Analytical Questions

Question 1:

One of the biggest advantages of Deep RL is that Transfer Learning techniques can be used in them, which is a very important feature. As we saw in the previous question in the "Let's Collaborate" section, it is possible to use a model trained on one problem and fine-tune it for a similar problem without needing to change the weights of the model. We observed that if the two tasks are similar, we can achieve good results.
Another advantage of Deep RL is that the input data to these networks is raw data. For example, in the case of a self-driving car, we can provide the images captured by the car from its surroundings as input to these algorithms, without the need for data engineering to create a meaningful representation of the input data. This allows the network to find the patterns in raw data that might be difficult for a human to detect, and use these patterns to improve the results further.
However, one of the problems of Deep RL is their sample inefficiency. Typically, in Deep RL, the agent needs more interactions with the environment to learn the best policy, and this large number of interactions can make the learning process time-consuming and costly.

Question 2:

The most important advantage of Deep Q-Learning over Q-Learning is its ability to handle high-dimensional and continuous state spaces. In Q-Learning, a Q-table was used, and based on it, we determined our policy. The size of this table grows exponentially based on the size of the state space. In Deep Q-Learning, instead of maintaining a Q-table, the goal is to use neural networks to map the state space to Q-values and, in fact, estimate the Q-function.
In the two figures below, you can see how, when the dimensionality of the state space increases, the Q-Learning algorithm suffers from overfitting.

Figure 1 shows the cumulative reward over different episodes for the CartPole-V0 task when the state space was discretized into 72 states, and Figure 2 shows the same task when the size of the state space is 5000.

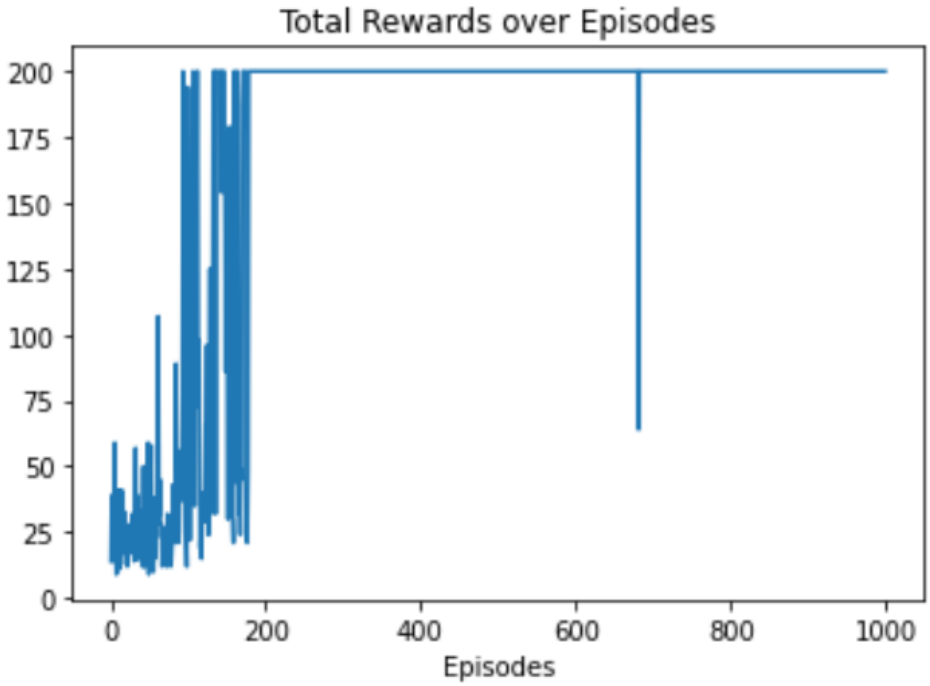


Figure 1

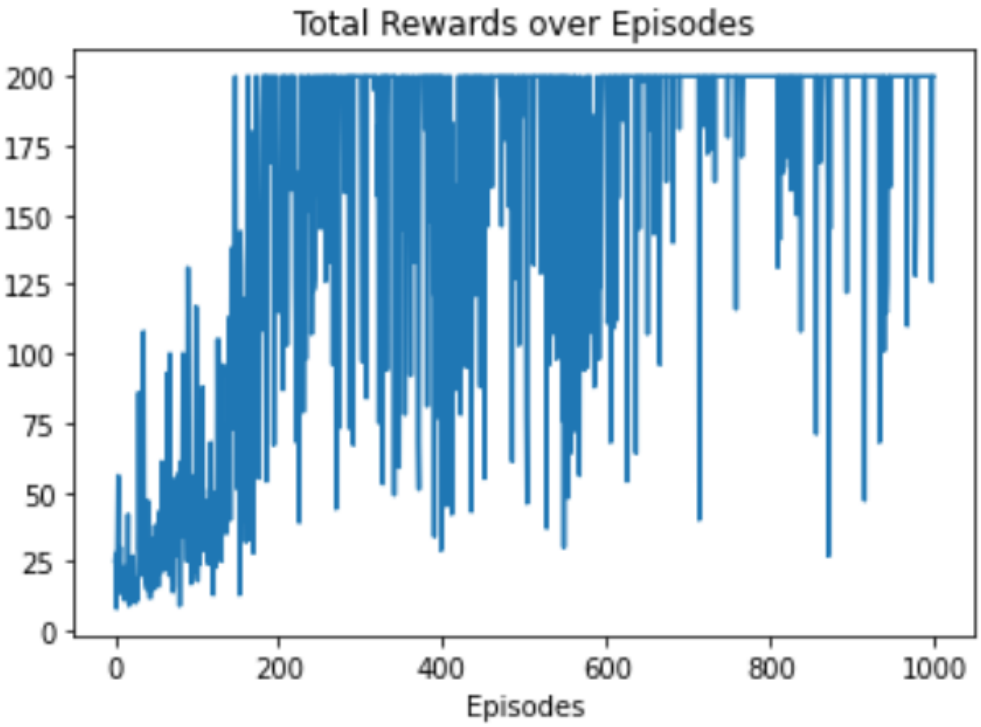


Figure 2

Question 3:

Using an experience buffer has several advantages, one of the most important of which is helping to increase the robustness of the training process. If we try to train our model after each transition from one state to another, given that consecutive transitions within an episode are dependent on each other, our model will learn based on this dependency. As a result, by using this buffer, we can break this dependency.

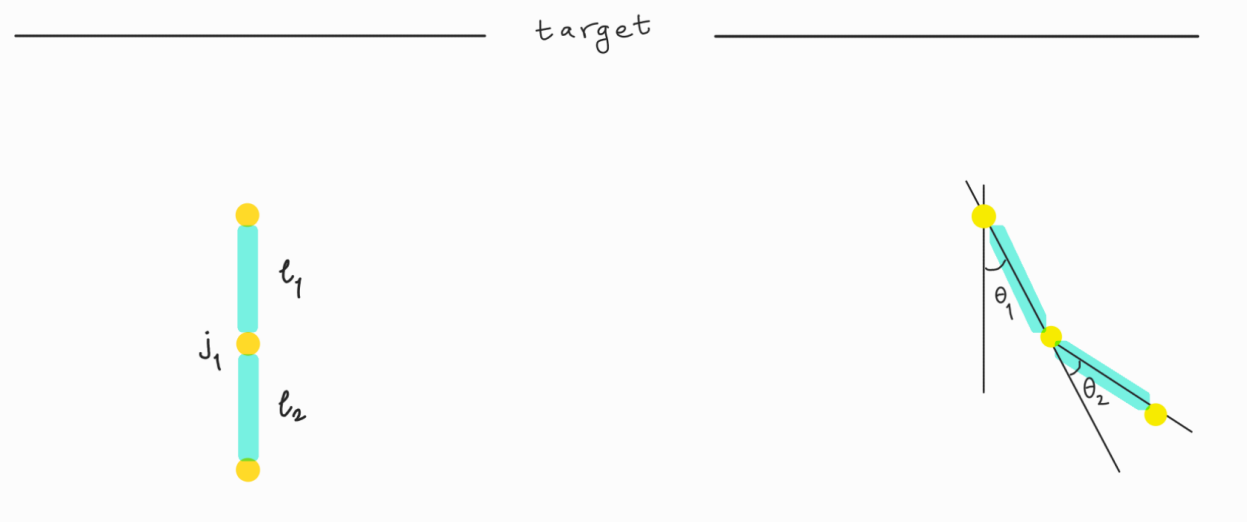
Implementation Question

Part 1-

Question 1:

In the acrobot-v1 model, each state is represented by a six-dimensional vector, where the elements of this vector are defined as follows:
- The first element: the cosine of angle θ1, which has a value between -1 and +1.
- The second element: the sine of angle θ1, which has a value between -1 and +1.
- The third element: the cosine of angle θ2, which has a value between -1 and +1.
- The fourth element: the sine of angle θ2, which has a value between -1 and +1.
- The fifth element: the angular velocity of θ1, which has a value between -4π and +4π radians per second.
- The sixth element: the angular velocity of θ2, which has a value between -9π and +9π radians per second.

Angles θ1 and θ2 correspond to the angle of link ℓ1 and the perpendicular direction, and the angle between link ℓ1 and link ℓ2, respectively. Thus, θ1 equals 0 when ℓ1 is aligned vertically downwards, and θ2 equals 0 when links ℓ1 and ℓ2 are aligned.
These details are shown more clearly in the figure below:



In this model, there are three actions, which are defined as the torque applied to joint j1 (the joint between the two links ℓ1 and ℓ2), known as the actuated joint.
- **Action 1:** A torque of +1 Newton meter applied to the actuated joint.
- **Action 2:** A torque of 0 Newton meters applied to the actuated joint.
- **Action 3:** A torque of -1 Newton meter applied to the actuated joint.

Regarding the reward, it should be noted that at each time step, if the robot has not yet reached its goal, the agent receives a reward of -1. If the robot reaches its goal, which is defined as the free end of the robot reaching the top of the black line, the agent receives a reward of 0.

The termination condition of an episode is met if one of the following two conditions occurs:
A) **Termination Condition:** If the free end of the acrobot reaches the top of the black line.
B) **Truncation Condition:** If the length of the episode exceeds 500 time steps.

Question 2:

As mentioned in the above explanation, the state space, which is defined by two angles θ1 and θ2 and the angular velocities of these two angles, is a continuous space, while the action space, which only has three elements, is a discrete space.

Part 2-

Question 1:

In this section, the deep network we used has two fully connected layers: the first layer contains 128 neurons, and the second layer contains 64 neurons. The input layer has 6 neurons, and the output layer has 3 neurons, which is determined by the problem. For this implementation, we used the PyTorch library. The activation function in the first and second layers is ReLU, and the activation function in the final layer is linear. The loss function is MSE (Mean Squared Error). As specified, this network takes the states as input, which is a 6-dimensional vector, and outputs the q-value for each action.
Here, we have two networks with the architecture described above. We use the first one to predict the q-values for each state, which is trained during each episode. This model is referred to as the "online network." The second model is used to generate q-values for the next states, and these values are then compared with those obtained from the online network using the Q-learning equation. We use the difference between the values for training the online network, which serves as our function approximator. The weights of the target network are updated after every 100 updates of the online network. The update happens by setting the weights of the online network to the target network at that moment.

The function or method “get_action” here, follows the greedy-ϵ policy. In each state, it returns an action, and the value of the epsilon parameter is set to 0.1. After the 100th episode, this value is multiplied by a decay factor

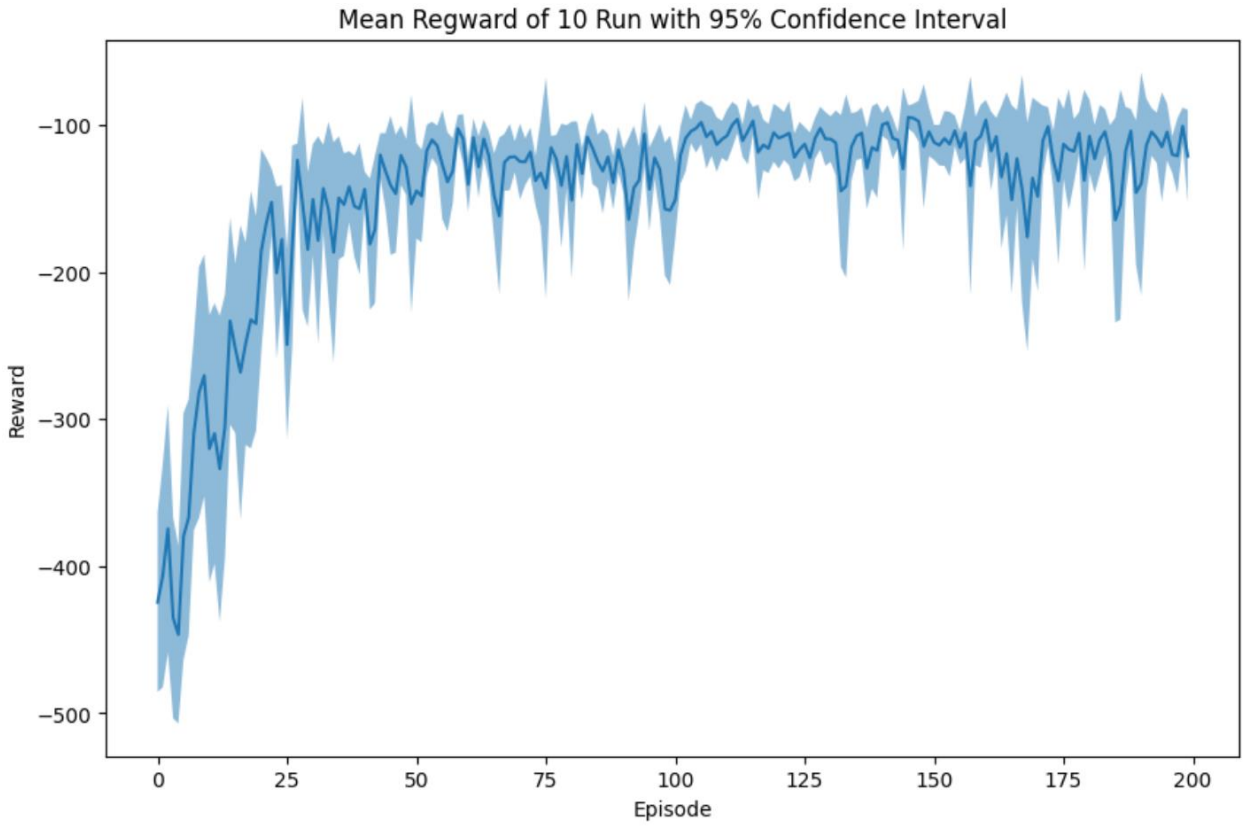
of 0.95 in each subsequent episode, gradually bringing the policy closer to a pure greedy policy.

The “train_network” function is used for training the network. Here, after every 100 updates, the weights of the target network are replaced with the weights of the online network.
For training the online network, we take a batch of size 100 from the replay buffer at each step and train the model using this batch.

The episode function contains a loop that repeats for “num_episode” times. After each time step in an episode, the experience from that time step, which is a 5-tuple (“state”, “action”, “reward”, “next_state”, “is_terminal”), is stored in the replay buffer. After that, the “train_network” function is called. Finally, the cumulative reward from that episode is added to the rewards list, which is one of the attributes of the agent.

Question 2:

As shown in the figure below, with high confidence, it can be said that after 200 episodes, the agent has managed to achieve the reward of the optimal policy, which is equal to -100:



Question 3:

Here, we used the “VideoRecorder” module from the Gym library to render an episode of the game. We selected the desired action for each state in a greedy manner based on the q-values corresponding to those actions in that state, using the “online_network” model.
The video generated is saved in the "Files" folder in the right toolbar of the Google Colab environment under the name training.mp4.

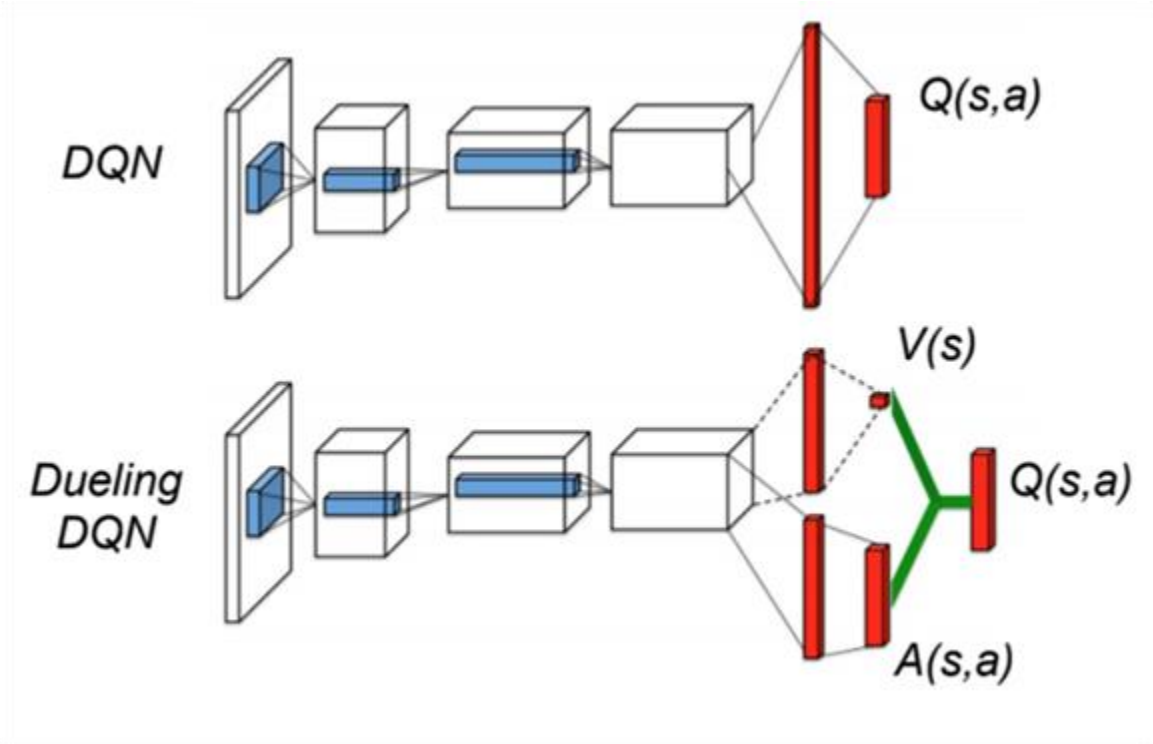
Question 4:

The parameters used in this problem are listed in the table below:

Variable name	Description	value
gamma	Discount factor	0.99
network_type	Type of the networ	simple - dueling
num_episode	Number of training episodes	200
epsilon	ε-Greedy parameter	0.1
rb_size	Size of the replay buffer	300
batch_rb_size	The batch size that is taken from the replay buffer for training the network	100
tar_net_update_period	The frequency of updating the target network weights	100
tar_net_update_index	The index that increases by one after each update of the online network, and when it reaches the value of tar_net_update_period, the weights of the target network are also updated.	0

Part 3-

The chosen implementation in this case was the Dueling Deep Q-Network. The difference between this network and the regular DQN can be clearly seen in the figure below:



The Dueling network, as seen in the figure above, has two separate layers at the final stage, which are at the same level. One of them (the value layer) estimates the value of the input state, and the other (the advantage layer) calculates the Q-value for each state-action pair. In this case, to prevent overestimation of the Q-value, which is influenced by the maximum Q-value in the next state, we calculate the Q-value in Dueling DQN as follows:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a))$$

To compare the results of these two types of networks (simple DQN and dueling DQN), we used a t-test. For each of these two networks, we trained the agent in the environment for a number of episodes equal to 200. The reward for each episode was recorded for both methods, and we performed a t-test on the rewards. The results showed that the p-value for this test was a very small number, almost equal to 0. As a result, the null hypothesis, which stated that there is no difference in rewards between the two methods, was rejected.