ECE 598: Interplay between Control and Machine Learning **Fall 2020**

Lecture 12

Value-Based RL Methods

Lecturer: Bin Hu, Date:10/20/2020

There are two main ways to solve the optimal policy for a given MDP. We can either learn the optimal Q function (value iteration) or iterate on the policy space directly (approximate policy iteration). In this lecture, we will follow the first idea and discuss how to estimate the optimal Q function from data when the model is unknown. We will talk about a few popular value-based RL methods including Q-learning, SARSA, and fitted Q-iteration.

Q-learning can be viewed as the data-driven implementation of value iteration on Q-factors. In TD learning, we try to fit the value function to enforce the left and right sides of the Bellman equation to be roughly equal on observed data. In Q-learning, we try to fit the value function to enforce the left and right sides of the **optimal Bellman equation** to be roughly equal on observed data.

Recall that a MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, P is the transition kernel, R is the reward, and γ is the discount factor. The optimal Bellman equation is given as

$$Q^*(s,a) = \bar{R}^\pi(s,a) + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} Q^\pi(s', a') \tag{12.1}$$

How to solve the above equation when we do not know $P_{ss'}^a$? We can extend the averaging idea in TD learning. Suppose we are using an iterative method and estimate Q^* as Q_k at step k. Suppose we use some behavior policy which can provide sufficient exploration and obtain a sample trajectory $\{(s_k, a_k)\}$ where a_k is generated using the behavior policy. Based on the optimal Bellman equation, $Q^*(s_k, a_k)$ can be estimated in two ways: either $Q_k(s_k, a_k)$ or $r(s_k, a_k) + \gamma \max_{a' \in \mathcal{A}} Q_k(s_{k+1}, a')$. Similar to TD(0), we can try to minimize the difference of these two things, and average them as

$$\begin{aligned} Q_{k+1}(s_k, a_k) &= (1-\varepsilon) Q_k(s_k, a_k) + \varepsilon \left(r(s_k, a_k) + \gamma \max_{a' \in \mathcal{A}} Q_k(s_{k+1}, a') \right) \\ &= Q_k(s_k, a_k) + \varepsilon \left(r(s_k, a_k) + \gamma \max_{a' \in \mathcal{A}} Q_k(s_{k+1}, a') - Q_k(s_k, a_k) \right) \end{aligned}$$

The calculation of $\max_{a' \in \mathcal{A}} Q_k(s_{k+1}, a')$ requires using the information of Q_k and s_{k+1}. The above algorithm is Q-learning. Usually ε is small, and this means that we do not want to make too much change for a given random sample point. To make Q-learning work, the behavior policy has to provide sufficient exploration: all the state-action pairs should be visited infinitely often! This is the key requirement on the behavior policy.

On-policy vs. off-policy. Q-learning adopts a prescribed behavior policy and hence is an off-policy method. No matter how a' is calculated, it is not going to change the sample data at all. In contrast, SARSA is an on-policy method which uses the followings iteration

$$Q_{k+1}(s_k, a_k) = Q_k(s_k, a_k) + \varepsilon \left(r(s_k, a_k) + \gamma Q_k(s_{k+1}, a_{k+1}) - Q_k(s_k, a_k)\right)$$

where a_{k+1} is calculated as $\arg\max_{a' \in \mathcal{A}} Q_k(s_{k+1}, a')$ with $(1 - \varepsilon)$ probability and chosen randomly with ε probability. This is called ε-greedy sampling. The difference here is that the sample data is generated using Q_k. Hence SARSA is an on-policy method. There are also ways to interpolate Q-learning and SARSA. For example, one can update the behavior policy once awhile by using the ε-greedy policy associated with the current Q_k.

Function approximation. Now we talk about the function approximation case. Suppose n is too large and we do not want to learn an n-dimensional vector for Q^*. Instead, we estimate $Q^*(s, a) \approx \theta^\mathsf{T} \phi(s, a)$ where ϕ is the feature vector. Here, after we get a sample trajectory, what shall we do? We can try to minimize $\frac{1}{2}\|\theta^\mathsf{T}\phi(s_k, a_k) - r(s_k, a_k) - \gamma \max_{a'} \theta_k^\mathsf{T} \phi(s_{k+1}, a')\|^2$. However, we do not want to completely solve this problem since we do not want to trust one sample point too much. Instead, we can perform one step gradient descent as

$$\theta_{k+1} = \theta_k + \varepsilon \left(r(s_k, a_k) + \gamma \max_{a'} \theta_k^\mathsf{T} \phi(s_{k+1}, a') - \theta_k^\mathsf{T}\phi(s_k, a_k)\right) \phi(s_k, a_k)$$

For SARSA, we can have

$$\theta_{k+1} = \theta_k + \varepsilon \left(r(s_k, a_k) + \gamma \theta_k^\mathsf{T} \phi(s_{k+1}, a_{k+1}) - \theta_k^\mathsf{T}\phi(s_k, a_k)\right) \phi(s_k, a_k)$$

where a_{k+1} is generated using the ε-greedy policy associated with Q_k. We can also introduce eligibility trace $z_k = \lambda\gamma z_{k-1} + \phi(s_k, a_k)$ and then update θ_k $\theta_{k+1} = \theta_k + \varepsilon(r(s_k, a_k) + \gamma\theta_k^\mathsf{T}\phi(s_{k+1}, a_{k+1}) - \theta_k^\mathsf{T}\phi(s_k, a_k))z_k$. This SARSA($\lambda$) method uses all the past information to update θ_k.

Online vs. Off-line: Fitted Q-iteration. Q-learning and SARSA are online methods. When a new data point (s_k, a_k) is observed, the weight θ_k is updated and then the data point is thrown away. How to make more efficient use of data? If all the data are available, then we can use off-line methods. For example, when the linear approximation is used, we can apply the least square method to fit θ directly. This method is the fitted Q-iteration. Now the data collection is decoupled from the algorithm iteration. Suppose $\{(s_k, a_k)\}$ has been sampled. At every step l, we solve the following finite-sum optimization

$$\theta_{l+1} = \arg\min_\theta \sum_k \frac{1}{2}\|\theta^\mathsf{T}\phi(s_k, a_k) - r(s_k, a_k) - \gamma \max_{a'} \theta_l^\mathsf{T}\phi(s_{k+1}, a')\|^2$$

For simplicity, we denote $y_k^{(l)} = r(s_k, a_k) + \gamma \max_{a'} \theta_l^\mathsf{T} \phi(s_{k+1}, a')$. Then at every l, we are exactly solving an least square problem and try to minimize the finite-sum function

$$\sum_k \frac{1}{2} \|\theta^\mathsf{T} \phi(s_k, a_k) - y_k^{(l)}\|^2.$$

This problem can be efficiently solved by SGD/ADAM or even in closed-form (by inverting matrix). The above method is different from LSTD-Q which is used to evaluate a value function for a given policy. Since the Bellman equation for Q-factors is still linear, hence one only needs to solve one least square problem to fit the Q-function for a given policy. Here, to find the optimal Q function, the nonlinear maximization is involved and we needs to apply the lease square method in a recursive manner.

Neural network as the function approximators. For practical problems, typically we will use neural networks to approximate the Q function, i.e. $Q^*(s,a) = W^{(1)} \sigma(W^{(0)} \phi(s,a))$. For simplicity, we augment all the weights as θ, and use the notation $V^\pi(s,a) = f_\theta(s,a)$. Again, the goal here is to find θ. At every step, we can try to minimize the following function.

$$\frac{1}{2} \|f_\theta(s_k, a_k) - r(s_k, a_k) - \gamma \max_{a'} f_{\theta_k}(s_{k+1}, a')\|^2$$

Again, we do not want to trust one data point too much and hence just do one step gradient descent as

$$\theta_{k+1} = \theta_k + \varepsilon \left(r(s_k, a_k) + \gamma \max_{a'} f_{\theta_k}(s_{k+1}, a') - f_{\theta_k}(s_k, a_k) \right) \nabla_\theta f_{\theta_k}(s_k, a_k)$$

We may try similar idea for SARSA. Unfortunately, for deep neural networks, this type of implementations does not work. Two tricks are needed, i.e. the experience replay and the frozen target. See the original paper by Mnih et al. for more details on the development of DQN.