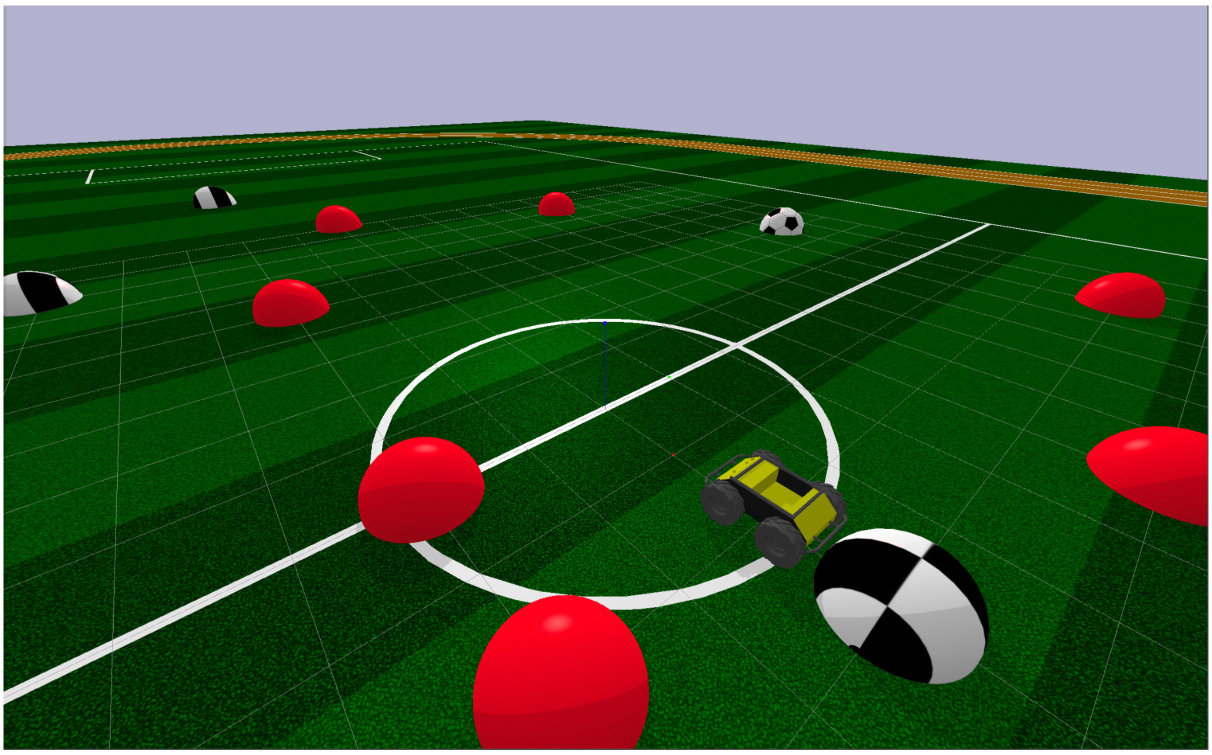
Implementation of Deep Q Network in Pybullet

Objective

Use DeepQNetworks to find the optimal path to the destination using an environment created using pybullet. The environment contains obstacles and a goal point(soccerball). The input states to the network are images and the total area of obstacles in the scene (found using segmentation masks).

Environment in PyBullet (Car with obstacles and Soccerball Goal point)



Theory

Instead of using Q-learning, which can only work for simple environments, we use deep Q networks which do not require a Q table for each state-action pair.

Deep Q Networks, a combination of neural network and RL, is used to approximate an optimal Q function. The network tries to find an "optimal Q function that will satisfy the **Bellman equation**".

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

The current state (images) can be input to the network, and it will output the predicted Q-value for each of the actions from the current state.

The network is trained for multiple episodes and the environment is reset at each new episode. Within each episode, the agent takes multiple steps. At each step, an action is selected.

While choosing an action, an **epsilon-greedy strategy** is used to balance between exploration and exploitation. The next state after performing the action and the reward obtained along with the current state, is stored in the replay memory.

A **replay memory** is used to store all the previous experiences of the agent, this helps to break correlation between two consecutive samples.

A batch of states is sampled from the replay memory and passed through the Policy Network. The corresponding next states from the sampled batch are passed to the Target Network.

The loss value is calculated between the output Q values from the Policy network and the target Q values from the Target network. Then, we update the network weights accordingly.

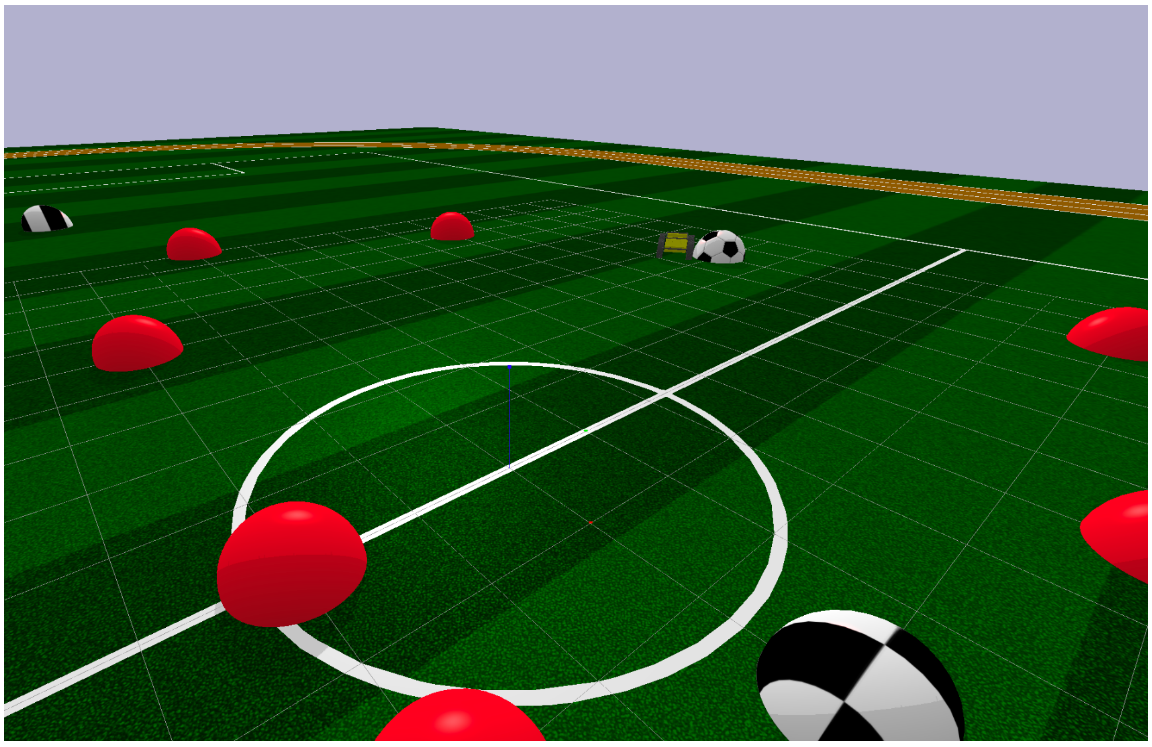
Algorithm

1. Initialize replay memory capacity.
2. Initialize the network with random weights.
3. *For each episode:*
    1. Initialize the starting state (image, area of obstacles).
    2. *For each time step:*
        1. Select an action.
            - *Via exploration or exploitation*
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass batch of preprocessed states to policy network.
        8. Calculate loss between output Q-values and target Q-values.
            - Requires a second pass to the network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.

Hyperparameters Used

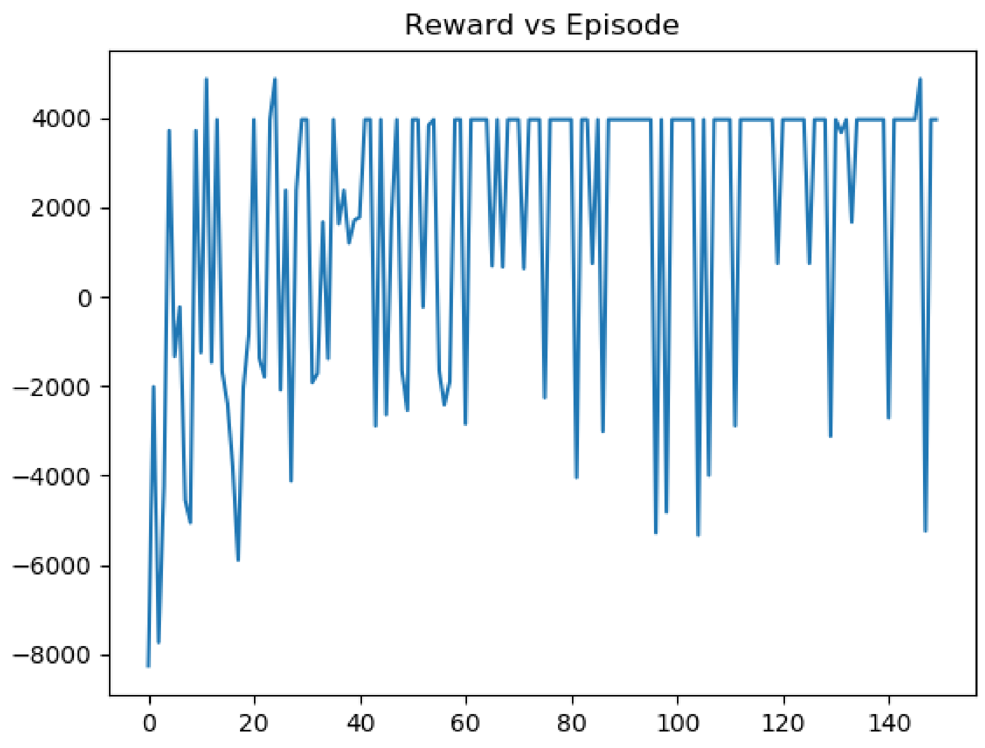
- Input State: **image captured + total area of all obstacles obtained using segmentation masks**
- Possible Actions - **4 directions (Left, Right, Front, Back)**
- Reward
  - On reaching goal = 5000
  - On collliding with obstacles = -0.5*area of obstacles
  - Outside boundary = -100
- Threshold distance to goal = 2
- Episodes = 150
- Max_Steps = 20
- Replaymemory size = 10000
- Epsilon_start = 0.9
- Epsilon_stop = 0.05

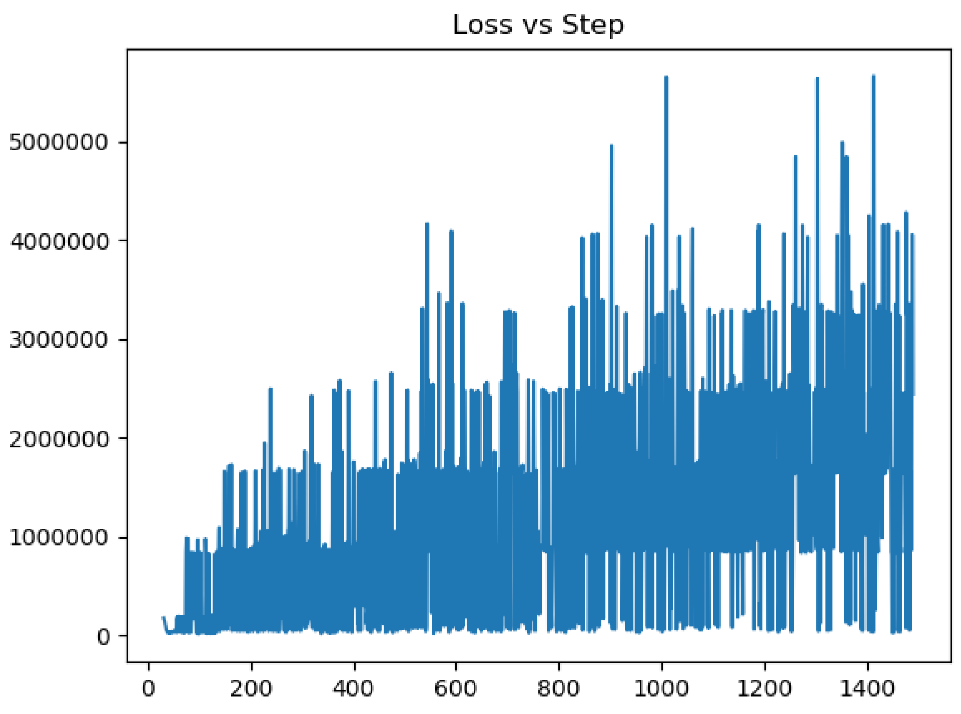
Results



Video - Car reaching the goal
https://iiitaphyd-my.sharepoint.com/:v:/g/personal/gowri_lekshmy_research_iiit_ac_in/EWbjOh3cbxdCjbECwfWEOzEB3FZoCWACQ4ieNp5LH-I5hg?e=iXKcXC

Plots


Reward vs Episode
4000
2000
0
−2000
−4000
−6000
−8000
0
20
40
60
80
100
120
140


Loss vs Step
5000000
4000000
3000000
2000000
1000000
0
0
200
400
600
800
1000
1200
1400