Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.S064 Introduction to Machine Learning

Phase 4: Reinforcement learning (lectures 20-21)

Learning from Feedback

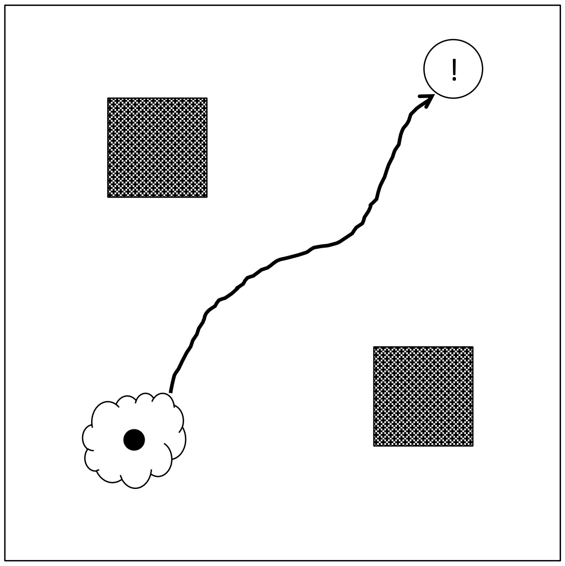

Figure 1: A robot navigation problem.

Let's consider a robot navigation problem (see the Fig 1 for illustration). At each point in time, our robot is located at a certain position on the grid. Our robot also has sensors which can help predict (with some noise) its position within the grid. Our goal is to bring the robot to its desired final destination (e.g., charging station). The robot can from one position to another in small increments. However, we assume that the movements are not deterministic. In other words, with some probability the robot moves to the desired next position, but there is also a chance that it ends up in another nearby location (e.g., applying a bit too much power). Let's assume for simplicity that the states are discrete. That is, the positions correspond to grid points.

At the first glance, this problem may resemble HMMs: we use states y_t to encode the position of the robot, and observations x_t capture the sensor readings. The process is clearly Markov in the sense that the transition to the next state is only determined by the current state.

In the case of HMMs, we represent transitions as $T(i,j) = p(y_{t+1} = j|y_t = i)$. What is missing? This parametrization does not account for the fact that the robot can select

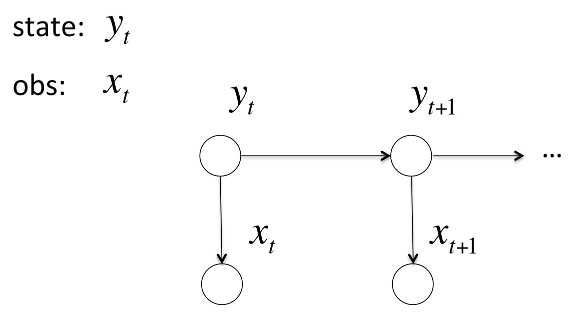

Figure 2: HMM.

actions such as the direction that it wants to move in. We will therefore expand the transition probabilities to incorporate selected actions — $T(i, k, j) = p(y_{t+1} = j | y_t = i, a_t = k)$ specifies the probability of transitioning to j from i after taking action a.

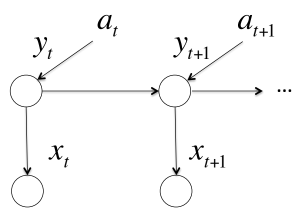

Figure 3: HMM with action.

In contrast to the more realistic setting discussed above, we will make the model simpler here by assuming that the states are directly observable. I.e., at every point, the robot knows its exact location in the greed. Put another way, the observation x_t fully determines y_t, and we will drop x_t as a result.

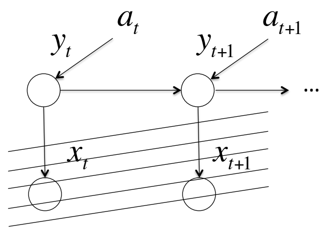

Figure 4: HMM without x_t.

The remaining piece in our robot navigation problem is to specify costs or rewards. Rewards can be associated with states $R(s_t)$ (e.g., the target charging station has a high reward) or they can also take into account the action that brings the robot to the follow-up state $R(s_t, a_t, s_{t+1})$. Here is a simple example of a reward function:

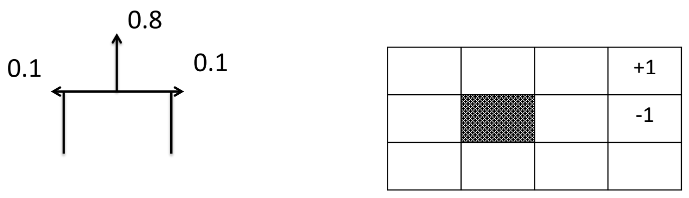

Figure 5: Robot and grid.

$$R_{s_t} = \begin{cases} 1 & \text{if on target} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Utility Function Intuitively, a utility function would be the sum of all the rewards that the robot accumulates. However, this definition may result in acquiring infinite reward (for instance, when the robot loops around). Also, it may be better to acquire high rewards sooner than later. One possible setting is to assume that the robot has a finite horizon: after N steps, the utility value doesn't change at all:

$$U([s_0, s_1, \ldots, s_{N+k}]) = U_n([s_0, s_1, \ldots, s_N]) \ \ \forall k$$

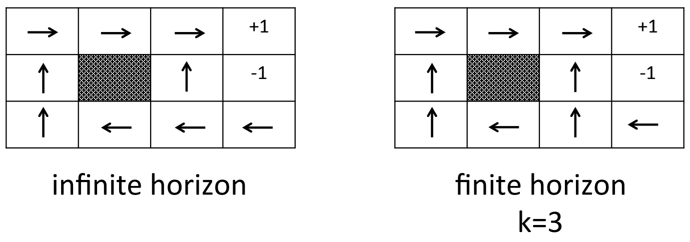

Figure 6: Policy with different horizon.

However, under this assumption the optimal strategy depends on how long the agent has to live (see Fig below). Thus, the optimal action would not only depend on the state that the robot is in, but would also depend on the time that it has left.

An alternative approach is to use so called *discounted rewards*. Even for infinite sequences, this utility function is guaranteed to have a finite value so long as the individual rewards are finite. For $0 \leq \gamma < 1$, we define it as

$$U([s_0, s_1, s_2 \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \tag{2}$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma} \tag{3}$$

This formulations assigns higher utilities to rewards that come early. The discounting also helps us solve for these utilities (important for convergence of the algorithms we will cover later in the lecture).

Another way to think of the discounted utility is by augmenting our state space with an additional *"halt"* state. Once the agent reaches this state, the experience stops. At any point, we will transition to this halt state with probability $1 - \gamma$. Thus the probability that we continue to move on is γ. Larger γ means longer horizon.

Policy A policy π is a function that specifies an action for each state. Our goal is to compute an optimal policy that maximizes the expected utility, i.e., the action that the robot takes in any state is chosen with the idea of maximizing the discounted future rewards. As illustrated in Fig 7, an optimal policy depends heavily on the reward function. In fact, it is the reward function that specifies the goal.

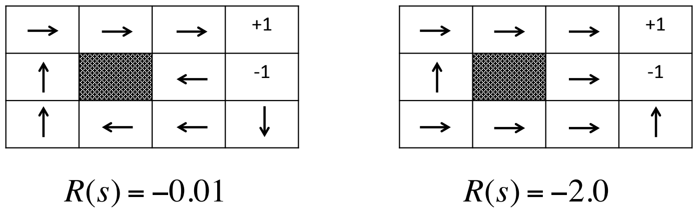

Figure 7: Policy with different rewards.

We will consider here two ways to learn the optimal policy:

- **Markov Decision Process (MDP)**. We assume that reward function and transition probabilities are known and available to the robot. More specifically, we are provided with
  - a set of states S
  - a set of actions A
  - a transition probability function $T(s, a, s') = p(s'|a, s)$
  - a reward function $R(s, a, s')$ (or just $R(s')$)
- **Reinforcement Learning** The reward function and transition probabilities are unknown (except for their form), and the robot only knows
  - a set of states S
  - a set of actions A

Value iteration

Value iteration is an algorithm for finding the (an) optimal policy for a given MDP. The algorithm iteratively estimates the value of each state; in turn, these values are used to compute the optimal policy. We will use the following notations:

- $V^*(s)$ – The value of state s, i.e., the expected utility of starting in state s and acting optimally thereafter.
- $Q^*(s,a)$ – The Q value of state s and action s. It is the expected utility of starting in state s, taking action a and acting optimally thereafter.
- $\pi^*(s)$ –The optimal policy. $\pi^*(s)$ specifies the action we should take in state s. Following policy π^* would, in expectation, result in the highest expected utility.

The above quantities are clearly related:

$$V^*(s) = \max_a Q^*(s,a) = Q^*(s,\pi^*(s)) \tag{4}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{5}$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{6}$$

$$= \sum_{s'} T(s,\pi^*(s),s')[R(s,\pi^*(s),s') + \gamma V^*(s')] \tag{7}$$

For example, in Eq.(5), we evaluate the expected reward of taking action a in state s. We sum over the possible next states s', weighted by the transition probabilities corresponding to the state s and action a. Each summand combines the immediate reward with the expected utility we would get if we started from s' and followed the optimal policy thereafter. The future utility is discounted by γ as discussed above. In other words, we have simple one-step lookahead relationship among the utility values.

Based on these equations, we can recursively estimate $V_k^*(s)$, the optimal value considering next k steps. As $k \to \infty$, the algorithm converges to the optimal values $V^*(s)$.

The Value Iteration Algorithm

- Start with $V_0^*(s) = 0$, for all $s \in S$
- Given V_i^*, calculate the values for all states $s \in S$ (depth $i+1$):

$$V_{i+1}^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i^*(s')]$$

- Repeat until convergence (until $V_{i+1}(s)$ is nearly $V_i(s)$ for all states)

The convergence of this algorithm is guaranteed. We will not show a full proof here but just illustrate the basic idea. Consider a simple MDP with a single state and a single action. In such a case:

$$V_{i+1} = R + \gamma V_i$$

We also know that for the optimal V* the following must hold:

$$V^* = R + \gamma V^*$$

By subtracting these two expressions, we get:

$$(V_{i+1} - V^*) = \gamma(V_i - V^*)$$

Thus, after each iteration, the difference between the estimate and the optimal value decreases by a factor $\gamma < 1$.

Once the values are computed, we can turn them into the optimal policy:

$$Q^*(s,a) = \sum_s T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

As we rely on the Q-values to get the policy, we could alternatively compute these values directly. Here is an algorithm that does exactly that.

The Q-Value Iteration Algorithm

- Start with $Q^*_0(s,a) = 0$ for all $s \in S$, $a \in A$.
- Given $Q^*_i(s,a)$, calculate the q-values for all states (depth $i+1$) and for all actions a:

$$Q^*_{i+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*_i(s',a')]$$

This algorithm has the same convergence guarantees as its value iteration counterpart.

Reinforcement learning

Now, we will consider a set-up where neither reward no transitions are known a priori. Our robot can travel in the grid, moving from one state to another, collecting rewards along the way. The model of the world in unknown to the robot other than the overall Markov structure. The robot could do one of two things. First, it could try to learn the model, the reward and transition probabilities, and then solve the optimal policy using the algorithms for MDPs described above. Another option is to try to learn the Q-values directly.

Model-based learning We first assume that we can collect information about transitions involving any state s and action a. Under this assumption, we can learn T and R through experience, by collecting outcomes for each s and a.

$$T(s,a,s') = \frac{count(s,a,s')}{\sum_{s'} count(s,a,s')}$$
$$R(s,a,s') = \frac{\sum_t R_t(s,a,s')}{count(s,a,s')}$$

where $R_t(s,a,s')$ is the reward we observed when starting in state s, taking action a, and transitioning to s'. If the reward is noisy, observed rewards $R_t()$ may vary from one instance to another. In reality, this naive approach is highly ineffective for any non-trivial state space. The best we can do is randomly explore, taking actions and moving from one state to another. Most likely, we will be unable to reach many parts of the state space in any complex environment. Moreover, the learned model would be quite large as we'd have to store all the states and possible transitions.

Model-free learning Can we learn how to act without learning a full model? Remember:

$$pi^*(s) = \arg\max_a Q^*(s,a)$$

We have shown that Q-values can be learned recursively, assuming we have access to T and R. Since this information is not provided to us, we will consider Q-learning algorithm, a sample based Q-value iteration procedure.

To better understand the difference between model-based and model-free estimation, consider the task of computing the expected value of a function $f(x)$: $E[f(x)] = \sum_x p(x)f(x)$

- **Model-based computation**: First estimate $p(x)$ from samples and then compute expectation:

$$x_i \sim p(x),\ i = 1, \dots, k$$
$$\hat{p}(x) = \frac{count(x)}{k}$$
$$E[f(x)] \approx \sum_x \hat{p}(x) f(x)$$

- **Model-free estimation**: estimate expectation directly from samples

$$x_i \sim p(x),\ i = 1, \dots, k$$
$$E[f(x)] \approx \frac{1}{k}\sum_{i=1}^{k} f(x_i)$$

Now we will apply the model-free learning approach to the estimation of Q-values. Recall,

$$Q^*_{i+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*_i(s',a')]$$

We will repeatedly take one action at a time, and observe the reward and the next state. We can compute:

$$sample_1 : R(s,a,s'_1) + \gamma \max_{a'} Q_i(s'_1, a')$$

$$sample_2 : R(s,a,s'_2) + \gamma \max_{a'} Q_i(s'_2, a')$$

$$\dots$$

$$sample_k : R(s,a,s'_k) + \gamma \max_{a'} Q_i(s'_k, a')$$

Now we can average all the samples, to obtain the Q-value estimate:

$$Q_{i+1}(s,a) = \frac{1}{k} \sum_{l=1}^{k} \left[R(s,a,s'_l) + \gamma \max_{a'} Q_i(s'_l, a') \right]$$

which, for large k, would be very close to the Q-value iteration step. We are almost there. In practice, we only observe the states when we actually move. Therefore, we cannot really collect all these sample at once. Instead, we will update the Q-values after every experience (s, a, s', r), where r is the reward.

Q-learning Algorithm

- Collect a sample: s, a, s', and $R(s,a,s')$.
- Update Q-values, by incorporating the new sample into a running average over samples:

$$\begin{aligned} Q(s,a) &\leftarrow (1-\alpha) Q(s,a) + \alpha \left[R(s,a,s') + \gamma \max_{a'} Q(s',a') \right] && (8) \\ &= Q(s,a) + \alpha \left[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] && (9) \end{aligned}$$

where the learning rate α takes the role of $1/k$ in the earlier sample average example. During the iterations of the algorithm, likely s' will contribute more often to the Q-values estimate. As the algorithm progresses, old estimates fade, making the Q-value more consistent with more recent samples.

You may have noticed that the form of the update closely resembles stochastic gradient decent. In fact, it has the same convergence conditions as the gradient ascent algorithm. Each sample corresponds to (s, a), i.e., being in state s and taking action a. We can assign a separate learning rate for each such case, i.e., $\alpha = \alpha_k(s,a)$, where

k is the number of times that we saw (s, a). Then, in order to ensure convergence, we should have

$$\sum_k \alpha_k(s, a) \rightarrow \infty$$

$$\sum_k \alpha_k^2(s, a) < \infty$$

Exploration/Exploitation Trade-Off In the Q-learning algorithm, we haven't specified how to select an action for a new sample. One option is to do it fully randomly. While this exploration strategy has a potential to cover a wide spectrum of possible actions, most likely it will select plenty of suboptimal actions, and leads to a poor exploration of the relevant (high reward) part of the state space. Another option is to exploit the knowledge we have already obtained during previous iterations. Remember that once we have Q estimates, we can compute a policy. Since our estimates are noisy in the beginning, and the corresponding policy is weak, we wouldn't want to follow this policy completely. To allow for additional exploration, we select a random action every once in a while. Specifically, with prob ε, the action is selected at random and with probability $(1 - \varepsilon)$, we follow the policy induced by the current Q-values. Over time, we can decrease ε, to rely more heavily on the learned policy as it improves based on the Q-learning updates.