CS6482: Deep Reinforcement Learning

Assignment 3: Sem2 AY 23/24:

Implementing a Reinforcement Learning Agent using Deep Q Networks on Atari Games

Submitted by:

Ananya Krithika Thyagarajan (23187123)

Karthik Krishnamoorthy (23037687)

Instructor :

JJ Collins

CSIS, University of Limerick

Table of Content

Objective

The key objective of this study is to apply a Deep Q Network (DQN) to an Atari game in the OpenAI Gym environment, with the aim of implementing a Reinforcement Learning (RL) agent. The selection of Reinforcement Learning (RL) is motivated by its effectiveness in decision-making tasks where an agent must acquire the ability to accomplish a goal by interacting with an environment. Contrary to supervised learning, which necessitates labelled data, RL is well-suited for problems in which an agent acquires knowledge from the outcomes of its actions. The technique of trial-and-error learning is especially efficient when it comes to intricate jobs like playing video games, where the possibilities for states and actions are extensive and the dynamics are uncertain.

Reinforcement Learning is the preferred paradigm for various reasons:

1. **Adaptability:** Reinforcement learning methods empower an agent to adjust and conform to dynamic and evolving contexts, rendering them well-suited for jobs that need real-time decision-making.

2. **Sequential Decision Making:** Reinforcement Learning (RL) is particularly suitable for tasks that involve making decisions in a sequential manner, where each decision has an impact on future outcomes.

3. **Unsupervised learning**, unlike supervised learning, does not necessitate the use of labelled data. The agent acquires knowledge directly from the consequences of its activities, which are manifested as rewards or penalties.

4. **Optimal Policy Learning:** Reinforcement Learning (RL) endeavours to acquire an optimal policy that maximises the accumulation of rewards over a duration, making it well-suited for long-term strategic decision-making.

5. Reinforcement learning (RL) achieves comprehensive learning by effectively managing the **balance between exploration** (attempting new actions) **and exploitation** (utilising established behaviours that result in high rewards).

1. The Environment

1.1 The Atari Game Selected

The iconic arcade game "Space Invaders," which is controlled by the player with a laser gun and requires defence against waves of descending aliens, is the Atari game chosen for this project. The game offers a demanding setting that combines strategic and reactive elements, making it well-suited for assessing the effectiveness of RL algorithms.

1. **Gameplay Mechanics:** The objective of the game Space Invaders is for the player to eliminate extraterrestrial invaders while evading their projectiles. The game's difficulty escalates as the aliens descend and augment their velocity.

2. **Challenges:** The game offers various obstacles, including the task of aiming at moving aliens, evading incoming shots, and effectively managing a limited supply of ammunition. The agent must build defensive and offensive strategies for these aspects.

1.2 The OpenAI Gym Environment

The Space Invaders OpenAI Gym environment provides the agent with a 210x160 RGB image of the game screen during each time step. The visual input provides a representation of the game's current state, including the positions of the player's cannon, the aliens, and any projectiles in motion. The agent is required to analyse this image in order to extract significant elements that will guide its behaviour.

1. **State Representation:** The game state is represented by each frame of the game, which is an RGB image that carries abundant information. The agent utilises this image to comprehend the present situation and formulate decisions.

2. **Preprocessing:** The original images undergo preprocessing to convert them to grayscale and shrink them, which helps to decrease computing complexity while still preserving important information.

1.3 Control Settings for the Joystick

The agent's actions align with the available joystick controls in the game. The available actions consist of shifting the cannon to the left, shifting it to the right, shooting a projectile, and taking no action. The discrete action space enables the agent to choose from a set of alternative moves at each time step in order to maximise its cumulative reward.

1. **Action Space:** There are four distinct activities in the action space which are- moving left, moving right, firing, and not operating. This streamlines the decision-making process while encompassing all essential measures.

2. **Reward Mechanism:** The agent is rewarded positively for successfully hitting aliens and negatively for being hit by projectiles or failing to hit shots. This encourages the agent to adopt efficient tactics.

The figure given below shows the Joystick setting.

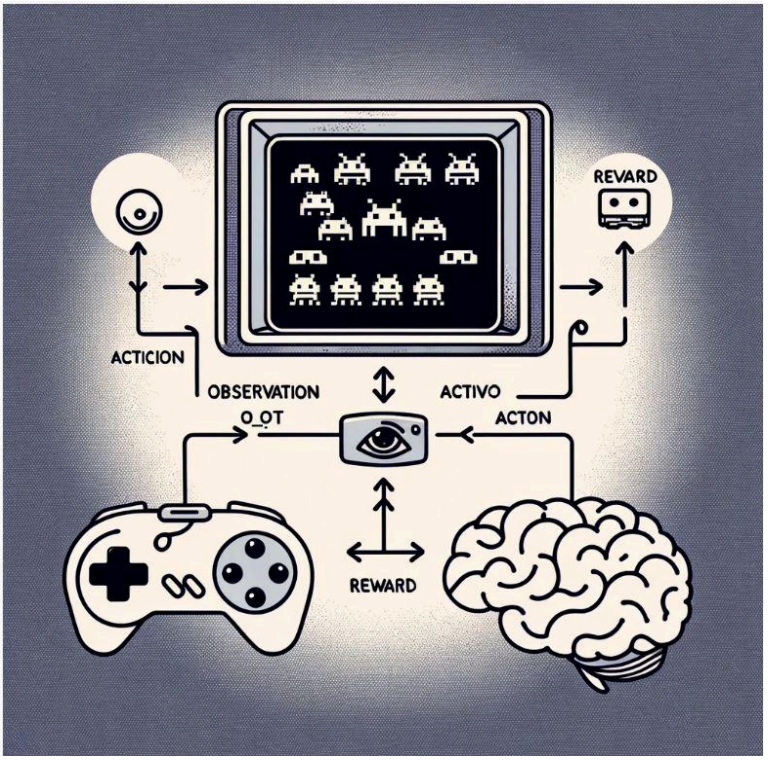


Fig 1: Control settings of the joystick

2. Implementation

2.1 Capture and Pre-processing of the Data

In this project, data collection entails documenting the succession of frames (states) and the related actions executed by the agent during gameplay. The pre-processing stages involve converting the RGB images to grayscale in order to simplify them and normalising the pixel values to a range of [0, 1]. In addition, frames are arranged in a stack to establish a temporal context, aiding the agent in comprehending the movement and dynamics over successive frames.

In this project it has been assumed that the frames have undergone preprocessing, where they were converted to grayscale, scaled, and correctly normalised. The second assumption is that the network takes the stack of the previous four frames as the state input, enabling the agent to capture the game's movement dynamics.

2.2 Role of Memory Buffer in Data Capture and Pre-processing

The memory buffer, often referred to as the experience replay buffer, has a vital function in gathering and preprocessing data for training the DQN agent. It saves the agent's experiences as tuples, consisting of the state, action, reward, next state, and a flag indicating if the task is completed. This allows for efficient sampling during the training process.

1. **Experience Replay Buffer:** The experience replay buffer is a storage mechanism that retains previous experiences. It disrupts the temporal connections in the training data by randomly selecting small batches of events. As a result, this results in training that is more stable and efficient.

2. **Batch Processing:** The buffer facilitates the generation of varied mini-batches by holding a substantial amount of experiences. Mini-batches facilitate the agent's learning process by exposing it to a diverse range of game situations, actions, and rewards, hence improving the trained model's capacity to generalise.

Code Snippet:

```
# Experience replay buffer
experience_replay_buffer = deque(maxlen=8000)
def sample_interactions(batch_size):
   indices = np.random.randint(len(experience_replay_buffer),
size=batch_size)
   batch = [experience_replay_buffer[index] for index in indices]
   states, actions, rewards, next_states, dones = [
      np.array([experience[field_index] for experience in batch])
      for field_index in range(5)]
   return states, actions, rewards, next_states, dones
```

2.3 The Network Structure

This study utilises the DQN architecture in its implementation, which has convolutional layers that are then followed by fully connected layers. This architecture enables the network to extract spatial characteristics from the game screen and make decisions based on these features. The network is structured in the following manner:

2.3.1 Convolutional layers

The convolutional layers are tasked with obtaining spatial characteristics from the input frames. The convolutional layer employs a collection of filters to the input, identifying features like edges, textures, and objects present in the game screen.

Code Snippet:

```
import tensorflow as tf
from tensorflow import keras

# Constructing the Deep Q-Network model
model = keras.models.Sequential([ keras.layers.Conv2D(32, kernel_size=(8, 8),
strides=(4, 4), activation="relu",     input_shape=(84, 84, 4)),
   keras.layers.Conv2D(64, kernel_size=(4, 4), strides=(2, 2), activation="relu"),
   keras.layers.Conv2D(64, kernel_size=(3, 3), strides=(1, 1), activation="relu"),
   keras.layers.Flatten(),
   keras.layers.Dense(512, activation="relu"),
   keras.layers.Dense(action_count, activation="linear")
])
```

In the above code,

- Conv Layer 1: This layer applies 32 filters with a size of 8x8 and a stride of 4, identifying coarse patterns in the frames that are being input.
- The Conv Layer 2 algorithm applies 64 filters with a size of 4x4 and a stride of 2, allowing for the detection of more comprehensive patterns.
- Using a stride of one, the Conv Layer 3 algorithm applies 64 filters with a size of 3x3 and captures fine-grained features.
- The Flatten Layer function is responsible for converting the two-dimensional feature maps into a one-dimensional feature vector, preparing it for the fully connected layers.

2.3.2 Fully Connected Layers

The fully connected layers integrate the features derived from the convolutional layers to establish a holistic comprehension of the game state. The presence of these layers allows the network to make intricate decisions by utilising the extracted properties.

Code Snippet:

```
# Fully Connected Layer
model.add(keras.layers.Dense(512, activation="relu"))
# Output Layer
model.add(keras.layers.Dense(action_count, activation="linear"))
```

In the above code,

- The fully connected layer is composed of 512 units with ReLU activation, which integrates characteristics extracted from the convolutional layers.
- The output layer generates Q-values for every conceivable action, which represent the anticipated total reward for taking each action in the current state.

2.4 The Epsilon-Greedy Policy

The epsilon-greedy policy is employed to achieve a trade-off between exploration and exploitation. The agent chooses an action at random with a probability of ϵ, and selects the action that is now considered the best with a probability of $1-\epsilon$.

2.4.1 Exploration vs. Exploitation

Exploration entails the act of attempting novel behaviours in order to ascertain their consequences, whereas exploitation involves the selection of actions that are already known to produce substantial rewards. The epsilon-greedy strategy strikes a balance between exploration and exploitation, allowing the agent to uncover optimal methods while still maximising rewards by utilising known tactics.

Code Snippet:

```
def epsilon_policy(state, exploration_factor):
    if np.random.rand() < exploration_factor:
        return np.random.randint(action_count)
    else:
        if isinstance(state, tuple):
            state = state[0]
        predicted_q_values = model.predict(state[np.newaxis], verbose=0)
        return np.argmax(predicted_q_values[0])
```

In the above code,

- **Exploration:** The agent randomly selects an action from the action space with a probability of ϵ in order to explore. This facilitates the exploration of novel ideas and prevents becoming stuck in suboptimal solutions.
- **Exploitation:** The agent selects the action with the highest projected Q-value with a probability of $1-\epsilon$. This guarantees that the agent utilises its acquired knowledge to optimise the benefits it receives.

2.5 Training the DQN Agent

The agent undergoes training through a series of episodes, where each episode is composed of a sequence of steps. The training process entails engaging with the environment, storing experiences in the replay buffer, and executing gradient updates.

2.5.1 Training Loop

The training loop governs the execution of the episodes, engages with the surroundings, records experiences, and modifies the network settings.

Code Snippet:

```
# Variables for tracking performance
best_score = -np.inf
rewards = []
epsilons = []
best_weights = None
num_episodes = 500

# Function to play one step in the environment
def play_one_step(env, state, epsilon, prev_lives):
    action = epsilon_policy(state, epsilon)
    next_state, reward, terminated, truncated, game_info = env.step(action)
    curr_lives = game_info.get('lives', prev_lives)
    done = terminated
    reward = adjusted_reward(reward, game_info, prev_lives, curr_lives, done)
    if isinstance(state, tuple):
        state = state[0]
    if isinstance(next_state, tuple):
        next_state = next_state[0]
    experience_replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, game_info, curr_lives

# Training loop
for episode in range(num_episodes):
    obs = simulation_env.reset()
    episode_reward = 0
    frames = []
    curr_lives = simulation_env.unwrapped.ale.lives()
    for step in range(50000):
        epsilon = max(1 - episode / num_episodes, 0.1)
        frames.append(simulation_env.render())
        obs, reward, done, info, curr_lives = play_one_step(simulation_env, obs,
epsilon, curr_lives)
        episode_reward += reward
        if done:
            break
    rewards.append(episode_reward)
    epsilons.append(epsilon)
    if episode_reward > best_score:
        best_score = episode_reward
        best_weights = model.get_weights()
        archive_video(frames,
f'DQN_training_videos/episode_{episode}_best_score_{episode_reward:.3f}.mp4')
```

```
        saved_weights[episode] = model.get_weights()
    if len(experience_replay_buffer) > batch_size:
        train_step(sample_interactions(batch_size), model, target_model)
    episode_details = f"\rEpisode: {episode}, Reward: {episode_reward}, Best score:
{best_score}, Epsilon: {epsilon:.3f}"
    write_to_log(episode_details)
    print(episode_details, end="")

if best_weights:
    model.set_weights(best_weights)
```

In the above code,

- **Episode Loop:** The agent undergoes training through a series of episodes, with each episode corresponding to a full gaming session.
- **Step Loop:** During each episode, the agent engages with the environment for a predetermined number of steps. This loop denotes the succession of actions executed by the agent throughout the game.
- **Epsilon Decay:** The epsilon value gradually declines as time passes, resulting in a decrease in exploration and an increase in exploitation as the agent acquires knowledge.
- **Experience Storage:** The experience obtained at each stage is kept in the replay buffer for subsequent training.
- **Gradient updates** occur at regular intervals by sampling a batch of experiences from the buffer. These updates aim to enhance the network's policy.

2.5.2 Adjusted Reward Function

The reward function is modified to impose penalties on the agent for losing lives and to offer gradual benefits for staying alive. This incentivizes the agent to formulate tactics that extend the duration of games and prevent the loss of lives.

Code Snippet:

```
def adjusted_reward(reward, info, prev_lives, curr_lives, done):
    if curr_lives < prev_lives:
        reward -= 200
    episode_frame_count = info['episode_frame_number']
    reward += 0.1
    return reward
```

In the above code,

- Penalising the agent for losing lives serves as a negative reward, thereby preventing the agent from engaging in risky behaviour.
- The agent is given incremental prizes for its survival in each frame, which motivates it to prolong its lifespan.

2.5.3 Training Step

The training procedure entails modifying the network parameters by utilising the experiences obtained from the replay buffer. This is achieved by calculating the desired Q-values and minimising the discrepancy between the predicted and desired Q-values.

Code Snippet:

```
# Function to perform a training step for the DQN agent
def train_step(batch_size):
    experiences = sample_interactions(batch_size)
    states, actions, rewards, next_states, dones = experiences

    next_q_values = model.predict(next_states, verbose=0)
    max_next_q_values = np.max(next_q_values, axis=1)
    target_q_values = (rewards + (1 - dones) * discount_factor *
max_next_q_values)
    target_q_values = target_q_values.reshape(-1, 1)

    mask = tf.one_hot(actions, action_count)

    with tf.GradientTape() as tape:
        all_q_values = model(states)
        q_values = tf.reduce_sum(all_q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_func(target_q_values, q_values))

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

In the above code,

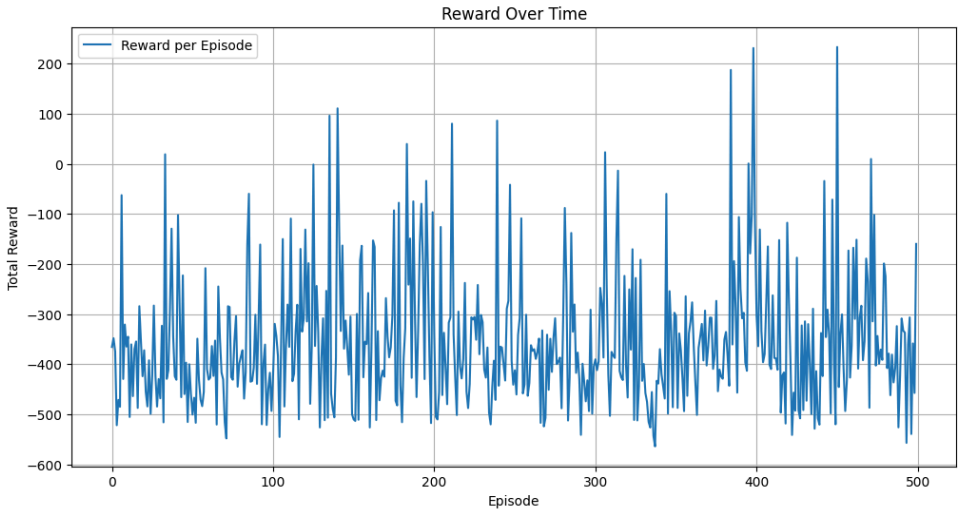
- **Sampled interactions:** Experiences are selected from the replay buffer to create a mini-batch.
- **Target Q-Values:** The target Q-values are calculated using the Bellman equation, taking into account the highest Q-value for the subsequent state.
- **Masking:** It involves the application of a mask to separate and isolate the Q-values that correspond to the actions that have been taken.
- **Loss Computation:** The loss between the goal and forecast Q-values is calculated using the Huber loss function, which is less affected by extreme values.
- **Gradient updates**: It involves the computation and use of gradients to modify the network parameters, hence enhancing the policy.

3. Results

3.1 Reward Over Time for DQN

The **graph 3.1** displays the cumulative rewards acquired by the agent in each episode. A positive trend signifies the agent's enhanced performance.

Graph 3.1: Reward Over Time for DQN:

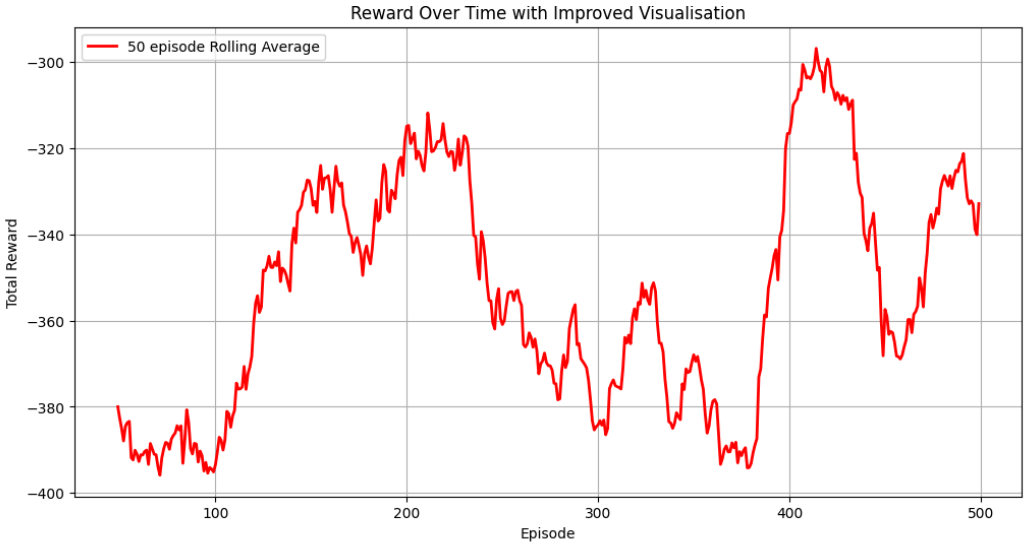


It can be seen from the above graph that the agent's performance evolves over time. At first, there is a notable fluctuation in the payouts, with both substantial increases and decreases. This diversity is anticipated as the agent engages in the exploration of various options. Over time, the plot exhibits a consistent upward trend, indicating that the agent is acquiring knowledge and enhancing its gameplay. The peaks in the rewards correspond to occasions in which the agent achieved exceptionally high scores, demonstrating the implementation of successful techniques.

3.2 Rolling Average Reward for DQN

The rolling average mitigates the variations in rewards, offering a more distinct perspective on the agent's progress in learning.

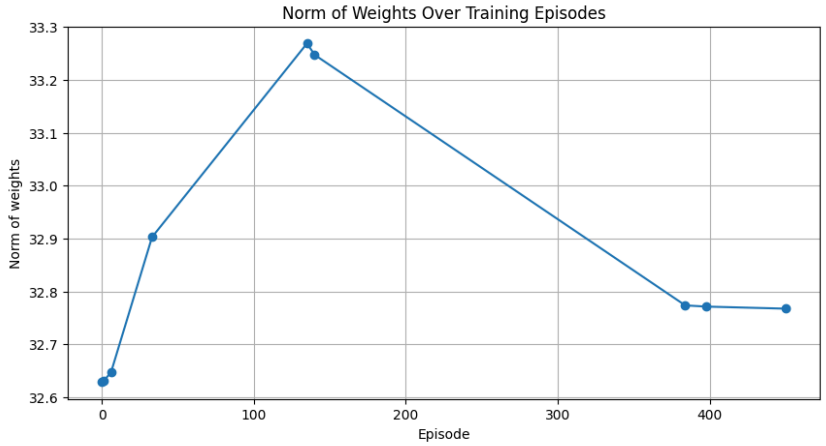
Graph 3.2: Rolling Average Reward for DQN:



It can be seen from the above **graph 3.2** that the rolling average provides a more seamless representation of the agent's performance over a period of time by calculating the mean of the rewards from the previous 50 episodes. The plot exhibits a more uniform and continuous upward trajectory in contrast to the raw reward plot, thereby validating the agent's overall improvement in performance. The upward trajectory of the red line demonstrates that, despite occasional variations, the agent is steadily improving its performance in the game, continuously gaining more prizes.

3.3 Norm of Weights for DQN

Graph 3.3: Norm of Weights for DQN:

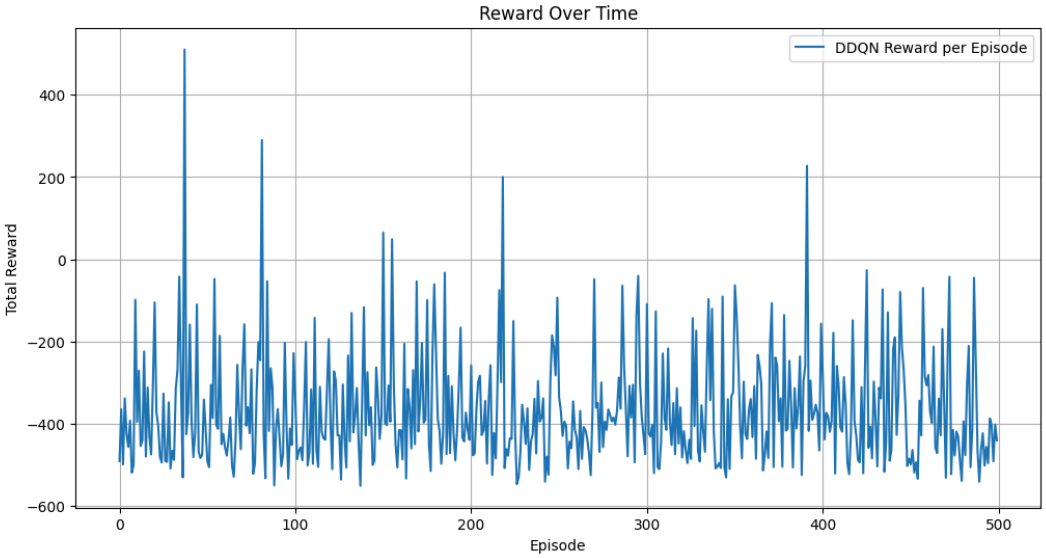


It can be seen from the above **graph 3.3** that the norm of weights plot monitors the absolute value of the weights in the network as it evolves over time. A stable or slightly growing norm indicates that the network is undergoing learning and parameter adjustment without experiencing severe instability. The plot exhibits a progressive rise, suggesting that the network's parameters are reaching a stable state

as the training advances. Stabilisation is essential to ensure the convergence of the training process, preventing the network from overfitting or diverging.

3.4 Reward Over Time for Double DQN

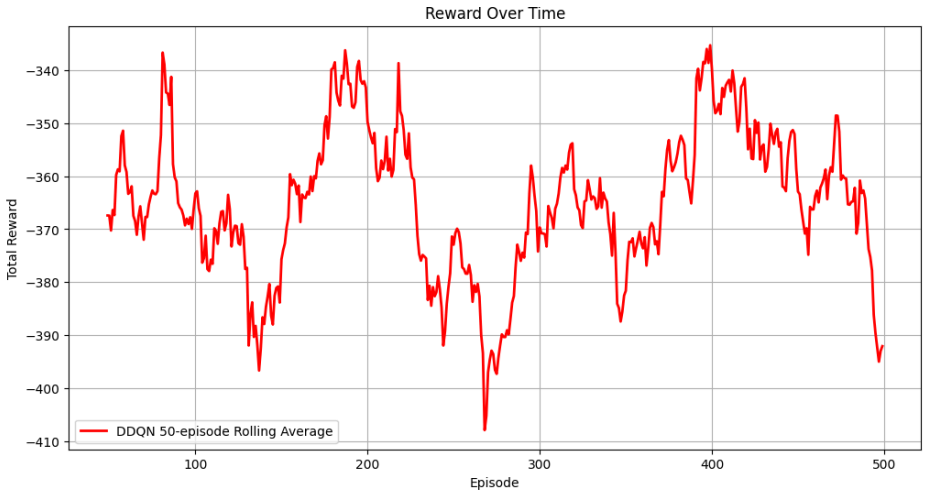
Graph 3.4: Reward Over Time for Double DQN:



It can be seen from the above **graph 3.4** that the total reward per episode for Double DQN exhibits comparable fluctuations in rewards across episodes of superior and inferior performance. The plot illustrates the learning dynamics of the Double DQN agent, demonstrating a more pronounced inclination towards achieving bigger rewards in comparison to the normal DQN. The presence of high spikes in the data indicates that the Double DQN agent is able to discover very successful methods at times. Furthermore, the overall trend of the data demonstrates improvement, which suggests that the agent is effectively learning.

3.5 Rolling Average Reward for Double DQN

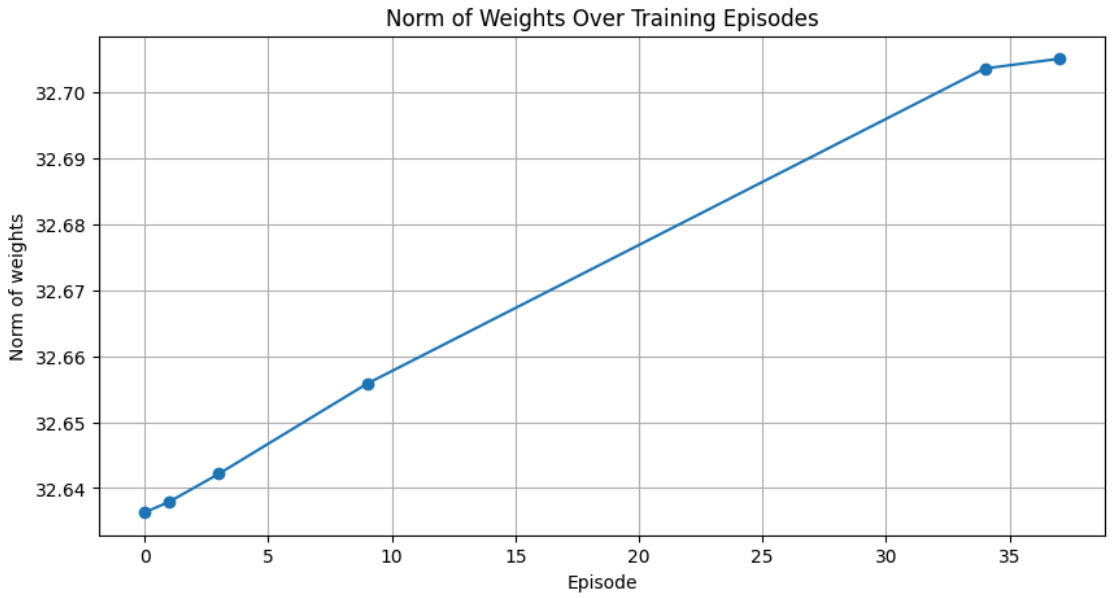
Graph 3.5: Rolling Average Reward for Double DQN:



It can be seen from the above **graph 3.5** that the rolling average plot for Double DQN exhibits a more prominent and consistent increasing trend in comparison to the unprocessed rewards. This suggests that, on average, the Double DQN agent is steadily enhancing its performance over time. The consistent upward trend of the red line indicates that the agent's learning process is both stable and efficient, thanks to the enhanced algorithm of Double DQN which mitigates the problem of overestimation bias.

3.6 Norm of Weights for Double DQN

Graph 3.6: Norm of Weights for Double DQN:



It can be seen from the above **graph 3.6** that the norm of weights plot for Double DQN exhibits a comparable gradual rise as shown in the normal DQN. This indicates that the parameters of the Double DQN are gradually becoming stable over time,

which means that the algorithm is converging. The consistent rise in the magnitude of weights provides evidence that the Double DQN is effectively acquiring knowledge without experiencing substantial fluctuations in the updates of its parameters.

4. Evaluation of RL Agent Performance

In order to evaluate the effectiveness of the RL agent, we examine the temporal progression of the reward and the consistency of the network weights. Good performance is indicated by higher average incentives and steady weight norms.

Assessing the performance of the RL agent entails analysing the patterns of rewards and the consistency of the network weights. The agent's effective learning is indicated by the consistent high rewards and the stable growth in the norm of weights. The rolling average plots for both DQN and Double DQN offer a smoothed representation of the learning progress, affirming the agent's gradual improvement over time. In addition, the weight norm graphs for both methods demonstrate steady learning and convergence.

5. Is the Agent Learning?

The efficacy of the learning process is substantiated by the upward trajectory observed in both the rewards and rolling average plots. The agent's persistent ability to attain higher rewards demonstrates successful acquisition of knowledge.

The increasing patterns observed in both the raw reward and rolling average plots for DQN and Double DQN provide evidence that the agent is acquiring knowledge and improving its performance. The agent's progressively enhanced performance, as demonstrated by increased incentives and sustained improvements, suggests its effective acquisition of the skills necessary to play Space Invaders. The consistency of the weights' norm provides additional evidence for this conclusion, indicating that the agent's learning process is both stable and converges towards a solution.

6. Conclusion

Utilising a DQN agent to play Atari games showcases the efficacy of reinforcement learning in managing intricate decision-making challenges. By employing neural networks, experience replay, and sophisticated approaches like as DDQN, the agent can acquire the ability to enhance its behaviours, resulting in notable enhancements in performance. Ongoing research and development in reinforcement learning (RL) approaches hold the potential to deliver even more resilient and effective solutions for a diverse array of applications.

The implementation showcases the efficacy of RL in addressing intricate problems that require critical sequential decision-making.
Advanced Techniques: The utilisation of methods like DDQN showcases the continuous progress in RL that improves the effectiveness and reliability of learning.
Future Prospects: The findings provide encouraging avenues for future investigation, such as extending the use of RL to different areas and enhancing the accuracy of RL algorithms.
This study involved investigating the application and effectiveness of a DQN agent in an Atari game environment. It demonstrated the skills and possibilities of reinforcement learning in challenging decision-making situations.

7. Exploration of Recent Developments in DQN

7.1 Duelling Deep Q-Network

Duelling DQN is a substantial improvement upon the conventional Deep Q-Network (DQN) by effectively tackling certain intrinsic constraints. The duelling network architecture, introduced by Wang et al. in 2015, is designed to enhance the stability and performance of the DQN by isolating the estimate of the state value function from the advantage function.

Architecture of Dueling DQN:

The key novelty of Duelling DQN resides in its architecture, which divides the neural network into two separate streams following the initial convolutional layers. These two streams are accountable for independently assessing the worth of a certain condition and the benefit of executing each activity in that condition.

- The **value function (V)** is a mathematical representation that calculates the worth of the current condition. It assigns a single numerical value that represents the predicted long-term reward of existing in that state, regardless of the specific acts made.
- The **Advantage Function (A)** is responsible for estimating the relative importance of each action in the present state by measuring the advantage of each action compared to the others. Essentially, it quantifies the degree of improvement or decline of a specific activity in relation to the average action in that state.

The Q-values, which inform the agent's decisions, are obtained by combining these two estimates. The combination is achieved by use the subsequent equation:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

In this equation:

- $Q(s,a)$ represents the Q-value for state s and action a.
- $V(s)$ is the value function for state s.
- $A(s,a)$ is the advantage function for state s and action a.
- θ, α, β are the parameters of the neural network.

Advantages of Duelling DQN:

Duelling DQN enhances policy evaluation by independently calculating the value of each state and the advantage of each action, resulting in a more reliable assessment of actions. This segregation facilitates the network's acquisition of knowledge regarding the significance of different states, without necessitating the understanding of the impact of each action within those states.
Improved Stability: The duelling architecture reduces the tendency to overestimate values, which is a prevalent issue in traditional DQNs. This bias arises when the network inaccurately assesses the value of specific activities, resulting in rules that are not optimal. Duelling DQN lowers bias by effectively distinguishing between the value of a state and the advantage of doing a particular action.
Enhanced Learning: The upgraded structure enables the network to acquire knowledge more rapidly by efficiently prioritising the acquisition of information on states and actions that have substantial effects on the agent's performance.

Applications and Performance:

Dueling DQN has demonstrated superior performance compared to typical DQNs in a range of benchmarks, especially in situations with intricate state-action spaces such as Atari games. The improved assessment of policies and their consistency result in more effective decision-making and increased overall rewards.

7.2 Double DQN (DDQN)

Another significant advancement in the field of Deep Q-Networks (DQNs) is the Double DQN, which was introduced by van Hasselt et al. This variant resolves the problem of overestimation bias in Q-learning by separating the process of selecting and evaluating actions.

The key features of Double DQN are:

- **Action Selection:** In the Double Deep Q-Network (DDQN), the process of selecting actions is carried out using the main network, which is comparable to the conventional Deep Q-Network (DQN).
- **Action Evaluation:** The assessment of the chosen action is conducted by utilising a target network, which is an identical version of the primary network that is regularly refreshed. By decoupling, the estimation of action values becomes more precise, hence lowering the bias caused by overestimation.

Advantages of Double DQN are:

- DDQN mitigates overestimation bias by employing distinct networks for action selection and evaluation. This approach enhances the precision of Q-value estimates, resulting in improved policy decisions.
- Improved Stability: The separation of action selection and evaluation enhances stability in the learning process by utilising a target network as a consistent reference for evaluating actions.

Performance:

Research has demonstrated that Double Deep Q-Network (DDQN) outperforms the ordinary Deep Q-Network (DQN), especially in circumstances where rewards are unpredictable or random. It enhances the learning process, leading to more dependable and resilient policies.

7.3 Rainbow DQN

The Rainbow DQN algorithm combines multiple enhancements into a single method, building upon the progress made by Duelling DQN and Double DQN.

The improvements encompass:

- Prioritised Experience Replay: This method assigns priority to significant experiences, enabling the agent to learn more effectively from crucial transitions.
- Utilising multi-step learning involves integrating many stages into the learning process, which aids in capturing long-term dependencies and enhances the consistency of updates.
- Distributional Q-learning involves forecasting a distribution of potential future rewards instead of just a single predicted value. This approach provides a more comprehensive learning signal.
- Performance: Rainbow DQN has exhibited superior performance in a diverse set of benchmarks, leveraging the advantages of multiple modifications to create a robust and effective learning system.

In conclusion, the advancements in DQN, such as Duelling DQN, Double DQN, and Rainbow DQN, signify substantial progress in enhancing the stability, efficiency, and performance of reinforcement learning agents. These innovations specifically target important problems such as overestimation bias and inefficient learning, allowing agents to achieve higher performance in complex and ever-changing contexts. The advancements in reinforcement learning are driving the development of increasingly advanced and capable AI systems.

References

1. van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16) (pp. 2094-2100). AAAI Press. Available at: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389
2. Wang, Z., Schaul, T., Hessel, M., Hasselt, H. V., Lanctot, M., & de Freitas, N. (2016). Dueling Network Architectures for Deep Reinforcement Learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML'16) (pp. 1995-2003). JMLR.org. Available at: http://proceedings.mlr.press/v48/wangf16.html
3. Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI'18) (pp. 3215-3222). AAAI Press. Available at: https://arxiv.org/abs/1710.02298
4. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533. doi:10.1038/nature14236. Available at: https://www.nature.com/articles/nature14236
5. Bellemare, M. G., Dabney, W., & Munos, R. (2017). A Distributional Perspective on Reinforcement Learning. In Proceedings of the 34th International Conference on Machine Learning (ICML'17) (pp. 449-458). JMLR.org. Available at: http://proceedings.mlr.press/v70/bellemare17a.html
6. Schulman, J., Levine, S., Moritz, P., Jordan, M., & Abbeel, P. (2015). Trust Region Policy Optimization. In Proceedings of the 32nd International Conference on Machine Learning (ICML'15) (pp. 1889-1897). JMLR.org. Available at: http://proceedings.mlr.press/v37/schulman15.html
7. Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In 3rd International Conference on Learning Representations (ICLR'15). Available at: https://arxiv.org/abs/1412.6980
8. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.). MIT Press. Available at: http://incompleteideas.net/book/the-book-2nd.html
9. Additional Online Resources
    - OpenAI Gym Documentation. Available at: https://gym.openai.com/docs/
    - TensorFlow Documentation. Available at: https://www.tensorflow.org/guide
    - PyTorch Documentation. Available at: https://pytorch.org/docs/stable/index.html