Implementation of Deep Q-Networks
AE598 - HW2

Maulik Bhatt *
Department of Aerospace Engineering, UIUC

Deep Q-Networks (DQNs) have achieved remarkable success in a variety of challenging tasks in reinforcement learning. DQNs employ a combination of deep neural networks and Q-learning to learn optimal policies for decision-making in complex environments. Due to their ability to handle high-dimensional input spaces, continuous states spaces and non-linear dynamics, DQNs have been widely used in many domains, including robotics, gaming, finance, and healthcare. In this assignment, I have explored the implementation of DQNs. Finally, I have used DQNs to control a discrete-action pendulum.

I. DQNs

We express the control problem of dynamical systems as a reinforcement problem which is controlling a Markov Decision Process(MDP). We achieve this by DQN architecture which combines deep neural networks and Q-learning. We try to approximate the state action value function($Q(s,a)$) through a neural network. Below is the pseudo code for the DQN algorithm.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Fig. 1 Pseudo Code

Following are the details of hyperparameters and various components of the code

*NetID: mcbhatt2

A. Neural Network

We use a neural network with two hidden layers that each have 64 units with a tanh activation function at both the hidden layers, with a linear activation function at the output layer.

```python
class DQN(nn.Module):

    def __init__(self, n_states, n_actions):
        super().__init__()
        self.layer1 = nn.Linear(n_states, 64)
        self.layer2 = nn.Linear(64, 64)
        self.out = nn.Linear(64,n_actions)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.tanh(self.layer1(x))
        x = F.tanh(self.layer2(x))
        x = self.out(x)
        return x
```

Fig. 2 DQN - class

B. Replay Memory

Replay memory is a component of Deep Q-Networks (DQNs), which enables them to learn from past experiences and improve their decision-making capabilities over time. Replay memory is essentially a buffer that stores transitions experienced by the agent during interaction with the environment. Each transition consists of the current state, the action taken, the resulting reward, and the next state. Furthermore, we also store if the a state terminated or not in the transitions.

```python
class ReplayMemory(object):

    def __init__(self, capacity, Transition):
        self.memory = deque([], maxlen=capacity)
        self.Transition = Transition

    def push(self, *args):
        """Save a transition"""
        self.memory.append(self.Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Fig. 3 Replay Memory

C. Updating the NN

We define a function to update the neural network which corresponds to sampling a random mini-batch from the replay memory, computing the targets y_j's performing a gradient descenet step on the loss funtion which is taken to be MSELoss.

```python
def update_network(policy_net, target_net, optimizer, memory, env, eps, batch_size, Transition, gamma):
    if len(memory) < batch_size:
        return

    batch_data = memory.sample(batch_size)

    batch = Transition(*zip(*batch_data))

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
    next_state_batch = torch.cat(batch.next_state)


    Q_s_a = policy_net(state_batch).gather(1, action_batch)

    y_js = torch.zeros(batch_size)

    with torch.no_grad():
        for i in range(batch_size):
            if batch.done[i]:
                y_js[i] = reward_batch[i]
            else:
                y_js[i] = reward_batch[i] + gamma*target_net(next_state_batch[i]).max()

    criterion = nn.MSELoss()

    loss = criterion(Q_s_a, y_js.unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Fig. 4 Update the network

D. Train the network

The corresponds to the outer loop in the pseudo code which consists of collecting data by taking action according to the policy network, updating the replay memory, calling the update network function, updating the target network according to the chosen hyper-parameter.

```
def train_network(num_episodes, policy_net, target_net, optimizer, memory, env, n_actions,
                  n_states, batch_size, eps_start, eps_end, gamma, Transition):
    reward_array = np.zeros(num_episodes)
    itr = 0
    for i_episode in range(num_episodes):
        # Initialize the environment and get it's state
        s = env.reset()
        s = torch.tensor(s, dtype=torch.float32).unsqueeze(0)
        done = False
        time = 0
        while not done:
            eps = eps_decay(itr,num_episodes, eps_start, eps_end)
            a = action(s, eps, n_actions, policy_net)
            s_next, r, done = env.step(a.item())

            r = torch.tensor([r])
            reward_array[i_episode] += gamma**(time)*r
            time += 1

            s_next = torch.tensor(s_next, dtype=torch.float32).unsqueeze(0)

            memory.push(s, a, s_next, r, done)

            # Move to the next state
            s = s_next

            # Perform one step of the optimization (on the policy network)
            update_network(policy_net, target_net, optimizer, memory, env, eps, batch_size, Transition, gamma)

            itr+=1

            if itr%1000 == 0:
                target_net.load_state_dict(policy_net.state_dict())

    return policy_net, reward_array
```

Fig. 5 Training the NN

E. List of hyper-parameters

Most of the hyperparamters are taken directly from [1].

Hyperparameter	Value
batch size	64
gamma(γ)	0.95
eps start	1
eps end	0.1
number of episodes	100
Optimizer(RMSprop) learning rate	0.00025
Optimizer(RMSprop) alpha	0.05
Replay memory size	1000000
No. of steps to update target network	1000
Number of training runs	20

II. Results

A. Learning curve

As we can see, reward is increasing over number of episodes.

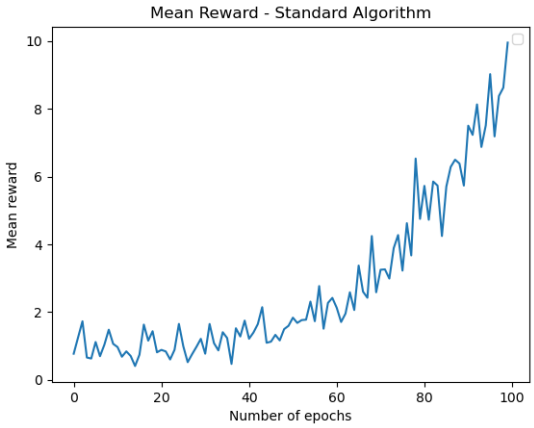

Fig. 6 Learning Curve

B. Example Trajectory

The agent learns policy to keep the pendulum in inverted vertical position.

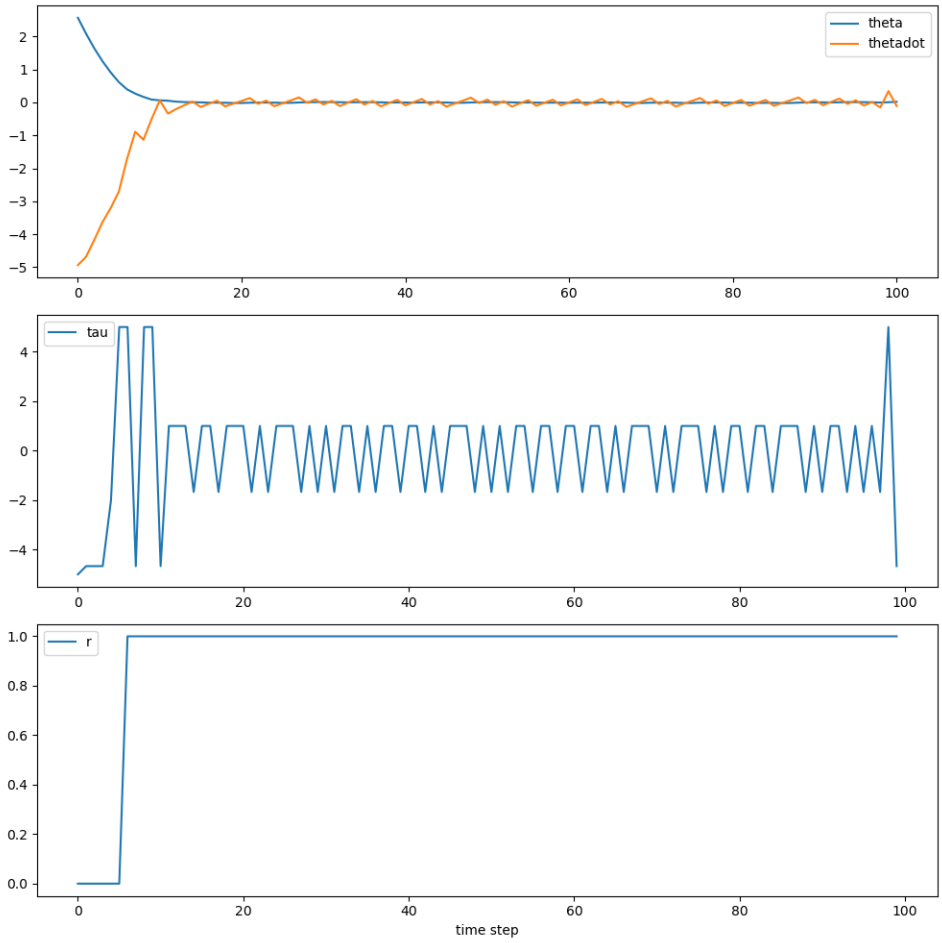

Fig. 7 Example Trajectory

C. Policy

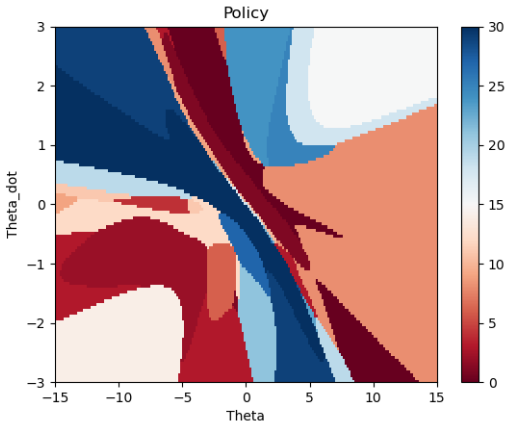

Fig. 8 Policy

D. State Value Function

As we can see from the plot, the state value function is very high around $\theta = 0$ and $\dot{\theta} = 0$, which is very interesting to see that DQN learns this from the reward given by the environment. Also, as we can see the state value is high when $\theta \cdot \dot{\theta} < 0$ i.e. if angle of pendulum is positive, the rate of change of angle should be negative and vice versa for a favorable state that will lead it to vertical position. The DQN is able to learn this also from the data which shows how power DQN is in general.

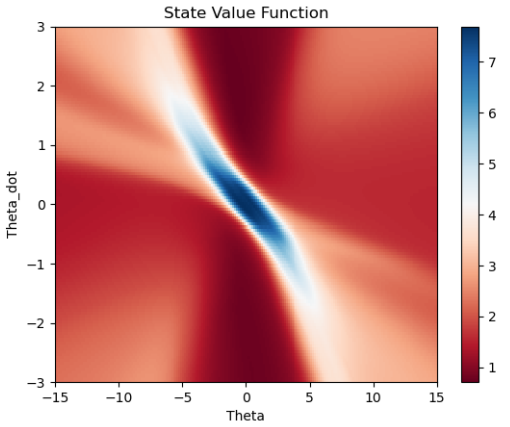

Fig. 9 State Value Function

E. Ablation Study

From the ablation study, we can observe that

- The standard DQN algorithm performs the best and obtains highest rewards
- If we remove the target network, the performance of DQN decreases slightly.
- By not having replay memory, the performance significantly decreases and almost simialr both with and without target network.

Therefore, we can conclude that the importance of having replay memory in DQN is much higher than having a target network

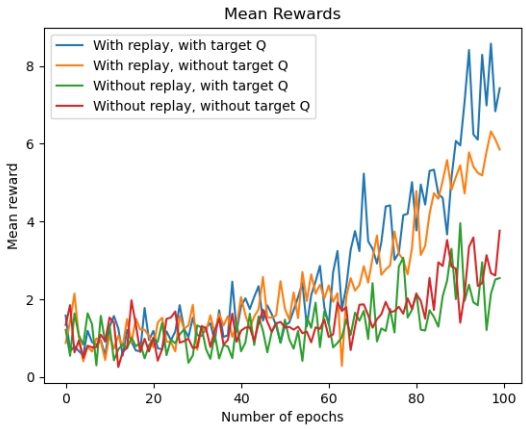

Fig. 10 Ablation Study

III. Conclusion

In conclusion, we implemented DQN for this homework. The key learnings were as follows,

- Pytorch is a powerful tool to implement machine learning algorithms. Learning and knowing various functionalities in pytorch is very important
- Finding good hyperparamters is crucial in RL algorithms and also rthe parameters are very problem specific. Even though most of the parameters are given in the the reference paper, I found out that using batch size of 64 was better for me than using 32. I also used number of steps to update target network to be 1000 as it worked best for me.
- Replay memory has a significant impact on the performance of DQN

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al., "Human-level control through deep reinforcement learning," *nature*, Vol. 518, No. 7540, 2015, pp. 529–533.