Deep Q-Learning

Mohammed Sharfuddin(201401205)

Recap

- The aim of our project is to try to build agents to play a game with a super-human level performance
- We have looked at how to model playing a game using an MDP,using Model-free and Model-based approaches,the tradeoff between exploration and exploitation while playing a game(ϵ-greedy)
- We have implemented an agent that uses a Q-table to make decisions in any particular state
- This works well in a discrete state-action space,however in real world applications with a huge state-action space constructing a Q-table will be an issue

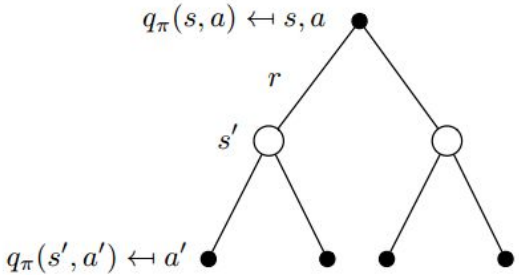
MDP Review

- A MDP is a Markov reward process with decisions. It is an environment in which all states are Markov.
- A Markov Decision Process is a tuple <S, A,P, R, γ>
- A policy π is a distribution over actions given states: π(a|s) = P [At = a | St = s]
- The action value function Q(s,a) gives the expected return on taking an action “a” in the state “s”

$$
\begin{aligned}
Q^{\pi}(s,a) &= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right] \\
&= \mathbb{E}_{s'}\left[r + \gamma Q^{\pi}(s', a') \mid s, a\right]
\end{aligned}
$$

Bellman Optimality Equation

The optimal Q-value function can be written as



$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

$$Q^*(s,a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a \right]$$

Deep Q-Learning

In Deep Q-learning we try to represent the state value function using a deep Q-network with weights ***w***

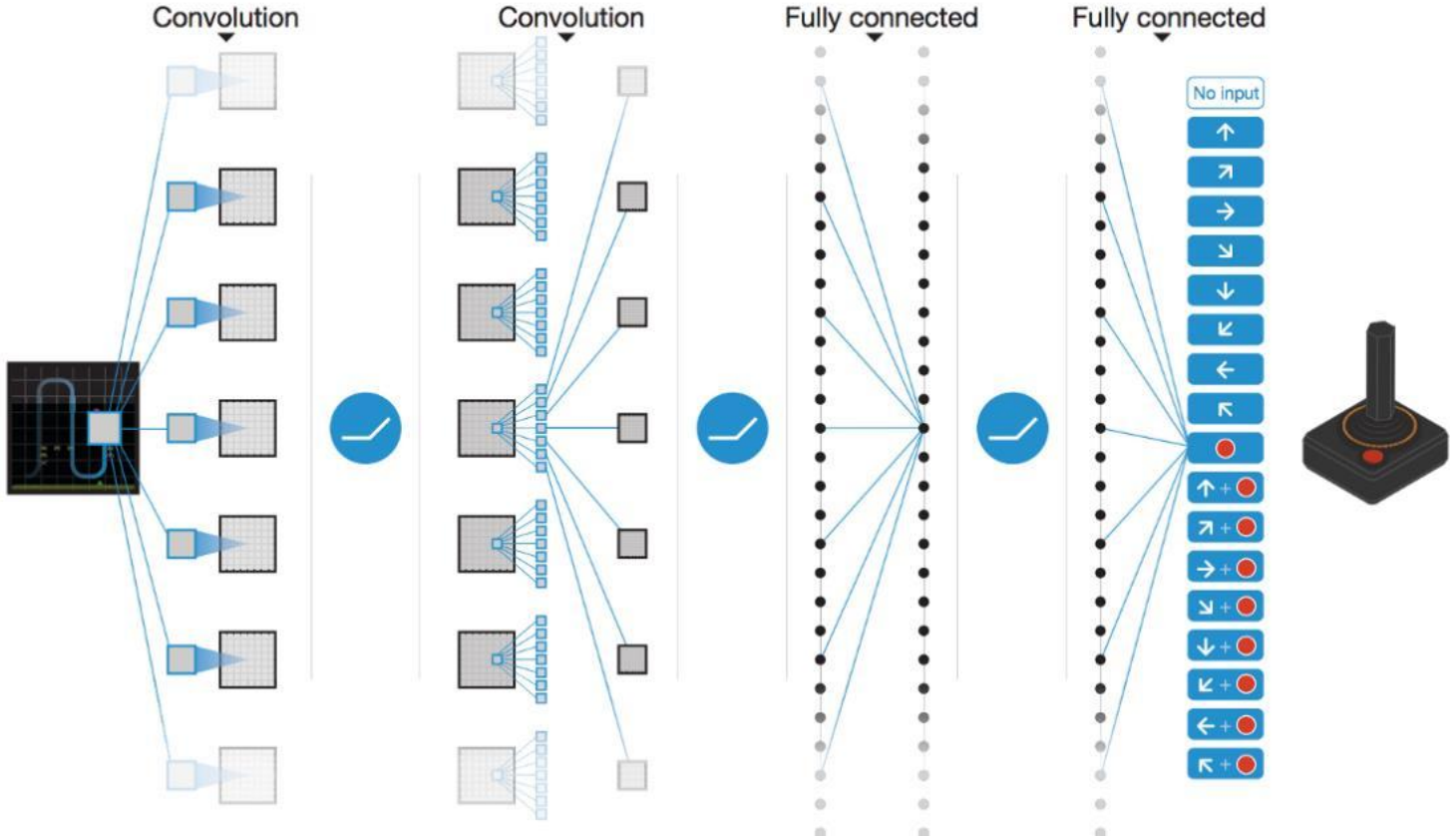
$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

For Q*(s,a) for any (s,a) the bellman optimality condition follows,thus we use the following loss function for the DQN

The below is called TD(0) error

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

Deep Q-Learning



Deep Q-Learning

1.We use a multi-layer convolutional network to estimate the action-value functions

2.Implementing Experience Replay, which will allow our network to train itself using stored memories from it’s experience.

3.Utilizing a second “target” network, which we will use to compute target Q-values during our updates

The Network

- Naive Q-learning oscillates and diverges with naive neural nets using a limited and discrete state space,to generate the Q-values*
- In DQN the solution provided is stable and generally improves along the direction of gradient of loss
- We use a Q-network to estimate Q(s,a;$\boldsymbol{\theta}$') and use a target network to estimate the value of Q(s',a';$\boldsymbol{\theta}$)
- At every step we update the Q-network using the step-loss and experience replay
- We accumulate the loss at every step and use this to update our target network after fixed number of steps

Exploration vs Exploitation

- To decide upon what action is to be performed, a set probability value(epsilon) is used
- A randomly generated number is used to determine whether the agent will take random action(Explore) or take the best greedy action(exploit)
- If the random number is above the probability threshold, the optimal action yielding the highest Q-value is selected (exploitation).
- Otherwise, a random action is selected (exploration)

Experience Replay

- To remove correlations, we build a data-set from agent's own experience
- We sample randomly from this dataset during training which breaks the correlations in data
- Store transition (st, at,rt+1,st+1) in replay memory D
- Sample random mini-batch of transitions (s, a,r,s0) from D

$$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w) \right)^2 \right]$$

Implementation on Cartpole Environment

CartPole-v0

- We built an agent for the CartPole environment in OpenAI gym that uses the DQN network to select actions in a given state
- The aim of the agent is to balance the pole such that it does not tilt more than 15 degrees and the cart doesn’t exceed the left and right boundaries of the frame while it moves
- An iteration ends when either of the above two happen
- The player can take two possible actions-move left,move right
- The agent receives a reward of +1 for every timestep spent in the environment

Pre-Processing

- We model this problem as an MDP,the state $\phi(.)$ in our MDP consists of the previous frame and the current one
- Though the image frames are of size 400x600,the information essential for making a decision is the Cart-Pole and it’s distance from the center of the screen
- So we clip unnecessary boundaries and resize the image by scaling it down by 10 times
- Doing this reduces the time required to train the network and forwarding through this network to take a decision also becomes faster
- We create a buffer of size 10,000 for storing our replay experiences
- We set the discount factor to 0.9

The design of the Q-Network

- Our Q-network consists of 3 convolutional layers,followed by a fully connected linear layer
- The RGB channels of the current and previous frames are input to the Q-network

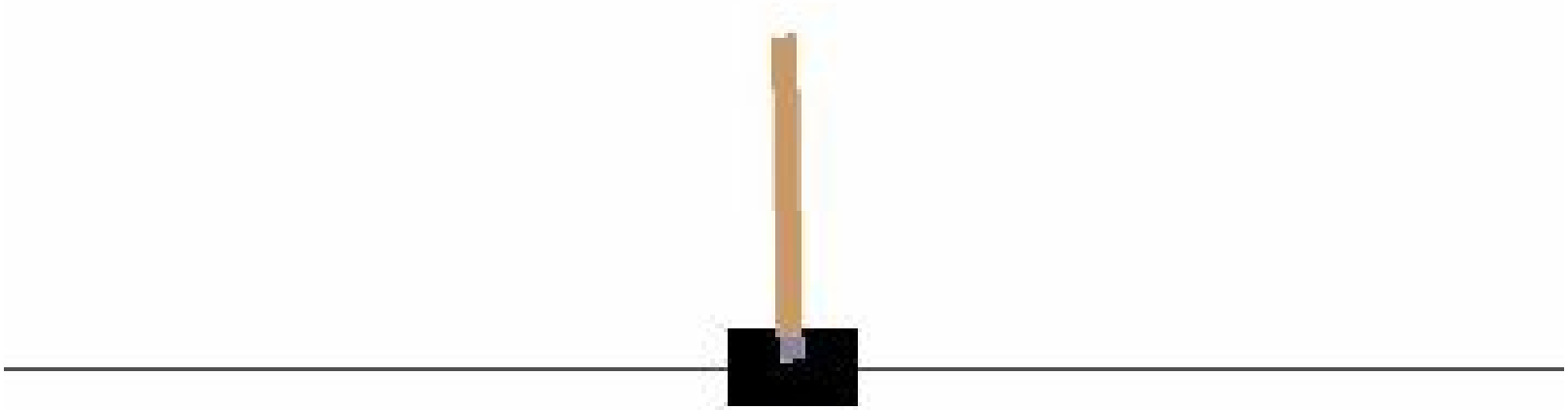
```
def __init__(self):
    super(DQN, self).__init__()
    self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
    self.bn1 = nn.BatchNorm2d(16)
    self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
    self.bn2 = nn.BatchNorm2d(32)
    self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
    self.bn3 = nn.BatchNorm2d(32)
    self.head = nn.Linear(448, 2)

def forward(self, x):
    x = F.relu(self.bn1(self.conv1(x)))
    x = F.relu(self.bn2(self.conv2(x)))
    x = F.relu(self.bn3(self.conv3(x)))
    return self.head(x.view(x.size(0), -1))
```

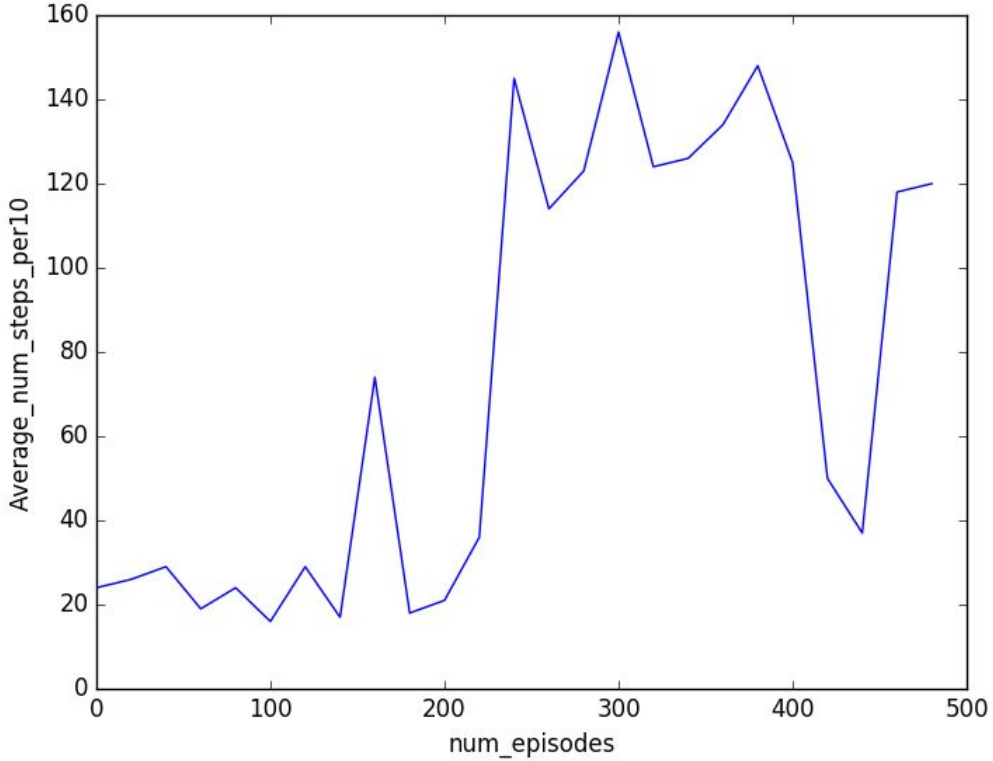
Playing the Game

- We follow the epsilon greedy approach while playing the game,the starting value of epsilon=0.9,this decays slowly as the number of iterations increase
- Two copies of the DQN networks are instantiated-One for the target network and the other for the Q-network
- At every step we randomly sample a batch of size 128 from the replay memory and use the loss function and do backpropagation on the Q-network
- The squared loss for a particular episode is accumulated and this loss is used by the target network to do backprop and update its parameters
- The loss function used by the Q-network is the mean-squared-loss(since the values are normalized,this is the same as mean squared error)

Demo and the Results



Performance



Actor-Critic Methods

- We plan to implement Model-based approaches,where we try to find an optimal stochastic policy for the game
- We use a CNN to generate a policy,the objective function for this network is the expected return from the given state
- The objective function for this CNN is the expected return on following a policy
- We use a DQN to estimate the expected return for a given state-action pair
- We train both of these networks independently