DS 102 Data, Inference, and Decisions **Spring 2020**

Lecture 24: Reinforcement Learning

Lecturer: Jacob Steinhardt

Last time, reviewed dynamic programming and Markov Decision Processes, and then saw how to combine these two ideas into the Value Iteration algorithm. Today, we will discuss Value Iteration in more details, as well as another useful variant of the algorithm called Q-iteration. We will then see how to improve these approaches to incorporate learning from data and handle large state spaces. Finally, having discussed all of the key conceptual ideas underlying reinforcement learning (RL), we will briefly discuss two more advanced approaches: temporal difference learning and policy gradient.

24.1 Recap: Value Iteration

Recall that, for Value Iteration, we have a Markov Decision Process (MDP) with transition probabilities $\mathbb{P}(s' \mid a, s)$ (note that in some places in the literature, these transition probabilities are instead denotes as $T(s', a, s)$), rewards $R(s', a, s)$, and possibly a discount parameter γ. Our goal is to compute the value function $V(s)$ which gives the expected reward starting from state s under the optimal policy. We can define V recursively,

$$V(s) = \max_{a \in A} \sum_{s' \in S} \mathbb{P}(s' \mid a, s)(R(s', a, s) + \gamma V(s')),$$

but to avoid computational issues with this circular definition, we also add a time component, $V_t(s)$, where $V_0(s) = 0$ and

$$V_{t+1}(s) = \max_{a \in A} \sum_{s' \in S} \mathbb{P}(s' \mid a, s)(R(s', a, s) + \gamma V_t(s')).$$

24.2 Q-iteration

Recall that, in addition to the value function V, we defined the Q function where $Q(s, a)$ is the expected reward if we start from state s taking action a, and then follow the optimal policy thereafter. Compared to V, Q is useful since it helps us understand which actions are optimal. However, the value and Q functions are very closely related:

$$V(s) = \max_{a \in A} Q(s, a),$$

and

$$\begin{aligned} Q(s,a) &= \sum_{s' \in S} \mathbb{P}(s' \mid a, s)(R(s', a, s) + \gamma V(s')) \\ &= \sum_{s' \in S} \mathbb{P}(s' \mid a, s) \left(R(s', a, s) + \gamma \max_{a' \in A} Q(s', a') \right). \end{aligned}$$

We can approach computing Q the same way we computed V in the previous section, namely by adding a time component and using dynamic programming: $Q_0(s, a) = 0$ and

$$Q_{t+1}(s,a) = \sum_{s' \in S} \mathbb{P}(s' \mid a, s)(R(s', a, s) + \gamma \max_{a' \in A} Q_t(s', a')).$$

24.3 Q-Learning

Value and Q-Iteration both assume that the MDP is fully known. What if the transition dynamics $\mathbb{P}(s' \mid a, s)$ are unknown? It turns out that, even when the transition probabilities are unknown, we can still estimate Q from data. This is called Q-learning.

Suppose we have data about several different trajectories, where a trajectory is a sequence

$$s_0, a_0, s_1, a_1, \ldots, s_i, a_i, \ldots$$

that comes from some policy $\pi : S \to A$. After each state-action pair (s_t, a_t) in a trajectory, we can update our estimate of Q:

$$Q(s_t, a_t) = (1 - \alpha) Q_{old}(s_t, a_t) + \alpha(R(s_{t+1}, a_t, s_t) + \gamma \max_{a' \in A} Q_{old}(s_{t+1}, a')).$$

This update is a weighted average of our previous estimate of the Q value and something looks similar to the update we used in Q-iteration. In fact, it turns out that the right-hand term of this Q-learning update is equal to the Q-iteration update in expectation:

$$\mathbb{E}_{s_{t+1}}[R(s_{t+1}, a_t, s_t) + \gamma \max_{a' \in A} Q_{old}(s_{t+1}, a')] = \sum_{s' \in S} P(s' | a_t, s_t)(R(s', a_t, s_t) + \gamma \max_{a' \in A} Q_{old}(s', a')).$$

Thus, our Q-learning update slowly averages in new estimates of the Q-function (by was of "stochastic" Q-iteration updates), while the weighted average with our previous estimate of the Q value helps stabilize our estimates.

There are convergence theorems which specify conditions under which this Q-learning algorithm will converge to the true Q-value function for the optimal policy. These theorems have two important requirements. First, they typically require that $\alpha \to 0$ over time. In practice, however, it usually works to fix α to a small value. Second, both in theory and in practice we need to make sure to explore the state space enough. There is an analogy to the multi-armed bandits setting from Lectures 17 and 21 where we needed to visit all of the states sufficiently often. One way to accomplish this is to explicitly define an exploration policy that does a good job of visiting all of

the states, but it can be challenging to design such a policy by hand if there are some states that can only be reached by very specific actions/trajectories. An alternative approach is to follow the induced policy from the current estimate of $Q(s,a)$, which means taking the action that looks the best given the current estimate of the Q function. However, this approach can get stuck and end up not exploring enough; it is better to make sure that some fraction of the time we take a random action. It often also helps to initialize $Q_0(s,a)$ to some optimistic (i.e. very large) value, which is similar in spirit to the UCB algorithm in that exploring states which have not yet been visited very often is incentivized.

24.4 Function Approximation

Q-learning allowed us to improve upon the Q-Iteration algorithm by incorporating learning, but both of these approaches will do poorly at handling large state spaces. In particular, each Q-learning update only updates $Q(s,a)$ for a single state-action pair, so if we have a very large number of states we will not be able to sufficiently explore them all. Intuitively, we would like to somehow update $Q(s,a)$ for all similar state-action pairs at once. We accomplish this by using function approximation: we parameterize Q as $Q_\theta(s,a)$. For example, we often model Q as a neural network, and θ are the parameters of the network.

Recall our Q-learning update,

$$\begin{aligned} Q(s_t,a_t) &= (1-\alpha)Q_{old}(s_t,a_t) + \alpha(R(s_{t+1},a_t,s_t) + \max_{a'\in A} Q_{old}(s_{t+1},a')) \\ &= Q_{old}(s_t,a_t) + \alpha\left(R(s_{t+1},a_t,s_t) + \max_{a'\in A} Q_{old}(s_{t+1},a') - Q_{old}(s_t,a_t)\right), \end{aligned}$$

where the second form of the update emphasizes the similarity to a stochastic gradient descent update with step size α. Instead of updating Q directly as in Q-learning, our new update will update θ:

$$\theta = \theta_{old} + \alpha\left(R(s_{t+1},a_t,s_t) + \max_{a'\in A} Q_{\theta_{old}}(s_{t+1},a') - Q_{\theta_{old}}(s_t,a_t)\right)\nabla_\theta Q_{\theta_{old}}(s_t,a_t).$$

This function approximation approach is an adaptation of Q-learning which is much better able to handle large state spaces.

One (not entirely rigorous) way to intuit the form of the update rule for θ is to imagine that we are preforming a prediction task where we would like to predict the Q value and we are using squared-error as our loss:

$$\left(R(s_{t+1},a_t,s_t) + \max_{a'\in A} Q_{\theta_{old}}(s_{t+1},a') - Q_{\theta_{old}}(s_t,a_t)\right)^2.$$

If we imagine that we are only updating the θ_{old} of the last term $Q_{\theta_{old}}(s_t,a_t)$ and take a gradient, we get

$$-2\left(R(s_{t+1},a_t,s_t) + \max_{a'\in A} Q_{\theta_{old}}(s_{t+1},a') - Q_{\theta_{old}}(s_t,a_t)\right)\nabla_\theta Q_{\theta_{old}}(s_t,a_t),$$

so that taking one gradient step would be equivalent to using our update rule for θ with $\alpha = 2$.

24.5 Temporal Difference Learning

It is often the case that we only see a reward at the very end of a trajectory. For example, if we want to use RL to play chess or Go, we do not see rewards until the end of the game when we either win or lose. This leads to a credit assignment problem wherein we have to determine which action we took along the way was really responsible for the reward obtained at the end of the trajectory. In regular Q-learning, this will slow down convergence immensely — we need T updates for a trajectory of length T to even get away from our initialization Q_0. Temporal Different Learning (TD(λ)) is a more advanced RL approach which helps deal with these situations by propagating rewards backwards retrospectively to all the previous states in the trajectory in some exponentially decaying way (according to the parameter λ) so that each update for a trajectory is more efficient. This approach was most famously used for TD-Gammon, an RL algorithm that was used to create a backgammon playing AI.

24.6 Policy Gradient

Policy gradient is an alternative way to approach RL problems. In Q-learning, we tried to build some approximate Q function, Q_θ, and then optimize θ to get good Q values. Policy gradient takes the alternative approach of learning a policy $\pi_\theta(a|s)$ that's a probability distribution of the action given the state. The goal is

$$\max_\theta \mathbb{E}_{\pi_\theta}[R(s_0, a_0, a_1) + \gamma R(s_1, a_1, s_2) + \ldots].$$

To maximize this, we want the gradient of this quantity with respect to θ. Taking this gradient is a bit tricky since the θ appears in the distribution with respect to which we are taking the expectation. To handle this, we can use the log-derivative trick, which says

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[R] = \mathbb{E}_{\pi_\theta}[R \nabla_\theta \log \pi_\theta],$$

thereby allowing us to compute a gradient we can use to update the parameters θ.