Statistical Techniques in Robotics (16-831, S22) Lecture #20 (Monday, April 4)

Function Approximators for Control, Policy Gradient Methods

Lecturer: Kris Kitani *Scribes: Young Woo Kim, Dakshit Agrawal*

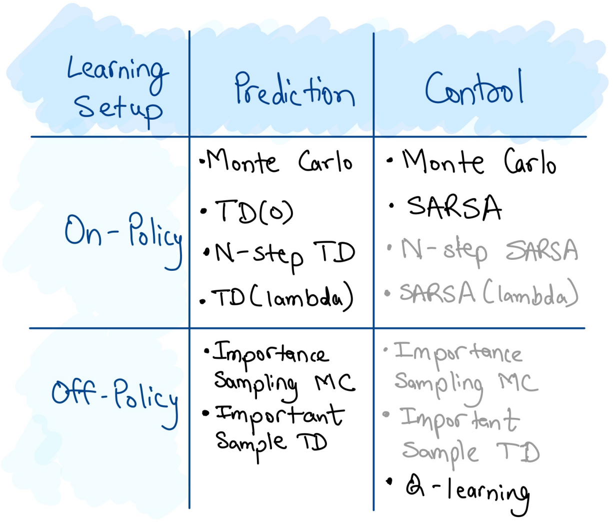
1 Review

In the past few lectures, we have learned many RL approaches (see Fig. 1) for environments where we don't have access to the reward function $R(s^{(t)}, a^{(t)})$ and the transition function $p(s^{(t+1)}|s^{(t)}, a^{(t)})$, also known as model-free environments. The general idea is to sample $(s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)})$ from the environment using the behavior policy $\mu(a^{(t)}|s^{(t)})$. These approaches can be used for:

- **Prediction** of the expected total rewards when at a state $s^{(t)}$ (value function $V(s^{(t)})$) or when at a state $s^{(t)}$ and taking an action $a^{(t)}$ (state-value function $Q(s^{(t)}, a^{(t)})$).
- **Control** by learning the target policy $\pi(a^{(t)}|s^{(t)})$ to maximize rewards in the model-free environment.

The approaches we have learned can alternatively be classified into the following types:

- **On-Policy:** the behavior policy μ is the same as the target policy π, i.e., we sample from the environment using the same policy for which we are estimating the value function.
- **Off-Policy:** the behavior policy μ is different from the target policy π, i.e., we use someone else's experience to sample from the environment.



Learning Setup	Prediction	Contool
On-Policy	•Monte Carlo •TD(0) •N-step TD •TD(lambda)	•Monte Carlo •SARSA •N-step SARSA •SARSA (lambda)
Off-Policy	•Importance Sampling MC •Important Sample TD	•Importance Sampling MC •Important Sample TD •Q-learning

Figure 1: Classification of model-free value-based RL approaches taught in class (bold font)

1.1 Value Function Approximation

While learning the approaches in the previous section, it was assumed we have a finite state and action space, thus leading to a tabular value function. In a scenario where the state and action space is continuous and infinite, we use a function approximator to approximate the value function. These function approximators may be:

- from continuous state space s to value function $V_\theta(s)$.
- from continuous state space s and action space a to action-value function $Q_\theta(s,a)$.
- from continuous state space s to action-value function $Q_\theta(s,a_1)$, $Q_\theta(s,a_2)$, $\ldots$, $Q_\theta(s,a_M)$ for M discrete actions.

The function approximators can be of any architecture such as:

- **Linear:** $V_\theta(s) = \theta^T f(s)$ and $Q_\theta(s,a) = \theta^T f(s,a)$ for handcrafted features $f(s)$ and $f(s,a)$.
- **NN:** $V_\theta(s) = NN_\theta(s)$ and $Q_\theta(s,a) = NN_\theta(s,a)$ for a neural network NN.
- **DNN:** $V_\theta(s) = DNN_\theta(s)$ and $Q_\theta(s,a) = DNN_\theta(s,a)$ for a deep neural network DNN.

1.2 Value Function Approximation for Prediction

The objective function to learn a function approximator for the value function is:

$$\hat{\theta} = \arg\min_\theta \mathbb{E}_p \left[(V(s) - V_\theta(s))^2 \right] \tag{1}$$

The objective function can be optimized using online gradient descent where:

$$\nabla_\theta J(\theta) \approx (V(s) - V_\theta(s)) \frac{\partial}{\partial \theta} V_\theta(s) \tag{2}$$

where $V_\theta(s)$ is the function approximator and $V(s)$ is the true value function. Algorithm 1 highlights the steps of value function approximation for prediction. The true value function ($G^{(t)}$) can be obtained using any of the prediction methods in Fig. 1.

Algorithm 1 Approximate-Prediction(π, α, V_θ)

```
for $e = 1, \ldots, E$ do
    $\{s^{(t)}, a^{(t)}, r^{(t)}\}_{t=0}^{T} \sim \mathcal{E}|\pi$
    for $t = 0, \ldots, T$ do
        $\theta = \theta + \alpha \left( G^{(t)} - V_\theta(s^{(t)}) \right) \frac{\partial}{\partial \theta} V_\theta(s^{(t)})$
    end for
end for
return $V_\theta$
```

2 Summary

In the class, we learned various algorithms for control in continuous spaces (Sec. 2.1). We then read about the basic paper introducing deep Q-learning (Sec. 2.2). We finally explored policy-based RL approaches (Sec. 2.3).

2.1 Function Approximation for Control

The methods learned for control in Fig. 1 can be reused for the continuous state space by doing the following:

- Replace the value function parameter update with the Q function parameter update (use a function approximator in the process).
- Do probabilistic behavior sampling instead of deterministic sampling since continuous state spaces require sampling. More specifically, select a random action with a probability of ϵ and select an action greedily with a probability of $1 - \epsilon$.

2.1.1 Monte-Carlo Control for Continuous Spaces

Algorithm 2 explains the tabular version of Monte-Carlo control.

Algorithm 2 MC-Control-Tabular(π, ϵ, α)

```
1: for $e = 0, \ldots, E$ do
2:   $\{s^{(t)}, a^{(t)}, r^{(t)}\}_{t=0}^{T} \sim \mathcal{E}|\pi$
3:   for $t = 0, \ldots, T$ do
4:     $G^{(t)} = \sum_{i=t}^{T} r^{(i)}$
5:     $Q(a^{(t)}, s^{(t)}) = Q(a^{(t)}, s^{(t)}) + \alpha \left[G^{(t)} - Q(a^{(t)}, s^{(t)})\right]$
6:     $\pi(a|s) = \frac{\epsilon}{|\mathcal{A}|} + \mathbf{1}\left[a = \arg\max_{a} Q(a, s^{(t)})\right](1 - \epsilon) \;\; \forall a, s$
7:   end for
8: end for
9: return $\pi$
```

Line 5 in Algorithm 2 is replaced with the function approximator update equation (line 6 in Algorithm 3) and line 6 in Algorithm 2 is replaced with a probabilistic behavior sampling method (line 3 in Algorithm 3) to obtain the Monte-Carlo Control method for continuous action spaces.

Algorithm 3 MC-Control-Continuous(π, ϵ, α)

```
for $e = 0, \ldots, E$ do
    $\pi \triangleq (1 - \epsilon)\pi_{greedy} + (\epsilon)\pi_{uniform}$
    $\{s^{(t)}, a^{(t)}, r^{(t)}\}_{t=0}^{T} \sim \mathcal{E}|\pi$
    for $t = 0, \ldots, T$ do
        $G^{(t)} = \sum_{i=t}^{T} r^{(i)}$
        $\theta = \theta + \alpha \left(G^{(t)} - Q_\theta(a^{(t)}, s^{(t)})\right) \frac{\partial}{\partial\theta} Q_\theta(a^{(t)}, s^{(t)})$
    end for
end for
return $\pi$
```

2.1.2 SARSA Control for Continuous Spaces

Algorithm 4 explains the tabular version of SARSA control.

Algorithm 4 SARSA-Control-Tabular(π, ϵ, α)

```
for $e = 0, \ldots, E$ do
    $\{s^{(0)}, a^{(0)}\} \sim \mathcal{E}|\pi$
    for $t = 0, \ldots, T$ do
        $\{s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)}, a^{(t+1)}\} \sim \mathcal{E}|\pi, s^{(t)}, a^{(t)}$
        $G^{(t)} = r^{(t)} + \gamma Q(a^{(t+1)}, s^{(t+1)})$
        $Q(a^{(t)}, s^{(t)}) = Q(a^{(t)}, s^{(t)}) + \alpha \left[G^{(t)} - Q(a^{(t)}, s^{(t)})\right]$
        $\pi(a|s) = \frac{\epsilon}{|\mathcal{A}|} + \mathbf{1}\left[a = \arg\max_a Q(a, s^{(t)})\right](1 - \epsilon) \;\; \forall a, s$
    end for
end for
return $\pi$
```

Algorithm 5 SARSA-Control-Continuous(π, ϵ, α)

```
for $e = 0, \ldots, E$ do
    $\{s^{(0)}, a^{(0)}\} \sim \mathcal{E}|\pi$
    for $t = 0, \ldots, T$ do
        $\pi \triangleq (1 - \epsilon)\pi_{greedy} + (\epsilon)\pi_{uniform}$
        $\{s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)}, a^{(t+1)}\} \sim \mathcal{E}|\pi, s^{(t)}, a^{(t)}$
        $G^{(t)} = r^{(t)} + \gamma Q_\theta(a^{(t+1)}, s^{(t+1)})$
        $\theta = \theta + \alpha \left(G^{(t)} - Q_\theta(a^{(t)}, s^{(t)})\right) \frac{\partial}{\partial\theta} Q_\theta(a^{(t)}, s^{(t)})$
    end for
end for
return $\pi$
```

Lines 5,6 in Algorithm 4 are replaced with the function approximator update equation (lines 6,7 in Algorithm 5) and line 7 in Algorithm 4 is replaced with a probabilistic behavior sampling method (line 4 in Algorithm 5) to obtain the SARSA Control method for continuous action spaces.

2.1.3 Q-learning Control for Continuous Spaces

Algorithm 6 explains the tabular version of Q-learning control.

Algorithm 6 Q-learning-Control-Tabular(μ, ϵ)

```
 1: for $e = 0, \ldots, E$ do
 2:   $s^{(0)} \sim \mathcal{E}|\mu$
 3:   for $t = 0, \ldots, T$ do
 4:     $\{s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)}\} \sim \mathcal{E}|\mu, s^{(t)}$
 5:     $a^* = \pi(s^{(t+1)}) \triangleq \arg\max_a Q(a, s^{(t+1)})$
 6:     $G^{(t)} = r^{(t)} + \gamma Q^\pi(a^*, s^{(t+1)})$
 7:     $Q^\pi(a^{(t)}, s^{(t)}) = Q^\pi(a^{(t)}, s^{(t)}) + \alpha \left[G^{(t)} - Q^\pi(a^{(t)}, s^{(t)})\right]$
 8:     $\mu(a|s^{(t+1)}) = \frac{\epsilon}{|\mathcal{A}|} + \mathbf{1}\left[a = \pi(s^{(t+1)})\right](1 - \epsilon)$
 9:   end for
10: end for
11: return $\pi$
```

Lines 6,7 in Algorithm 6 are replaced with the function approximator update equation (lines 7,8 in Algorithm 7) and line 8 in Algorithm 6 is replaced with a probabilistic behavior sampling method (line 4 in Algorithm 7) to obtain the Q-learning Control method for continuous action spaces.

Algorithm 7 Q-learning-Control-Continuous(μ, ϵ)

```
 1: for $e = 0, \ldots, E$ do
 2:   $s^{(0)} \sim \mathcal{E}|\mu$
 3:   for $t = 0, \ldots, T$ do
 4:     $\mu \triangleq (1 - \epsilon)\pi_{greedy} + (\epsilon)\pi_{uniform}$
 5:     $\{s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)}\} \sim \mathcal{E}|\mu, s^{(t)}$
 6:     $a^* = \pi_{greedy}(s^{(t+1)}) \triangleq \arg\max_a Q(a, s^{(t+1)})$
 7:     $G^{(t)} = r^{(t)} + \gamma Q_\theta(a^*, s^{(t+1)})$
 8:     $\theta = \theta + \alpha\left(G^{(t)} - Q_\theta(a^{(t)}, s^{(t)})\right)\frac{\partial}{\partial\theta}Q_\theta(a^{(t)}, s^{(t)})$
 9:   end for
10: end for
11: return $\pi$
```

2.2 Deep Q-learning

Minh et. al. [1] successfully used a deep neural network as the function approximator for Q-learning (naming it Deep Q-learning) to play like professional human players in Atari games. The algorithm they used (Algorithm 8) resembles the continuous Q-learning control method learned in the previous section, albeit with a few additional hacks which proved to be the key points for its success.

Algorithm 8 Deep Q-learning with Experience Replay

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
 4: for episode = 1, ..., $M$ do
 5:     Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 6:     for $t = 1, \dots, T$ do
 7:         With probability $\epsilon$ select a random action $a_t$ otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
 8:         Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
 9:         Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10:         Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
11:         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
12:         Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
13:         Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
14:         Every C steps reset $\hat{Q} = Q$
15:     end for
16: end for
17: return $\pi$
```

Here, x is an image of the Atari game and $\phi(s)$ is a deterministic pre-processed feature of this image. The similarities between Algorithm 7 and Algorithm 8 are as follows:

- Lines 1, 2, 3 of Algorithm 7 correspond to lines 4, 5, 6 of Algorithm 8.
- Line 4 of Algorithm 7 corresponds to line 7 of Algorithm 8.
- Line 5 of Algorithm 7 corresponds to lines 8, 9 of Algorithm 8.
- Lines 6, 7 of Algorithm 7 corresponds to line 12 of Algorithm 8.
- Line 8 of Algorithm 7 corresponds to line 13 of Algorithm 8.

The few additional hacks which proved to be the key points for the success of Algorithm 8 are as follows:

- **Replay Buffer:** A memory D of capacity N stores the previous N observed transitions $(\phi_t, a_t, r_t, \phi_{t+1})$. While training, a random minibatch of transitions is sampled from D to break correlation and increase stability of the algorithm. Lines 1, 10, 11 in Algorithm 8 correspond to this modification.
- **Delayed Updates:** Instead of updating the target policy in each time step, it is updated every C steps to increase the stability of the algorithm as well as to ensure split behavior policy and target policy. Line 14 of Algorithm 8 corresponds to this modification.

Codebase for the above algorithm is explained in the Appendix (Sec. 3.1). Regret analysis of RL algorithms is still an ongoing area of research and thus not discussed in class.

2.3 Policy Gradient Methods

2.3.1 Objective of Policy Gradient Methods

In contrast to value-based reinforcement learning, policy gradients aim to optimize the expected return without explicitly estimating a value function. Instead, it learns directly from the policy by maximizing the expected return of a learning policy. It maximizes the following objective:

$$\begin{aligned}\hat{\theta} &= \arg\max_{\theta} \mathbb{E}_{p_\theta(\zeta)}\left[\sum_{t=0}^{T} r^{(t)}\right] \\ &= \arg\max_{\theta} J(\theta)\end{aligned}$$

This expected return term can be written more explicitly as

$$\mathbb{E}_{p_\theta(\zeta)}\left[\sum_{t=0}^{T} r^{(t)}\right] = \int_{\theta} p_\theta(\theta) r(\zeta) d\zeta$$

where $r(\zeta) = \sum_{t=0}^{T} r^{(t)}$
Recall the following notations in the expected value term:

$$\begin{aligned}&p_\theta(\zeta) = p(s^{(0)})\Pi_{t=0}^{T}\pi_\theta(a^{(t)}|s^{(t)})p(s^{(t+1)}|s^{(t)}, a^{(t)}) \\ &\zeta = \{s^{(0)}, a^{(0)}, s^{(1)}, a^{(1)}, ..., s^{(T)}, a^{(T)}\} \\ &r^{(t)} \triangleq r(s^{(t+1)}, a^{(t)}, s^{(t)})\end{aligned}$$

2.3.2 Gradient Descent Update

By taking the linear approximation of this objective with L2 regularization, and then solving for θ which optimizes the Lagrangian, we achieve the gradient descent update for θ:

$$\hat{\theta} = \arg\max_{\theta} \left\{\alpha(J(\theta^{'}) + \langle\theta - \theta^{'}, \nabla_{\theta^{'}} J(\theta^{'})\rangle - \frac{1}{2}||\theta - \theta^{'}||^2)\right\}$$

$$\nabla_\theta\{\alpha(J(\theta^{'}) + \langle \theta - \theta^{'}, \nabla_{\theta^{'}} J(\theta^{'})\rangle - \frac{1}{2}||\theta - \theta^{'}||^2)\} = 0$$
$$\alpha\nabla_{\theta^{'}} J(\theta^{'}) - \theta + \theta^{'} = 0$$
$$\theta \leftarrow \theta^{'} + \alpha\nabla_{\theta^{'}} J(\theta^{'})$$

2.3.3 Policy Gradient Derivation

We now compute the $\nabla_{\theta^{'}} J(\theta^{'})$ term.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int_\zeta p_\theta(\zeta) r(\zeta) d\zeta \\
&= \int_\zeta \nabla_\theta p_\theta(\zeta) r(\zeta) d\zeta \\
&= \int_\zeta p_\theta(\zeta) \frac{\nabla_\theta p_\theta(\zeta)}{p_\theta(\zeta)} r(\zeta) d\zeta \\
&= \int_\zeta p_\theta(\zeta) \nabla_\theta \ln p_\theta(\zeta) r(\zeta) d\zeta
\end{aligned}$$

By expanding the natural log term, we will see that some terms that do not depend on θ can be eliminated from the gradient.

$$\begin{aligned}
\nabla_\theta \ln p_\theta(\theta) &= \nabla_\theta \left[\ln p(s^{(0)}) + \sum_{t=1}^{T} \ln \pi_\theta(a^{(t)}|s^{(t)}) + \ln p(s^{(t+1)}|s^{(t)}, a^{(t)}) \right] \\
&= \nabla_\theta \left[\sum_{t=1}^{T} \ln \pi_\theta(a^{(t)}|s^{(t)}) \right] \\
\nabla_\theta J(\theta) &= \int_\theta p_\theta \nabla_\theta \ln p_\theta(\theta) r(\zeta) d\zeta \\
&= \int_\theta p_\theta \nabla_\theta \left(\sum_{t=1}^{T} \ln \pi_\theta(a^{(t)}|s^{(t)}) \right) r(\zeta) d\zeta \\
&= \mathbb{E}_{p_\theta} \left[\left(\sum_{t=1}^{T} \nabla_\theta \ln \pi_\theta(a^{(t)}|s^{(t)}) \right) \left(\sum_{t=1}^{T} r^{(t)} \right) \right]
\end{aligned}$$

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

3 Appendix

3.1 Example Code for Deep Q-learning

```python
import numpy as np
import random
import gym
import tensorflow as tf
import keras

# Example Python code for training DQN corresponding to lines of Algorithm 8
def TrainDQN():
    # Line 1: Initialize replay buffer D to capacity N
    D = collections.deque(maxlen = N)
    # Line 2: Initialize action value function Q with random weights
    Q = DQN()
    dqn.random_init()
    # Line 3: Initialize target action value function Q_hat with same weights
    Q_hat = DQN()
    Q_hat.weights = Q.weights
    # Line 4: for each episode...
    for i in range(M):
        # Line 5: Initialize sequence s1 and preprocessed sequence phi
        x = env.reset()
        s = [x]
        phi = preprocess(x)
        # Line 6: for each time step...
        for t in range(T):
            # Line 7: select random action or the action that maximizes Q
            q_values = dqn.predict(phi)
            if (random.random() < epsilon):
                a = np.random.choice(np.arange(q_values.shape[0]))
            else:
                a = np.argmax(q_values)
            # Line 8: observe r and image x from action a
            next_x, r, done = env.step(a)
            # Line 9: set s_{t+1} and preprocess phi_{t+1}
            s.extend([a, next_x])
            next_phi = preprocess(next_s)
            # Line 10: store transition in replay memory
            D.append((phi, a, r, next_phi, done))
            # Line 11: Randomly sample minibatch of transitions
            transitions = random.sample(D, batch_size)
            batch_phi = np.array([t[0] for t in transitions])
            batch_a = np.array([t[1] for t in transitions])
            batch_r = np.array([t[2] for t in transitions])
            batch_next_phi = np.array([t[3] for t in transitions])
            batch_done = [t[4] for t in transitions]
            # Line 12: set y_i
            batch_y = batch_r
            for i,done in enumerate(batch_done):
                if not done:
                    batch_y[i] += gamma*np.max(Q_hat.predict(batch_phi[i]))
            # Line 13: gradient descent on squared error with respect to theta
            DQN.backprop((batch_y - Q.predict(batch_phi)[batch_a])**2)
            # Line 14: copy weights every C steps
            if (t % C == 0):
                Q_hat.weights = Q.weights
    # Line 17: return
    return Q
```

Listing 1: Deep Q-learning