**A blue shield with a cross and a book

Description automatically generated**

**Project 3 For Diploma in AI and its application in Business:**

**Reinforcement Learning with Q and Deep Q-Learning**

**Project Report**

*Reinforcement Learning and Example Implementation on Q and Deep Q-learning*

Prepared by: Chen Yijie

Supervisor: Zhou Changxin

Table of Contents

[Covering Concepts: 3](#_Toc164095314)

[Potential Use Case: 5](#_Toc164095315)

[Environment Used: 6](#_Toc164095316)

[Q-Learning 7](#_Toc164095317)

[**Environment Used:** 7](#_Toc164095318)

[**Environment Understanding:** 8](#_Toc164095319)

[**Q-Learning Implementation:** 11](#_Toc164095320)

[**Training Parameter:** 13](#_Toc164095321)

[**Initial Results:** 14](#_Toc164095322)

[**Learning Rate Decay** 16](#_Toc164095323)

[**Intermediate Result** 17](#_Toc164095324)

[**Episode Number** 18](#_Toc164095325)

[**Final Result** 19](#_Toc164095326)

[**Potential Improvements** 20](#_Toc164095327)

[Deep Q-Learning Network (DQN) 20](#_Toc164095328)

[**Environment Used** 20](#_Toc164095329)

[**Environment Understanding:** 21](#_Toc164095330)

[**Deep Q-Learning Implementation:** 24](#_Toc164095331)

[**Training Parameter:** 29](#_Toc164095332)

[**Results:** 31](#_Toc164095333)

[**Potential Improvements to be Done:** 33](#_Toc164095334)

[Conclusion: 34](#_Toc164095335)

**Project Overview:**   
The project aims to implement reinforcement learning techniques, focusing on Q-learning and Deep Q-learning networks (DQN), to develop an autonomous decision-making system. Q-learning is a foundational algorithm in reinforcement learning that enables agents to learn optimal policies by iteratively updating action-value functions based on observed rewards. Deep Q-learning extends Q-learning by employing deep neural networks to approximate Q-values, allowing for more complex state-action mappings. The project will explore both algorithms independently using separate environments to understand their principles and implementations. While real-world applications are limited due to the lack of a specific dataset, the project serves as an educational endeavor to deepen understanding of reinforcement learning concepts and methodologies.

## **Covering Concepts:**

1. **Agent and Environment:** In reinforcement learning, an agent interacts with an environment, where the agent takes actions, and the environment responds with rewards and new states. The agent's goal is to learn a policy that maximizes cumulative rewards over time.
2. **Temporal Difference Learning:** Temporal Difference (TD) learning is a method used in reinforcement learning for updating value estimates based on the observed rewards and predictions of future rewards. It combines ideas from dynamic programming and Monte Carlo methods and is the basis for algorithms like Q-learning and SARSA.
3. **Propagation:** In the context of reinforcement learning, propagation refers to the process by which information about rewards and state transitions propagates backward through time. This is essential for updating value estimates and learning optimal policies.
4. **Bellman's equation:** Bellman's equation is a fundamental concept in reinforcement learning, providing a recursive relationship between the value of a state and the values of its successor states. It expresses the value of a state as the sum of the immediate reward obtained in that state and the expected value of the successor states. Bellman's equation serves as the basis for many reinforcement learning algorithms, including Q-learning, by guiding the update of action values towards optimality. It facilitates the estimation of state values and the derivation of optimal policies through iterative value iteration or policy iteration methods.
5. **Q-learning:** Q-learning is a model-free reinforcement learning algorithm used to learn the quality of actions in a given state. It updates action values based on the difference between the estimated and observed rewards, following the Bellman equation. Q-learning is off-policy and has been widely applied in various domains.
6. **Deep Q-Network (DQN):** DQN is a deep reinforcement learning algorithm that combines Q-learning with deep neural networks. It uses a neural network to approximate the Q-function, enabling it to handle high-dimensional state spaces efficiently. DQN introduced techniques like experience replay and target networks to stabilize training and improve convergence.
7. **Exploration vs. Exploitation:** In reinforcement learning, agents face the exploration-exploitation dilemma, where they must balance between trying new actions to discover optimal strategies (exploration) and exploiting known good actions to maximize immediate rewards (exploitation). Strategies for addressing this trade-off are crucial for effective learning.
8. **Reward Function:** The reward function in reinforcement learning defines the goal of the agent by specifying the immediate feedback or reinforcement it receives after taking an action in a particular state. Designing an appropriate reward function is essential for guiding the learning process towards desired outcomes and behaviors.
9. **Policy Networks:** Policy networks are used in reinforcement learning to directly learn the policy function, mapping states to actions without explicitly computing value functions. These networks can be trained using techniques like policy gradients to optimize for long-term rewards.
10. **Experience Replay:** Experience replay is a technique used in deep reinforcement learning, particularly in DQN, where the agent stores experiences (state, action, reward, next state) in a replay buffer and samples mini-batches during training. This helps in breaking the temporal correlation in the data and stabilizing learning.
11. **Target Networks:** Target networks are used in DQN to stabilize training by decoupling the target Q-network from the network being updated. This involves periodically updating the parameters of the target network to prevent the moving target problem, where the target values oscillate during training.

## **Potential Use Case:**

Reinforcement learning (RL) has a wide range of potential use cases across various domains due to its ability to learn optimal behaviors through trial and errors. The potential usages are as such:

1. **Autonomous Robotics**: RL can be used to train robots to perform complex tasks such as navigation, manipulation of objects, or assembly line operations. Robots can learn from their interactions with the environment to optimize their actions and adapt to dynamic situations.
2. **Game Playing**: RL algorithms have demonstrated impressive performance in playing complex strategy games like chess, Go, and video games. These algorithms can learn effective strategies and tactics by playing against opponents or exploring different game environments.
3. **Recommendation Systems**: RL can be used to personalize recommendations in various domains such as e-commerce, content streaming, and online advertising. By learning from user interactions, RL algorithms can optimize recommendations to maximize user engagement or satisfaction.
4. **Finance and Trading**: RL algorithms can be applied to optimize trading strategies in financial markets by learning from historical data and market dynamics. They can adapt to changing market conditions and optimize portfolio management to maximize returns while managing risk.
5. **Supply Chain Management**: RL can optimize inventory management, logistics, and routing decisions in supply chain operations. By learning from historical data and real-time information, RL algorithms can optimize decision-making to reduce costs, minimize delays, and improve overall efficiency.
6. **Healthcare**: RL can be used for personalized treatment planning, drug discovery, and medical diagnosis. RL algorithms can learn optimal treatment policies by analyzing patient data and treatment outcomes, leading to more effective and personalized healthcare interventions.
7. **Energy Management**: RL can optimize energy consumption, production, and distribution in smart grid systems. RL algorithms can learn to control energy generation and storage systems to maximize efficiency, reduce costs, and minimize environmental impact.
8. **Advertising and Marketing**: RL can optimize online advertising campaigns by learning to allocate resources effectively across different channels and targeting strategies. RL algorithms can adapt to changing user preferences and market trends to maximize advertising ROI.
9. **Automated Decision Making**: RL can automate decision-making processes in various domains such as customer service, resource allocation, and scheduling. RL algorithms can learn to make optimal decisions in complex and uncertain environments, reducing the need for human intervention.
10. **Drug Discovery and Development**: RL can be used to optimize drug discovery pipelines by guiding the selection and optimization of candidate compounds. RL algorithms can learn to design and test molecules in silico, accelerating the drug discovery process and reducing the cost of bringing new drugs to market.

## **Environment Used:**

Python programming language with:

1. gym = ^0.26.2

2. swig = ^4.1.1

3. box2d-py = ^2.3.8

4. pygame =^ 2.5.2

6. tensorflow = ^2.15.0

7. keras = ^2.15.0

8. tensorboard = ^2.15.0

9. numpy = ^1.26.3

## **Q-Learning**

### **Environment Used:**

For Q-learning, we will be using the Blackjack Environment from Gymnasium (Gym). Gym's Toy Text Blackjack environment offers a simplified rendition of the classic card game, providing an ideal setting for learning Q-learning algorithms. In this environment, the agent's objective is to achieve a hand value as close to 21 as possible without exceeding it. With a straightforward state space defined by the player's hand value, the dealer's visible card, and the presence of a usable ace, the environment enables the agent to explore and learn efficiently. The limited action space, consisting solely of "hit" or "stick," simplifies decision-making and allows the agent to focus on learning optimal strategies. Immediate rewards, directly tied to the agent's actions, provide clear feedback and facilitate reinforcement learning. Additionally, the episodic nature of the game, with distinct start and end points for each round, allows the agent to learn from individual experiences and episodes, gradually improving its policy over time. Overall, Gym's Toy Text Blackjack environment offers a conducive learning environment for Q-learning algorithms, enabling agents to efficiently navigate the game space and converge towards optimal gameplay strategies.

A screenshot of a game

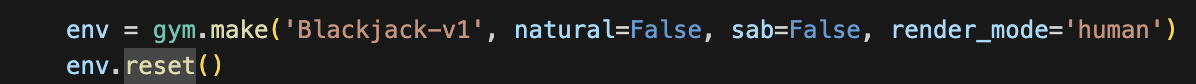
Description automatically generated

Fig 1. Sample Game Play from Black Jack Environment

### **Environment Understanding:**

Learning the environment thoroughly is essential to implement solutions for it. This project used some of the common practices including reading documentations and implementing codes to learn about blackjack environment and collect information crucial for developing Q-Learning solution.

1. **Rendering the Game Window:** Rendering the game window involves displaying the environment or game interface to the agent so it can observe and interact with it. This step typically includes rendering graphics, if applicable, and providing visual feedback to the agent about the current state of the environment.

Fig 2. Code Implementation to Render the Environment for Visual Analysis

A screenshot of a game

Description automatically generated

Fig 3. PyGame Window Visualisation of Blackjack Game

1. **Finding out the Action Space:** The action space represents the set of all possible actions that the agent can take in the environment. In the Blackjack Environment’s setting, the action shape is an integer or either 0 or 1, indicating whether to stick or hit. 0 is stick and 1 means hit.

A screen shot of a computer

Description automatically generated

Fig 4. Code Implementation to Find out the Action Space

1. **Finding out the Observation Space:** The observation space defines the set of all possible states or observations that the agent can perceive from the environment. In the blackjack environment, the observation is defined as a 3-tuple containing: the player’s current sum, the value of the dealer’s one showing card (1-10 where 1 is ace), and whether the player holds a usable ace (0 or 1). The observation is returned as a tuple of 2 integers and a Boolean.

A black screen with blue text

Description automatically generated

Fig 5. Code Implementation to Find out the Observation Shape

1. **Determining the Game Reward:** The game reward is the feedback provided to the agent after each action, indicating the immediate consequences of its decisions. Rewards can be positive, negative, or zero, depending on the outcome of the agent's actions and their alignment with the desired objectives. In the context of Blackjack, agent will score a reward of +1 for winning a game, -1 for losing a game and 0 for drawing the game.
2. **Defining Termination Conditions:** Termination conditions determine when an episode or game ends, indicating the completion of a task or achieving a specific objective. In the Blackjack environment, termination happens when the player hits and the sum of hand exceeds 21 or when the player decide to stick.

A black screen with white text

Description automatically generated

Fig 6. Code Implementation to Find out Reward and Termination

Bottom of Form

### **Q-Learning Implementation:**

A few essential steps are implemented to achieve the Q-learning implementation.

1. **Initializing the Q-Table:** The implementation begins by creating a Q-table using NumPy's zeros function. This table serves as a data structure to store action values for each state-action pair in the environment. The dimensions of the table are the total number of all possible observations and actions. In Blackjack’s observation, the first integer representing player’s current sum have 32 possible permutations. While the value of opponent’s showing card contains 11 possibilities, and a Boolean of whether the player has an Ace card. The action space of the environment is a Boolean of whether to hit or stick as well, making the dimension of our Q-table containing all possibilities to be (32,11,2,2).



Fig 7. Code Implementation to Initialize Q-Table of Dimension 32,11,2,2

1. **Defining the Get Action Function:** The next step involves defining a function to select actions based on the current state and the Q-table. The Get Action Function serves as a pivotal component in the Q-learning implementation, responsible for guiding the agent's decision-making process. By leveraging the Q-table, which encapsulates the agent's learned knowledge, this function selects actions based on the highest expected reward, thus exploiting the known information. However, to ensure a balance between exploiting learned knowledge and exploring new possibilities, the function incorporates an epsilon-greedy strategy. This strategy enables the agent to occasionally choose random actions, fostering exploration and allowing the discovery of potentially better strategies. Through this delicate balance, the Get Action Function drives the agent's learning process, facilitating optimal decision-making in diverse and dynamic environments.

A screen shot of a computer code

Description automatically generated

Fig 8. Code Implementation to Look Up Q-Table and return Corresponding Action

1. **Defining the Update\_Q Function with Bellman Equation:** The update\_Q function is defined to update the Q-values in the Q-table based on observed rewards and transitions. This function implements the Bellman equation, which updates the Q-value for a state-action pair by combining the immediate reward with the discounted maximum future rewards. By iteratively applying the Bellman equation through running the environments, the agent learns new observation and rewards pair and the Q-values in the Q-table are gradually refined to approximate the optimal action-value function.

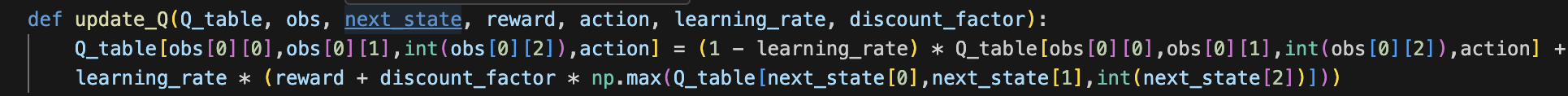


Fig 9. Code Implementation that Utilizes Bellman’s Equation to Update Q Value

1. **Implementing a Decay Function:** A decay function is implemented to gradually reduce the exploration rate over time. The exploration rate (Epsilon) determines the balance between exploration (trying new actions) and exploitation (using learned knowledge) in the Q-learning algorithm. By decaying the exploration rate over episodes, the agent gradually shifts from exploring to exploiting as it gains more experience in the environment. This helps in improving the convergence and stability of the Q-learning algorithm.

A computer screen with text on it

Description automatically generated

Fig 10. Code Implementation to Decay Exploration Rate

### **Training Parameter:**

In training the Q-learning agent for reinforcement learning, specific hyperparameters are chosen based on their impact on the learning process and the characteristics of the environment. Here's an explanation of the reasons for the chosen hyperparameters:

1. **Epochs**: The number of epochs is set to 10,000 to ensure that the agent has sufficient time to explore the environment, learn from its experiences, and converge towards optimal policies. This value strikes a balance between training time and the agent's ability to learn complex behaviors.
2. **Learning Rate (Alpha)**: A learning rate of 0.1 is chosen as it provides a moderate step size for updating the Q-values. This value allows for significant learning progress in each epoch while ensuring stability by preventing overly drastic updates that may lead to oscillations or divergence in the learning process.
3. **Discount Factor (Gamma)**: With a discount factor of 1, future rewards are given full consideration, indicating that the agent values long-term rewards equally to immediate rewards. This choice is suitable when the environment does not involve delayed rewards or when the agent needs to prioritize long-term planning over short-term gains.
4. **Exploration Rate (Epsilon)**: Starting with an exploration rate of 1 allows the agent to explore the environment extensively at the beginning of training, ensuring that it explores a wide range of actions and states. The exploration rate decays over time according to a decay rate of 0.999, gradually shifting the agent's focus towards exploiting learned knowledge as training progresses. The minimum exploration rate is set to 0.1 to ensure that the agent continues to explore the environment even after significant training, preventing premature convergence to suboptimal policies.
5. **Decay Rate**: A decay rate of 0.999 is chosen to ensure a slow and gradual decrease in the exploration rate over epochs. This gradual decay allows the agent to balance exploration and exploitation effectively, gradually transitioning from exploration to exploitation as it gains more experience in the environment.

A screen shot of a computer code

Description automatically generated

Fig 11. Hyperparameters Used for Training

A computer screen with white text

Description automatically generated

Fig 12. Code Implementation for Q-Agent Training

### **Initial Results:**

As there is no validation dataset for reinforcement learning, the only way to test out the model is through running the agent through the game and view its reward. To ensure fairness, the agent is put to perform in the Blackjack game for 100 times, the indicator for its effectiveness will be the average reward it scored over 100 game episodes.

The trained agent achieved an average reward of -0.25, which is an improvement from the baseline reward of -0.53 for using all random actions through 100 episodes. However, this score can still be improved as a negative reward means that we are still losing money.

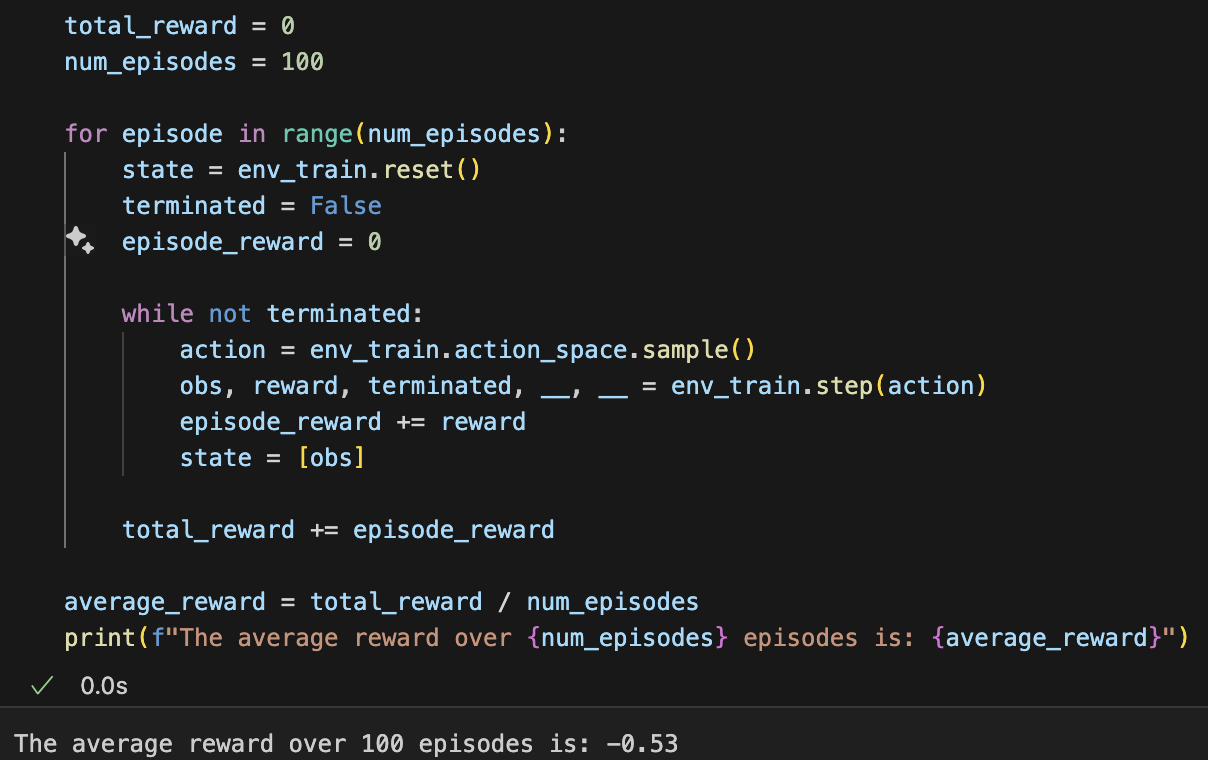


Fig 13. Baseline Average Reward of -0.53

A screen shot of a computer program

Description automatically generated

Fig 14. Agent Achieving Score of -0.25 after Training.

### **Learning Rate Decay**

The Agent’s suboptimal performance might be a result of high learning rate, having a high learning rate in Q-learning can introduce instability and hinder the convergence of the learning process. This instability arises from the overly aggressive updates to the Q-values, which may oscillate or diverge, preventing the agent from accurately learning optimal policies.

To explore the effect of learning rates, the learning rate hyperparameter is put through the same decay function as the epsilon function with the lowest learning rate set to 0.005. The agent is then put through a fresh 10,000 episodes of training.

A screen shot of a computer program

Description automatically generated

Fig 15. Hyperparameters and Code Implementation for Re-training

### **Intermediate Result**

Putting the retrained agent through 100 episodes of game achieved an average reward of -0.25, which barely improved from the previous model.

A screen shot of a computer program

Description automatically generated

Fig 16. Code Implementation and Results for Re-Trained Agent

### **Episode Number**

The next logical suspicion is that the episode number is too low and thus the agent was not able to converge. Having a low number of episodes in Q-learning can limit the agent's exposure to the environment and hinder its ability to learn optimal policies effectively. With a limited number of training episodes, the agent may not have sufficient opportunities to explore different states and actions, resulting in incomplete learning and suboptimal decision-making. Additionally, a low episode number may lead to a lack of diversity in the agent's experiences, causing it to overlook potentially valuable strategies or solutions to the given task. As a result, the agent's learned policies may be less robust and adaptable to variations in the environment, limiting its overall performance and effectiveness.

To explore the effect of episode number, a third agent is trained with the exact same parameters except that the episode number is tuned to 1,000,000 from 10,000.

A screen shot of a computer program

Description automatically generated

Fig 17. Hyperparameters Used and Code Implementation for Third Agent’s Training

### **Final Result**

The third agent achieved an average reward of 0.09 through 100 episodes of game. This is a huge improvement from the first and second agent’s performance, showing that having a higher number of training episodes do introduce effect to our agent’s performance. On top of that, it is even more significant as a positive average reward means that the agent will be earning money when it is deployed to a real-life scenario. Given that Blackjack is not only a luck game, the agent can be considered as doing well.

### **Potential Improvements**

Some potential improvements that can be implemented includes:

* 1. Further finetune the agent by changing hyperparameters.
  2. Average across more game episodes to get a fairer indicator.

## **Deep Q-Learning Network (DQN)**

### **Environment Used**

For DQN, we will be using the mountain car environment from Gymnasium(Gym). The Mountain Car environment from Gymnasium presents a classic reinforcement learning challenge where an underpowered car must navigate a steep uphill slope to reach a goal position. The car has limited engine power, making it unable to reach the goal by directly driving up the mountain. Instead, the car must learn to build momentum by moving back and forth, utilizing the principles of physics and momentum to overcome the gravitational pull and reach the goal. This environment poses a challenging task for reinforcement learning agents, requiring them to learn efficient strategies for balancing exploration and exploitation to achieve successful navigation of the mountain terrain.

A black and grey line with a black object on it

Description automatically generated with medium confidence

Fig 18. Sample Screenshot from Mountain Car Environment

### **Environment Understanding:**

To effectively implement solutions for the Mountain Car environment, thorough understanding of its mechanics is crucial. This project utilized common practices such as reading documentation and implementing code to gather essential information necessary for developing a Q-Learning solution.

1. **Rendering the Game Window:** Rendering the game window involves presenting the environment or game interface to the agent for observation and interaction. In the case of the Mountain Car environment, visualization involve displaying a 2D representation of the mountain terrain and the position of the car.

A screenshot of a video game

Description automatically generated

Fig 19. PyGame Window Visualisation of MountainCar Game

1. **Finding out the Action Space:** The action space in the Mountain Car environment represents the set of all possible actions that the agent can take. In this setting, actions include applying force to the car in different directions, such as accelerating to the left, accelerating to the right, or maintaining the current velocity.

A screen shot of a computer

Description automatically generated

Fig 20. Code Implementation and Result to Find out the Action Space

1. **Finding out the Observation Space:** The observation space defines the set of all possible states or observations that the agent can perceive from the environment. In the Mountain Car environment, observations may include the position and velocity of the car, as well as information about the terrain, such as the slope of the hill.

A black screen with white text

Description automatically generated

Fig 21. Code Implementation and Results to Find out the Observation Shape

1. **Determining the Game Reward:** The game reward is the feedback provided to the agent after each action, indicating the immediate consequences of its decisions. In Mountain Car, the agent may receive a negative reward for each time step until it reaches the goal position, motivating it to minimize the number of time steps required to reach the goal.

A black background with white text

Description automatically generated

Fig 22. Gymnasium’s Documentation on Mountain Car Environment’s Reward

1. **Defining Termination Conditions:** Termination conditions determine when an episode or game ends, indicating the completion of a task or achieving a specific objective. In the Mountain Car environment, termination occurs when the car reaches the goal position or when a maximum number of time steps is reached.

A black background with white text

Description automatically generated

Fig 23. Gymnasium’s Documentation on Mountain Car Environment’s Stop ConditionBottom of Form

### **Deep Q-Learning Implementation:**

This project implements Deep-Q learning with an object-oriented approach consists of the following steps.

1. **Introducing an Object-Oriented Agent**:
   * Creating an agent class that interacts with the environment and learns to make decisions based on the observations received.
   * The agent class contains methods for actions like selecting an action, updating the Q-network, and training.
   * Initialize all required parameters

A screen shot of a computer program

Description automatically generated

Fig 24. DeepQAgent’s Initialization

1. **Define Replay Buffer (Deque)**:
   * The replay buffer is a data structure used to store experiences (state, action, reward, next state) encountered by the agent during interactions with the environment.
   * It helps in breaking the temporal correlation in the training data and making the learning process more stable.
   * The replay buffer is implemented using a deque (double-ended queue) from the collections module. (Refer to Fig 24, “Self.replaybuffer”)
2. **Define the Neural Network Used Consisting of 3 Dense Layers**:
   * Deep Q-learning utilizes a neural network to approximate the Q-function, which estimates the expected future rewards for taking a particular action in a given state.
   * The neural network typically consists of multiple dense layers (also known as fully connected layers). Due to the simplicity of the project, the structure of DQN is set to only consists of 3 fully connected dense layers.
   * The input size of the network is determined by the dimensionality of the state space, in MountainCar’s case it will be tuple of 2 floating number. The output size is determined by the dimensionality of the action space, in MountainCar’s case it will be a discrete integer of either 0,1 or 2.
   * The model is then compiled with gradient descent optimization and the loss function of mean squared error between the predicted Q-values and the target Q-values.

A screen shot of a computer code

Description automatically generated

Fig 25. Code Implementation to Create Network

1. **Define Get Action**:
   * The get\_action method is responsible for selecting actions based on the current state and the Q-network's predictions (Output).
   * Epsilon-greedy strategy is employed to balance between exploring and exploiting, where with a certain probability (epsilon), the agent selects a random action to explore the environment, while with probability (1 - epsilon), it outputs the action with the highest Q-value.

A screen shot of a computer code

Description automatically generated

Fig 26. Code Implementation to Get Action

1. **Define Network Updating**:
   * The network\_update method updates the parameters of the Q-network based on sampled experiences from the replay buffer.
   * Current Q-Values are predicted and calculated.
   * Bellman equation is implemented on current Q-Values to estimate the future Q-values.
   * Target Q-Values are predicted using the observation from 1 time step later.
   * The network is then trained using the predicted Q-values and future Q-values to minimize the differences, making the model predicting the “Future” more accurately

A screenshot of a computer program

Description automatically generated

Fig 27. Code Implementation to Update Network

1. **Define Training Step**:
   * The training\_step method performs a single step of the training process.
   * It samples a batch of experiences from the replay buffer, updates the Q-network based on these experiences, and optionally updates the exploration rate (epsilon).
   * This method is called repeatedly during the training process to iteratively improve the Q-network's performance.

A computer screen shot of a program code

Description automatically generated

Fig 27. Code Implementation for Training Step

1. **Define Playing**:
   * The playing method allows the trained agent to interact with the environment and make decisions based on the learned Q-values.
   * It typically involves running the agent in the environment for a certain number of episodes or until a termination condition is met.
   * During playing, the agent selects actions based on the current state and the learned Q-values, without further updating the Q-network parameters.

A computer screen shot of text

Description automatically generated

Fig 28. Code Implementation for Agent to Perform in the Environment

### **Training Parameter:**

In training the Deep Q-learning agent for reinforcement learning, specific hyperparameters are chosen based on their impact on the learning process, similar to the Q-Learning training process:

1. **Epsilon (Exploration Rate):**
   * The exploration rate (epsilon) is initially set to 1. This high exploration rate ensures that the agent explores the environment extensively at the beginning of training, helping it to discover various states and actions.
   * As training progresses, the epsilon value decays gradually over time. A decay rate of 0.999 is chosen, indicating that the exploration rate decreases slowly over epochs.
   * The minimum exploration rate is set to 0.1, ensuring that the agent continues to explore even after significant training. This prevents premature convergence to suboptimal policies and allows the agent to exploit learned knowledge effectively.
2. **Learning Rate:**
   * A learning rate of 0.0001 is chosen. This low learning rate provides a small step size for updating the Q-network parameters. It helps in stabilizing the learning process and prevents drastic updates that may lead to oscillations or divergence.
   * With a low learning rate, the agent can make small adjustments to the Q-values based on the observed experiences, ensuring gradual learning and convergence towards optimal policies.
3. **Discount Factor (Gamma):**
   * The discount factor is set to 0.99. This value indicates that the agent values future rewards slightly less than immediate rewards, encouraging it to consider both short-term and long-term consequences of its actions.
   * A discount factor of 0.99 strikes a balance between giving due consideration to future rewards and ensuring stability in the learning process. It allows the agent to prioritize near-term rewards while still planning for long-term goals.
4. **Epochs:**
   * The number of epochs is set to 2000. This value determines the number of iterations or training episodes the agent undergoes.
   * With 2000 epochs, the agent has a sufficient number of training episodes to explore the environment, learn from experiences, and converge towards optimal policies without excessively prolonging the training process.
5. **Batch Size:**
   * The batch size is set to 32. This batch size determines the number of experiences sampled from the replay buffer for each training iteration.
   * A batch size of 32 strikes a balance between computational efficiency and stability in the learning process. It ensures that the agent learns from a diverse set of experiences while avoiding excessive memory usage or computational overhead.

A screen shot of a computer code

Description automatically generated

Fig 29. Hyperparameters Used for Training



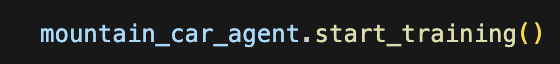


Fig 30 &31. Code Implementation to Start Training

### **Results:**

The Deep Q-Learning Agent has first successfully complete the environment by reaching the flag in episode 1030, and the frequency of success increases steadily, showing signs of converge.

A screen shot of a computer

Description automatically generated

Fig 32. Training Results for DQN Agent

Further testing the agent through playing it in the environment achieved around 8 successes out of 10 tries. Through rendering the game play, a dead spot can be found on the game screen where agent spawned near the location is unable to act. The model is able to perform at any other spawn locations, achieving the goal in less than 130 steps.

A screen shot of a computer program

Description automatically generated

Fig 33. Testing Results for Trained DQN Agent

A line with a red circle

Description automatically generated

Fig 34. Spawning Dead Spot that the Model is Unable to Perform

### **Potential Improvements to be Done:**

1. The DeadSpot might be due to the effect of simple neural network structure. A more complex structure of the neural network can be introduced at the expenses of computing resources to achieve better results.
2. Hyperparameter tuning: Further optimize model performance by adjusting parameters such as batch size, and optimizer choice.

## **Conclusion:**

In conclusion, this project has provided a thorough exploration into the field of reinforcement learning (RL), specifically focusing on the implementation of Q-learning and Deep Q-learning Network (DQN). Beginning with foundational concepts, the project elucidated the principles underlying RL, where agents iteratively learn to make decisions in order to maximize cumulative rewards.

The study progressed to Q-learning, a fundamental RL algorithm, which facilitates agents in updating action values based on observed rewards and transitions, gradually converging towards optimal policies. Subsequently, the project extended its scope to Deep Q-learning, a sophisticated technique that employs neural networks to approximate the Q-function, enabling agents to effectively handle complex environments with high-dimensional state spaces.

Through an object-oriented approach, agents were designed to interact with the environment, select actions, and update their knowledge using experiences stored in a replay buffer. The definition of neural network architectures, training processes, and hyperparameters facilitated the agents in learning effective strategies while balancing exploration and exploitation.

Moreover, the project elucidated critical concepts such as exploration-exploitation trade-offs, epsilon-greedy strategies, and replay buffers, which are fundamental to understanding and implementing RL algorithms proficiently. By bridging theory with practical implementation, learners gained valuable experience in developing and training RL agents, equipped with the skills to address diverse challenges across different environments.

In summary, this project has offered a comprehensive understanding of Q-learning and DQN methodologies, providing a solid foundation for further exploration and application of reinforcement learning techniques in various real-world scenarios. From fundamental principles to advanced implementations, learners are now prepared to engage in the field of RL with confidence and proficiency.Top of Form

Bottom of Form