Reinforcement Learning Final Project Report

Project goals

Implementing both DQN and Double DQN algorithms to solve three discrete action space tasks. The environments are Cartpole (v1), Acrobot (v1) and Mountain Car (v0).

Implementation of DQN and Double DQN

Implementation of DQN

The network is a one hidden layer architecture with 256 neurons. And the activation function is the Leaky relu activation function to prevent dying relu.

```
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(QNetwork, self).__init__()
        # input  : (b_size, state_dim)
        # output : (b_size, action_dim)
        self.hidden_dim = 256
        self.layer1 = nn.Linear(state_dim, self.hidden_dim)
        self.layer2 = nn.Linear(self.hidden_dim, action_dim)

    def forward(self, s):
        #
        s_x = F.leaky_relu(self.layer1(s))
        q = self.layer2(s_x)

        return q
```

Two networks are initialized: the main q network and a target network. The state dict of the weights of the main q network are loaded to the target network. The loss is the MSE loss and for the optimizer the Adam optimizer.

```
# Define and initialize your networks and optimizer
self.q_network =  QNetwork(self.state_size, self.action_size).to(device)
self.target_network =  QNetwork(self.state_size, self.action_size).to(device)
self.target_network.load_state_dict(self.q_network.state_dict())
self.optimizer = optim.Adam(self.q_network.parameters(), lr=self.learning_rate)
self.criterion =  nn.MSELoss()
```

The target network updating is a hard updating of the weights.

```
def update_target_network(self):
    # implement target Q network update function
    self.target_network.load_state_dict(self.q_network.state_dict())
```

For action selection when interacting with the environment the get_action function uses the epsilon greedy algorithm. This returns a random action with probability epsilon or a greedy action probability 1-epsilon. More specifically, the greedy action is an action chosen by inputting the state to the main network. That is, based on the values of the state for each action, the action that has the highest value given the state is selected as the (greedy) action that will be taken.

```
def get_action(self, state, use_epsilon_greedy=True):

    state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)

    if use_epsilon_greedy:
        # implement epsilon greedy policy given state
        if random.random() < self.epsilon:

            action = random.randrange(self.action_size)
        else:

            with torch.no_grad():
                action = self.q_network(state).max(1).indices.item()
    else:
        # implement greedy policy given state
        # this greedy policy is used for evaluation
        with torch.no_grad():
                action = self.q_network(state).max(1).indices.item()

    return action
```

For the replay memory, the state, action, reward, next state and done are appended to the buffer.

```
def append_sample(self, state, action, reward, next_state, done):
    # implement storing function given (s,a,r,s',done) into the replay memory.
    self.memory.append((state, action, reward, next_state, done))
```

The sampling function is more complex as this is where the tensors must be sent to the gpu. (**Note**: When I first implemented DQN, I mistakenly loaded the tensors to the gpu when saving the data to the replay buffer and this made my implementation 2x slower. Thus, I learned that it is more efficient to load the tensors to the gpu only after sampling.)

```
def get_samples(self, n):
    # implement transition random sampling function from the replay memory,
    # and make the transiton to batch.

    transition_batch = random.sample(self.memory,n)

    states, actions, rewards, next_states, dones = zip(*transition_batch)

    s_batch = torch.tensor(states, dtype=torch.float32, device=self.device)
    a_batch = torch.tensor(actions, dtype=torch.long, device=self.device).unsqueeze(1)
    r_batch = torch.tensor(rewards, device=self.device).unsqueeze(1)
    s_next_batch = torch.tensor(next_states, dtype=torch.float32, device=self.device)
    done_batch = torch.tensor(dones, dtype=torch.bool, device=self.device)

    # i.e.) s_batch : (batch_size, state_dim)
    return s_batch, a_batch, r_batch, s_next_batch, done_batch
```

The epsilon decay update simply multiplies the decay rate for the current epsilon value and selects the maximum between this multiplication and the min threshold value.

```
def epsilon_decay(self):
    # implement epsilon decaying function
    self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay_rate)
```

The train function is where the core of the DQN algorithm (apart from the memory replay of course, but here it is where it is used). First, a batch of samples are sampled from the memory replay buffer, and sent to the q_network to obtain the value of the given state for the taken action (that is why .gather(1, a_batch) is used). Then by using the target network (and excluding the terminal next states which are given a value of zero) the value of the next states for the highest valued next action is obtain. In other words, the values of the next states are computed and the value with the highest value is selected. Implicitly these are two phases: 1) a greedy next action is chosen and 2) the next state value for this next action is estimate. Just that in other order.

```
def train(self):
    s_batch, a_batch, r_batch, s_next_batch, done_batch = self.get_samples(self.batch_size)

    # implement DQN training function.
    # You can return any statistics you want to check and analize the training.  (i.e. loss, q values, target q values, ... etc)

    q_values = self.q_network(s_batch).gather(1,a_batch)
    next_state_values = torch.zeros(self.batch_size, 1, device=self.device)

    with torch.no_grad():
        next_state_values[~done_batch] = self.target_network(s_next_batch[~done_batch]).max(1).values.unsqueeze(1)

    expected_q_values = (next_state_values * self.discount_factor) + r_batch
    loss = self.criterion(q_values, expected_q_values)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    return None
```

Implementation of Double DQN

It is basically all the same as DQN except for two things. First Double DQN explicitly selects the next action and then obtains the value for taking that next action in the given next state (reverse order than DQN). Second, Double DQN uses the main q network to select the greedy next action and the target network only computes the value for taking that action in the given the next state.

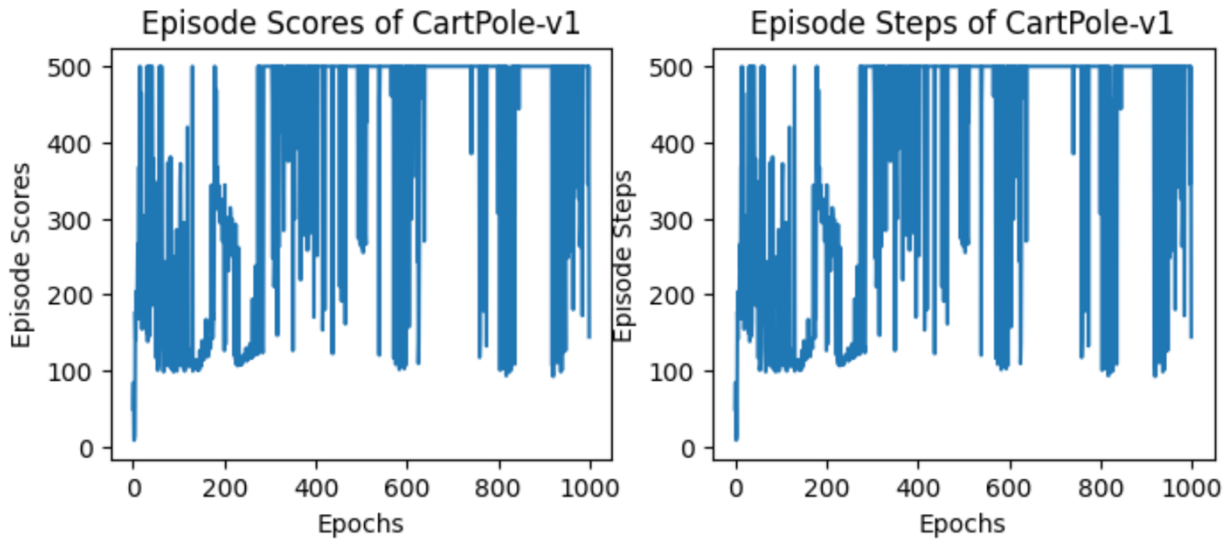
```
############################################################
# Q network chooses the best action for next states
next_actions = self.q_network(s_next_batch).max(1).indices.detach().unsqueeze(1)

# Target network estimates Q values of the selected actions
next_state_values = torch.zeros(self.batch_size, 1, device=self.device)
with torch.no_grad():
    next_state_values[~done_batch] = self.target_network(s_next_batch[~done_batch]).gather(1, next_actions[~done_batch])

############################################################
```

Train results and analysis of DQN and Double DQN

The epsilon decay rate was set the same across DQN and Double DQN for comparison. However, the rate was set per environment. Also, the performance for comparison was the average reward and average step obtained during evaluation from episode 500 to 1000.

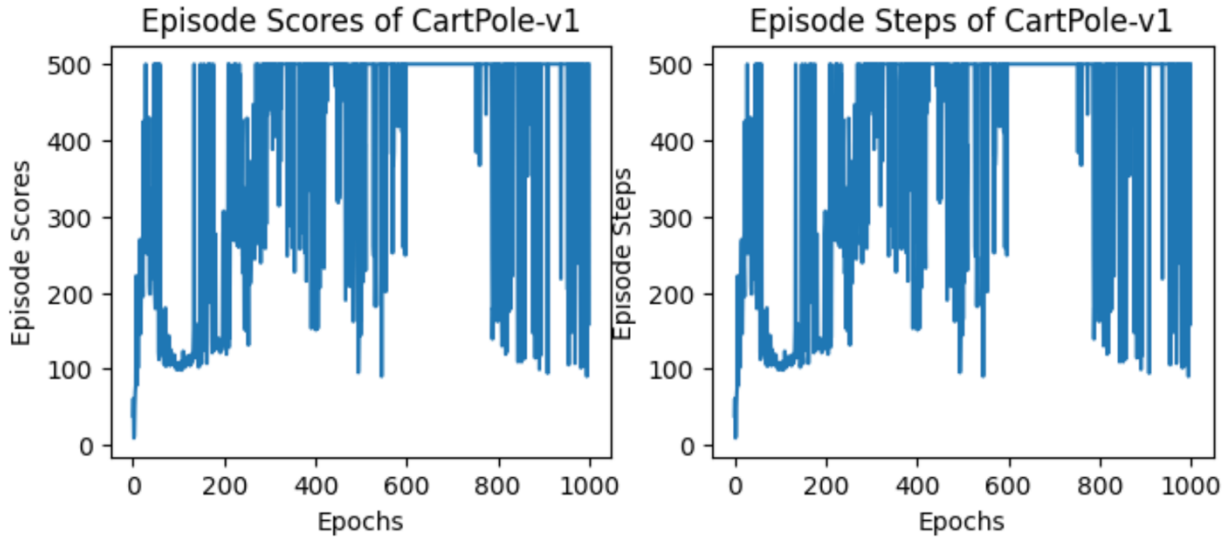
Cartpole: DQN

The epsilon decay rate of was set to 0.995 for DQN. The average reward of the evaluated episodes was 449. This means DQN solved the task successfully as it is greater than 195.



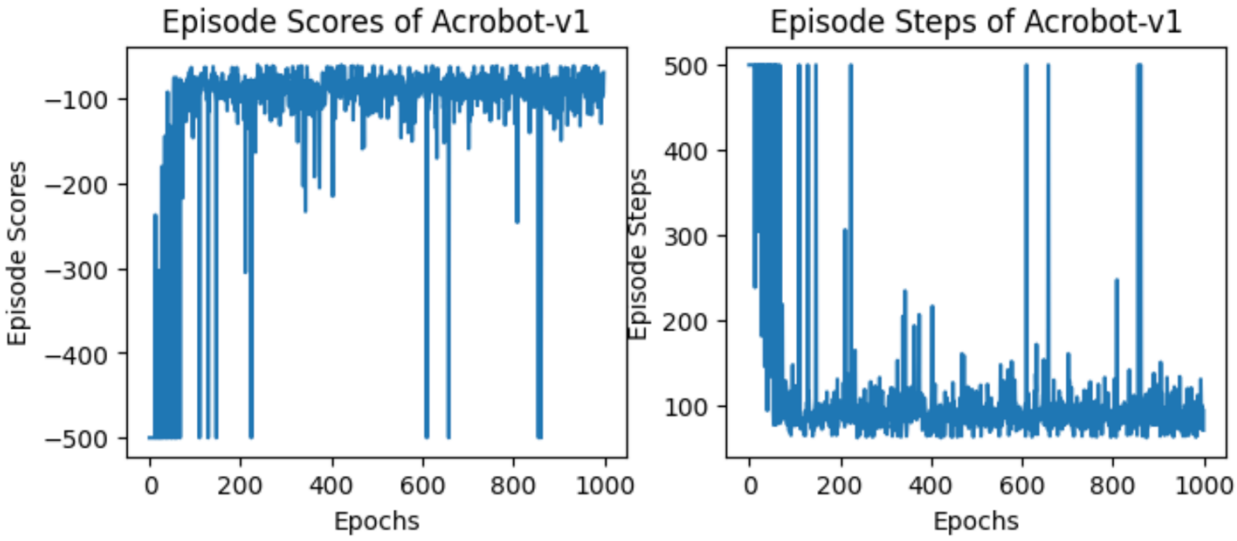
Cartpole: Double DQN

Double DQN achieved an average reward score of 457, showing a better performance than DQN for this task.



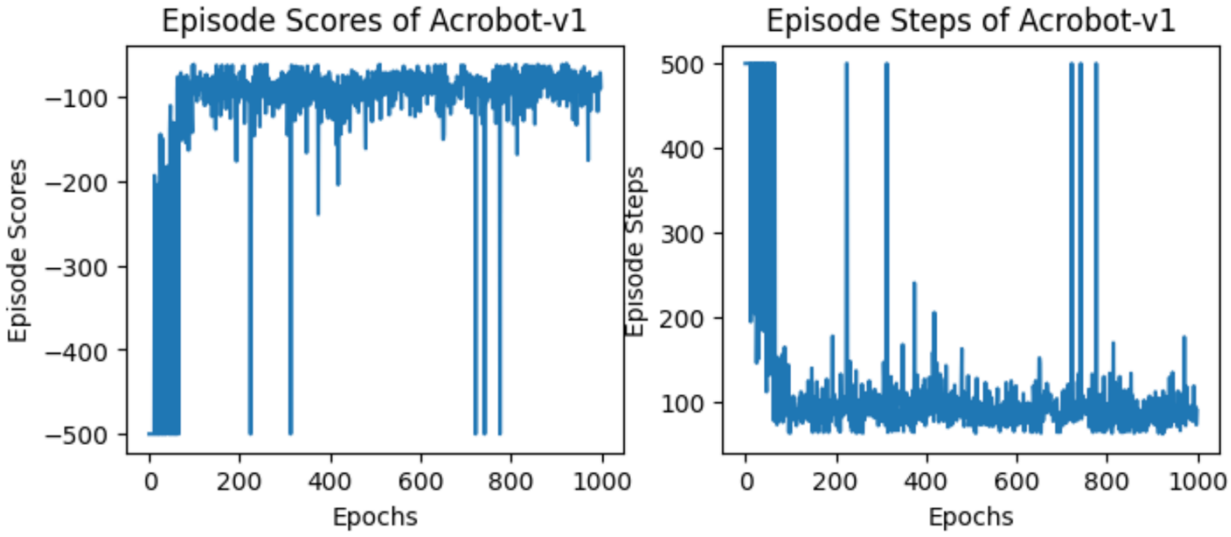
Acrobot: DQN

The epsilon decay rate was set to 0.9. The average reward was -92 and average steps were 93. Since most of the rewards are greater than -100. Even though there is not a known threshold DQN can be considered successful with this score.



Acrobot: Double DQN

For Acrobot, Double DQN achieved an average reward score of -90 and average steps of 91. Again, Double DQN showed a better performance.



Mountain Car

Mountain Car is an environment with sparse reward that usually requires reward reshaping or a long training to be well trained. Therefore, unlike the cartpole and acrobot where finding the epsilon rate did not require much searching, for mountain car many experiments were required. The following are the tables with the resulting average reward for both DQN and Double DQN for several epsilon decay rates.

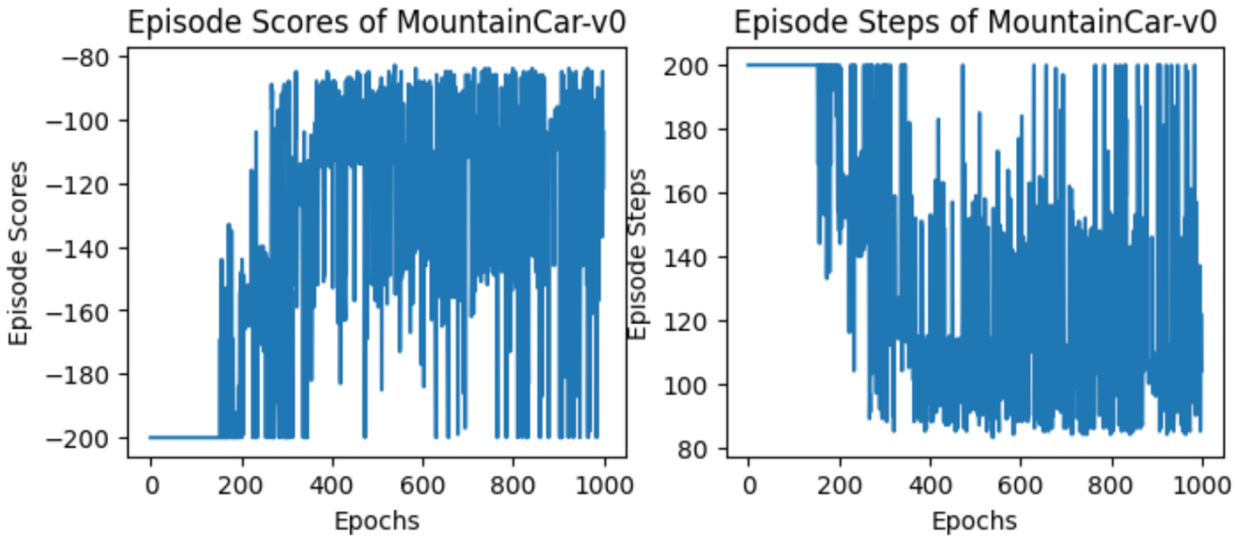
It is important to notice that neither DQN or Double DQN were able to successfully solve this task (as the threshold is -110). This is because of the sparse reward nature of the environment. Thus, it would require a longer training and possible reward reshaping.

DQN

For DQN, the best epsilon decay rate was 0.95.

Epsilon decay rate	Average reward (500~1000 episodes)
0.999	**-136**
0.995	**-124**
0.99	**-126**
0.95	**-120**
0.9	**-133**
0.85	**-121**
0.8	**-129**

The reward graph obtained was the following.

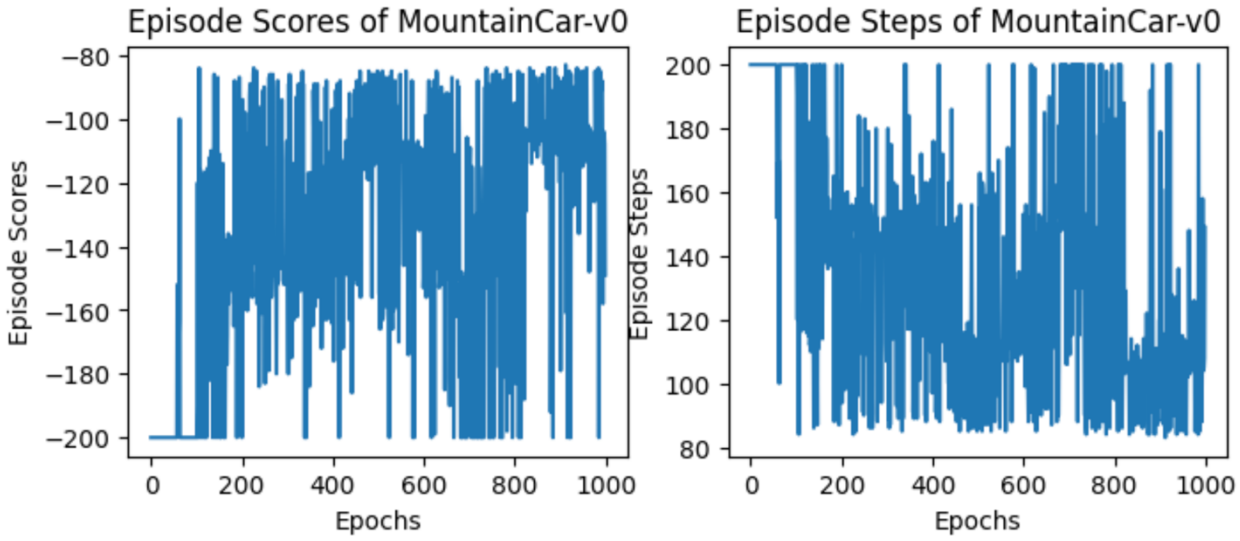


Double DQN

In the case of Double DQN, these are the average rewards given the same Epsilon decay rates.

Epsilon decay rate	Average reward (500~1000 episodes)
0.999	**-148**
0.995	**-125**
0.99	**-127**
0.95	**-127**
0.9	**-124**
0.85	**-132**
0.8	**-127**

For the rate of 0.95 (best of DQN) the graph of rewards is the following.



Conclusion

DQN is one of the core RL algorithms that ignited attention to the area due to the combination of RL with deep neural networks. It is also a simple to implement algorithm that can solve discrete action spaces tasks that have no sparse rewards.

Cartpole and acrobot can be solved using DQN and improved by Double DQN. However, in the case of the MountainCar, some further strategy to deal with the sparse reward is vital.