UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CS6482: Deep Reinforcement Learning

Assignment 2: Sem2 AY 23/24: DQN for Classic Control

Submitted by:

Ananya Krithika Thyagarajan (23187123)

Karthik Krishnamoorthy (23037687)

Instructor :

JJ Collins

CSIS, University of Limerick

Table of Content

Introduction

This research explores the application of a Reinforcement Learning (RL) model to improve the control strategy for an Acrobot, which is a simulation of a robotic arm consisting of two linked segments. This task exemplifies a category of problems that are very suitable for reinforcement learning (RL), in which an autonomous agent acquires the ability to navigate an intricate environment by means of feedback mechanisms. In this report, we provide a comprehensive account of the systematic approach utilised, elucidating every element of the implementation and assessing the performance of the model at different phases of training.

1. Efficacy of Reinforcement Learning for the Acrobot Task

Paradigm of Choice:

Reinforcement Learning (RL) is the best paradigm for the Acrobot problem because of its intrinsic ability to learn from interactions. Reinforcement learning performs exceptionally well in situations where it is not possible to provide clear and detailed instructions on how to achieve a specific objective. The Acrobot challenge is that it allows an agent to discover effective strategies through trial and error, optimising actions in accordance with the rewards received.

2. The Gym Environment

The gym environment can be considered a standardised testing ground.
It is an open-source toolset for creating and evaluating RL algorithms. The platform offers a collection of standardised settings, such as 'Acrobot-v1', that follow a consistent interface for agents to engage with.

Code Snippet

```
acro_env = gym.make('Acrobot-v1', render_mode='rgb_array', new_step_api=True)
```

The line ***'gym.make'*** creates a new instance of the Acrobot environment. The ***'render_mode'*** set to ***'rgb_array'*** specifies that the rendered picture should be received as an array for visualisations, and ***new_step_api=True*** guarantees that the latest step API is used, which gives a cleaner structure for returned values from each step.

3. Implementation

3.1 Capture and Sampling of the Data

In the realm of reinforcement learning (RL), particularly in the context of a deep Q-learning framework, the process of capturing and sampling events is crucial for successful learning policies. A replay buffer is a crucial data structure used in the Acrobot task's code to facilitate effective and quick learning. In this section, we will examine the buffer's role and function in great depth.

Role and Function of the Experience Replay Buffer

A replay buffer is a mechanism used to store the experiences of an agent at each timestep. An experience typically includes the state (***'state'***), the action taken (***'action'***), the reward obtained (***'reward'***), the next new state (***'next_state'***), and a done flag (***'done'***) that indicates the conclusion of an episode.

Code Snippet

```
replay_buff = deque  (maxlen=2000)
```

In the above code snippet,
'deque': It is a data structure that enables the efficient insertion and removal of elements from both ends, with a time complexity of O(1).
This property is crucial for ensuring high efficiency.
'maxlen = 2000': This setting establishes a predetermined capacity for the buffer, guaranteeing that it can only hold the most recent 2000 experiences. Once the buffer surpasses this amount, the oldest experiences are eliminated, guaranteeing a limited memory usage.

Capturing Data and Experiences

At every instance of interaction with the environment, the agent's encounter is recorded and saved in the buffer.

Code Snippet:

```
replay_buff.append((state, action, reward, next_state, done))
```

In the above code snippet,
'append()': This function replenishes the replay buffer with a fresh experience. Every experience is a tuple that includes the done flag, the next state, the action taken, the reward received, and the present state.
Storing experiences as tuples preserves the link between the many components of an experience, which is essential for learning.

Data Sampling Technique for Random Sampling

In order to update the agent's policy during training, random mini-batches are sampled from the replay buffer. It is crucial to use random sampling to disrupt the correlation that may arise from successive encounters, as this might cause learning to become unstable or divergent.

Code Snippet:

```
indices = np.random.randint(len(replay_buff), size=batch_size)
batch = [replay_buff[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)]
```

In the above code snippet,

‘np.random.randint()’: This function ensures that every sampled batch has a random subset of experiences by creating a random selection of indices.
‘group’: Using the randomly generated indices, a list comprehension is used to pull the experiences from the replay buffer.
The last four lines take each experience's component (states, actions, etc.) and stack them into different numpy arrays so the neural network may process them in batches while training.

Relevance to Deep Q-Learning: What Makes a Replay Buffer Effective?

- Reducing Correlation: The agent's ability to generalise depends on minimising the correlation between successive encounters, which is achieved by employing a replay buffer and randomly selecting mini-batches.
- Training for Stabilisation: It smoothes the learning updates over a larger distribution of previous experiences, preventing the agent from overfitting to the most recent experiences.
- Efficiency: As the network is updated with batches of data, sampling from the replay buffer enables a more economical use of computational resources than online, one-by-one updates.

Thus, the replay buffer has to be carefully curated and used in order for the agent to learn from its interactions with the environment. By offering a varied and representative collection of experiences, it guarantees a consistent and robust learning process, which serves as the foundation for the agent's gradual advances over time.

3.2 The Network Structure and Hyperparameters

The selection of network design and the optimisation of hyperparameters play a crucial role in determining the effectiveness of the learning algorithm in the Acrobot challenge inside the reinforcement learning framework. Here, we provide a comprehensive explanation of the complex elements of the network and hyperparameters used in the code.

The Neural Network Architecture: Design of a Multi-Layer Perceptron (MLP)

The neural network architecture employed in this assignment is a Multi-Layer Perceptron (MLP), which belongs to the category of feedforward artificial neural networks.

Code Snippet:

```
acro_model = tf.keras.Sequential([
   tf.keras.layers.Dense(32, activation="relu", input_shape=[input_shape]),
   tf.keras.layers.Dense(32, activation="relu"),
   tf.keras.layers.Dense(n_outputs)])
```

In the above code snippet, a layer-by-layer analysis has been done:

Input layer:

The ***'input_shape'***, which is the environment's observation space, determines the size of the input layer.
The initial layer of the network indirectly conveys its responsibility of accepting state information from the environment.

Hidden layers:

There are two designated hidden layers, each containing 32 neurons.
The activation function "relu" signifies the utilisation of the Rectified Linear Unit (ReLU) activation function. This function incorporates non-linearity into the learning process, enabling the network to acquire intricate patterns.

Output layer:

The size of the output layer, denoted as***' n_outputs'***, corresponds to the set of possible actions in the Acrobot environment.
The process generates a Q-value for every potential action, which the policy then uses to determine the subsequent action to be taken.

Specification of Hyperparameters

Hyperparameters are the modifiable parameters that govern the process of training a model. They are important because they have a significant impact on the trained model's performance.

Code Snippet:

```
dis_rate = 0.95  # Discount rate for future rewards
learn_rate = 0.01  # Learning rate for the optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=learn_rate)
epsilon = 1.0
epsilon_decay = 0.995
min_epsilon = 0.01  # Minimum value for epsilon
```

In the above code snippet,

Discounting Rate (dis_rate):

This value represents the significance of forthcoming rewards.
A discount rate of 0.95 indicates that the agent places a high value on future rewards, although somewhat less than immediate rewards.

Learning Rate (learn_rate):

During training, the learning rate regulates how frequently the neural network weights are updated.

A rate of 0.01 is a judicious option, striking a balance between the pace of acquisition and the requirement for consistent convergence.

Optimizer:

The Adam optimizer is renowned for amalgamating the most advantageous characteristics of the AdaGrad and RMSProp algorithms.
The learning rate is adjusted for each weight separately, resulting in a variable learning rate that aids in navigating the loss landscape with greater efficiency.

Exploration Rate (epsilon, min_epsilon, epsilon_decay):

The ***epsilon-greedy*** approach to action selection is controlled by these factors.
The initial value of epsilon is set to 1.0, which enables the agent to freely explore the action space in the early stages.
Epsilon_decay controls the rate at which the level of exploration is reduced. A value of 0.995 will gradually decrease the level of exploration as the agent acquires more knowledge about the area.
The value of ***min_epsilon*** represents the minimum threshold for epsilon, guaranteeing the presence of exploration at all times.

Importance of Hyperparameters and Networks in Learning

The MLP network, consisting of predetermined layers and neurons, functions as the function approximator for the Q-values. The concealed layers encapsulate the intricate connections between the input states and the resultant Q-values.

Hyperparameters determine the rate and consistency of the learning process. For example, the learning rate should be sufficiently high to enable the network to make considerable adjustments to the weights in response to errors, but not so high that it leads to unpredictability in the weights. The discount rate impacts the agent's policy by determining the extent to which it prioritises immediate benefits over long-term gains, thereby affecting its level of shortsightedness or farsightedness.

It can be concluded that the network architecture and hyperparameters of the reinforcement learning model are carefully adjusted to ensure that the agent can efficiently process and learn from the environmental conditions. By carefully calibrating the agent, it is ensured that it not only acquires knowledge but also applies it effectively to achieve optimal performance in the given task.

3.3 Q-Learning Update Mechanism

Within the domain of reinforcement learning (RL), specifically in the implementation for the Acrobot environment, the Q-learning update mechanism plays a crucial role in determining the agent's policy evolution. We will analyse this mechanism by breaking it down into its most detailed steps, using the code as a reference to provide clarity on every process.

Calculation of Temporal-Difference Error Sequential Procedure

1. **Q-value prediction:**
   The agent initiates the process by making predictions for the Q-values of current states using the neural network model.
   Code Snippet:
   ```
   all_Q_values = acro_model(states)
   ```

   This line utilises the ***'acro_model'*** to create predictions of Q-values for all feasible actions corresponding to the provided states.

2. **Selecting Q-values for the Actions Performed:**
   Subsequently, the Q-values associated with the activities that were effectively executed need to be extracted. This is achieved by utilising a one-hot encoding technique on the actions and subsequently employing it as a mask to selectively filter the projected Q-values.
   Code Snippet:
   ```
   mask = tf.one_hot(actions, n_outputs)
   Q_val = tf.reduce_sum(all_Q_values * mask, axis=1)
   ```

   In the above code snippet, ***'tf.one_hot'*** generates a binary mask in which only the indices of the executed operations are represented as 1. The Q-values for the chosen actions are isolated by performing element-wise multiplication with all_Q_values and then applying a ***'tf.reduce_sum'*** operation across the action dimension.

3. **Calculation of Target Q-values:**
   The target Q-values are determined by adding the immediate reward to the discounted maximum expected Q-value for the next state, but only if the following state is not terminal.

   Code Snippet:
   ```
   max_next_Q_val = np.max(next_Q_val, axis=1)
   target_Q_val = rewards + (1 - dones) * dis_rate * max_next_Q_val
   ```

   The variable ***"max_next_Q_val"*** represents the highest anticipated Q-value for the next state, which indicates the optimal action to be taken in the future. The ***target_Q_val*** represents the desired Q-value for the state-action pair, taking into account the immediate reward and the discounted predicted future rewards.

4. **Loss Calculation:**
   The temporal-difference (***TD***) error is computed by taking the average of the squared differences between the predicted Q-values (***Q_val***) and the target Q-values (***target_Q_val***).

   Code Snippet:
   ```
   loss = tf.reduce_mean(tf.square(target_Q_val - Q_val))
   ```

The ***tf.square*** function calculates the square of the difference, resulting in a stronger penalty for larger inaccuracies. The process of taking the mean ***(tf.reduce_mean***) yields a solitary scalar number that represents the average mistake throughout the batch.

Weight Update and Gradient Computation

5. **Backpropagation**:
   Gradients of the loss with respect to the model parameters are computed after the loss is calculated. Backpropagation is the fundamental process in which the erroneous signal is transmitted in reverse through the network to modify the weights.

   Code Snippet:
   grads = tape.gradient(loss, acro_model.trainable_variables)

   TensorFlow's tf module.GradientTape is employed to document processes for the purpose of automatic differentiation. When the function tape.gradient is used, it computes the gradients of the loss function with respect to the trainable variables of the ***acro_model***.

6. **Applying the gradients:**
   The gradients are subsequently employed to modify the weights in a manner that would diminish the loss.

   Code Snippet:
   optimizer.apply_gradients(zip(grads, acro_model.trainable_variables))

   The optimizer, in this instance represented by tf.keras.optimizers.Adam, is tasked with modifying the weights. The function takes a list of gradient-variable pairings, obtained by using the zip() function on the gradients and the trainable variables of the acro_model. It then applies the gradients to the corresponding variables.

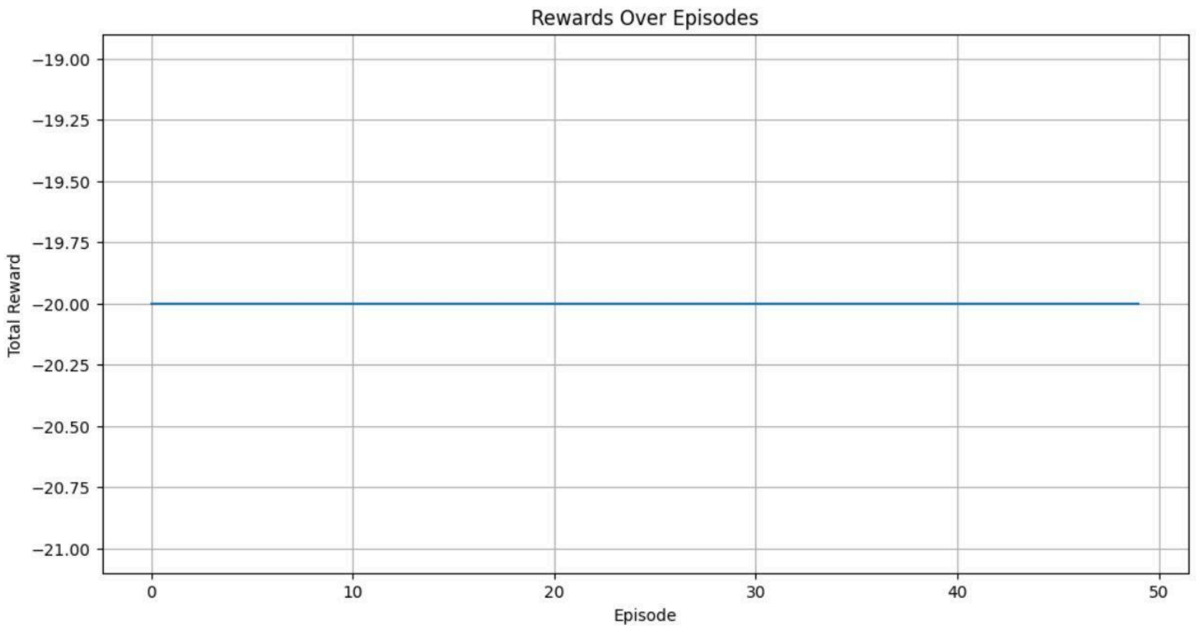
Thus, during this process, the agent gradually acquires the ability to estimate Q-values, which represent the anticipated total of future rewards for every combination of state and action. The iterative improvement of the Q-value predictions enables the agent to create a strategy that aims to maximise the total reward, hence enhancing its performance in the Acrobot challenge as time progresses.

4. Evaluation of Results

During the training of a reinforcement learning agent, it is essential to evaluate its performance via visual analysis of important indicators. Now, with the help of graphical aids, let us analyse in detail how the Acrobot job learned both before and after the introduction of personalised rewards.

Stagnation of Performance in Initial Training

Graph 4.1: Rewards Across Episodes Prior to the Implementation of Custom Rewards:



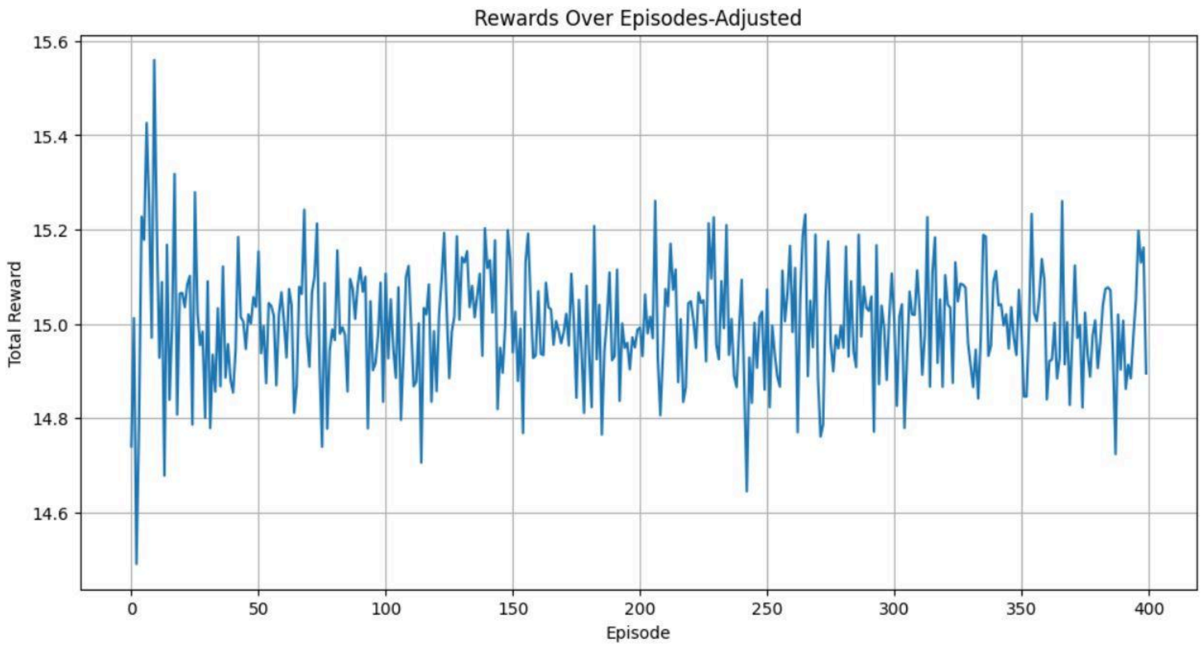
During the first training of the Acrobot agent without personalised rewards, the rewards over episodes plot exhibited a practically constant line, suggesting that the agent's performance did not demonstrate substantial improvement over time. The overall reward each episode remained consistent, with only marginal enhancements reported.

Investigation of Stagnation

The lack of a sufficient reward signal is responsible for the stagnation in performance, as it fails to provide guidance to the agent for achieving the desired behaviour. In intricate settings, particularly ones with limited or delayed rewards, an agent may encounter difficulty establishing a connection between its actions and the long-term outcomes, leading to a point in the learning process where no substantial enhancement in the agent's strategy is detected.

Optimising Learning with Tailored Incentives

Graph 4.2: After the introduction of custom rewards:



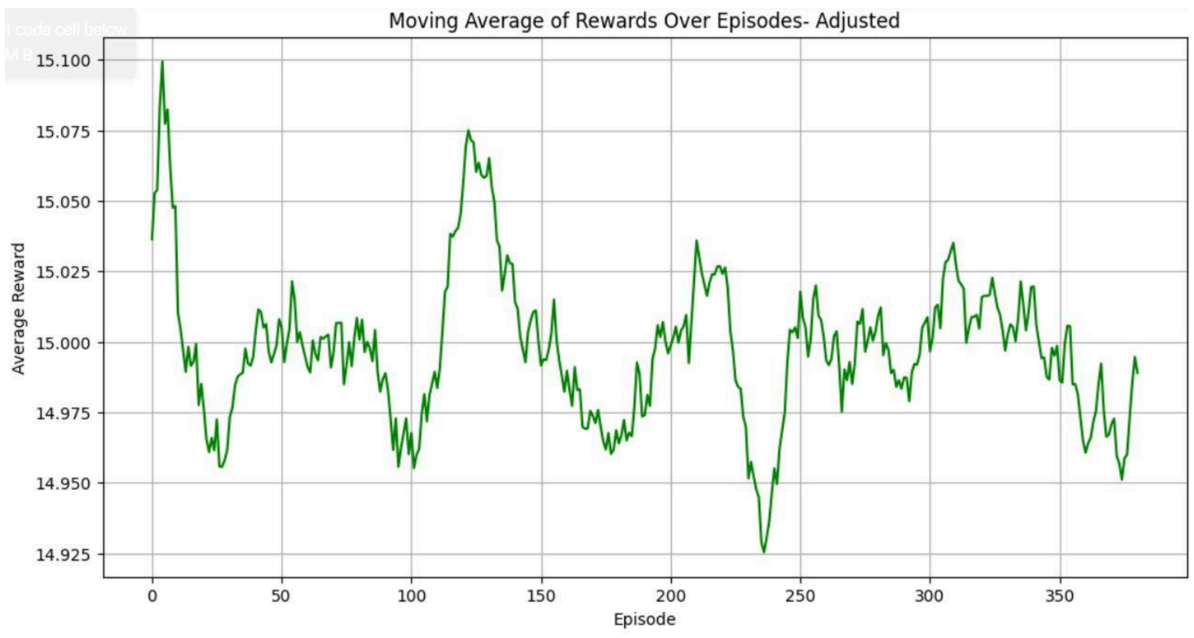
The modified rewards graph clearly demonstrates a significant shift in the agent's rate of learning. Fluctuations and increasing patterns in the overall reward per episode suggest that the agent has begun to acquire a more optimal strategy.

Effects of Customised Reward System

Implementing a customised reward system alters the dynamics of learning by offering the agent more frequent and meaningful feedback. The customised incentives provide a more distinct progression for the agent to adhere to, resulting in enhanced policy enhancements.

Reducing Variability to Uncover Patterns

Graph 4.3: Moving Average of Rewards Over Episodes:



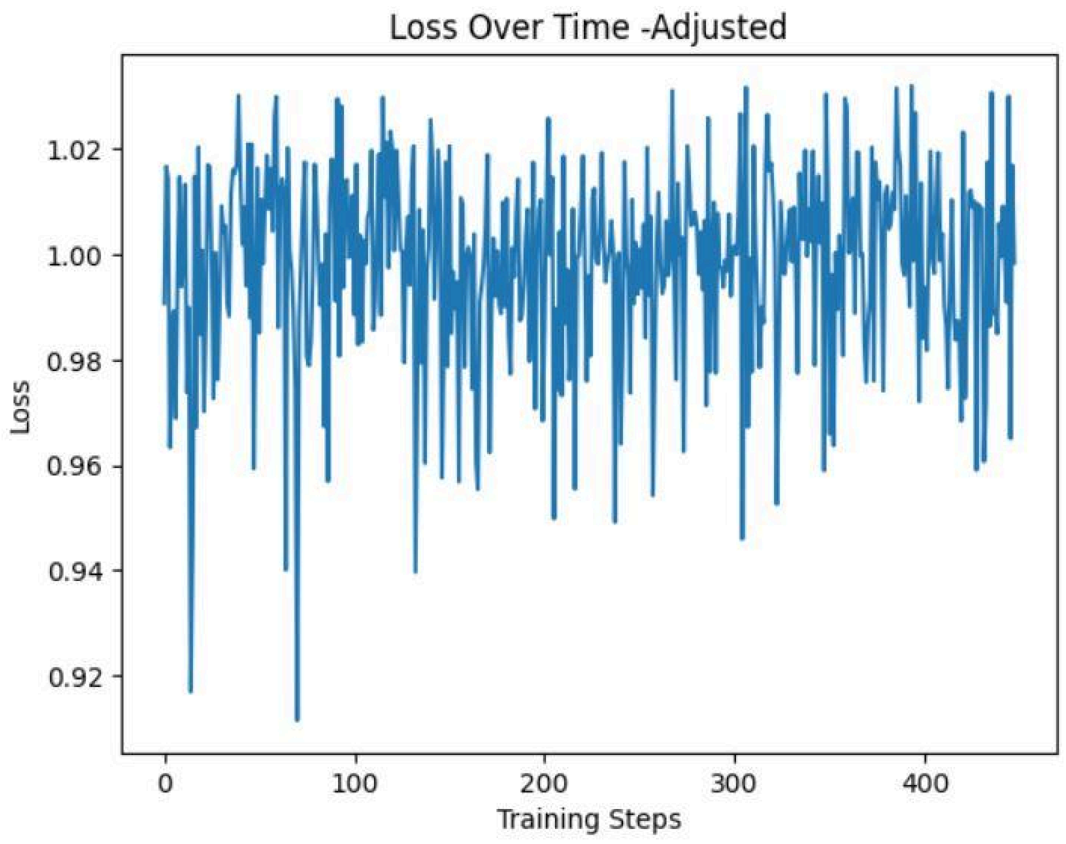
The moving average plot mitigates the inherent fluctuation in the per-episode incentives, providing a more distinct perspective on the underlying patterns in the agent's performance over time.

Findings from Trend Analysis

Examining the moving average of incentives enables us to discern the overall trend in the agent's learning trajectory. Upon making modifications, we observe periods of gradual acquisition of knowledge and occasional declines, which are inherent in the process of exploring and exploiting. The overall positive trend indicates that the agent is progressively enhancing its performance in the task.

Loss Reduction as a Measure of Learning Efficiency

Graph 4.4: Loss Over Time:



The loss over time graph functions as a gauge of the degree to which the Q-values anticipated by the neural network of the agent correspond with the target Q-values. It quantifies the agent's ability to learn effectively.

Reduction in Temporal-Difference Error

The declining trend in loss indicates that the agent is improving its ability to predict future rewards based on its actions, suggesting that the neural network's weights are being adjusted efficiently. The reduction in the temporal-difference error is an encouraging indication of the learning process.

Analysis of Linear Graphs Pre-Adjustment

Inadequate Reward Signals result in Limited Learning Progress

Before implementing personalised incentives, the reward graphs exhibit linear patterns, indicating a lack of learning. This occurrence can be attributed to the absence of informative input for the agent.

Implications of Inadequate Reward Structure

In the initial context, the incentive signal may have been insufficiently frequent or not sufficiently structured to effectively direct the agent towards the objective. The agent's ability to successfully learn which acts are desirable and which are not is hindered without clear and quick feedback on its actions.

Recognising the Significance of Reward Shaping in Reinforcement Learning

The graphical analysis after adjustment clearly illustrates the significance of reward shaping in reinforcement learning. Custom incentives enhance the agent's understanding and navigation of its environment towards the intended objective by offering a prompt and consistent indicator of the correct behaviours.

In short, the changes that were made to the reward system and the analysis of the data that came from them show that reward signals are very important for the learning process of a reinforcement learning agent. The transition from a linear trend in the rewards graph to a more fluid curve captures the fundamental nature of an enhanced learning path.

5. Animation of Acrobot

The YouTube video of the animation has been posted below:

https://youtu.be/3cVoL1ogJ6E?si=RWXIVatoaLwr-QPw

6. In-depth Analysis of Independently Researched Concepts

6.1 The effect of altering the number of neurons in a layer

The Architecture of Learning

The neural network's architecture, particularly the number of neurons in each layer, is crucial in determining the agent's capacity to understand and adapt to its surroundings in reinforcement learning.

Ability to Handle Complicated Situations

The capacity of a neural network to capture intricacies within the data is determined by the number of neurons in its hidden layer. Increasing the number of neurons in a network allows for the representation of more complex patterns. However, this also increases the risk of overfitting, when the network becomes too specialised in learning the training data, even irrelevant noise, leading to poor performance in new scenarios.

Achieving Balance

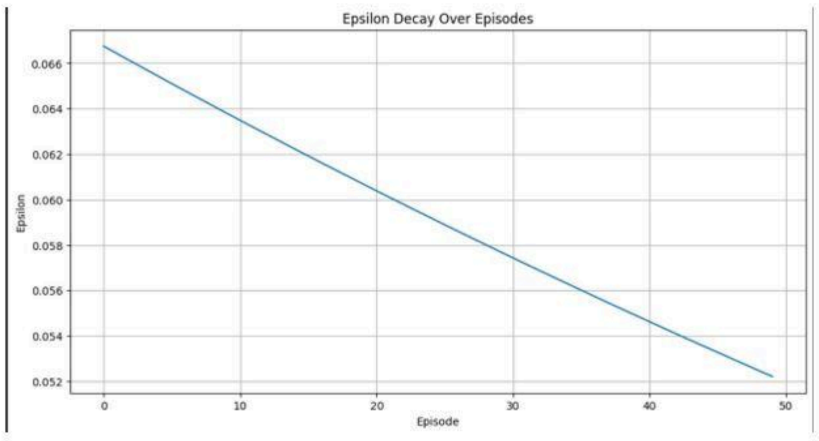
In contrast, a deficiency in the number of neurons can result in underfitting, a situation in which the neural network lacks the capacity to comprehend the fundamental structure of the issue domain. One must meticulously choose the optimal amount to strike a balance between overfitting and underfitting, which can be achieved by experimentation or model validation procedures.

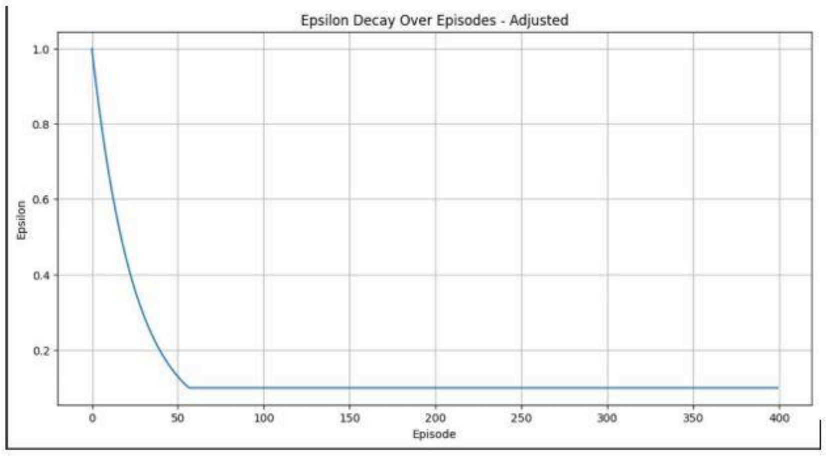
6.2 Epsilon decay and the trade-off between exploration and exploitation

Improving the Process of Exploration

Epsilon decay is a method employed in reinforcement learning to systematically decrease the level of exploration as time progresses. At first, a large epsilon value promotes exploration in order to uncover a maximum number of states. As the learning process advances, the value of epsilon decreases, and the agent gradually transitions towards exploitation, leveraging the acquired knowledge to make more informed judgements.

Epsilon Decay Analysis





The graphs presented illustrate the decline rate of epsilon over successive occurrences. In the unmodified technique, the degradation follows a linear and progressive pattern, ensuring a consistent transition from exploration to exploitation. In the modified strategy, the epsilon value decreases significantly at the beginning, enabling a quick transition to exploitation, which demonstrates a high level of trust in the acquired values.

Strategic Decision-Making amid Deterioration

The strategic decision regarding the rate at which epsilon decays has substantial consequences. If the decay rate is too fast, the agent may not be able to thoroughly explore all possible states, which might result in a suboptimal policy. Conversely, if the decay rate is too slow, the agent may keep exploring instead of utilising its acquired knowledge, resulting in a delay in reaching an optimal strategy.

It can be concluded that, the relationship between the neural network's size and the pace at which epsilon decays is a subtle feature of reinforcement learning that necessitates meticulous deliberation. Modifying these factors can significantly impact the learning results and overall performance of the agent in the activity.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). TensorFlow: A system for large-scale machine learning. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 265-283).
2. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D.,... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2
3. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. arXiv preprint arXiv:1606.01540.
4. Sutton, R.S., & Barto, A.G. (2018). *Reinforcement Learning: An Introduction*. MIT Press. Foundational understanding of RL concepts.
5. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. For comprehensive details on neural network architectures and capacity.
6. Mnih, V., et al. (2015). *Human-level control through deep reinforcement learning*. Nature, 518 (7540), 529–533. - Demonstrates practical application of varying neural network sizes and epsilon decay in the context of deep Q-Networks.
7. https://www.gymlibrary.dev/environments/classic_control/acrobot/