DDA4230 Reinforcement learning Model-free policy evaluation

Lecture 13

Lecturer: Baoxiang Wang *Scribe: Baoxiang Wang*

1 Goal of this lecture

Starting from LN13 we investigate model-free algorithms for reinforcement learning. In this lecture we start with methods for policy evaluation which given a policy output its corresponding value function.
Suggested reading: Chapter 5, 6 and 13 of *Reinforcement learning: An introduction*;

2 Model-free policy evaluation

In previous lectures on discrete MDPs, all of the methods demand the assumption that we know both the rewards and probabilities for every transition, up to if the model is known (RL with known model) or is estimated (model-based RL). However, in many cases, such information is not readily available to us, which necessitates model-free algorithms. In this lecture, we will be discussing model-free policy evaluation. That is, we will be given a policy and will attempt to learn the value of that policy without leveraging knowledge of the rewards or transition probabilities.

Policy evaluation is an important foundation of reinforcement learning. In combination with how to improve our policies in the model-free case (to be discussed in the next lecture) we will obtain model-free RL algorithms.

2.1 Monte-Carlo policy evaluation

We now describe our first model-free policy evaluation algorithm which uses a popular computational method called the Monte-Carlo method. We first walk through an example of the Monte-Carlo method outside the context of reinforcement learning, then discuss the method more generally, and finally apply Monte Carlo to reinforcement learning. We emphasize here that this method only works in episodic environments, and we will see why this is as we examine the algorithm more carefully in this section.

Suppose we want to estimate how long the commute from your house to the campus will take today. Suppose we also have access to a commute simulator which models our uncertainty of how bad the traffic will be, the weather, construction delays, and other variables, as well as how these variables interact with each other. One way to estimate the expected commute time is to simulate our commute many times on the simulator and then take an average over the simulated commute times. This is called a Monte-Carlo estimate of our commute time.

In general, we get the Monte-Carlo estimate of some quantity by observing many iterations of how that quantity is generated either in real life or via simulation and then

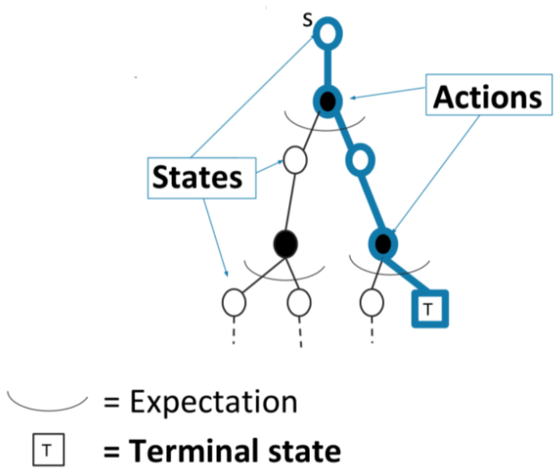

Figure 1: Backup diagram for the Monte-Carlo policy evaluation algorithm.

averaging over the observed quantities. By the law of large numbers, this average converges to the expectation of the quantity.

In the context of reinforcement learning, the quantity we want to estimate is $V^\pi(s)$, which is the average of returns G_t (which equals R_t without n-step truncate or eligibility traces) under policy π starting at state s. We can thus get a Monte-Carlo estimate of $V^\pi(s)$ through three steps:

1. Execute a rollout of policy π until termination many times;
2. Record the returns G_t that we observe when starting at state s;
3. Take an average of the values we get for G_t to estimate $V^\pi(s)$.

The backup diagram for Monte-Carlo policy evaluation can be seen in Figure 1. The new blue line indicates that we sample an entire episode until termination starting at state s.

There are two forms of Monte-Carlo policy evaluation, which are differentiated by whether we take an average over just the first time we visit a state in each rollout or every time we visit the state in each rollout. These are called first-visit Monte-Carlo and every-visit Monte-Carlo policy evaluation, respectively.

More formally, we describe the first-visit Monte-Carlo in Algorithm 1 and the every-visit Monte-Carlo in Algorithm 2.

Note that in the body of the for loop in Algorithms 1 and 2, we can remove vector S and replace the update for $V^\pi(s_{j,t})$ with

$$V^\pi(s_{j,t}) \leftarrow V^\pi(s_{j,t}) + \frac{1}{N(s_{j,t})}\left(G_{j,t} - V^\pi(s_{j,t})\right).$$

This is because the new average is the average of $N(s_{j,t}) - 1$ of the old values $V^\pi(s_{j,t})$ and

Algorithm 1: First-visit Monte-Carlo policy evaluation

```
Input: $h_1, \dots, h_j$
For all states $s$, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
for each episode $h_j$ do
    for $t = 1, \dots, L_j$ do
        if $s_{j,t} \neq s_{j,u}$ for $u < t$ then
            $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
            $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
            $V^\pi(s_{j,t}) \leftarrow S(s_{j,t})/N(s_{j,t})$
return $V^\pi$
```

Algorithm 2: Every-visit Monte-Carlo policy evaluation

```
Input: $h_1, \dots, h_j$
For all states $s$, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
for each episode $h_j$ do
    for $t = 1, \dots, L_j$ do
        $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
        $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
        $V^\pi(s_{j,t}) \leftarrow S(s_{j,t})/N(s_{j,t})$
return $V^\pi$
```

the new return $G_{j,t}$, giving us

$$\frac{V^\pi(s_{j,t}) \cdot (N(s_{j,t}) - 1) + G_{j,t}}{N(s_{j,t})} = V^\pi(s_{j,t}) + \frac{1}{N(s_{j,t})}\left(G_{j,t} - V^\pi(s_{j,t})\right),$$

which is precisely the new form of the update.

Replacing $1/N(s_{j,t})$ with α in this new update gives us the more general incremental Monte-Carlo policy evaluation. Algorithms 3 and 4 detail this procedure in the first-visit and every-visit cases, respectively.

Algorithm 3: Incremental first-visit Monte-Carlo policy evaluation

```
Input: $\alpha, h_1, \dots, h_j$
For all states $s$, $N(s) \leftarrow 0$, $V(s) \leftarrow 0$
for each episode $h_j$ do
    for $t = 1, \dots, terminal$ do
        if $s_{j,t} \neq s_{j,u}$ for $u < t$ then
            $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
            $V^\pi(s_{j,t}) \leftarrow V^\pi(s) + \alpha(G_{j,t} - V^\pi(s))$
return $V^\pi$
```

Setting $\alpha = 1/N(s_{j,t})$ recovers the original Monte-Carlo policy evaluation algorithms given in Algorithms 1 and 2, while setting $\alpha > \frac{1}{N(s)}$ gives a higher weight to newer data,

Algorithm 4: Incremental every-visit Monte-Carlo policy evaluation

```
Input: $\alpha, h_1, \dots, h_j$
For all states $s$, $N(s) \leftarrow 0$, $V(s) \leftarrow 0$
for each episode $h_j$ do
    for $t = 1, \dots, terminal$ do
        $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
        $V^\pi(s_{j,t}) \leftarrow V^\pi(s) + \alpha(G_{j,t} - V^\pi(s))$
return $V^\pi$
```

which can help learning in non-stationary domains. If we are in a truly Markovian-domain, every-visit Monte Carlo will be more data efficient because we update our average return for a state every time we visit the state.

3 Monte-Carlo off-policy evaluation

In the section above, we discussed the case where we are able to obtain many realizations of G_t under the policy π that we want to evaluate. However, in many costly or high stakes situations, we are unable to obtain rollouts of G_t under the policy that we wish to evaluate. For example, we may have data associated with one medical policy, but want to determine the value of a different medical policy. In this section, we describe Monte-Carlo off-policy policy evaluation, which is a method for using data taken from one policy to evaluate a different policy.

3.1 Importance sampling

The key ingredient of off policy evaluation is a method called importance sampling. The goal of importance sampling is to estimate the expected value of a function $f(x)$ when x is drawn from distribution q using only the data $f(x_1), \dots, f(x_n)$, where x_i are drawn from a different distribution p. In summary, given $q(x_i), p(x_i), f(x_i)$ for $1 \leq x_i \leq n$, we would like an estimate for $\mathbb{E}_{x \sim q}[f(x)]$. We can do this via the following approximation, as

$$\begin{aligned}
\mathbb{E}_{x \sim q}[f(x)] &= \int_x q(x) f(x) dx \\
&= \int_x p(x) \left[\frac{q(x)}{p(x)} f(x) \right] dx \\
&= \mathbb{E}_{x \sim p} \left[\frac{q(x)}{p(x)} f(x) \right] \\
&\approx \sum_{i=1}^{n} \left[\frac{q(x_i)}{p(x_i)} f(x_i) \right] .
\end{aligned}$$

The last equation gives us the importance sampling estimate of f under distribution q using samples of f under distribution p. Note that the first step only holds if $q(x)f(x) > 0$ implies $p(x) > 0$ for all x.

3.2 Importance sampling for off-policy policy evaluation

We now apply the general result of importance sampling estimates to reinforcement learning. In this instance, we want to approximate the value of state s under policy π_1, given by $V^{\pi_1}(s) = \mathbb{E}[G_t \mid s_t = s]$, using n histories $h_1, \dots, h_n$ generated under policy π_2. Using the importance sampling estimate result gives us that

$$V^{\pi_1}(s) \approx \frac{1}{n} \sum_{j=1}^{n} \frac{\mathbb{P}(h_j \mid \pi_1, s)}{\mathbb{P}(h_j \mid \pi_2, s)} G(h_j)\,,$$

where $G(h_j) = \sum_{t=1}^{L_j - 1} \gamma^{t-1} r_{j,t}$ is the total discounted sum of rewards for history h_j.

Now, for a general policy π, we have that the probability of experiencing history h_j under policy π is

$$\begin{aligned}
\mathbb{P}(h_j \mid \pi, s = s_{j,1}) &= \prod_{t=1}^{L_j - 1} \mathbb{P}(a_{j,t} \mid s_{j,t}) \mathbb{P}(r_{j,t} \mid s_{j,t}, a_{j,t}) \mathbb{P}(s_{j,t+1} \mid s_{j,t}, a_{j,t}) \\
&= \prod_{t=1}^{L_j - 1} \pi(a_{j,t} \mid s_{j,t}) \mathbb{P}(r_{j,t} \mid s_{j,t}, a_{j,t}) \mathbb{P}(s_{j,t+1} \mid s_{j,t}, a_{j,t})\,,
\end{aligned}$$

where L_j is the length of the j-th episode. The first line follows from looking at the three components of each transition. The components are

1. $\mathbb{P}(a_{j,t} \mid s_{j,t})$ - probability we take action $a_{j,t}$ at state $s_{j,t}$;
2. $\mathbb{P}(r_{j,t} \mid s_{j,t}, a_{j,t})$ - probability we experience reward $r_{j,t}$ after taking action $a_{j,t}$ in state $s_{j,t}$;
3. $\mathbb{P}(s_{j,t+1} \mid s_{j,t}, a_{j,t})$ - probability we transition to state $s_{j,t+1}$ after taking action $a_{j,t}$ in state $s_{j,t}$.

Now, combining our importance sampling estimate for $V^{\pi_1}(s)$ with our decomposition of the history probabilities, $\mathbb{P}(h_j \mid \pi, s = s_{j,1})$, we get that

$$\begin{aligned}
V^{\pi_1}(s) &\approx \frac{1}{n} \sum_{j=1}^{n} \frac{\mathbb{P}(h_j \mid \pi_1, s)}{\mathbb{P}(h_j \mid \pi_2, s)} G(h_j) \\
&= \frac{1}{n} \sum_{j=1}^{n} \frac{\prod_{t=1}^{L_j-1} \pi_1(a_{j,t} \mid s_{j,t}) \mathbb{P}(r_{j,t} \mid s_{j,t}, a_{j,t}) \mathbb{P}(s_{j,t+1} \mid s_{j,t}, a_{j,t})}{\prod_{t=1}^{L_j-1} \pi_2(a_{j,t} \mid s_{j,t}) \mathbb{P}(r_{j,t} \mid s_{j,t}, a_{j,t}) \mathbb{P}(s_{j,t+1} \mid s_{j,t}, a_{j,t})} G(h_j) \\
&= \frac{1}{n} \sum_{j=1}^{n} G(h_j) \prod_{t=1}^{L_j-1} \frac{\pi_1(a_{j,t} \mid s_{j,t})}{\pi_2(a_{j,t} \mid s_{j,t})}\,.
\end{aligned}$$

Notice we can now explicitly evaluate the expression without the transition probabilities or rewards since all of the terms involving model dynamics canceled out in the second step of the equation. In particular, we are given the histories h_j, so we can calculate $G(h_j) = \sum_{t=1}^{L_j-1} \gamma^{t-1} r_{j,t}$, and we know the two policies π_1 and π_2, so we can also evaluate the second term.

4 Temporal difference learning

We have discussed two methods for policy evaluation: dynamic programming (with a known model) and Monte Carlo. Dynamic programming leverages bootstrapping to help us get value estimates with only one backup. On the other hand, Monte Carlo samples many histories for many trajectories which frees us from using a model. Now, we introduce a new algorithm that combines bootstrapping with sampling to give us a second model-free policy evaluation algorithm.

To see how to combine sampling with bootstrapping, we go back to our incremental Monte-Carlo update

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(G_t - V^\pi(s_t))\,.$$

Recall that G_t is the return after rolling out the policy from time step t to termination starting at state s_t. We now replace G_t with a Bellman backup like in dynamic programming. That is, we replace G_t with $r_t + \gamma V^\pi(s_{t+1})$, where r_t is a sample of the reward at time step t and $V^\pi(s_{t+1})$ is our current estimate of the value at the next state. Making this substitution gives us the temporal difference (TD) learning update

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t))\,.$$

The difference

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

is commonly referred to as the TD error, and the sampled reward combined with the bootstrap estimate of the next state value,

$$r_t + \gamma V^\pi(s_{t+1})\,,$$

is referred to as the TD target. The full TD learning algorithm is given in Algorithm 5. We can see that using this method, we update our value for $V^\pi(s_t)$ directly after witnessing the transition (s_t, a_t, r_t, s_{t+1}). In particular, we do not need to wait for the episode to terminate like in Monte Carlo.

Algorithm 5: TD Learning to evaluate policy π

```
Input: step size $\alpha$, number of trajectories $n$
For all states $s$, $V^\pi(s) \leftarrow 0$
while $n > 0$ do
    Begin episode $E$ at state $s$
    while episode $E$ has not terminated do
        $a \leftarrow$ action at state $s$ under policy $\pi$
        Take action $a$ in $E$ and observe reward $r$, next state $s'$
        $V^\pi(s) \leftarrow V^\pi(s) + \alpha(R + \gamma V^\pi(s') - V^\pi(s))$
        $s \leftarrow s'$
    $n \leftarrow n - 1$
return $V^\pi$
```

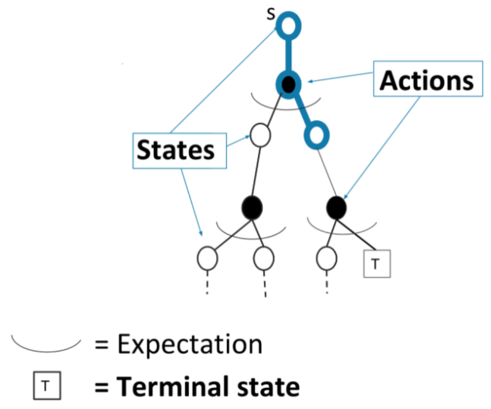

Figure 2: Backup diagram for the TD Learning policy evaluation algorithm.

We can again examine this algorithm via a backup diagram as shown in Figure 2. Here, we see via the blue line that we sample one transition starting at s, then we estimate the value of the next state via our current estimate of the next state to construct a full Bellman backup estimate.

There is actually an entire spectrum of ways we can blend Monte Carlo and dynamic programming using a method called TD(λ). When $\lambda = 0$, we get the TD learning formulation above, hence giving us the alias TD(0). When $\lambda = 1$, we recover Monte-Carlo policy evaluation, depending on the formulation used. When $0 < \lambda < 1$, we get a blend of these two methods. For a more thorough treatment of TD(λ), we refer the interested reader to Sections 7.1 and 12.1-12.5 of *Reinforcement learning: An introduction*, or wait until a future lecture, which detail n-step TD learning and TD(λ)/eligibility traces, respectively.

4.1 Batch Monte-Carlo sampling and temporal difference

We now look at the batch versions of the algorithms in today's lecture, where we have a set of histories that we use to make updates many times. We consider the batch cases of Monte Carlo and TD(0). In the batch case, we are given a batch, or set of histories $h_1, \ldots, h_n$, which we then feed through Monte Carlo or TD(0) many times. The only difference from our formulations before is that we only update the value function after each time we process the entire batch. Thus in TD(0), the bootstrap estimate is updated only after each pass through the batch.

A motivating example Before looking at the batch cases in generality, we first look at Example 6.4 from *Reinforcement learning: An introduction* to more closely examine the difference between Monte Carlo and TD(0). Suppose $\gamma = 1$ and we have eight histories generated by policy π, take action a_1 in all states:

$$\begin{aligned} h_1 &= (A, a_1, +0, B, a_1, +0, terminal) \\ h_j &= (B, a_1, +1, terminal) \text{ for } j = 2, \ldots, 7 \\ h_8 &= (B, a_1, +0, terminal). \end{aligned}$$

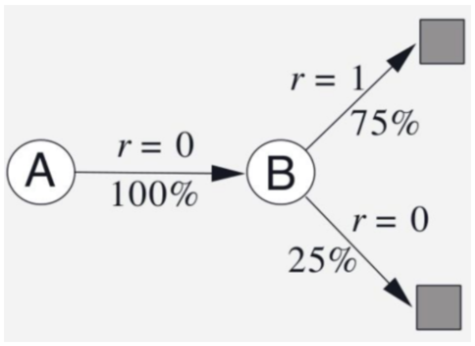

Figure 3: Example 6.4 from *Reinforcement learning: An introduction.*

Then, using either batch Monte Carlo or TD(0) with $\alpha = 1$, we see that $V(B) = 0.75$. However, using Monte Carlo, we get that $V(A) = 0$ since only the first episode visits state A and has return 0. On the other hand, TD(0) gives us $V(A) = 0.75$ because we perform the update $V(A) \leftarrow r_{1,1} + \gamma V(B)$. Under a Markovian domain like the one shown in Figure 3, the estimate given by TD(0) makes more sense.

Convergence of batch TD(0) In the Monte-Carlo batch setting, the value at each state converges to the value that minimizes the mean squared error with the observed returns. This follows directly from the fact that in Monte Carlo, we take an average over returns at each state, and in general, the MSE minimizer of samples is precisely the average of the samples. That is, given samples $y_1, \ldots, y_n$, the value $\sum_{i=1}^n (y_i - \hat{y})^2$ is minimized for $\hat{y} = \sum_{i=1}^n y_i$. We can also see this in the example at the beginning of the section. We get that $V(A) = 0$ for Monte Carlo because this is the only history visiting state A.

In the TD(0) batch setting, we do not converge to the same result as in Monte Carlo. In this case, we converge to the value V^π that is the value of policy π on the maximum likelihood MDP model where

$$\hat{P}(s' \mid s, a) = \frac{1}{N(s,a)} \sum_{j=1}^{n} \sum_{t=1}^{L_j - 1} \mathbb{1}(s_{j,t} = s, a_{j,t}, s_{j,t+1} = s')$$

$$\hat{r}(s, a) = \frac{1}{N(s,a)} \sum_{j=1}^{n} \sum_{t=1}^{L_j - 1} \mathbb{1}(s_{j,t} = s, a_{j,t}) r_{j,t} \,.$$

In other words, the maximum likelihood MDP model is the most naive model we can create based on the batch - the transition probability $\hat{P}(s' \mid s, a)$ is the fraction of times that we see the transition (s, a, s') after we take action a at state s in the batch, and the reward $\hat{r}(s, a)$ is the average reward experienced after taking action a at state s in the batch.

We also see this result in the example from the beginning of the section. In this case, our maximum likelihood model is

$$\begin{aligned}\hat{P}(B \mid A, a_1) &= 1 \\ \hat{P}(terminal \mid B, a_1) &= 1 \\ \hat{r}(A, a_1) &= 0 \\ \hat{r}(B, a_1) &= 0.75\,.\end{aligned}$$

This gives us $V^\pi(A) = 0.75$, like we stated before.

The value function derived from the maximum likelihood MDP model is known as the certainty equivalence estimate. Using this relationship, we have another method for evaluating the policy. We can first compute the maximum likelihood MDP model using the batch. Then we can compute V^π using this model and the model-based policy evaluation methods discussed in last lecture. This method is highly data efficient but is computationally expensive because it involves solving the MDP which takes time $O(|\mathcal{S}|^3)$ analytically and $(|\mathcal{S}|^2|\mathcal{A}|)$ via dynamic programming.

Acknowledgement

This lecture notes partially use material from *Reinforcement learning: An introduction* and *CS234: Reinforcement learning* from Stanford.