DS 102 Data, Inference, and Decisions **Spring 2021**

Lecture 23: Reinforcement Learning

Lecturer: Ramesh Sridharan

These notes are from a previous iteration of Data 102: the ideas are the same, but they may contain some additional content that wasn't covered in depth this semester, and may use slightly different notation (e.g., $\mathbb{P}(s' \mid s, a)$ for transition probabilities).

23.1 Dynamic Programming

Dynamic programming is a general tool that makes recursion more efficient by intelligently reusing the answer to previous function calls.

We will build up the idea of dynamic programming by considering a simple form of recursion: the Fibonacci numbers. The sequence of Fibonnaci numbers is defined by:

$$\begin{aligned} &f(0) = 0 \\ &f(1) = 1 \\ &f(n) = f(n-1) + f(n-2) \ \forall\ n > 1 \end{aligned}$$

A naive program to compute this would have the following structure:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

The recursive tree for this function call will explode fairly quickly. For example, what happens if we try to call `fib(50)`?

$$\begin{tikzcd}
& & & \texttt{fib(50)} \arrow[dll] \arrow[drr] & & & \\
& \texttt{fib(49)} \arrow[dl] \arrow[dr] & & & & \texttt{fib(48)} \arrow[dl] \arrow[dr] & \\
\texttt{fib(48)} & & \texttt{fib(47)} & & \texttt{fib(47)} & & \texttt{fib(46)} \\
\vdots & & \vdots & & \vdots & & \vdots
\end{tikzcd}$$

This exponential blow-up in the computation tree is not ideal for processors, memory usage, *et cetera*. However, one key observation we can make is that the same computation gets made several times; for example, `fib(48)` gets computed times in the tree.

Memoization improves upon this naive recursive approach by caching and reusing the results of previous function calls. The memoized version of our Fibonacci function would look something like this:

```
memo = dict()
def fib(n):
    if n in mem.keys():
        return memo[n]
    if n == 0:
        memo[n] = 0
    if n == 1:
        memo[n] = 1
    else:
        memo[n] = fib(n − 1) + fib(n − 2)
    return memo[n]
```

One nice this about the memoization approach is that it not too difficult to adapt a naive recursive approach to a memoized approach. In fact, in Python it is particularly easy: there are function decorators (e.g. `@memoize`) that will automatically memoize a recursive function. The memoized version of a recursive function will blow-up only linearly rather than exponentially. While this is a big win in terms of the amount of computation performed, memoization can still be problematic in cases where the cache is large and the look-up step is very slow.

Dynamic programming is another improvement on top of memoization. Dynamig programming "unrolls" the recursion into a for-loop by performing the computations in such a way that the results of any previous computations required for the current step of the loop are always known. This is preferable over naive recursions since it reuses computation, and over memoization since it bypasses the need for performing look-ups. For our Fibonnaci example, a dynamic programming approach would have the following structure:

```
fib = zeros(shape=(n+1, 1))
fib[0] = 0
fib[1] = 1
for n in range(2, n+1):
    fib[n] = fib[n−1] + fib[n−2]
```

Dynamic programming is fast compared to other approaches to recursion because it reduces the problem to a loop. However, in order to take a dynamic programming approach, the computation must be laid out in some nice way. For some problems, determining the right order to do the computations is sufficiently complicated that one might prefer the ease of a memoization approach even though its use of a cache is slower.

We will end this section with a slightly more complicated dynamic programming example that build towards MDPs and RL. For this example, suppose we are driving a car along a linear route and are trying to pick the optimal set of gas stations at which to stop. We start at location 0 and our goal is to reach location N (which is N units to our right) without running out of gas. It costs 1 unit of gas to move 1 unit to the right, and each gas machine at location i has g_i units of gas available for purchase for a cost of c_i dollars per unit. How much gas should we buy at each location to minimize the total cost?

We can frame this as a dynamic programming problem. First, we should consider what state we need to keep track of to make a decision at each step. In this case, the two factors that affect our decision are our current location and how much gas is currently in our tank. If we were to solve this using recursion, our function `f(loc, gas)` should return the minimum cost to get to the end, given the current state `(loc, gas)`. This is an example of a **cost-to-go** function, which we will see again in several of the upcoming MDP and RL examples. The pseudo-code for a naive recursive approach would have the following structure:

```
def f(loc, gas):
    # Base case: we already reached the goal
    if loc == N:
        return 0
    # Base case: we ran out of gas
    if gas < 0:
        return math.inf

    # Option 1: Buy 1 unit of gas and stay where we are
    cost1 = f(loc, gas + 1) + price[loc]
    # Option 2: Go forward 1 unit
    cost2 = f(loc + 1, gas - 1)

    return min(cost1, cost2)
```

Note that if we start at the end and work our way backwards, we will always either already know the answer (e.g. in the case of `loc == N`) or have previously done the computations required to figure out the answer. Thus, a dynamic programming approach would have the following basic structure, with one or two additional edge cases:

```
f = zeros(shape=(N + 1, N + 1))
for loc in range(N, 0, -1):
    for gas in range(N, 0, -1):
        cost1 = f[loc, gas + 1] + price[loc]
        cost2 = f[loc + 1, gas - 1]
        f[loc, gas] = min(cost1, cost2)
```

23.2 Markov Decision Processes

MDPs generalize the dynamic programming idea of working backwards to *stochastic* decision-making processes.

An MDP is defined by a sequence of states $s \in S$, a set of actions $a \in A$, and a transition function $T(s, a, s')$ that gives the probability that the action a from state s leads to the new state s' (e.g. $\mathbb{P}(s' \mid s, a)$, which is also called a model or the dynamics). We usually also have some reward function $R(s, a, s')$, and would like to have a large cumulative reward. For example, the current state s might denote where we currently are in the world, the action a might specify the direction in which we'd like to move, and the transition function would give the probability distribution over what outcome actually happens when we try to move this direction from our current location.

The "Markov" in Markov Decision Process refers to the fact that the transition function only depends on the current state and the current actions, and not on the entire history of the process. One intuition is to think of an MDP like a stochastic version of a search problem where we're trying to make an optimal action at each step based on all the things that could happen in the future (e.g. trying to decide what move to make in a Chess game based on possible reactions to each move), but in an MDP there is randomness involved in the potential future outcomes.

Often, we are interested in understanding what actions we should take at each step of the MDP. A policy is a function $\pi : S \to A$ that specifies an action for each state. For any policy, π, it makes sense to think about what the expected reward is under that policy, or about the distribution over rewards under that policy. An optimal policy, denoted π^*, is the policy that maximizes the expected reward. Typically, we are trying to maximize the expected cumulative reward over all time steps. It is also reasonable, though, to prefer rewards now to rewards later. **Discounting** accounts for this by down-weighting future time steps by some multiplicative factor γ, which can be interpreted as accounting for some chance γ that the process could end at every step. Practically, discounting is often used because it help algorithms converge.

23.3 Solving Markov Decision Processes with Dynamic Programming

There are several quantities of interest when working with MDPs, including:

- The value (or utility) of a state s, denoted $V^*(s)$, is the expected utility when starting in state s and acting optimally
- The value (or utility) of a q-state (s, a), denoted $Q^*(s, a)$, is the expected utility starting out taking action a from state s and thereafter acting optimally

Note that the Q^* function is useful compared to V^* since the value function tells us what happens when we act optimally, but does not help us understand which action to take. If we know the Q^* function, we can determine a policy π^*.

We can define these quantities recursively. For example, we can write the value function as

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

or if we want to do discounting we can incorporate the discounting factor γ,

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

Since this is a recursive formula, we can start thinking about what recursive approaches to solving for the value function would look like. In order to avoid infinite recursion issues due to the randomness in the problem, we introduce a time horizon T:

```
def V(s, t):
    if t == T:
        return 0
    else:
        val = -math.inf
        for a in actions:
            v = sum([P(s' | s, a) (R(s, a, s') + gamma * V(s', t + 1))
                     for s' in states])
            val = max(val, v)
        return val
```

We could use memoization on this recursion, but we could also think about using a dynamic programming approach instead. This might looks something like the following pseudo-code:

```
V = zeros(shape=(S, T+1))
for t in range(T-1, 0, -1):
    for s in states:
        V[s, t] = max([
            sum([P(s' | s, a) (R(s, a, s') + gamma * V[s', t + 1])
                for s' in states])
            for a in actions])
```

This lets us compute the value function over some time horizon. We can improve the efficiency of this approach by noting that we do not actually need to store the full V array; at time t we only ever need to know the value function at time $t+1$. This means that we can just store a small slice of V and run the algorithm for a very long time horizon without using additional memory. This algorithm is known as the **Value Iteration algorithm**.

23.4 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

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

23.5 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.

23.6 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 $\alpha = 2$.

23.7 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.

23.8 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.