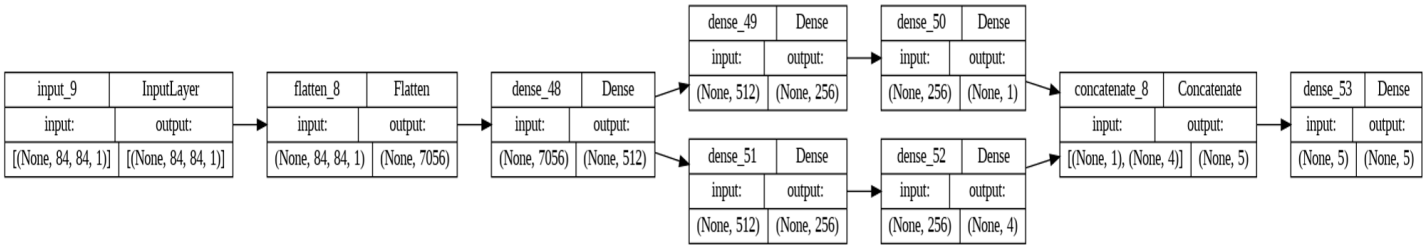


Figure 11. Diagram of Network Structure (generated using TensorBoard)

Duelling DQN Architecture:

1. **Input Layer:**
    - The input layer's shape is the preprocessed observations space of the Breakout Environment, which is (84, 84, 1) meaning 84x84 pixels and it has 1 colour channel (grayscale).
2. **Flattening Layer**
    - Flattens the input layer into a 1-D array.
    - The output is a 1-D array.
3. **Dense Layer (512 units, ReLU activation)**
    - Process flattened input with a fully connected layer.
    - Output is 512 units
4. **Value Stream**
    - **Dense Layer** (256 units, ReLU activation):
        - Extracting features related to the value/advantage of being in a certain state.
    - **Dense Layer** (1 unit):
        - Outputs the estimated state value.
5. **Advantage Stream**
    - **Dense Layers** (256 units, ReLU activation):
        - Extracting features related to the advantage of each action.
    - **Dense Layer** (Number of Actions units):
        - Number of action units corresponds to the number of actions in the Breakout environment (NOOP, FIRE, LEFT, RIGHT)(4)
        - Outputs estimated advantage values for each of the possible actions
6. **Combining Value and Advantage Streams**
    - Combining the value and advantage streams to get a more accurate Q-value estimation.
7. **Duelling Layer**
    - **Dense Layer** (Number of Actions +1 units, Linear Activation):
        - Produces the final Q values, including the baseline average
8. **Model Compilation:**
    - **Optimiser:** Adam optimiser with a learning rate of 0.001 (can be adjusted)
    - **Loss Function:** Mean Squared Error (MSE)

(c). Q-Learning Update Applied to Weights

In order to discuss where the Q-learning update is applied to the weights there are some key components in the code that I'll cover first.

DQNAgent Class:

The *DQNAgent* is initialised/defined with various parameters including, the size of the state and action spaces, a replay memory buffer, a discount factor (gamma), exploration rate parameters (epsilon), and two neural network models (the model and target model)

```
# Define the agent
class DQNAgent:
    def __init__(self, state_size, action_size):
        # Agent initialization (More Hyperparameters)
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=10000)  # Replay memory buffer
        self.gamma = 0.99  # Discount factor for future rewards
        self.epsilon = 1.0  # Exploration-exploitation trade-off
        self.epsilon_decay = 0.995  # Decay rate for exploration
        self.epsilon_min = 0.01  # Minimum exploration rate
        self.model = build_model(state_size, action_size)  # Main DQN model
        self.target_model = build_model(state_size, action_size)  # Target DQN model
```

Figure 12. DQNAgent Hyperparameters

The following are methods in the *DQNAgent class and their purpose:*

The *preprocess_state* method (as previously mentioned) is used to convert the game state into a grayscale image, resize it to 84x84 pixels, and normalise the pixel values in the range of [0, 1] (Fig. 8)

The *remember* method stores experiences (state, action, reward, next_state, done) in the replay memory.

```
def remember(self, state, action, reward, next_state, done):
    # Extract the actual state and next_state from the tuple if needed
    state = state[0] if isinstance(state, tuple) else state
    next_state = next_state[0] if isinstance(next_state, tuple) else next_state

    # Store experience in replay memory
    self.memory.append((state, action, reward, next_state, done))
```

Figure 13. Extracting states to be used in replay memory

The *act* method implements the epsilon-greedy (ϵ-greedy) approach for action selection. With a probability of 'epsilon', a random action is chosen, otherwise the action with the maximum Q-value predicted by the current model is selected.

```python
def act(self, state):
    # Exploration-exploitation trade-off when selecting an action
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)

    # The state is a tuple, so extract the actual state
    state = state[0] if isinstance(state, tuple) else state

    act_values = self.model.predict(state)

    # Ensure the action is within the valid range
    action = np.argmax(act_values[0])
    action = np.clip(action, 0, self.action_size - 1)

    return action
```

Figure 14. Code for act method, used to implement ϵ-greedy

Q-Learning Update in the *replay* method

Experience replay:

- The agent takes a sample of experiences (*minibatch*) from its replay memory.

Q-Value Calculation:

- For each experience in the minibatch, it calculates the target Q-value using the current Q-network (*self.model*).
- When the episode is complete (done = True), the target Q-value for that action is set to the reward.
- If the episode is not complete, it calculates the Q value using the target Q-network (*self.target_model*) for the next state (*next_state*)

Bellman Equation:

- The target Q-value is calculated using the Bellman equation:

$$Q(s,a) = R + \gamma \cdot maxa\prime Q(s\prime, a\prime)$$

Figure 15. Formula for the Bellman equation

- Where R is the immediate reward, γ (gamma) is the discount factor, s' is the next state and a' is the action that maximises the Q-value for the next state.

Training the Model:

- The target Q-value is used to update the Q-value for the chosen action in the current state.

- The model is then trained for one epoch using the current state it's in and the updated target Q-values
- The Keras *fit* method is used for training, where the input is the current state, and the target is the update Q-values.

Error Calculation:

- The error between the predicted and target Q-values is calculated implicitly during the training process, and the model's weights are adjusted in order to minimise this error. The error is the difference between the predicted Q-value and the target Q-value for a given state-action pair.

```
    def replay(self, batch_size):
        # Experience replay to train the agent
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = self.model.predict(state)
            if done:
                target[0][action] = reward
            else:
                a = self.model.predict(next_state)[0]
                t = self.target_model.predict(next_state)[0]
                target[0][action] = reward + self.gamma * t[np.argmax(a)]
            self.model.fit(state, target, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def update_target_model(self):
        # Update target model weights with main model weights
        self.target_model.set_weights(self.model.get_weights())
```

Figure 16. The replay buffer & where the target Q values are updated

(d). Independently Researched Concepts

(i) Random Seed Initialisation:

Random seed initialisation is a common practice used in machine learning to ensure results are reproducible. This is very important as there are many aspects of training a machine learning model that involve randomness (e.g., initialising weights, shuffling data). Setting a random seed ensures that these random processes are able to produce the same results each time the code is run. This is essential for debugging and evaluation of the model.

```
tf.random.set_seed(42) #set a specific seed value
```

Figure 17. Random seed initialisation

(ii) Impact of Regularisers on Scores:

Regularisation is a machine learning technique employed to mitigate overfitting in models (Hastie, et al., 2009). Two common regularisation types are L1 regularisation known as Lasso, and L2 regularisation known as Ridge (Hastie, et al., 2009).

L1 regularisation introduces the absolute values of coefficients as a penalty term in the loss function, which encourages sparsity in the weight vectors (Hastie, et al., 2009). This can lead to feature selection, where less important features have zero weights (Hastie, et al., 2009). However, L2 regularisation adds the squared values of the coefficients to the loss function, penalising large weights without enforcing exact zero values (Hastie, et al., 2009).

The impact of regularisers on scores during training is significant. Models with regularisation often exhibit lower training scores, showing a reduced tendency to fit training data closely. This penalty term encourages a more generalised model, enhancing its ability to generalise new, unseen data (Bishop, 2006).

Moreover, regularisation simplifies machine learning models by reducing the magnitudes of training weights. Particularly, L2 regularisation can effectively handle multicollinearity by preventing weights from becoming too large (Bishop, 2006).

Regularisation is a fundamentally used tool in training a machine learning model, to make sure they are robust and effective (Hastie et al., 2009; Bishop, 2006).

(iii) Techniques to Counter Catastrophic Forgetting and Maximisation Bias

Catastrophic Forgetting:

Catastrophic forgetting is a phenomenon in machine learning where a model that is trained sequentially on multiple tasks, tends to forget information from earlier tasks when learning new ones. As the model updates its weights during the training of new tasks, it may unintentionally overwrite previously learned patterns, leading to a drop in performance on earlier tasks.

Techniques to Counter Catastrophic Forgetting:

1. **Replay**
    - This involves periodically revisiting old data during training to prevent the model from forgetting previously learned information.
    - Storing subsets of pasta data/using a combination of old and new data in mini batches during training helps mitigate catastrophic forgetting.

```python
def replay(self, batch_size):
    # Experience replay to train the agent

    # Sample a minibatch from the replay memory
    minibatch = random.sample(self.memory, batch_size)

    # Iterate through the minibatch
    for state, action, reward, next_state, done in minibatch:
        # Calculate the target Q-value
        target = self.model.predict(state)

        # Update the target Q-value based on whether the episode is done
        if done:
            target[0][action] = reward
        else:
            # Calculate the Q-values for the next state from both the model and target model
            q_values_next_state_model = self.model.predict(next_state)[0]
            q_values_next_state_target = self.target_model.predict(next_state)[0]

            # Update the target Q-value using the Q-value of the action with the highest Q-value in the next state
            target[0][action] = reward + self.gamma * q_values_next_state_target[np.argmax(q_values_next_state_model)]

        # Update the model using the calculated target Q-value
        self.model.fit(state, target, epochs=50, verbose=0)

    # Update epsilon for epsilon-greedy exploration
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Figure 18. Updated replay buffer method (used to counter catastrophic forgetting)

2. **Progressive Neural Networks (PNN)**
    - PNNs maintain a network for each task, and when faced with a new task, a new network is added (Fayek, Cavedon and Wu, 2020).
    - Connections between the old and new networks are established to transfer knowledge, which prevents catastrophic forgetting (Fayek, Cavedon and Wu, 2020).

Maximisation Bias

Maximisation Bias is a cognitive bias in reinforcement learning where an agent, driven by the goal of maximising immediate rewards, tends to neglect exploration of less familiar actions. This can hinder the agent from discovering optimal policies, resulting in suboptimal decision making.

Techniques to Counter Maximisation Bias

1. **Exploration Strategies**
    - Using techniques to encourage the agent to explore less familiar states, reducing the tendency to exploit a suboptimal policy (Sutton & Barto, 2018).
    - Techniques like the epsilon greedy approach balance exploration and exploitation (Sutton & Barto, 2018).

```python
def act(self, state):
    # Exploration-exploitation trade-off when selecting an action
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)

    # The state is a tuple, so extract the actual state
    state = state[0] if isinstance(state, tuple) else state

    act_values = self.model.predict(state)

    # Ensure the action is within the valid range
    action = np.argmax(act_values[0])
    action = np.clip(action, 0, self.action_size - 1)

    return action
```

Figure 19. Act method

2. **Intrinsic Motivation**
    - The introduction of signals to encourage the agent to explore novel states based on curiosity/novelty.
    - This counters maximisation bias by promoting exploration beyond immediate rewards.

4. Plots/Results

During the training process, I included three live plots in the loop to give me updates on the learning rate (reward per episode) along with the average reward over the previous 10 episodes, the exploration rate, and the length of each episode (to show how long the agent lasts in the breakout game per episode). I also made sure to include a number of metrics in the output to assess the quality of the training. I've also included a video of the training stage (attached in the submission)

1. The **reward/score** to show whether the agent lost or not
2. the **value of epsilon** to display the rate of exploration vs exploitation
3. The **Q-values** to show if the agent is getting better at predicting expected rewards
4. The size of the **Experience Replay Buffer:** to know when it's almost at capacity and how many past experiences it's using to learn from (capped at 10000)

Plots/Results Output

Episode 10

```
Episode: 10/300, Score: -3.0, Epsilon: 0.95
1/1 [==============================] - 0s 17ms/step
Q-values: [[ 0.04881438  0.04879528  0.04952756  0.04892961 -0.10657103]]
Loss: None
Experience Replay Buffer Size: 2171
Episode Done. Total Reward: -3.0
Exploration Rate (Epsilon): 0.9511101304657719
```

Figure 20. Episode 10 output

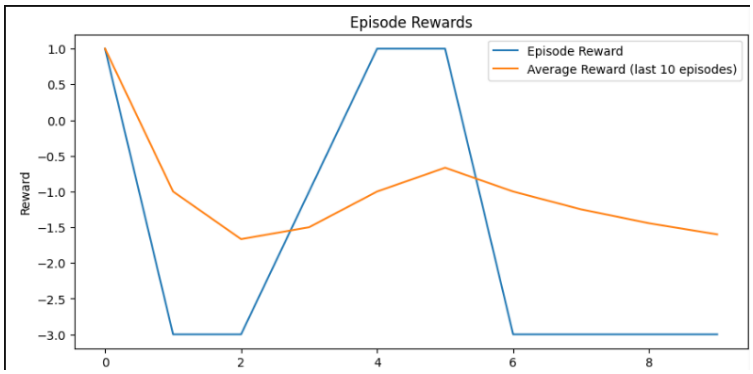


Figure 21. Episode 10 learning curve

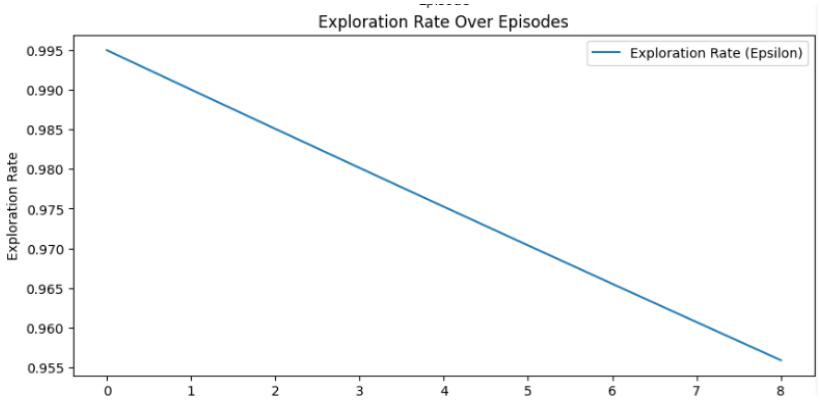


Figure 22. Exploration rate of Episode 10

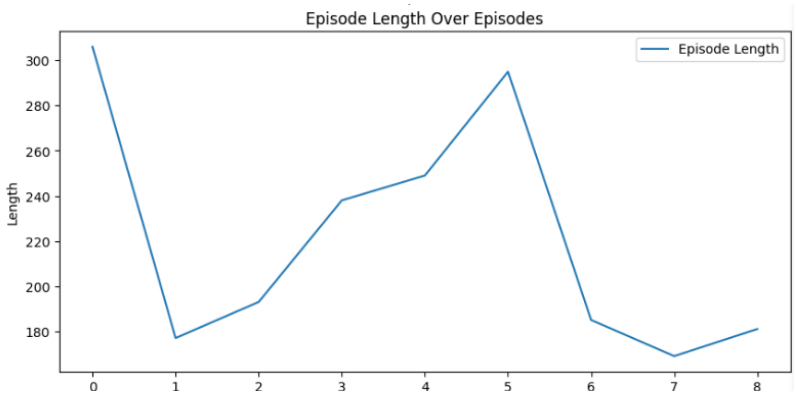


Figure 23. Episode length over episodes (after 10)

Episode 100

```
Episode: 100/300, Score: -1.0, Epsilon: 0.61
1/1 [==============================] - 0s 18ms/step
Q-values: [[ 1.5209687   1.5667877   1.5209359   1.5118533  -0.54856974]]
Loss: None
Experience Replay Buffer Size: 10000
Episode Done. Total Reward: -1.0
Exploration Rate (Epsilon): 0.6057704364907278
```

Figure 24. Episode 100 output

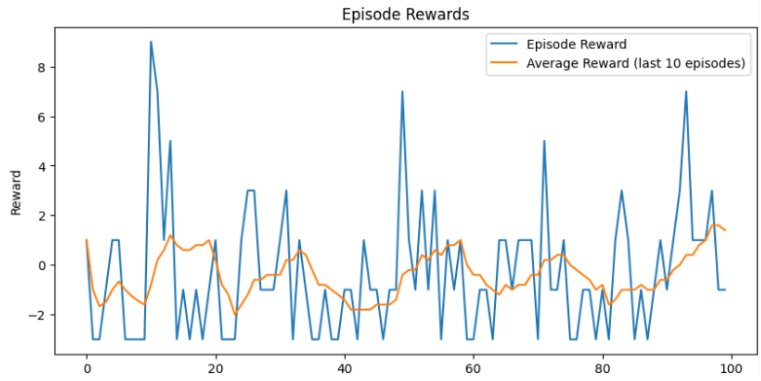


Figure 25. Learning curve after 100 episodes

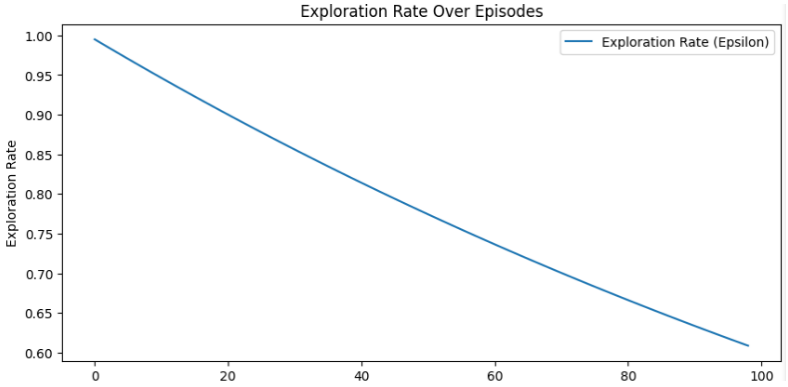


Figure 26. Exploration Rate after 100 episodes

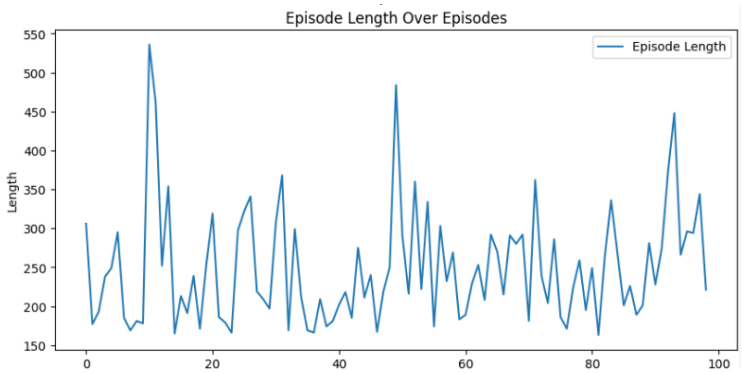


Figure 27. Episode length per episode (after 100)

Episode 200

```
Episode: 200/300, Score: 1.0, Epsilon: 0.37
1/1 [==============================] - 0s 15ms/step
Q-values: [[ 1.8904792   1.8900787   1.8899893   1.8892667  -0.70218915]]
Loss: None
Experience Replay Buffer Size: 10000
Episode Done. Total Reward: 1.0
Exploration Rate (Epsilon): 0.3669578217261671
```

Figure 28. Episode 200 output

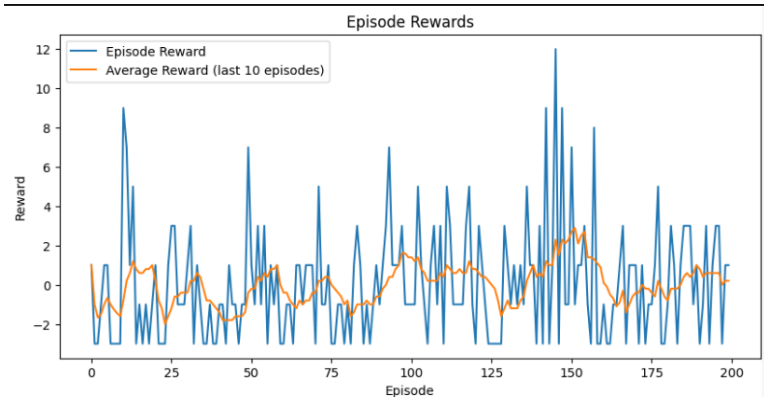


Figure 29. Learning curve after 200 episodes

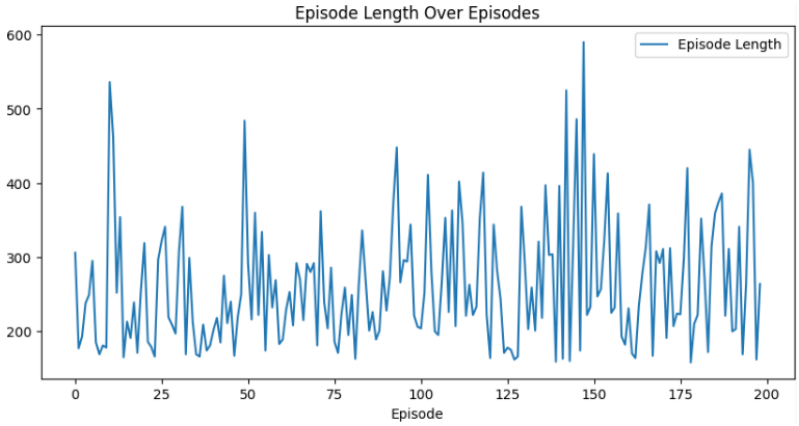


Figure 30. Episode Length per episode (after 200)

Episode 300

```
Episode: 300/300, Score: -1.0, Epsilon: 0.22
1/1 [==============================] - 0s 16ms/step
Q-values: [[ 1.8759763  1.8751003  1.8758181  1.8770015 -0.6540594]]
Loss: None
Experience Replay Buffer Size: 10000
Episode Done. Total Reward: -1.0
Exploration Rate (Epsilon): 0.22229219984074702
```

Figure 31. Episode 300 output

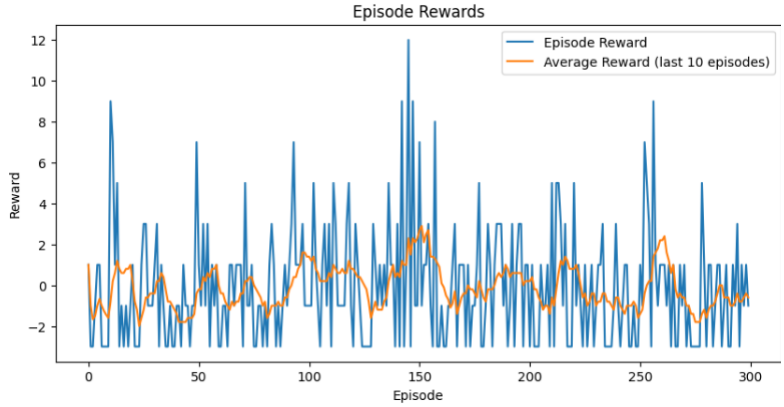


Figure 32. Learning Curve after 300 episodes

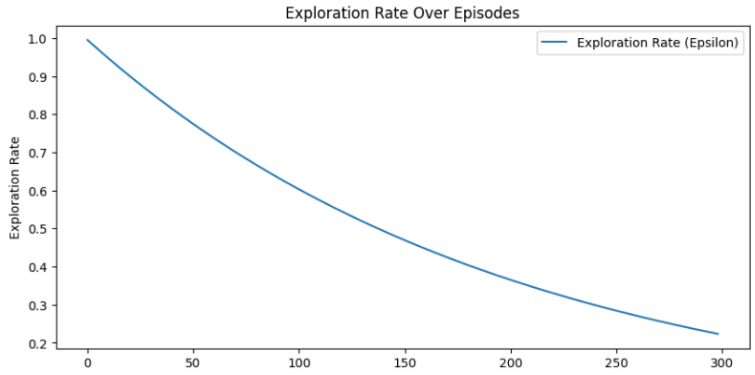


Figure 33. Exploration rate after 300 episodes

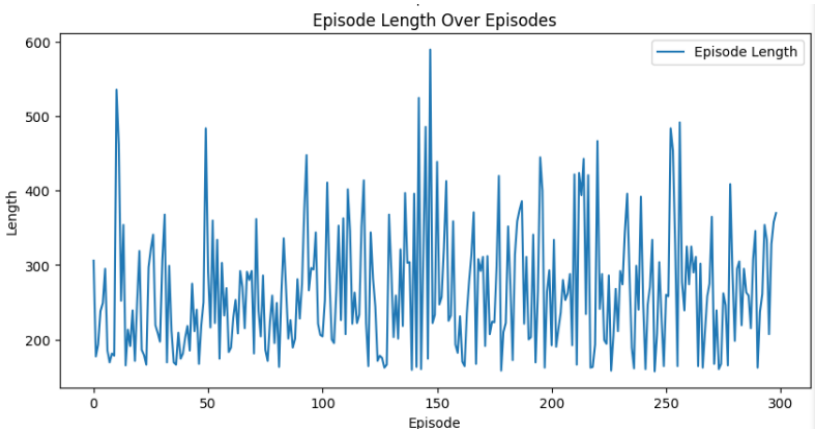


Figure 34. Episode Length per episode (after 300)

5. Evaluation

(a). Evaluating performance of the RL Agent

Overall, I believe that the RL agent performed well. There are signs that it is learning although it may not be clear by simply looking at the graphs, which is exactly why I chose to include certain metrics in the output.

Proof of Learning:

Q-values

The Q-values represent the expected future rewards for each action in a given state. To show signs of learning, the Q-values would be expected to increase overtime as the agent learns. In the output logs of the Jupyter Notebook, there is a noticeable increase in the Q values from start to finish. For example, if we compare the Q-values at episode 10 (Fig.20) with those of episode 300 (Fig. 31), we see that the Q-values of episode 10 are relatively small in comparison and some are even negative, while episode 300's are much higher indicating the agent has begun to learn better action values.

Exploration rate:

The decreasing trend in the graph is expected, as the agent starts with a high exploration rate (epsilon) and gradually shifts to exploiting the learned knowledge (resulting in lower epsilon values).

Total Reward
The reward can be seen fluctuating between positive and negative values. This can be an indicator that the agent is trying to learn from mistakes/navigating through difficult scenarios in the environment.

(b). Are the Metrics Relevant?

I believe the chosen metrics are relevant for assessing the learning progress of the RL agent. The **total reward (score)** is relevant as it makes it easy to see where the agent is succeeding and where it is encountering challenges

The **exploration rate (epsilon)** represents the tradeoff between exploration and exploitation. The agent is made to explore more at the beginning and then later on it exploits the learned knowledge, so a decrease in the curve is evidence that the agent is shifting towards its learned policy

Monitoring the **Q-values** was vital as this was the major indicator for me, that the agent was learning better action values. Positive Q for good actions and negative Q for less optimal actions are good indicators of learning.

Reflections:

To get a clearer idea about whether or not the agent was learning, I would have liked to train the model on more episodes but due to time constraints this wasn't possible. Initially, I implemented the model on a Google CoLab Notebook. There were some errors which took a few days to sort out before the model was training smoothly.

Upon training it, the Google CoLab was unable to handle the memory requirements of the DQN model. It took even more time to set up the python environment on my local machine, as I decided it would be best to make use of a high-end GPU in order to speed up training. This involved installing NVIDIA libraries that supported primitives for Deep Neural Networks.

The model was finally training but time was limited so I decided to train it using 300 episodes. Deepmind trained their agent over 10 million frames, which was able to surpass human ability in breakout. Compared to the 300,000 frames mine was trained over, it’s clear why the evidence of learning is minimal.

Potential Changes:

Reward Shaping

Initially there was no negative reward in the training loop for losing a game of breakout. I added this in the later stages of testing out the model in order to encourage the agent not to lose.

```
game_loss_reward = -3.0  # penalty value for losing game
```

Figure 35. Initialising the game loss reward variable (-3.0 as 3 lives lost means game end)

```
 # Apply a penalty if the game is lost and the total reward is 0 or negative
if done and reward <= 0:
    reward += game_loss_reward
```

Figure 36. Application of the game end penalty

If there was enough time I would have liked to enhance this by adding a penalty whenever a life is lost rather than when the game is lost, to reinforce that losing lives along with losing the game is also bad.

Replay Buffer

I experimented by training the model with a replay memory size of 10 rather than 10,000. The training only took around 3 minutes, but the results were very scattered, due to the agent only having the previous 10 experiences to learn from.

There were no signs of learning as the Q-values stayed low and the exploration rate stayed the same from start to finish.

```
Starting Episode 300
Episode: 300/300, Score: -3.0, Epsilon: 1.00
1/1 [==============================] - 0s 15ms/step
Q-values: [[ 0.21517871 -0.05681433 -0.39591336  0.16582125 -0.2710856 ]]
Loss: None
Experience Replay Buffer Size: 10
Episode Done. Total Reward: -3.0
Exploration Rate (Epsilon): 1.0
```

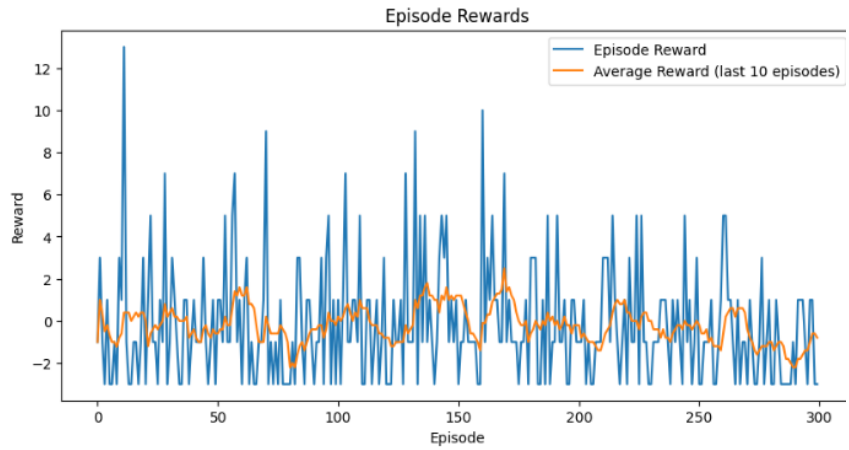


Figure 37. Results with a Replay Buffer Size of 10

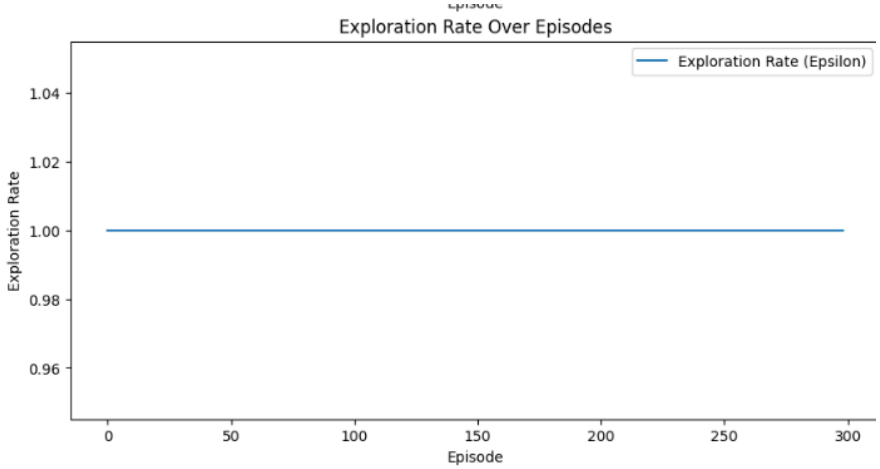


Figure 38. Exploration Rate with a Replay Buffer Size of 10

I would have liked to try a larger replay memory size (e.g. 50,000 or 100,000) to see the effects it would have on the learning curve.

Other Hyperparameters/Plots

With enough time, I would have liked to experiment by making changes to the learning rate, optimizers and loss functions etc., as I am very curious about the effects they would have had on the results.

Looking back on it, I would have also plotted the convergence of the Q values, as a graphical representation would be much easier to understand.

References

Bishop, C. M. (2006). "Pattern Recognition and Machine Learning." Springer.

Catastrophic Forgetting: Fayek, H.M., Cavedon, L. and Wu, H.R. (2020). Progressive learning: A deep learning framework for continual learning. Neural Networks, 128, pp.345–357. doi:https://doi.org/10.1016/j.neunet.2020.05.011.

Gym (2022). Gym Documentation. [online] Gym Library. Available at: https://www.gymlibrary.dev/ [Accessed 1 Dec. 2023]

Hastie, T.; Tibshirani, R.; Friedman, J. (2009). "The Elements of Statistical Learning." Springer.
Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). 'Playing Atari with Deep Reinforcement Learning.' arXiv preprint arXiv:1312.5602.

Random Seed Initialization: TensorFlow (2023). tf.random.set_seed | TensorFlow v2.14.0. [online] TensorFlow. Available at: https://www.tensorflow.org/api_docs/python/tf/random/set_seed [Accessed 8 Dec. 2023].

Sutton, R. S., & Barto, A. G. (2018). 'Reinforcement Learning: An Introduction' (2nd ed.). MIT Press.