Q-Learning & Game-Playing

Research and Development Project

Sudeshna Merugu
200010050
Mentored By Prof. Prabuchandran K J

Index

0.1 Introduction

0.1.1 About the project

The aim of this project is to learn and apply various Q-Learning algorithms in game playing. To observe and infer on how different each learning variant in Q-Learning performs on different games. The choice of a domain for such applications has been made since the environment is easy to observe, infer and draw conclusions from.

0.1.2 Components of the project

The following path has been followed to maintain the continuity in the learning stages:

- Introduction to AI (course by Berkeley)
- Introduction to Object Oriented Programming in Python
- Application of Q-Learning on PACMAN
- Application of Function Approximation on PACMAN
- Limitations and need for Advanced algorithms
- Deep Q-Learning
- Application of DQNs in games
- DQN for ATARI Games
- Introduction to Offline Reinforcement learning

0.2 Introductory Information

0.2.1 Reinforcement Learning

Reinforcement learning is based on the idea of getting rewarded for good actions and punished for bad ones. The learning agent is required to interact with the environment and hence learn by receiving rewards or punishments.

How is Reinforcement Learning different from standard Machine Learning?

- The key distinguishing factor is how the agent is trained
- Instead of inspecting the data provided, the model interacts with the environment, seeking ways to maximize reward
- In the case of deep reinforcement learning, a neural network is in charge of storing the experiences, improving the way the task is performed.

RL Agents learn from exploitation and exploration, a continuous trial-error based learning, where the agent tries to apply different combination of actions on a state to find the highest cumulative reward. The exploration when deployed in the real world (for higher order tasks like autonomous driving) becomes really expensive and nearly impossible. We know that such tasks are expensive since it is not feasible to take best actions where the dynamics of the environment change very frequently. For a brief idea of how learning methods have been modified to arrive at a solution to the above problem, other mechanisms have been applied (offline-reinforcement learning), like learning by mimicking the desired behaviour, learning through demonstrations are being tried on robots to learn the environment in simulations.

0.2.2 MDPs

An MDP is a mathematical framework that is used for modelling decision making problems where the outcomes are partly random and partly controllable.

Note that if an action has low probability doesn't mean that it won't be picked at all, Just that it is less likely to be picked (definitions of the technical terms can be referred to in the following section under prerequisites)

A Markov Decision Process (MDP) models a sequential decision-making problem. The most important characteristic of an MDP is that the state transition and reward function depend on only the current state and the applied action and not on previous observations.

An MDP is essentially comprised of a reward function **R** and a state transition function **P**. In case we have these functions, we can solve the MDP using Dynamic Programming.

Markov Process: given a state S_t being s, the probability of the next state S_{t+1} being s' should only be dependent on the current state S_t = s, and not on the rest of the past states $S_1 = s_1$, $S_2 = s_1$, ...

Policy Iteration

Policy Iteration has two aspects to it, that are, Policy evaluation and Policy improvement. In this, we evaluate the pre-assumed policy in an iterative fashion. Once this evaluation is done, we proceed to improve this policy. We again loop through this process of evaluating and improving the policy.

Value Iteration

In Value Iteration, we try to reduce the computation for solving the MDP by taking an advantage at every iteration. The idea behind the Value Iteration al-

gorithm is to merge a truncated policy evaluation step (as shown in the previous example) and a policy improvement step into the same algorithm.

0.2.3 Q-Learning

However, for most cases, we cannot precisely predict **R** and **P**. If those functions are not known, Q-Learning is an algorithm that can be use to solve MDPs with unknown reward and transition functions.

The main idea of Q-Learning is to "explore" all possibilities of state-action pairs and estimate the long-term reward that will be received by applying an action in a state (the **Q(s,a)** value).

0.3 PACMAN

0.3.1 Prerequisites

Framework

The PACMAN Framework provided by the University of California, Berkeley (UCB), Berkeley United States was used to simulate the learnt algorithms. `(FRAMEWORK LINK)`. Provides you with the necessity framework and user guide to get you started. Make sure you are running the framework using python's second version.

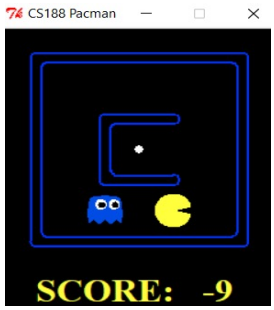
CS 188: Intro To AI

A prior knowledge of what Markov Decision Processes are. The open lectures from UCB are available on YouTube and proceeding by going through their course would give a better insight into what Q-learning is and how Q-learning works.
`(CS: 188)`

The following book provided me with a higher insight into algorithm implementations and the concepts:
`(Reference Material)`

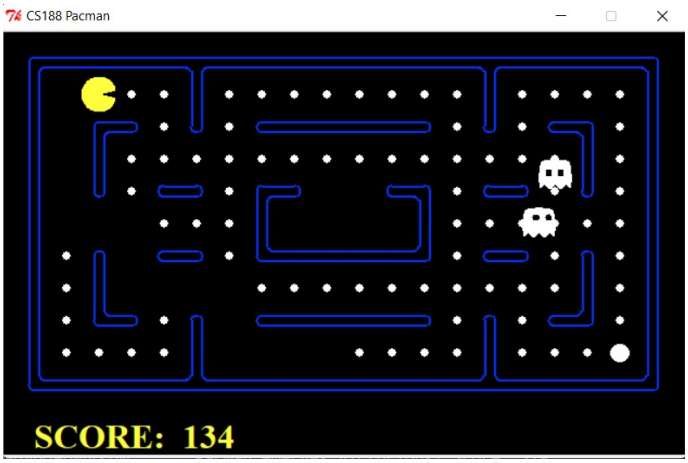
Object Oriented Programming

A considerable amount of efficiency is required to understand and implement ideas using the framework provided. The framework uses these extensively and prior knowledge in OOPs is expected.



(a) Layout 1

(b) Layout 2



(c) Layout 3

Figure 1: Layouts

Important Terminologies

Agent: An RL agent is the entity which we are training to make correct decisions (for eg: a Robot that is being trained to move around a house without crashing).

Environment: The environment is the surrounding with which the agent interacts (for eg: the house where the Robot moves). The agent cannot manipulate the environment; it can only control its own actions (for eg: the Robot cannot control where a table is kept in the house, but it can walk around it to avoid crashing).

State: The state defines the current situation of the agent (for eg: it can be the exact position of the Robot in the house, or the alignment of its two legs, or its current posture; it depends on how you address the problem).

Action: The choice that the agent makes at the current time step (for eg: it can move its right or left leg, or raise its arm, or lift an object, turn right or left, etc.). We know the set of actions (decisions) that the agent can perform in advance.

Policy: A policy is the thought process behind picking an action. In practice, it is a probability distribution assigned to the set of actions. Highly rewarding actions will have a high probability and vice versa.

0.3.2 Q-Learning for PACMAN

To begin with, this section shows the implementation of PACMAN using Q-Learning, the algorithm, the layout, the graphs of how the agent has learnt over experience, and also a brief understanding of why and how Q-learning works.

In scenarios where the agent isn't provided with the necessary reward function and the transition probability matrix, that is **R** and **P** respectively, we can't follow the regular dynamic programming methods. The agent is expected to learn from experience.

We know that if we have the optimal Q* values for all state-actions pairs, we can deduce an optimal policy from that. The objective of Q-Learning is to find the optimal policy, that is to arrive at Q*.

How is Value Iteration Different From Q-Learning?
In Q-Learning, the agent doesn't know the state transition probabilities or the rewards. The agent only discovers that there is a reward for going from one state to another via a given action when it does so and receives a reward. Similarly, it only figures out what transitions are available from a given state by ending up in that state and looking at its options. Whereas in value iteration you are given the state transition probabilities and the reward function.

$$q^*(s,a) = \max_{\pi} q_\pi(s,a)$$

What is a Q function exactly?
Given a state and an action, gives the expected return from taking that action in that state and following the given policy thereafter.

One of the most fundamental property of Q* is that it follows the Bellman Equation:

$$q^*(s,a) = E[R_{t+1} + \gamma \max_{a'} q_*(s',a')]$$

Exploration Vs Exploitation:
To this point it is clear that the agent has to learn from experience. How is this happening exactly? We have mentioned previously that we want to find the

optimal Q-values for all state-actions pairs. For this reason we build something called a Q-table that initializes all the corresponding Q-values to be 0, which will be updated over iterations. The table is of the size (no.of states)X(no.of actions). We will keep updating these slots based on our experience.

$$q_*(s,a) - q(s,a) = loss$$

$$E[R_{t+1} + \gamma \max_{a'} q_*(s',a')] - E[\sum_{k=0}^{\infty} {}_{t+k+1}] = loss$$

To gain experience, the agent has to take an action being in a given state and observing the reward function. How will the agent decide on what on actions to take? If the agent picks the action returning the maximum Q-value, then we say it is exploiting the information it has gained yet, but if it chooses an action randomly, that means it is exploring the environment instead using the information it already has. To have the right usage of both strategies for learning, we use the Epsilon-Greedy Strategy that makes the best use of both worlds.

NOTE: In Q-Learning our Q function is linear

Q-Learning Algorithm

Epsilon-Greedy Strategy is where we initially let the agent pick an action based on exploration. After it gains much information through exploration, we force it to be greedy to exploit this information towards the end of the training. This is done by taking an epsilon value and a random number generator, this random number generated, if is greater than the epsilon, the agent will exploit the environment and if less, then explore the environment. We generally start out with epsilon to be equal to 1 and gradually cut down the agents flexibility to explore, but push it to exploit the information.

We have stated that the Q-table is initialized with zero. Now with a strategy to get the action, the agent executes it and observes the reward gained taking that action. It computes the loss of prediction using the Q-table which has our prediction and Q* for that state-action pair using the Bellman Equation and the reward obtained.

Learning rate: We have stated that the Q-values will keep getting updated, but we don't expect to completely over-write the previous one. We do need the previous information along with the new information.And to account for this, we have a learning rate, alpha. If more is the learning rate the agent will adopt the new Q value for the state-action pair. That is if alpha is set to 1, then the values keep getting over written and the Q-values are equal to the recently computed Q-value and we lose information regarding the previous one.

$$Q^{new}(s,a) = (1-\alpha) \underbrace{Q(s,a)}_{\text{from Q table}} + \alpha \underbrace{(R_{t+1} + \gamma \max_{a'} q(s',a'))}_{\text{learned value}}$$

Q-learning on SmallGrid Layout

States:
The states in our PACMAN framework, is the grid itself, with the score included.

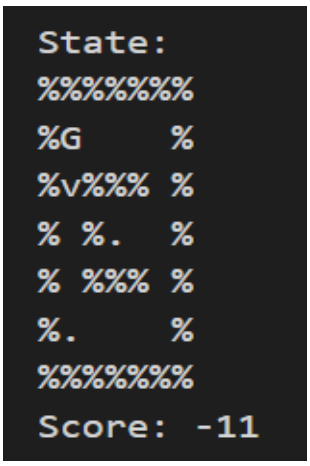
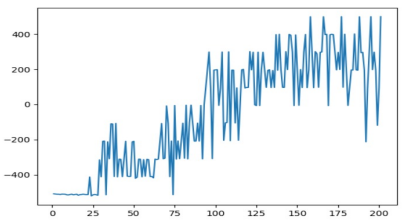


Figure 2: State encoding in the Framework

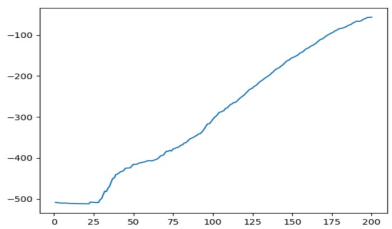
Actions: The agent has 5 actions to chose from: "West", "East", "North", "South", "Stop"

Graph of Rewards Calculated

The following are the two graphs plotted throughout the course of the learning phase.



(a) Average of previous 10 episodes



(b) Average of all the previous episodes

Figure 3: Graphs

Observations

- Almost every learning phase has shown that the agent visits nearly 5K states during training.
- For the small grid layout itself, the state space was very large and to run 2000 episodes solely for training was turning out to consume a lot of computational power and space.
- Running the same Q-learning algorithm on the larger layouts provided, have turned out to be really slow and hadn't converged even after 1 hour of training.
- As the layout size increases, the time taken to complete 100 episodes (the frequency at which the training had been tracked) was gradually decreasing.

0.3.3 Function Approximation for PACMAN

Previously we have seen that the number of states being visited was over 5K and maintaining a Q-table for such a large state space, with their corresponding actions, was seeming to look like a lot of computation for the simple and small grid itself. What if we wanted to train the agent to play on a larger layout?

To overcome the problem of having a large state space and the number of training sessions required to train our agent, we use the approximate Q-Learning method. If one thinks about what Q-Learning is doing, the agent blindly memorizes each state and updates their values but by doing this, in a way it fails to know the environment.

We can observe that when the state space becomes too large, the tabular methods become insufficient and unsuitable. In order to address this shortcoming, we can adopt a new approach based on the features of each state. The aim is to use these set of features to generalise the **estimation** of the value at states that have similar features.

We used the word estimation to indicate that this approach will never find the true value of a state, but an approximation of it. This will achieve faster computation and much more generalisations.

Here our Q-values are represented as the weighted linear sum of feature values. The agent in this algorithm learns by updating these weights given to each feature

Advantages include the states sharing a few features and even unseen and unvisited states can be generalized using these features. Also helps the agent learn to take similar decisions when in similar states and also less computation due

to the reduced Q table. A disadvantage might be that we need to define such features and require domain knowledge to do so, which is often done manually.

$$Q(s,a) = \sum_{i}^{n} f_i(s,a) w_i$$

$$w_i \leftarrow w_i + \alpha[correction] f_i(s,a)$$

$$correction = (R(s,a) + \gamma V(s')) - Q(s,a)$$

0.3.4 Variants of Feature Extractors

The framework provided has already defined a simple extractor which I have exploited to run Approximate Q-Learning.

Simple Extractor Version 1

The Q-Value is calculated by considering the state, taking the action, and where would Pacman end up (the position), and from there the feature values are calculated.
Pacman in position (x1,y1) in its given state, on taking an action will end up in (x2,y2).

The features are calculated using these coordinates, for example:
1. Distance of the closest power pellet is calculated from (x2,y2) to the power pellet.
2. Number of ghosts one step away from (x2,y2).
3. Distance of the closest food is also calculated from (x2,y2).
4. If the agent can eat the food or not

The following are the features the framework has pre-defined:

- "bias" which is always 1.0
- "#-of-ghosts-1-step-away" is the number of ghosts (regardless of whether they are safe or dangerous) that are 1 step away from PACMAN
- "closest-food" the distance in PACMAN steps to the closest food pellet (does take into account walls that may be in the way)
- "eats-food" either 1 or 0 if PACMAN will eat a pellet of food by taking the given action in the given state

All the other versions too do the same as above.

Version 2: If the ghosts were scared, then the existence of the ghosts have been neglected.

Version 3: If the ghosts were scared, then their position was modified to be equivalent to food.

Version 4: The power pellets weren't considered as food particles, hence modified to be food.

FINAL: Added a new feature to know if both ghosts would be scared, given a state and an action.

Test Game No.	Score
1	1315
2	1318
3	1318
4	1333
5	1338

Table 1: Version 1

Test Game No.	Score
1	1705
2	1733
3	1737
4	1732
5	1545

Table 2: Version 2

Test Game No.	Score
1	1545
2	1527
3	1925
4	1523
5	2131

Table 3: Version 3

Test Game No.	Score
1	1337
2	2130
3	1529
4	1730
5	2119

Table 4: Version 4

Test Game No.	Score
1	1711
2	1930
3	2117
4	1931
5	1526

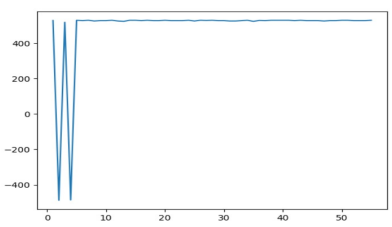
Table 5: Version 5(i)

Test Game No.	Score
1	1731
2	2129
3	2134
4	1307
5	2113

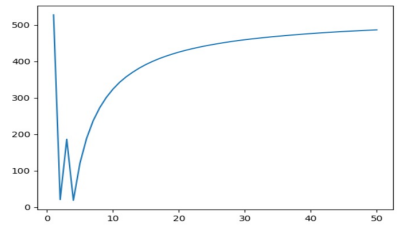
Table 6: Version 5(ii)

Rewards on MediumGrid

The following graphs show the rewards collected using the modified feature extractor on Medium Grid Layout.



(a) Average of previous 10 episodes

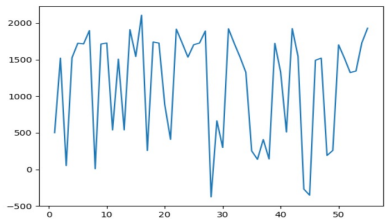


(b) Average of all the previous episodes

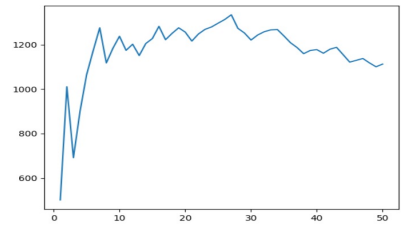
Figure 4: Graphs

Rewards on MediumClassic

The following graphs show the rewards collected using the modified feature extractor on Medium Classic Layout.



(a) Average of previous 10 episodes



(b) Average of all the previous episodes

Figure 5: Graphs

Version	**Avg. Score**
Simple Extractor	1324.4
Version 2	1690.4
Version 3	1730.2
Version 4	1769.0
Final Version	1882.8

Table 7: Extractor Variants

0.3.5 Off-Policy versus On-Policy Learning

The difference between Off-policy and On-policy methods is that in Off-Policy learning the agent does not to follow any specific policy, the agent could even behave randomly and despite this, off-policy methods can still find the optimal policy. On the other hand, on-policy methods are dependent on the policy used.

Example:
The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s' and the greedy action a'. In other words, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy.
The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state s' and the current policy's action a". It estimates the return for state-action pairs assuming the current policy continues to be followed.
The distinction disappears if the current policy is a greedy policy. However, such an agent would not be good since it never explores.

0.4 DQN

We have always known artificial neural networks to approximate functions. In our function approximator that we've discussed about in the previous module, the Q-function is defined linearly. To bring the sense of non-linearity into our function approximation, DQN introduces neural networks. This algorithm makes use of a deep neural network to estimate the Q values for each state action pair in a given environment and in turn the network will approximate the optimal Q function. The active combination of deep neural networks and Q-learning is called deep Q-learning. The network that approximates the Q-function is called a deep Q network or a DQN.

Suppose we have some arbitrary deep neural network that accepts states as input. For each state input, the network outputs the Q-values estimations for each action possible from that state. The objective of this function is to approximate the optimal Q-function. We have already seen that the optimal Q-function will satisfy the Bellman Equation. We make use of this equation to compute loss. The objective here is to minimize this loss. The main difference between DQN and Q-learning is that we have been using a Q-table to solve for the Q-function problem, whereas here we use a neural network.

0.4.1 DQN: with an example

Assume the following visual in a game:

Figure 6: Game Visual Raw

We crop this out and turn the RGB image into grayscale since this would not affect the learning process. This is not the whole state, we take a sequence of images, pre-process them as mentioned earlier and together these comprise of one state. Why do we need to take multiple images to comprise one state to be passed through the network? This is to capture the direction of movement, in this game. The agent will get a sense of the ball either going down or up or which direction, etc. Refer figure 19 for the same.

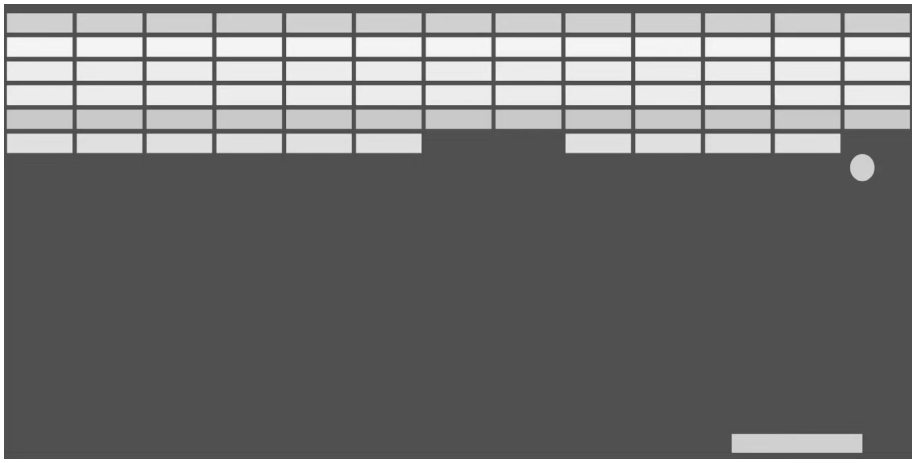


Figure 7: GrayScale Conversion

A stack of frames represent a single input and now we send this state through the neural network. **Experience Replay Replay Memory:**
With experience replay, we store the experiences at each time step in a replay memory. At time t, the experience e_t is defined as the following tuple:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

The replay memory has a pre-defined size, so we do not store all the experiences at every time step. It will only store the last N experiences, where N is pre-set by us.

This replay memory of experiences is where we'd be sampling from to train the network.

But why are we chosing random samples from replay memory to sample and train the network rather than not needing to store but just learn from sequential experiences? A key reason is to break the correlation between the consecutive samples. If the network only learned from consecutive samples, the samples would then be highly correlated and therefore lead to inefficient learning.

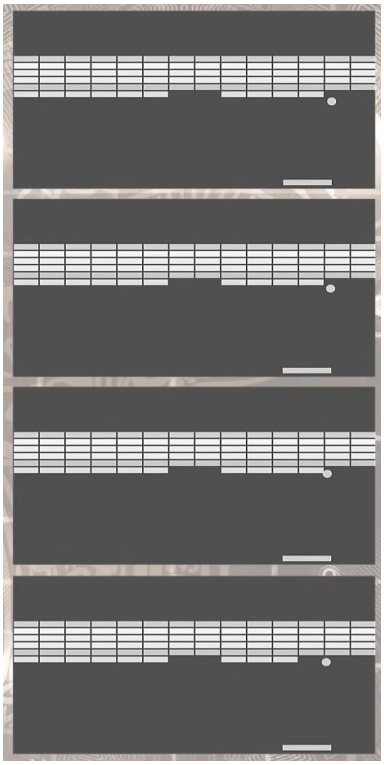


Figure 8: Form a state

0.4.2 Algorithm

```
1. Initialize replay memory capacity
2. Initialize the network with random weights
3. for each episode:
    1. Initialize the starting state
    2. For each time step:
        1. select action (via exploration or exploitation)
        2. Execute
        3. Observe reward and next state
        4. Store experience in replay memory
        5. Sample random batch
        6. Pre-process all states from batch
        7. Pass these through the policy network
        8. Calculate Loss (requires second pass for corresponding next state)
        9. Gradient descent
```

Explanation of the Algorithm:
For a time step in an episode of our training, we chose an action, execute it, observe it and put it in the replay memory. Suppose we have 6 experiences stored soo far. We randomly sample a batch from them, say a size of 3, 4, etc. We then preprocess all these states using the method discussed above (grayscale conversion and stacking consecutive frames). Now we pass these states through our network that we've initialized. For a better understanding, let us take one experience to begin with:

$$e_4 = (s_4, a_4, r_5, s_5)$$

Once we pass the state s_4 through the policy network, we get the corresponding Q-values for the actions available to it. We now take the action that has been taken in our e_4, and check the corresponding Q-value returned by the policy network for that action. This is what we say is the $Q(s_4, a_4)$. We still need to get the new Q-value to calculate the loss with respect to this equivalent of old Q-value in our Q-learning algorithm.

$$q_*(s, a) - q(s, a) = loss$$

$$E[R_{t+1} + \gamma \max_{a'} q_*(s', a')] - E[\sum_{k=0}^{\infty} {}_{t+k+1}] = loss$$

Note that to calculate this, we need the following:

$$\max_{a'} q_*(s', a')$$

To calculate this part of the equation, we again send in the state s' through the policy network and take the maximum from the values the policy network returns. (Equivalent to how we take the Q-values from the Q-table in Q-learning)

Once this is calculated, we can compute the loss and implement gradient descent to update the weights in the policy network. The explanation accounted for only one experience tuple, but we need to pass in all the experiences from the batch that we've sampled. All this will occur for one time step, the next time-steps we do this process again. After the end of an episode, we move onto another episode and the same continues.

0.4.3 Application on CARTPOLE problem

The cartpole game is where we try to balance the pole on a slab that is movable. The actions available are right and left, which can be controlled manually using the keys. Here in this problem we try to simulate a network and see how using DQN, the agent learns to play the game efficiently.

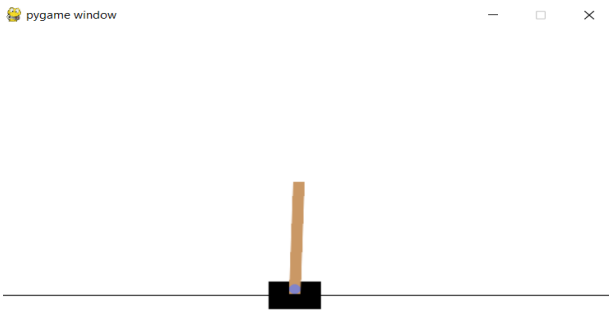


Figure 9: Display

Eps	Score
1	15.0
2	44.0
3	46.0
4	18.0
5	21.0
6	22.0
7	12.0
8	14.0
9	17.0
10	20.0

Table 8: Untrained

Eps	Score
1	200.0
2	103.0
3	200.0
4	200.0
5	200.0
6	200.0
7	199.0
8	200.0
9	200.0
10	179.0

Table 9: Trained

In this implementation of DQN on cartpole, the Gym library has been used for creating our game environment. From the stable_baselines3 library, we import DQN training algorithms to be exploited, where it simplifies our task by defining the size for the neural network by itself, that is the number layers, the size of each layer, and the type of layers used. It initializes trains and builds the model by itself.

This game has been modelled by using libraries for training.

0.4.4 Application on MOUNTAIN-CAR problem

Mountain Car, a standard testing domain in Reinforcement learning, 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. Here the actions available are accelerate forward, backward, or hold still.

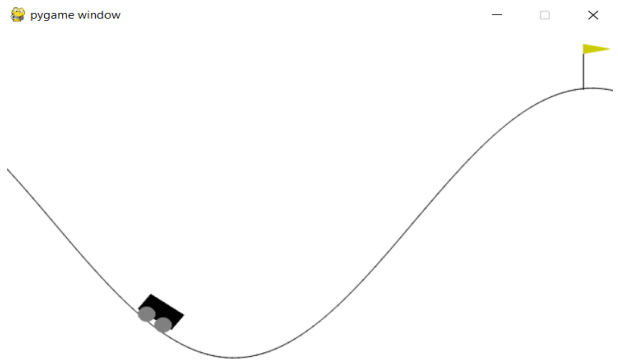


Figure 10: Display

Since the previous game has been implemented using the libraries for predefining the network and for training, this time, the Q-network, weights, update rules, etc. have been defined from scratch. We use the keras library to build our model.

- The first layer is to take in the states
- The last layer of the network is to get corresponding Q-values for actions
- The Q-network is made of 2 hidden layers
- Each layer is fully connected
- The first fully connected layer comprises of 20 neurons

Eps	Scores
2	54.32225837901821
5	72.97582806508282
6	62.79769246157908
7	55.53191686625779
8	81.75593144810617
9	73.46967355427854
10	62.91564051528104
72	96.29876274056622
78	107.41182562260856
83	117.84680966190157
91	122.02783778229286
116	166.14879082851954

Table 10

- The second fully connected layer comprises of 25 neurons
- The last layer is of size 3 corresponding to 3 actions
- The activation functions used are Relu for the hidden layers
- The last layer uses linear activation function
- The loss function is the minimum squared error

0.4.5 Application on LUNAR-LANDER problem

Lunar Lander problem is the task to control the fire orientation engine to help the lander land in the landing pad. Using 4 actions to land on pad: do nothing, fire the left engine, fire the main engine, fire the right engine. The landing pad is always at coordinates (0,0). If the lander moves away from the landing pad it loses reward. The episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. To get a stable and successful agent, landing safely on the pad with an average rewards over 200 in 100 successive episodes is needed.

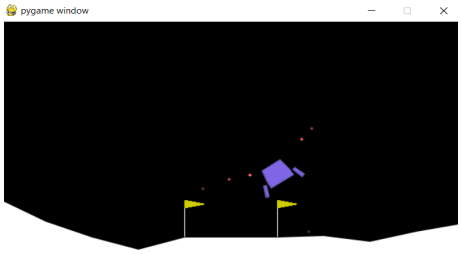


Figure 11: Display

Eps	**Average Scores**
100	-147.49
200	-122.75
300	-63.986
400	-28.43
500	38.087
600	96.094
700	79.706
800	152.26
900	157.29
1000	171.37
1100	172.49
1200	160.63
1300	168.39
1400	156.63
1500	186.59
1600	171.96
1700	191.22
1800	191.56
1858	201.68

Table 11

This problem too has been implemented without the stable_baselines3 library for DQN implementation. Here torch library has been used to build the networks.

- The first layer is to take in the states
- The last layer of the network is to get corresponding Q-values for actions
- The first fully connected layer comprises of 64 neurons
- The second fully connected layer comprises of 64 neurons

- The last layer is of size, corresponding to the 4 actions
- The activation functions used are Relu for the hidden layers
- The last layer uses linear activation function
- The loss function is the minimum squared error