MOUNTAIN CAR PROBLEM USING REINFORCEMENT LEARNING

A Project Report

Submitted by

HANIEL EDWARD JACOB T - 312318104052

JAGADEESH D - 312318104064

in partial fulfilment for the award of the Degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

St. JOSEPH'S COLLEGE OF ENGINEERING

(An Autonomous Institution)

St. Joseph's Group of Institutions

Jeppiaar Educational Trust

OMR, Chennai 600 119

ANNA UNIVERSITY: CHENNAI 600 025

JUNE 2022

ANNA UNIVERSITY

BONAFIDE CERTIFICATE

Certified that this project report on ***MOUNTAIN CAR PROBLEM USING REINFORCEMENT LEARNING*** *is the bonafide work of* ***HANIEL EDWARD JACOB.T (312318104052)*** *and* ***JAGADEESH.D (312318104064)*** *who carried out the project under my supervision during* ***the academic year 2021 - 2022.***

Signature of the
Head of the Department

Dr. A. Chandrasekar
M.E., Ph.D

Professor and Head

Department of Computer Science and Engineering

St. Joseph's College of Engineering

OMR, Chennai- 600119

Signature of the Supervisor

Dr. F. Sangeetha Francelin Vinnarasi

M.Tech., Ph.D

SUPERVISOR

Associate Professor

Department of Computer Science and Engineering

St. Joseph's College of Engineering

OMR, Chennai- 600119

CERTIFICATE OF EVALUATION

COLLEGE NAME **:** St. Joseph's College of Engineering,

BRANCH **:** B.E. - Computer Science and Engineering

SEMESTER **:** VIII

Sl. No	Name of the Student	Title of The Project	Name of the Supervisor with Designation
1. 2.	Haniel Edward Jacob. T (312318104052) Jagadeesh. D (312318104064)	**MOUNTAIN CAR PROBLEM USING REINFORCEMENT LEARNING** **.**	**Dr. F. Sangeetha Francelin Vinnarasi** M.Tech., Ph.D Associate Professor

The report of the project work submitted by the above students in partial fulfillment for the award of the Degree of Bachelor of Engineering in Computer Science and Engineering at Anna University is confirmed to be report of the work done by the above students and then evaluated.

Submitted to Project and Viva Examination held on_____.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

At the outset, we would like to express our sincere gratitude to the **ALMIGHTY** and our beloved ***Chairman, Dr. B. Babu Manoharan M.A., M.B.A., Ph.D St. Joseph's Group of Institutions*** for his constant guidance and support to the student community and the Society.

We would like to express our hearty thanks to our respected ***Managing Director, Mrs. S. Jessie Priya M.Com. St. Joseph's Group of Institutions*** for her kind encouragement and blessings. We wish to express our sincere thanks to our ***Executive Director Mr. B. Shashi Sekar, M.Sc. St. Joseph's Group of Institutions*** for providing ample facilities in the institution.

We would like to express sincere gratitude to our beloved and highly respected ***Principal Dr. Vaddi Seshagiri Rao M.E., M.B.A., Ph.D., F.I.E.*** for his inspirational ideas during the course of the project. We would like to express sincere gratitude and our utmost respect to our beloved ***Dr. B. Parvathavarthini M.E., M.B.A., Ph.D., Dean (Research)*** for her inspirational ideas during the course of the project.

We would like to acknowledge our gratitude to our supervisor **Dr. F. Sangeetha Francelin Vinnarasi, M.Tech., Ph.D** for her excellent guidance and connoisseur's suggestion throughout the study carried out successfully.

Finally, we thank the **Faculty Members** and **our Family**, who helped and encouraged us constantly to complete the project successfully.

ABSTRACT

RL algorithms have made significant progress in recent years in the fields of gaming, healthcare, robotics and much more. In this project, we solve the mountain car problem using QL, DQL and SARSA algorithms. We also compare the efficiency of each of these algorithms to determine the best algorithm to solve this problem. These algorithms are model free RL algorithms which means that they learn from previous experiences. The Mountain Car problem is a simple but intriguing problem which is used to test different reinforcement learning algorithms. It involves a virtual mountain car which must drive up a steep hill in order to reach the other side of the hill (goal). The car cannot climb the entire slope even with full speed because of gravity. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. This means that the agent must learn to take initial actions that seem to prevent it from reaching the goal but are nevertheless mandatory for solving the problem.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVATIONS

ACRONYM	ABBREVATIONS
RL	Reinforcement Learning
QL	Q-Learning
SARSA	State Action Reward State Action
DQL	Deep Q-Learning
AI	Artificial Intelligence

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Reinforcement Learning

RL is an area of machine learning training which involves rewarding desired and progressive actions and punishing undesired ones. An RL agent is able to perceive and interpret its environment, take actions and learn through trial and error. Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. The model learns to perform the task to maximize the reward. The model starts by making some random actions and learns to take the more optimal actions based on the reward obtained after every action. The optimal action is the path which has the highest reward value. By leveraging the power of search and many trials, RL is currently the most effective way to train self-learning agents using AI. AI can gather experience from thousands of parallel gameplays if a RL algorithm is run on a sufficiently powerful computer infrastructure. This is time efficient when compared to doing the same task manually. There are two main types of RL algorithms which are model free and model-based RL algorithms. The algorithms (QL, SARSA, DQL) used in this project are all model free algorithms. A model free algorithm learns about the environment from the actions it takes as opposed to a model-based algorithm which constructs a model which is able to think ahead and see what would happen for a range of possible choices.

1.2 Elements of Reinforcement Learning

Any reinforcement learning problem includes the following elements:

1. **Agent** – the program controlling the object of concern.

2. **Environment** –this defines the outside world programmatically. Everything the agent(s) interacts with is part of the environment. It's built for the agent to make it seem like a real-world case. It's needed to prove the performance of an agent, meaning if it will do well once implemented in a real-world application.
3. **Rewards** – this gives us a score of how the algorithm performs with respect to the environment. It's represented as 1 or 0. '1' means that the policy network made the right move, '0' means wrong move. In other words, rewards represent gains and losses.
4. **Policy** – the algorithm used by the agent to decide its actions. This is the part that can be model-based or model-free.
5. **Epsilon** – Initially the value of epsilon is 1 which means that while training an agent it is more likely to explore more states than exploit the rewards. As the agents gains more experience the value of epsilon is gradually decreased which increases the probability for the agent to exploit the rewards instead of exploring states.

1.3 Problem Statement

The mountain car problem is a RL problem which has been solved in the past by using traditional QL. QL is a model free RL algorithm which is used to learn the value of an action in a particular state. Although traditional QL could be used to solve the mountain car problem it is a slow process and require a large amount of memory for the RL agent to reach the desired outcome. Also, it was only effective on smaller and less complex environments. To overcome the problems of traditional QL a new on Policy approach called SARSA was introduced. SARSA used the action performed by the current policy to learn the Q-Value. RL algorithms have made significant progress in recent years due to the power of deep neural networks which has led to DQL. DQL uses a deep neural network to approximate Q-values. SARSA and DQL are more efficient than traditional QL when it comes to solving the mountain car problem.

1.4 Objectives

RL is developing rapidly as algorithms more efficient than its predecessors are being found. Traditional implementations for RL involved QL. However, deep learning and SARSA have quickly become preferred approaches to solve RL problems. This is mainly because deep RL or DQL does not require as much storage as traditional QL models for complex environments and SARSA takes a safer path when compared to QL.

The aims of this project are as follows:

- To create a mountain car environment
- To implement the mountain car problem using traditional QL
- To implement the mountain car problem using SARSA
- To implement the mountain car problem using DQL
- To compare the performances of traditional QL and SARSA.

1.5. Scope of the Project

RL is training existing ML models so that they can produce a sequence of decisions. With every industry looking to apply AI in their domain, studying RL opens world of opportunities to develop cutting edge applications in various fields. We as students of computer science want to explore this trending technology. As a beginner our knowledge on this field was limited. We started with an online course. By digging more and more into RL, we got to know about the different algorithms used to implement RL agents. Our interests in RL grew gradually and lead us to take up a project on that. The mountain car problem has always been a relatively tough task for QL models. Recent use of DQL and SARSA approaches has demonstrated substantial improvement in results and reliability. The mountain car problem is used to test various RL algorithms. RL is used in various fields such as Marketing, Broadcast,

Journalism, Healthcare, Robotics, Gaming, Image Processing and Manufacturing.

1.6 Project Overview

The project is essentially a solution to the mountain car problem. Three major algorithms which are QL, DQL and SARSA are used to solve this problem. All these algorithms are model free algorithms which means that they learn directly from the actions taken by the agent as opposed to a model-based algorithm which creates a model that predicts future actions. To visualise the results the code is executed in Jupyter notebook.

CHAPTER 2

LITERATURE SURVEY

2.1. Introduction

To carry out this project it was crucial to obtain a thorough grasp on the basics of RL. We learned this through online courses such as the “Reinforcement Learning Specialization” on Coursera and viewing the TensorFlow documentation. It was also very important to understand how neural networks worked. For this purpose, we took up courses on Deep learning to help understand more about neural networks. This understanding was very beneficial to our study.

We discovered relevant work in the subject using Google Scholar and IEEE explore, that gave us insight in design patterns used and showed that there are many different approaches and algorithms which can be used to solve the mountain car problem. To acquire a full picture of RL, we looked at forums and blog postings where developers discuss their experiences with the technology. This helped us gain a practical understanding of RL and the algorithms present to solve RL problems.

Most researchers use traditional QL to solve the mountain car problem. Even though this problem can be solved in this way it is very slow and the requires a considerably large number of episodes to reach the goal state. While DQL is more accurate it is not required to solve a simple environment such as the mountain car environment, it also takes a long time to train a neural network which updates the q-table. We have identified SARSA as the most efficient algorithm to solve this problem.

2.2. Literature studies on RL

[1] Andreea-Iulia Patachi, Florin Leon, "A Comparative Performance Study of Reinforcement Learning Algorithms for a Continuous Space Problem", 2019

Three solutions to the Mountain Car problem were presented: using standard QL with ϵ-greedy exploration, Deep QL to optimize the table representation of the Q function with the aid of a neural network, and Deep Q-Network with experience replay.

[2] Mohd Azmin Samsuden, Norizan Mat Diah, Nurazzah Abdul Rahman, "A Review Paper on Implementing Reinforcement Learning Technique in Optimizing Games Performance", 2019

This paper made it evident that Implementing AI in games can help players in making better decisions and improve their performance. Implementing an AI agent will also help make the game more competitive.

[3] Ahmad Hammoudeh, "A Concise Introduction to Reinforcement Learning", 2018

RL is a huge topic, with a long history, a wide range of applications, an elegant theoretical core, distinguished successes, novel algorithms, and many open problems.

[4] Dongbin Zhao, Haitao Wang, Kun Shao, Yuanheng Zhu, "Deep Reinforcement Learning with experience replay based on SARSA", 2016

SARSA learning has some advantages when being applied to decision making

problems. It makes learning process more stable and is more suitable to some complicated systems. Given these facts, a new DRL algorithm based on SARSA, called deep SARSA learning, is proposed to solve the control problems of video games.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, Anil Anthony Bharath, "A Brief Survey of Deep Reinforcement Learning", 2017

Despite the successes of DRL, many problems need to be addressed before these techniques can be applied to a wide range of complex real-world problems.

[6] Christopher J Gatti , "The Mountain Car Problem", 2015

In general, we find that the ranges and locations of parameters that allow for convergence are consistent with intuition and are explainable. However, once individual subregions are explored on a more detailed level, we find that some subregions can be quite simple with a single parameter that dominates performance, whereas others can be quite complex with multiple parameters that have an effect on performance. Additionally, in these complex subregions, the reasons why each of these parameters have the effects that are seen is not intuitively obvious and will likely require further experimentation.

[7] B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications", 2019

Q-learning algorithms are off policy reinforcement learning algorithms that try to perform the most profitable action given the current state. However, these powerful set of algorithms are not fully exploited at their full potential.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, "Playing Atari with Deep Reinforcement Learning", 2013

This paper introduced a new deep learning model for reinforcement learning, and demonstrated its ability to master difficult control policies for Atari 2600 computer games, using only raw pixels as input. The authors also presented a variant of online Q-learning that combines stochastic minibatch updates with experience replay memory to ease the training of deep networks for RL. Their approach gave state-of-the-art results in six of the seven games it was tested on, with no adjustment of the architecture or hyperparameters.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning", 2015

A new neural network architecture was proposed that decouples value and advantage in deep Q-networks, while sharing a common feature learning module. The new dueling architecture, in combination with some algorithmic improvements, leads to dramatic improvements over existing approaches for deep RL in the challenging Atari domain.

[10] Hado van Hasselt, Arthur Guez, David Silver, "Deep Reinforcement Learning with Double Q-learning", 2015

This paper has five contributions. First, the authors have shown why Q-learning can be overoptimistic in large-scale problems, even if these are deterministic, due to the inherent estimation errors of learning. Second, by analysing the value estimates on Atari games the authors have shown that these overestimations are more common and severe in practice than previously acknowledged. Third, the

authors have shown that Double Q-learning can be used at scale to successfully reduce this overoptimism, resulting in more stable and reliable learning. Fourth, the authors have proposed a specific implementation called Double DQN, that uses the existing architecture and deep neural network of the DQN algorithm without requiring additional networks or parameters. Finally, the authors have shown that Double DQN finds better policies, obtaining new state of the art results on the Atari 2600 domain.

2.3. Conclusion

Based on the studies carried out, we have found that QL and DQL are suboptimal algorithms to solve the mountain car problem. Thus, we decided to implement the mountain car problem using SARSA and compare it with the traditional Q-Learning algorithm.

CHAPTER 3

SYSTEM ANALYSIS

3.1. Existing System

The mountain car problem has been implemented in the past by using QL and deep QL. These algorithms can often be slow in learning. This is mainly because it may require a considerable number of transitions to obtain a satisfying policy. QL has a high per-sample variance which can cause problems in model converging. Traditional QL approaches also ignore possible penalties for the exploratory moves of the agent which makes it expensive to train and implement.

3.1.1. Advantages of Existing System

- Easy to implement and use.

3.1.2. Disadvantages of Existing System

- QL requires a considerably large amount of memory space to reach the goal for complex environments.
- Training the agent is time consuming.

3.2. Proposed System

In our solution we use the SARSA algorithm to solve the mountain car problem. A SARSA agent interacts with the environment and updates the policy based on actions taken. The Q value for a state-action is updated by an error and adjusted by the learning rate. SARSA is more conservative than QL as it punishes negative actions. An example is walking near a cliff. QL will take the shortest path because it is optimal (with the risk of falling), while SARSA will take the longer, safer route (to avoid unexpected falling).

3.2.1. Advantages of Proposed System

- The policy gives more accurate and efficient results
- SARSA is more conservative than QL

3.3. Requirements

3.3.1. Functional Requirements

The functions or features that must be included in any system to satisfy business needs and be acceptable to users are known as functional requirements.
Based on this, the functional requirements that the system must require are as follows:

• The system should be able to run the PyGame application.

• The system should be able find the optimal set of actions required to reach the goal state. (top of the hill)

3.3.2. Non-Functional Requirements

A non-functional requirement is a description of the system's features, characteristics, and attributes, as well as any limitations that may limit the proposed system's bounds.
Performance, information, economy, control, and security efficiency and services are the main non-functional criteria.

Based on these the non-functional requirements are as follows:

• The system should provide better accuracy to perform efficiently in a short amount of time.

3.3.3. Hardware Requirements

- i5/i7 CPU, 8GB RAM, Nvidia CUDA GPU

3.3.4. Software Requirements

Language: Python

IDE: Jupyter Notebook

Front end: Keras

Back end: Tensorflow

Libraries: Gym, Numpy, MatplotLib, Collections, Random

3.4 LANGUAGE SPECIFICATION

3.4.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. Since there is no compilation step, the edit-test-debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on.

3.4.2 Frameworks

TensorFlow

Google TensorFlow is a python library. This library is a great choice for building commercial grade deep learning applications. Tensor Flow is based on computational graphs where a node represents persistent data or math operation and edges represents persistent data or math operation and edges represents the

flow between nodes, which is a multidimensional array or tensor hence the name TensorFlow.

TensorFlow grew out of another library DistBelief V2 that was a part of Google Brain Project. The library aims to extend the probability of deep learning. The output form of an operation or set of operations is fed an input into the next operation. Even though TensorFlow was designed for neural networks, it works well for other nets where computation can be modelled as data flow graph.

TensorFlow also uses several features from Theano such as common and subexpression elimination, auto differentiation, shared and symbolic variables. Different types of deep nets can be build using Tensor Flow like convolutional nets, Autoencoders, RNTN, RNN, RBN and DBM/MLP.

Keras

Keras is a python library for developing and evaluating deep learning models. It has a minimalist design that allows us to build a net layer by layer, to train it, and run it. Keras wraps the efficient numerical computation libraries. Theano and TensorFlow and allows us to define and train neural network models. It is a high-level neural network API, helping to make wide use of deep learning and Artificial intelligence. Its code is portable and used to implement neural networks. It is a deep learning API written in Python, running on top of machine learning platform. It was developed with focus of enabling fast experimentation.

3.4.3 Libraries

NumPy

NumPy is a library for python programming language that provides the supports for large, multi-dimensional arrays. Using NumPy, we can express images as multidimensional arrays.

Representing images as NumPy arrays is not only computational and resource efficient, but many other image processing and machine learning libraries use NumPy array representations as well. Furthermore, by using NumPy's build-in high-level mathematical functions, we can quickly perform numerical analysis on an image.

Gym

Gym is an open-source Python Library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API. Since its release gym's API has become the field standard for doing this. The gym API's API models environments as simple Python classes. Creating environment instances and interacting with them is very simple.

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible.

Collections

The Collection module in python provides a wide variety of containers. A container is an object that is used to store different objects and provide a way to access these objects and iterate over them. Some of the built-in containers are Tuples, Lists, Sets and Dictionaries.

Random

The Random library in python helps generate pseudo random numbers for different distributions. For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement. On the real line, there are functions to compute

uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

CHAPTER 4

SYSTEM DESIGN

The proposed method is to solve the mountain car problem by using SARSA. SARSA has proven to be the most effective way to solve this problem. The model chooses an action from the initial state by using a policy. Once the model chooses an action from a particular state it moves to another state. The reward is calculated and the next state which the model moves to is calculated by using the policy. This process is continued until the terminal or goal state is reached. The average, maximum and minimum rewards after every episode is recorded. This data is used in graphs which are plotted to compare the QL algorithm with the SARSA algorithm for the mountain car problem.

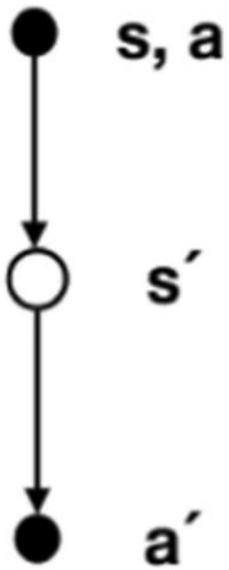

Fig. 4.1. SARSA

4.1. Design

a) The gym library is used to create the mountain car environment.
b) A Q-table is created and the learning rate, epsilon, discount, and bucket size is defined.
c) An action is selected by using the epsilon greedy policy which is used to determine the new state and new reward.
d) The SARSA algorithm is applied to determine the new Q-values and update the Q-table.

4.2 Elements of Reinforcement Learning

Any reinforcement learning problem includes the following elements:

1. **Agent** – the program controlling the object of concern.
2. **Environment** –this defines the outside world programmatically. Everything the agent(s) interacts with is part of the environment. It's built for the agent to make it seem like a real-world case. It's needed to prove the performance of an agent, meaning if it will do well once implemented in a real-world application.
3. **Rewards** – this gives us a score of how the algorithm performs with respect to the environment. It's represented as 1 or 0. '1' means that the policy network made the right move, '0' means wrong move. In other words, rewards represent gains and losses.
4. **Policy** – the algorithm used by the agent to decide its actions. This is the part that can be model-based or model-free.
5. **Epsilon** – Initially the value of epsilon is 1 which means that while training an agent it is more likely to explore more states than exploit the rewards. As the agents gains more experience the value of epsilon is gradually decreased which increases the probability for the agent to exploit the rewards instead of exploring states.

4.3 Q-Table

Q-Table is a lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state. As the agent performs exploration and learns more about the environment, the table is updated with new Q-values. These Q-values will help determine future actions.

	A0	A1	A2	A3
S0	**0.05**	**0.92**	**0.12**	**0.35**
S1	**0.07**	**0.09**	**0.35**	**0.25**
S2	**0.09**	**0.36**	**0.58**	**0.96**
S3	**0.99**	**0.11**	**0.87**	**0.47**
S4	**0.89**	**0.23**	**0.93**	**0.75**
S5	**0.29**	**0.98**	**0.63**	**0.74**
S6	**0.41**	**0.56**	**0.03**	**0.02**
S7	**0.06**	**0.51**	**0.15**	**0.19**
S8	**0.33**	**0.83**	**0.08**	**0.18**
S9	**0.99**	**0.22**	**0.11**	**0.71**
S10	**0.67**	**0.91**	**0.69**	**0.16**
S11	**0.23**	**0.39**	**0.63**	**0.26**
S12	**0.91**	**0.73**	**0.09**	**0.97**
S13	**0.98**	**0.87**	**0.90**	**0.08**
S14	**0.56**	**0.89**	**0.77**	**0.13**
S15	**0.01**	**0.04**	**0.45**	**0.11**
S16	**0.44**	**0.22**	**0.55**	**0.34**
S17	**0.77**	**0.22**	**0.33**	**0.67**
S18	**0.12**	**0.13**	**0.78**	**0.02**

Table 4.1. Q-Table

4.4. The Mountain Car Problem

The mountain car is a standard testing domain in RL, it is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. The state is represented by using two variables which are velocity and position, and the car has three possible actions which are to move left, stay at rest or move right.

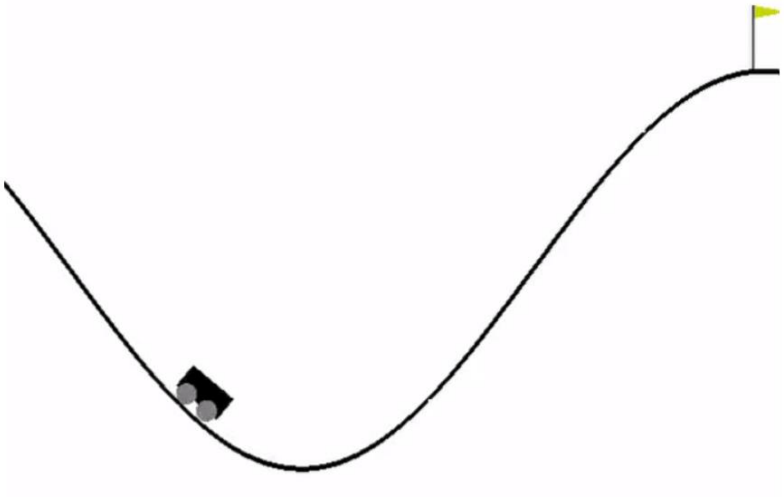

Fig. 4.2. Mountain Car Problem

4.5 Epsilon greedy policy

Epsilon greedy is a simple method which helps determine whether the agent should explore new unvisited states or use the knowledge it has of the states it has already explored. This trade-off between exploration and exploitation is managed by the variable epsilon which decides when the agent should explore and when the agent should exploit. Initially the agent has to explore more because it has no knowledge of other states and thus the value of epsilon is bigger. The value of epsilon is reduced as the agent explores and gains knowledge of the environment.

4.6 Experience Replay

Experience Replay is the act of storing and replaying game states (the state, action, reward, next state) that the RL algorithm is able to learn from. Experience Replay can be used in Off-Policy algorithms to learn in an offline fashion. Off-policy methods are able to update the algorithm's parameters using saved and stored information from previously taken actions. Deep Q-Learning uses Experience Replay to learn in small batches in order to avoid skewing the dataset distribution of different states, actions, rewards, and next states that the neural network will see. Importantly, the agent doesn't need to train after each step.

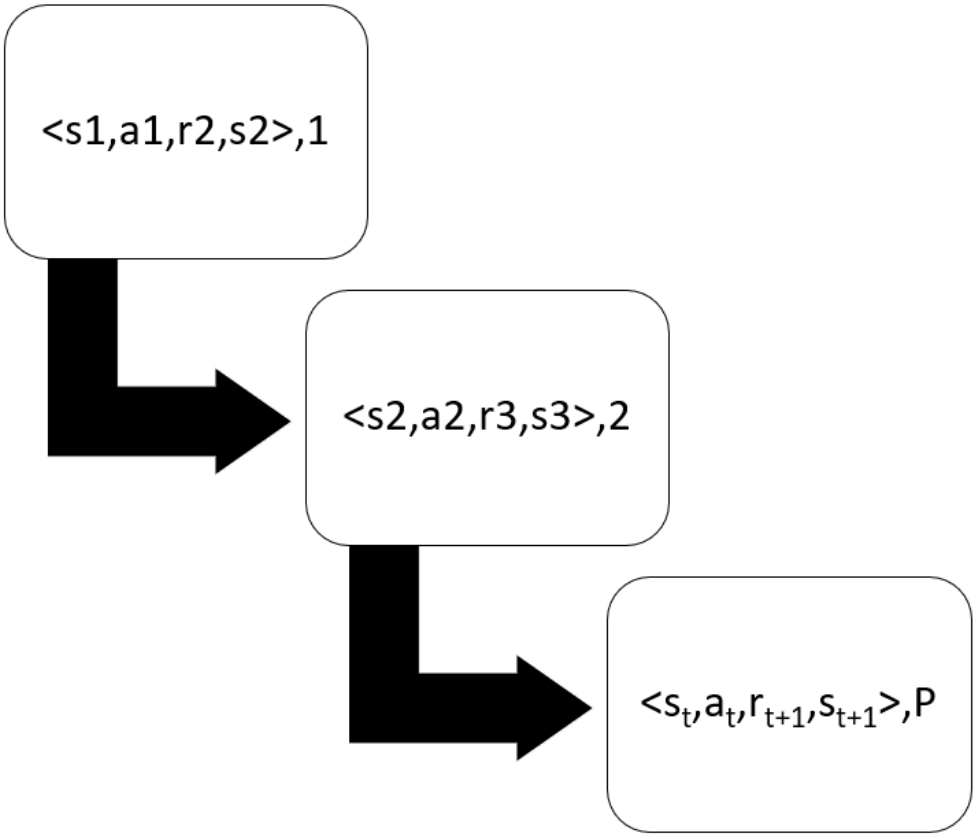

Fig. 4.3. Experience Relay

4.7 Q-Learning

Q-values are set state-action pairs and the algorithm chooses an optimal action for the current state based on estimates of this value. The reward and next state for this action is observed which allows for the Q value to be updated. Over many episodes this algorithm can learn the best path to take for this problem as long as the strategy balances exploration and exploitation correctly. Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. To implement Q-Learning in our mountain car problem we must follow Initialize random Q values and create the Q-table. An action is selected by using the epsilon greedy policy. After the action is performed a reward is generated and the agent moves to a new state. The Q value is updated using a bellman equation. This process is repeated until a terminal state is reached.

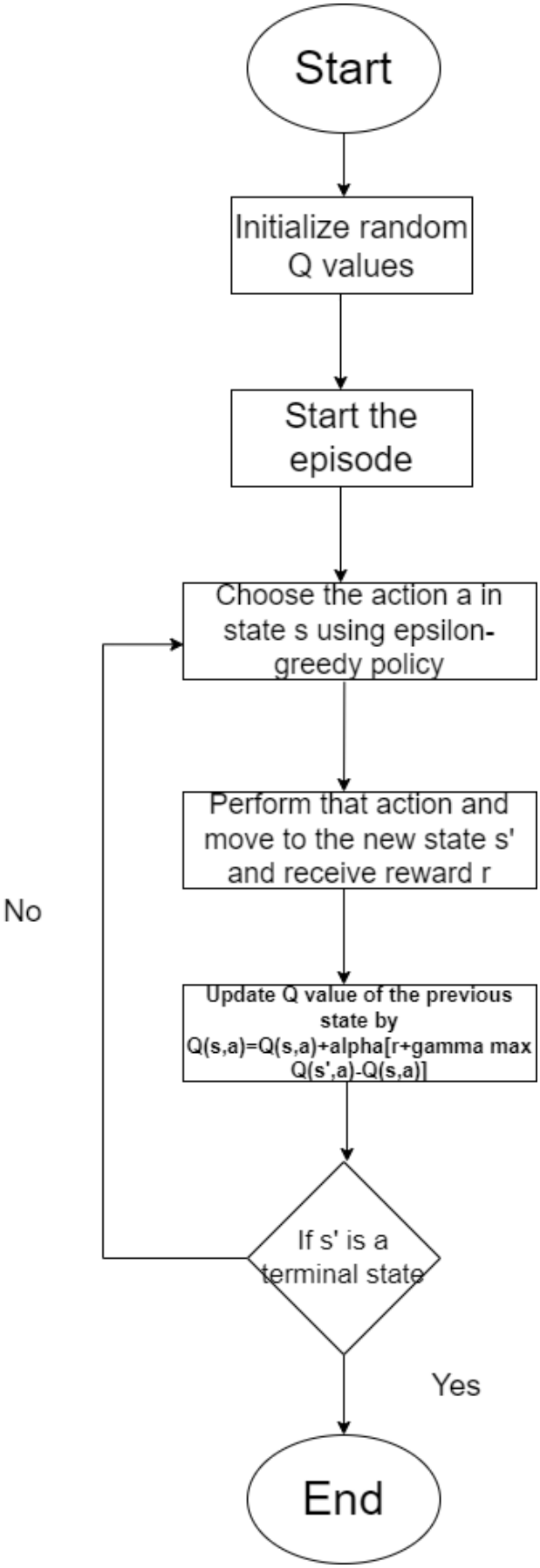

Fig. 4.4. Q-Learning Flowchart

4.8 SARSA

SARSA is an algorithm for learning a Markov decision process policy, used in the RL area of machine learning. This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent "S_1", the action the agent chooses "A_1", the reward "R" the agent gets for choosing this action, the state "S_2" that the agent enters after taking that action, and finally the next action "A_2" the agent chooses in its new state. To implement SARSA in our mountain car problem we must follow Initialize random Q values and create the Q-table. An action is selected by using the epsilon greedy policy. After the action is performed a reward is generated and the agent moves to a new state. The Q value is updated using a bellman equation. This process is repeated until a terminal state is reached.

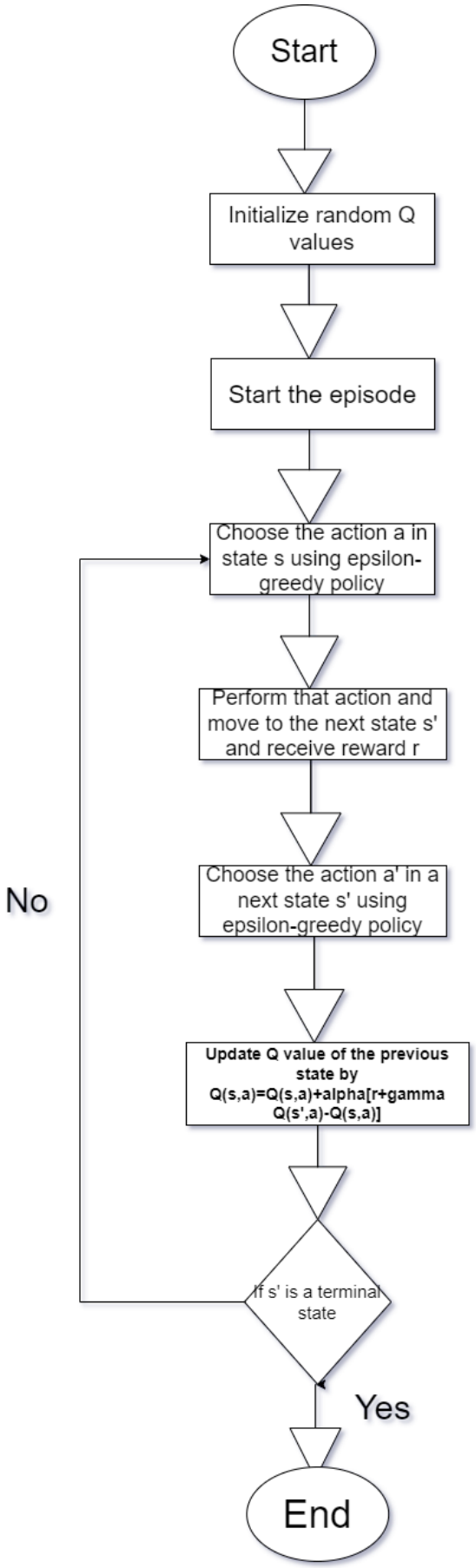

Fig. 4.5. SARSA Flowchart

4.9 Deep Q-Learning

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The user stores all past experiences in memory and the future action defined by the output of the Q-Network. It is used to solve problems which involve complicated environments.

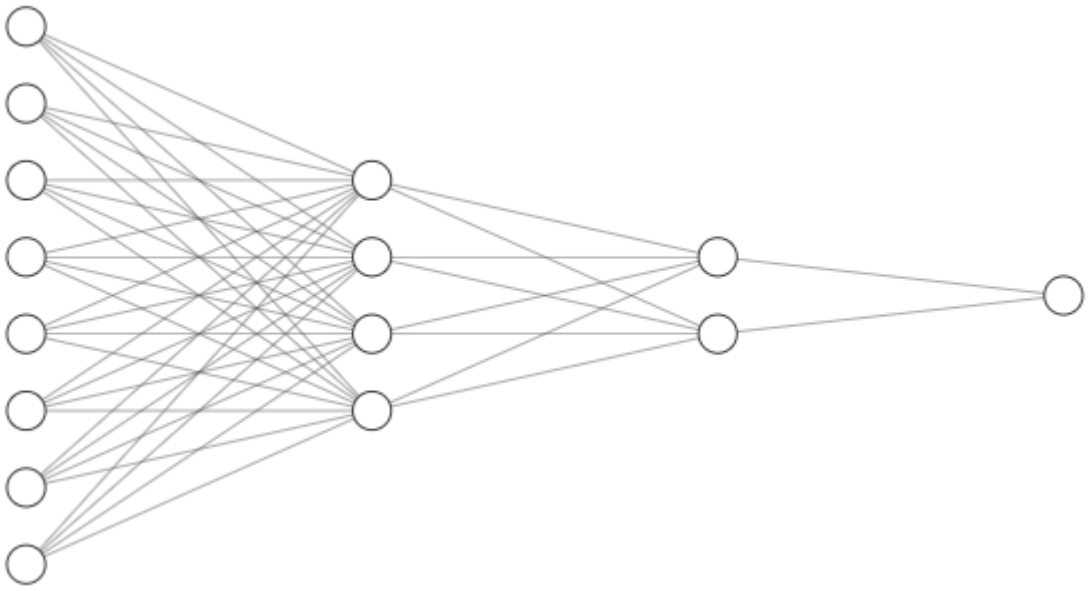

Fig. 4.6. Neural Network

CHAPTER 5

SYSTEM IMPLEMENTATION

5.1. Q-Learning Implementation

In the Q-Learning algorithm, the goal is to learn iteratively the optimal Q-value function using the Bellman Optimality Equation. To do so, we store all the Q-values in a table that we will update at each time step using the following formula:

$$q(s_t,a_t) = q(s_t,a_t) + \alpha(R_{t+1} + \gamma max\ q(s_{t+1},a) - q(s_t,a_t))$$

Where,

$q(s_t,a_t)$ = q value

α = learning rate

R_{t+1} = reward after performing action a_t from state s_t

γ = gamma

$max\ q(s_{t+1},a)$ = max q value given state s_{t+1} and action a

The steps involved are:

1) The Mountain car environment is created using the gym library in python. The environment provides valuable information such as the observation space and the action space for the mountain car problem. The observation space indicates the number of states in the problem and the action space is the number of possible actions possible in the environment. In the case of the mountain car problem the mountain car can take 3 actions which are moving to the left (-1), moving to the right (1) and staying at the same state (0).

2) The number of episodes to run the agent and the learning rate is set to a constant value. For every state in the environment a discrete state is created.
3) The value of epsilon is set. Epsilon determines whether the agent should exploit the rewards or explore the states. Initially, the value of epsilon is set as 1 which means that the agent will explore the different states in the environment. Gradually the value of epsilon is reduced which increases the probability of the agent to use the knowledge it has learnt and find the most optimal set of actions.
4) After every episode the q value in the q table is updated using the equation above and the reward gained after every episode is stored in a local variable.
5) The average reward after every episode is displayed and a graph between the episode number and reward is plotted.

5.2. SARSA Implementation

SARSA very much resembles Q-learning. The key difference between SARSA and Q-learning is that SARSA is an on-policy algorithm. It implies that SARSA learns the Q-value based on the action performed by the current policy instead of the greedy policy.

$$q(s_t,a_t) = q(s_t,a_t) + \alpha(R_{t+1} + \gamma q(s_{t+1},a_{t+1}) - q(s_t,a_t))$$

Where,

$q(s_t,a_t)$ = q value
α = learning rate
R_{t+1} = reward after performing action a_t from state s_t
γ = gamma
$max\ q(s_{t+1},a_{t+1})$ = max q value given state s_{t+1} and action a_{t+1}

The steps involved are:

1) The Mountain car environment is created using the gym library in python. The environment provides valuable information such as the observation space and the action space for the mountain car problem. The observation space indicates the number of states in the problem and the action space is the number of possible actions possible in the environment. In the case of the mountain car problem the mountain car can take 3 actions which are moving to the left (-1), moving to the right (1) and staying at the same state (0).
2) The number of episodes to run the agent and the learning rate is set to a constant value. For every state in the environment a discrete state is created.
3) The value of epsilon is set. Epsilon determines whether the agent should exploit the rewards or explore the states. Initially, the value of epsilon is set as 1 which means that the agent will explore the different states in the environment. Gradually the value of epsilon is reduced which increases the probability of the agent to use the knowledge it has learnt and find the most optimal set of actions.
4) After every episode the q value in the q table is updated using the equation above and the reward gained after every episode is stored in a local variable.
5) The average reward after every episode is displayed and a graph between the episode number and reward is plotted.

5.3. DQL Implementation

DQL leverages a Neural Network to estimate the Q-value function. The input for the network is the current, while the output is the corresponding Q-value for each of the action. The steps involved are,

1) DQL replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a q-value, a neural network maps input states to (action, Q-value) pairs. The number of episodes to run the agent and the

2) learning rate is set to a constant value. For every state in the environment a discrete state is created. One of the interesting things about Deep Q-Learning is that the learning process uses 2 neural networks. These networks have the same architecture but different weights. Every N steps, the weights from the main network are copied to the target network. Using both of these networks leads to more stability in the learning process and helps the algorithm to learn more effectively.

3) In the Epsilon-Greedy Exploration strategy, the agent chooses a random action with probability epsilon and exploits the best-known action with probability 1 - epsilon. In our implementation we use this strategy to help the agent perform exploration and exploitation tasks. Both the Main model and the Target model map input states to output actions. These output actions actually represent the model's predicted Q-value. In this case, the action that has the largest predicted Q-value is the best-known action at that state.

4) After choosing an action, it's time for the agent to perform the action and update the Main and Target networks according to the Bellman equation. Deep Q-Learning agents use Experience Replay to learn about their environment and update the Main and Target networks.

CHAPTER 6

SYSTEM TESTING

6.1 Testing the QL model

To test the QL model we run 2000 episodes with a learning rate of 0.1 and epsilon as 1. The model is able to reach the goal state around the 900th episode. A graph is plotted between the number of episodes on the X-axis and average, maximum and minimum episode rewards on the Y-axis. From the graph we can conclude that the model performs well in the case of maximum rewards and not so well in the case of average and minimum rewards. This makes the model inconsistent.

Fig 6.1. Q-Learning Graph

6.2. Testing the SARSA model

To test the SARSA model we run 2000 episodes with a learning rate of 0.1 and epsilon as 1. The model is able to reach the goal state around the 300th episode. A graph is plotted between the number of episodes on the X-axis and average, maximum and minimum episode rewards on the Y-axis. From the graph we can conclude that the model performs well in the case of maximum rewards and not so well in the case of average and minimum rewards. This makes the model inconsistent.

Fig. 6.2. SARSA Graph

6.3. Testing the DQL model

To test the DQL model we run 10000 episodes with a batch size of 32 and epsilon as 1. The neural network is trained initially with random weights. The weights are updated after every episode to give a more accurate value. After training this network we observed that this was a time-consuming process and wasn't time efficient to solve the simple mountain car problem. We saved the weights in a local file to make testing and using the model easier.

The model was tested by running 10 episodes of the entire mountain car scenario. The model was able to reach the goal state efficiently in all 10 scenarios.

```
Episode 0 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 1
Episode 1 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 1
Episode 2 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 1
Episode 3 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 1
Episode 4 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 1
Episode 5 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.99
Episode 6 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9801
Episode 7 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9702989999999999
Episode 8 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.96059601
Episode 9 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9509900498999999
Episode 10 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9414801494009999
Episode 11 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9320653479069899
Episode 12 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.92274469442792
Episode 13 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9135172474836407
Episode 14 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.9043820750088043
Episode 15 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.8953382542587163
Episode 16 Episodic Reward -200.0 Maximum Reward -200.0 EPSILON 0.8863848717161291
```

Fig. 6.3. training the neural network

CHAPTER 7

CONCLUSION AND FUTURE ENHANCEMENT

7.1 Conclusion

The aim of this thesis was to solve the mountain car problem by using different RL algorithms. Based on the literature studies, we figured out several drawbacks of the traditional QL model and tried to overcome those. We were able to overcome these drawbacks by implementing the mountain car problem using SARSA and DQL. SARSA proved to be more efficient than DQL for solving the mountain car problem as it took lesser time to train and still managed to arrive at the desired result.

7.2 Future Enhancement

Reinforcement Learning consists of many algorithms. In this project we have compared only 3 of these algorithms. An exhaustive comparison of all the reinforcement learning algorithms for the mountain car problem would give us a more thorough comparison.

APPENDIX-A

SAMPLE CODING

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import tensorflow as tf
import random
import tensorflow.compat.v1.keras.backend as K
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras import Model
from tensorflow.keras.optimizers import Adam

env = gym.make("MountainCar-v0")
DISCRETE_BUCKETS = 20
EPISODES = 2000
DISCOUNT = 0.95
SHOW_EVERY = 500
LEARNING_RATE = 0.1
epsilon = 0.5
EPSILON_DECREMENTER = epsilon/(EPISODES//4)
np.random.seed(1)

Q_TABLE =
np.random.randn(DISCRETE_BUCKETS,DISCRETE_BUCKETS,env.action_
space.n)
ep_rewards = []
aggr_ep_rewards = {'ep': [], 'avg': [], 'max': [], 'min': []}
```

```
def discretised_state(state):
    DISCRETE_WIN_SIZE = (env.observation_space.high-
env.observation_space.low)/[DISCRETE_BUCKETS]*len(env.observation_spa
ce.high)
    discrete_state = (state-env.observation_space.low)//DISCRETE_WIN_SIZE
    return tuple(discrete_state.astype(np.int))

for episode in range(EPISODES):
    episode_reward = 0
    done = False

    if episode % SHOW_EVERY == 0:
        render_state = True
    else:
render_state = False
curr_discrete_state = discretised_state(env.reset())
    if np.random.random() > epsilon:
        action = np.argmax(Q_TABLE[curr_discrete_state])
    else:
        action = np.random.randint(0, env.action_space.n)

    while not done:
        new_state, reward, done, _ = env.step(action)
        new_discrete_state = discretised_state(new_state)

        if np.random.random() > epsilon:
            new_action = np.argmax(Q_TABLE[new_discrete_state])
        else:
            new_action = np.random.randint(0, env.action_space.n)
if render_state:
```

```
env.render()

        if not done:
            current_q = Q_TABLE[curr_discrete_state+(action,)]
            max_future_q = Q_TABLE[new_discrete_state+(new_action,)]
            new_q = current_q +
LEARNING_RATE*(reward+DISCOUNT*max_future_q-current_q)
            Q_TABLE[curr_discrete_state+(action,)]=new_q
        elif new_state[0] >= env.goal_position:
            print(f"Goal state reached at episode:{episode}")
            Q_TABLE[curr_discrete_state + (action,)] = 0

        curr_discrete_state = new_discrete_state
        action = new_action

        episode_reward += reward

    epsilon = epsilon - EPSILON_DECREMENTER

    ep_rewards.append(episode_reward)

    if not episode % SHOW_EVERY:
        avg_reward = sum(ep_rewards[-SHOW_EVERY:])/len(ep_rewards[-
SHOW_EVERY:])

    ep_rewards.append(episode_reward)
    if not episode % SHOW_EVERY:
        average_reward = sum(ep_rewards[-SHOW_EVERY:])/len(ep_rewards[-
SHOW_EVERY:])
        aggr_ep_rewards['ep'].append(episode)
```

```
        aggr_ep_rewards['avg'].append(average_reward)
        aggr_ep_rewards['max'].append(max(ep_rewards[-SHOW_EVERY:]))
        aggr_ep_rewards['min'].append(min(ep_rewards[-SHOW_EVERY:]))
        print(f'Episode: {episode:>5d}, average reward: {average_reward:>4.1f},
current epsilon: {epsilon:>1.2f}')
env.close()

plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['avg'], label="average
rewards")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['max'], label="max rewards")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['min'], label="min rewards")
plt.legend(loc=2)
plt.show()

from IPython.core.display import HTML
HTML("<script>Jupyter.notebook.kernel.restart()</script>")

env = gym.make("MountainCar-v0")
np.random.seed(1)

LEARNING_RATE = 0.1
DISCOUNT = 0.95
EPISODES = 2000
SHOW_EVERY = 500
DISCRETE_OS_SIZE = [20, 20]
discrete_os_win_size = (env.observation_space.high -
env.observation_space.low)/DISCRETE_OS_SIZE
epsilon = 1
START_EPSILON_DECAYING = 1
END_EPSILON_DECAYING = EPISODES//2
```

```
epsilon_decay_value = epsilon/(END_EPSILON_DECAYING -
START_EPSILON_DECAYING)
q_table = np.random.uniform(low=-2, high=0, size=(DISCRETE_OS_SIZE +
[env.action_space.n]))
ep_rewards = []
aggr_ep_rewards = {'ep': [], 'avg': [], 'max': [], 'min': []}
def get_discrete_state(state):
    discrete_state = (state - env.observation_space.low)/discrete_os_win_size
    return tuple(discrete_state.astype(np.int))
for episode in range(EPISODES):
    discrete_state = get_discrete_state(env.reset())
    done = False
    episode_reward = 0

if episode % SHOW_EVERY == 0:
        render = True
    else:
        render = False

    while not done:
        if np.random.random() > epsilon:
            action = np.argmax(q_table[discrete_state])
        else:
            action = np.random.randint(0, env.action_space.n)

        new_state, reward, done, _ = env.step(action)
        episode_reward += reward
        new_discrete_state = get_discrete_state(new_state)

        if episode % SHOW_EVERY == 0:
```

```
env.render()
        if not done:
            max_future_q = np.max(q_table[new_discrete_state])
            current_q = q_table[discrete_state + (action,)]
            new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE *
(reward + DISCOUNT * max_future_q)
            q_table[discrete_state + (action,)] = new_q

        elif new_state[0] >= env.goal_position:
            print(f"Goal state reached at episode:{episode}")
            q_table[discrete_state + (action,)] = 0

        discrete_state = new_discrete_state

    if END_EPSILON_DECAYING >= episode >=
START_EPSILON_DECAYING:
        epsilon -= epsilon_decay_value

    ep_rewards.append(episode_reward)
    if not episode % SHOW_EVERY:
        average_reward = sum(ep_rewards[-SHOW_EVERY:])/len(ep_rewards[-
SHOW_EVERY:])
        aggr_ep_rewards['ep'].append(episode)
        aggr_ep_rewards['avg'].append(average_reward)
        aggr_ep_rewards['max'].append(max(ep_rewards[-SHOW_EVERY:]))
        aggr_ep_rewards['min'].append(min(ep_rewards[-SHOW_EVERY:]))
        print(f'Episode: {episode:>5d}, average reward: {average_reward:>4.1f},
current epsilon: {epsilon:>1.2f}')

env.close()
```

```
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['avg'], label="average
rewards")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['max'], label="max rewards")
plt.plot(aggr_ep_rewards['ep'], aggr_ep_rewards['min'], label="min rewards")
plt.legend(loc=2)
plt.show()

from IPython.core.display import HTML
HTML("<script>Jupyter.notebook.kernel.restart()</script>")

class DQNAgent:
    def __init__(self, sess, action_dim, observation_dim):
        K.set_session(sess)
        self.sess = sess
        self.action_dim = action_dim
        self.observation_dim = observation_dim
        self.model = self.create_model()

    def create_model(self):
        state_input = Input(shape=(self.observation_dim))
        state_h1 = Dense(400, activation='relu')(state_input)
        state_h2 = Dense(300, activation='relu')(state_h1)
        output = Dense(self.action_dim, activation='linear')(state_h2)
        model = Model(inputs=state_input, outputs=output)
        model.compile(loss='mse', optimizer=Adam(0.005))
        return model

EPISODES = 10_000
REPLAY_MEMORY_SIZE = 1_00_000
```

```
MINIMUM_REPLAY_MEMORY = 1_000
MINIBATCH_SIZE = 32
EPSILON = 1
EPSILON_DECAY = 0.99
MINIMUM_EPSILON = 0.001
DISCOUNT = 0.99
VISUALIZATION = False

env = gym.make('MountainCar-v0')
action_dim = env.action_space.n
observation_dim = env.observation_space.shape

sess = tf.compat.v1.Session()
replay_memory = deque(maxlen=REPLAY_MEMORY_SIZE)
agent = DQNAgent(sess, action_dim, observation_dim)

def train_dqn_agent():
    minibatch = random.sample(replay_memory, MINIBATCH_SIZE)
    X_cur_states = []
 X_next_states = []
    for index, sample in enumerate(minibatch):
        cur_state, action, reward, next_state, done = sample
        X_cur_states.append(cur_state)
        X_next_states.append(next_state)

    X_cur_states = np.array(X_cur_states)
    X_next_states = np.array(X_next_states)

    cur_action_values = agent.model.predict(X_cur_states)
    next_action_values = agent.model.predict(X_next_states)
```

```
for index, sample in enumerate(minibatch):
        cur_state, action, reward, next_state, done = sample
        if not done:
            cur_action_values[index][action] = reward + DISCOUNT *
np.amax(next_action_values[index])
        else:
            cur_action_values[index][action] = reward
    agent.model.fit(X_cur_states, cur_action_values, verbose=0)

max_reward = -999999
for episode in range(EPISODES):
    cur_state = env.reset()
    done = False
    episode_reward = 0
    episode_length = 0
    while not done:
        episode_length += 1
        if VISUALIZATION:
            env.render()

        if(np.random.uniform(0, 1) < EPSILON):
            action = np.random.randint(0, action_dim)
        else:
            action = np.argmax(agent.model.predict(np.expand_dims(cur_state,
axis=0))[0])

        next_state, reward, done, _ = env.step(action)

        episode_reward += reward
```

```
        if done and episode_length < 200:

            reward = 250 + episode_reward
            if(episode_reward > max_reward):
                agent.model.save_weights(str(episode_reward)+"_agent_.h5")
        else:
            reward = 5*abs(next_state[0] - cur_state[0]) + 3*abs(cur_state[1])
replay_memory.append((cur_state, action, reward, next_state, done))
        cur_state = next_state

        if(len(replay_memory) < MINIMUM_REPLAY_MEMORY):
            continue

        train_dqn_agent()



    if(EPSILON > MINIMUM_EPSILON and len(replay_memory) >
MINIMUM_REPLAY_MEMORY):
        EPSILON *= EPSILON_DECAY

    max_reward = max(episode_reward, max_reward)
    print('Episode', episode, 'Episodic Reward', episode_reward, 'Maximum
Reward', max_reward, 'EPSILON', EPSILON)

model_weight_file = "./-134.0_agent_.h5"

sess = tf.compat.v1.Session()
K.set_session(sess)
env = gym.make('MountainCar-v0')
action_dim = env.action_space.n
```

```
observation_dim = env.observation_space.shape
agent = DQNAgent(sess, action_dim, observation_dim)
agent.model.load_weights(model_weight_file)
episodes_won = 0
TOTAL_EPISODES = 10

for _ in range(TOTAL_EPISODES):
    cur_state = env.reset()
    done = False
    episode_len = 0
    while not done:
        env.render()
        episode_len += 1
        next_state, reward, done, _ =
env.step(np.argmax(agent.model.predict(np.expand_dims(cur_state, axis=0))))
        if done and episode_len < 200:
            episodes_won += 1
        cur_state = next_state

print(episodes_won, 'EPISODES WON AMONG', TOTAL_EPISODES,
'EPISODES')

from IPython.core.display import HTML
HTML("<script>Jupyter.notebook.kernel.restart()</script>")
```

APPENDIX-B

SCREENSHOT

```
Episode:     0, average reward: -200.0, current epsilon: 0.50
Goal state reached at episode:271
Goal state reached at episode:275
Goal state reached at episode:311
Goal state reached at episode:312
Goal state reached at episode:385
Goal state reached at episode:386
Goal state reached at episode:387
Goal state reached at episode:389
Goal state reached at episode:390
Goal state reached at episode:391
Goal state reached at episode:396
Goal state reached at episode:398
Goal state reached at episode:399
```

Fig B.1 SARSA output

```
Episode:     0, average reward: -200.0, current epsilon: 1.00
Episode:   500, average reward: -200.0, current epsilon: 0.50
Goal state reached at episode:908
Goal state reached at episode:918
Goal state reached at episode:919
Goal state reached at episode:920
Goal state reached at episode:922
Goal state reached at episode:923
Goal state reached at episode:926
Goal state reached at episode:933
Goal state reached at episode:936
Episode:  1000, average reward: -199.5, current epsilon: -0.00
Goal state reached at episode:1014
Goal state reached at episode:1020
Goal state reached at episode:1025
```

Fig B.2 Q-Learning output

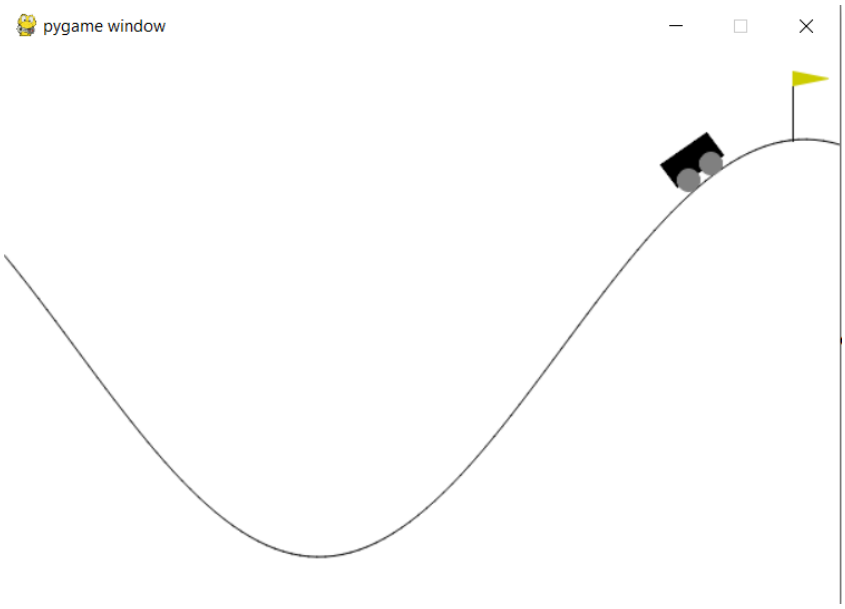

Fig B.3 Mountain car using Pygame

REFERENCES

1. Ahmad Hammoudeh, A Concise Introduction to Reinforcement Learning, 2018

2. Andreea-Iulia Patachi, Florin Leon, A Comparative Performance Study of Reinforcement Learning Algorithms for a Continuous Space Problem, 2019

3. Anil Anthony Bharath, Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, A Brief Survey of Deep Reinforcement Learning, , 2017

4. B. Jang, M. Kim, G. Harerimana and J. W. Kim, Q-Learning Algorithms: A Comprehensive Classification and Applications, 2019

5. Christopher J Gatti, The Mountain Car Problem, 2015

6. Dongbin Zhao, Haitao Wang, Kun Shao, Yuanheng Zhu, Deep Reinforcement Learning with experience replay based on SARSA, 2016

7. Hado van Hasselt, Arthur Guez, David Silver, Deep Reinforcement Learning with Double Q-learning, 2015

8. Mohd Azmin Samsuden, Norizan Mat Diah, Nurazzah Abdul Rahman, A Review Paper on Implementing Reinforcement Learning Technique in Optimizing Games Performance, 2019

9. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing Atari with Deep Reinforcement Learning, 2013

10. Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, Dueling Network Architectures for Deep Reinforcement Learning, 2015