Deep Reinforcement Learning (Sp25)

Instructor: Dr. Mohammad Hossein Rohban

Summary of Lecture 5: Function Approximation & Policy Gradient methods

Summarized By: Benyamin Naderi

- **Off-policy Learning** is a method in which the policy used to generate behavior, by sampling trajectories (S_t, A_t, R_t, S_{t+1}), differs from the policy being evaluated and improved. This separation enables the agent to learn from experiences produced by a different policy, known as the behavior policy, which samples trajectories from the environment. These experiences can include past interactions or exploratory actions, allowing the agent to improve its target policy without being constrained by the behavior policy.

- In contrast, **On-policy Learning** is where the agent learns and improves the same policy that it uses to interact with the environment.

- Recall **Bellman's optimality equation** where we wanted to sample the expected :

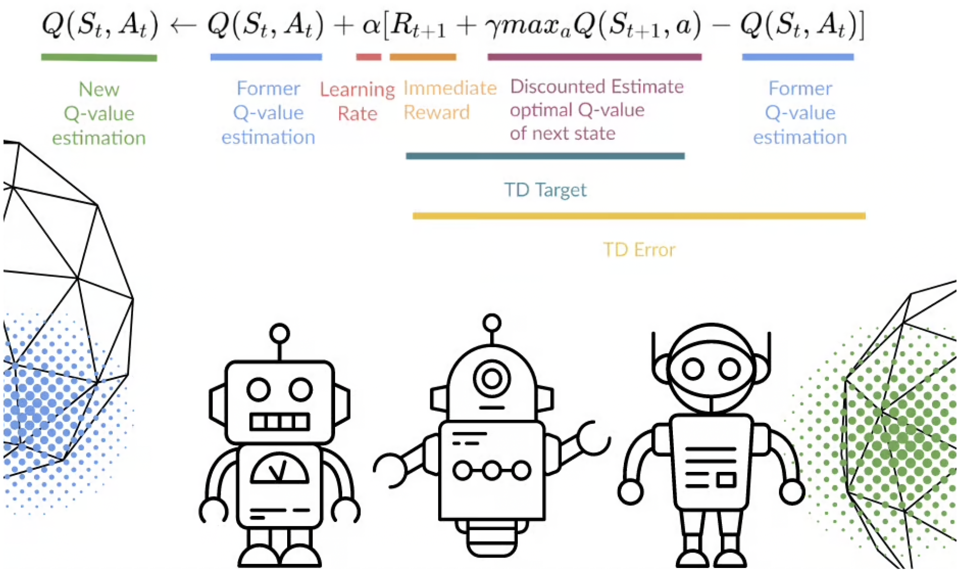
$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right]$$

  we can estimate this expectation by any behavior policy that samples trajectories containing sate-action pair S_t, A_t. this way we can meet an Off-policy method for value based methods.

- **Q-Learning** is a value-based reinforcement learning (RL) method that iteratively updates Q-values to seek the optimal policy. To understand this algorithm, consider sampling (S_t, A_t, R_t, S_{t+1}) trajectories, which allows us to estimate the expected value in Bellman's optimality equation. By iterating through these samples, we can obtain an unbiased estimate of the return. Pay attention to the Q-learning update rule, which iteratively converges to the average return in each step (Q-value) based on these sampled trajectories:

$$q_{t+1}(s,a) = q_t(S_t,A_t) + \alpha_t\left(R_{t+1} + \gamma \max_{a'} q_t(S_{t+1},a') - q_t(S_t,A_t)\right)$$

  note that for each sate, action pair the above equation is rolls over samples for approximating, by averaging action value functions.


$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1},a) - Q(S_t,A_t)]$
New Q-value estimation
Former Q-value estimation
Learning Rate
Immediate Reward
Discounted Estimate optimal Q-value of next state
Former Q-value estimation
TD Target
TD Error

- **Q-learning is really sample inefficient** : In the scenarios where we have a large state space, we need a great number of samples for Q-values to be meaningful, since we are averaging over samples (enough samples!) for Q-values.

- **ϵ-greedy** the importance of sampling actions by ϵ-greedy policy is that we can explore more promising states for achieving better rewards or discovering some strategies during this process. Although running a greedy policy leaves some promising actions undiscovered, in some sense it's really important to let the agent converge to the optimal policy as soon as possible and the speed of convergence important so with a probability the agent acts random and otherwise it acts greedy to rapidly find great rewards.

$$a = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases}$$

- **Function Approximation** instead of storing values in a table (e.g., a Q-table), a neural network is used to approximate the action-value function. The network takes the state as input and outputs the estimated action-value function if actions space is continuous or inputs the action and state to approximate it and helps to generalize over mostly identical states.

- **Training Q-Network** : we need the target Q-values to minimize the loss function using **Stochastic Gradient decent**, there are 2 options : 1) using Monte-carlo estimation for target values which have high variance and is not preferred, since the gradient might have a great variance and makes convergence issues. 2) TD target which is better since it has lower variance w.r.t Monte-carlo. DQN loss functions can be seen below:

$$L(\theta) = \mathbb{E}\left[\left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta^-) - Q(S_t, A_t; \theta)\right)^2\right]$$

- **Note the difference between gradients in such choices of targets** For SARSA, instead use a TD target:

$$\Delta w = \alpha \left(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; w) - \hat{Q}(s_t, a_t; w)\right) \nabla_w \hat{Q}(s_t, a_t; w)$$

For Q-learning:

$$\Delta w = \alpha \left(r + \gamma \max_a \hat{Q}(s_{t+1}, a; w) - \hat{Q}(s_t, a_t; w)\right) \nabla_w \hat{Q}(s_t, a_t; w)$$

- **2 Major problems in DQN** :The **Deep Q-Network (DQN)** algorithm faces two major challenges during training:

  1. **Non-IID Data**:
     - **Problem**: The data collected from the environment is not independent and identically distributed (non-IID) because consecutive samples are highly correlated (e.g., sequential states in an episode).
     - **Solution**: **Experience Replay** is used to break the correlation. Transitions (s_t, a_t, r_t, s_{t+1}) are stored in a replay buffer, and mini-batches are sampled randomly from this buffer for training. This ensures that the data used for updates is more IID.

2. **Non-Stationary Targets**:
   - **Problem**: The target Q-values (used to compute the TD error) are non-stationary because the same network parameters θ are used to compute both the current Q-values and the target Q-values. This can lead to instability during training.
   - **Solution**: A **Target Network** is introduced. This is a separate network with parameters θ^- that are periodically copied from the main Q-network. The target network is used to compute the target Q-values, providing more stable targets and improving convergence.

Summary:

- **Non-IID Data**: Solved by **Experience Replay**.
- **Non-Stationary Targets**: Solved by **Target Network**.

These two innovations were key to stabilizing and improving the performance of DQN in deep reinforcement learning.

- Can treat the target as a constant scalar, but the weights will get updated on the next round, changing the target value!
- **DQN algorithm**:

Algorithm 1 DQN Algorithm

```
1: Input $C, \alpha, D = \{\}$. Initialize $w, w^- = w, t = 0$
2: Get initial state $s_0$
3: loop
4:     Sample action $a_t$ given $\epsilon$-greedy policy for current $Q(s_t, x; w)$
5:     Observe reward $r_t$ and next state $s_{t+1}$
6:     Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $D$
7:     Sample random minibatch of tuples $(s_i, a_i, r_i, s_{i+1})$ from $D$
8:     for each tuple $(s_i, a_i, r_i, s_{i+1})$ in minibatch do
9:         if episode terminated at step $i + 1$ then
10:            $y_i = r_i$
11:        else
12:            $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; w^-)$
13:        end if
14:        Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; w))^2$ for parameters $w$:
               $\Delta w = \alpha(y_i - \hat{Q}(s_i, a_i; w))\nabla_w \hat{Q}(s_i, a_i; w)$
15:    end for
16:    $t = t + 1$
17:    if mod$(t, C) == 0$ then
18:        $w^- \leftarrow w$
19:    end if
20: end loop
```

Deep Reinforcement Learning (Sp25)

Instructor: Dr. Mohammad Hossein Rohban

Summary of Lecture 5: Function Approximation & Policy Gradient methods

Summarized By: Benyamin Naderi

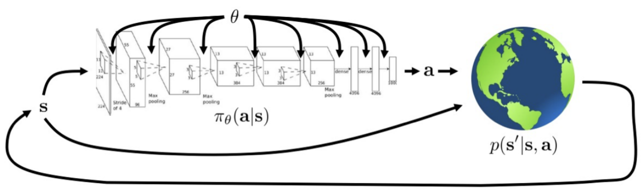
- **DQN vs Q-Learning** : DQN improves upon traditional Q-learning by using a neural network to approximate the Q-function, enabling it to handle high-dimensional state spaces like images. It stabilizes training with experience replay, which breaks correlations in data, and a target value estimation that uses fixed target for every C mini-batch and updates the Q-Network weights, which reduces instability by providing consistent target Q-values. These innovations make DQN more scalable and stable for complex environments.

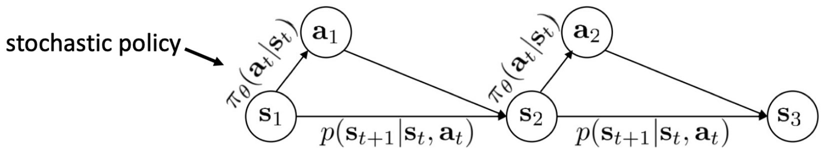
- **DQN results in Atari game** make sure you can interpret these results :

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

- **Value-Based RL:** Focuses on learning value functions such as $Q(s, a)$ to evaluate the quality of actions. The policy is derived implicitly by selecting actions that maximize the value function.

- **Policy Gradient:** Instead of approximating Q-values that optimizes the policy, directly learn the optimal policy using policy network. In this case we need transition probabilities to access the objective gradients:


θ
s
a
$\pi_\theta(\mathbf{a}|\mathbf{s})$
$p(\mathbf{s}'|\mathbf{s},\mathbf{a})$



- **Reinforce Algorithm:**

  The REINFORCE algorithm is a policy gradient method that optimizes the policy directly by maximizing the expected return. The objective is to maximize the expected reward:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} r_t\right]$$

  where $\tau = (s_0, a_0, s_1, a_1, \dots, s_T, a_T)$ is a trajectory sampled from the policy π_θ, and r_t is the reward at time step t.

To optimize the policy, we need to compute the gradient of the objective function $J(\theta)$ with respect to the policy parameters θ. Using the log trick, the gradient can be derived as follows:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} r_t\right]$$

Using the likelihood ratio trick, we can rewrite the gradient as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\left(\sum_{t=0}^{T} r_t\right) \nabla_\theta \log \pi_\theta(\tau)\right]$$

The trajectory probability $\pi_\theta(\tau)$ can be expressed as:

$$\pi_\theta(\tau) = p(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Taking the log of both sides, we get:

$$\log \pi_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)$$

Since $p(s_0)$ and $p(s_{t+1}|s_t, a_t)$ are not dependent on θ, their gradients are zero. Therefore, the gradient of the log-probability of the trajectory simplifies to:

$$\nabla_\theta \log \pi_\theta(\tau) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Substituting this back into the gradient expression, we get:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\left(\sum_{t=0}^{T} r_t\right)\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right)\right]$$

In practice, the expectation is approximated using Monte Carlo sampling. We sample N trajectories $\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(N)}$ from the policy π_θ, and compute the gradient as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N}\left[\left(\sum_{t=0}^{T} r_t^{(i)}\right)\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\right)\right]$$

where $r_t^{(i)}$ and $a_t^{(i)}$ are the reward and action at time step t in the i-th trajectory.