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

Lecture 23: Introduction to Reinforcement Learning

Lecturer: Jacob Steinhardt

Today, we will start to think about how to make decisions in a time-series or sequential setting. We will begin with a refresher on recursion and dynamic programming, and discuss Markov Decision Processes (MDPs). MDPs will act as a jumping-off point for Reinforcement Learning (RL); RL combines ideas of MDPs, learning from data, and function approximation. In the next lecture, we will understand how these different pieces fit together.

23.1 Dynamic Programming

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

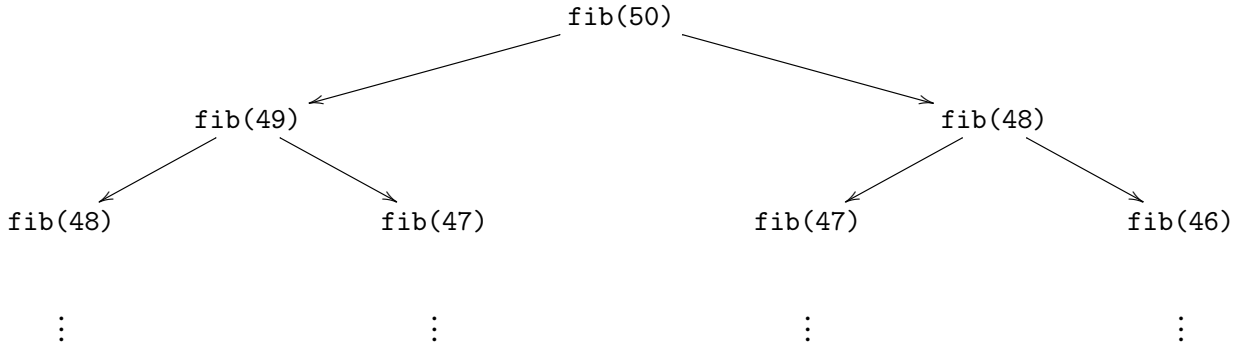
We will build up the idea of dynamic programming by considering a simple form of recursion: the Fibonacci numbers. The sequence of Fibonnaci numbers is defined by:

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

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)`?



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), denotes $Q^*(x, a)$, 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**. In our next discussion, we will see this algorithm in more detail and build up to a fuller picture of RL.