Introduction to Reinforcement Learning

1 What is Reinforcement Learning?

So far we have seen unsupervised and supervised learning. Unsupervised learning tries to find patterns or representations for unlabeled data. In that setting, we often try to find clusters or low-dimensional representations for our data. Supervised learning algorithms try to make their outputs mimic the labels y given in the training set. In that setting, the labels give an unambiguous right answer for each of the inputs x. In contrast, for many sequential decision making and control problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the correct actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

In the reinforcement learning framework, we will instead provide our algorithms only a reward function, which indicates to the learning agent when it is doing well, and when it is doing poorly. In the four-legged walking example, the reward function might give the robot positive rewards for moving forwards, and negative rewards for either moving backwards or falling over. It will then be the learning algorithms job to figure out how to choose actions over time so as to obtain large rewards. In general, reinforcement learning (RL) deals with sequential decision making with time.

Reinforcement learning has been successful in applications as diverse as playing atari games, playing the game of Go, autonomous helicopter flight, robot legged locomotion, cell-phone network routing and marketing strategy selection. We will begin studying RL by defining Markov decision processes (MDPs), which provide the formalism in which RL problems are usually posed.

RL typically applies to scenarios with an environment, and an agent trying to learn a task by interacting with the environment (see figure 1). Assuming discrete timesteps, at each timestep the

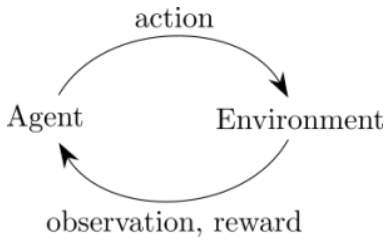

Figure 1: The reinforcement learning scenario

agent gets an observation x_t and responds by taking an action a_t. The environment responds by giving the agent a reward r_t, and the next observation x_{t+1}. Note that the environment has an internal state for itself s_t which might not be fully observable by the agent. Note that for MDPs, the environment state is fully observable i.e. $x_t = s_t$, e.g. Chess and Go. But many interesting examples of RL involve only partial observations i.e. $x_t = \phi(s_t)$. In such a case the framework is called a partially observable markov decision process (POMDP). For now, we will restrict ourselves to MDPs i.e. we will assume that the agent receives full state s_t in the observation at time t.

2 Markov Decision Processes

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of a helicopter.)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopters control sticks.)
- $P_{sa} : S \times A \times S \to \mathbb{R}$ are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space i.e. $\sum_{s'} P_{sa}(s') = 1 \ \forall s \in S, a \in A$ and $0 \leq P_{sa}(s') \leq 1 \ \forall s, s' \in S, a \in A$. Well say more about this later, but briefly, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \to \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state s only, in which case we would have $R : S \to \mathbb{R}$).

The dynamics of an MDP proceeds as follows: We start in some state s_0, and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$. Then, we get to pick another action a_1. As a result of this action, the state transitions again, now to some $s_2 \sim P_{s_1 a_1}$. We then pick a_2, and so on ... Pictorially, we can represent this process as follows:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_3} s_3 \xrightarrow{a_3} \cdots$$

Total Payoff: Upon visiting the sequence of states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, our total payoff is given by:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots .$$

Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots .$$

For most of our development, we will use the simpler state-rewards $R(s)$, though the generalization to state-action rewards $R(s, a)$ offers no special difficulties.

Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$\mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots].$$

Note that the reward at timestep t is **discounted** by a factor of γ^t. Thus, to make this expectation large, we would like to accrue positive rewards as soon as possible (and postpone negative rewards as long as possible). In economic applications where $R(\cdot)$ is the amount of money made, γ also has a natural interpretation in terms of the interest rate (where a dollar today is worth more than a

dollar tomorrow). Mathematically, γ makes the total payoff bounded even for infinite timesteps as long each individual reward $R(s) \in [-R_{max}, R_{max}]$ for some $R_{max} < \infty$ i.e.

$$\begin{aligned}
&|R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots| \\
\leq &|R(s_0)| + \gamma|R(s_1)| + \gamma^2|R(s_2)| + \cdots \\
\leq &R_{max}(1 + \gamma + \gamma^2 + \cdots) \\
\leq &\frac{R_{max}}{1-\gamma}
\end{aligned}$$

Policy: A policy is a probability distribution $\pi : S \times A \to \mathbb{R}$ over actions in any state s i.e. $\sum_{a \in A} \pi(s, a) = 1 \ \forall s \in S$ and $0 \leq \pi(s, a) \leq 1 \ \forall s \in S, a \in A$. Such a policy is called a stochastic policy. We can also have deterministic policies which recommend a single action a in a given state s. In such a case the policy is a function $\pi : S \to A$ mapping the states to actions. From here on, we will only work with deterministic policies (more on why later). We say that we are executing some policy π if, whenever we are in state s, we take our next action $a = \pi(s)$.

Value Function: We also define the value function for a policy π as:

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s; \pi]$$

$V^\pi(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.

Bellman's Equations: Given a fixed policy π, its value function V^π satisfies the following recurrence relation:

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_{s_0, s_1, s_2, \ldots}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s; \pi] \\
&= R(s) + \gamma \mathbb{E}_{s_0, s_1, s_2, \ldots}[R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s; \pi] \\
&= R(s) + \gamma \sum_{s' \in S} \left\{ P_{s\pi(s)}(s') \ \mathbb{E}_{s_0, s_1, s_2, \ldots}[R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s, s_1 = s'; \pi] \right\} \\
&= R(s) + \gamma \sum_{s' \in S} \left\{ P_{s\pi(s)}(s') \ \mathbb{E}_{s_1, s_2, \ldots}[R(s_1) + \gamma^2 R(s_2) + \cdots | s_1 = s'; \pi] \right\} \\
&= R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')
\end{aligned}$$

The final result:

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')$$

is called the **Bellman's equation**. It says that the expected sum of discounted rewards $V^\pi(s)$ for starting in s consists of two terms: First, the immediate reward $R(s)$ that we get right away simply for taking any action in state s, and second, the expected sum of future discounted rewards. Examining the second term in more detail, we see that the summation term above can be rewritten as $\mathbb{E}_{s' \sim P_{s\pi(s)}}[V^\pi(s')]$. This is the expected sum of discounted rewards for starting in state s', where s' is distributed according to $P_{s\pi(s)}$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s. Thus, the second term above gives the expected sum of discounted rewards obtained after the first step in the MDP.

Optimal Value Function: We also define the optimal value function according to

$$V^*(s) = \max_\pi V^\pi(s)$$

In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy starting from the state s. Take some time to ponder over the meaning of such a maximization, since this might make you believe that there is a different policy π_s which maximizes $V^\pi(s)$. It turns out that there is actually a single deterministic policy which can maximize the value function V^π starting from any state s. This is precisely why we began only with deterministic policies.

Note that π^* has the interesting property that it is the optimal policy for all states s. Specifically, it is not the case that if we were starting in some state s then there'd be some optimal policy for that state, and if we were starting in some other state s' then there'd be some other policy that's optimal policy for s'. The same policy π^* attains the maximum value function for all states s (this follows directly from the Bellman recurrence equation). This means that we can use the same policy π^* no matter what the initial state of our MDP is.

So we can now rewrite the definition of V^* as:

$$V^* = \max_\pi V^\pi$$

where the vector-valued objective is maximized simultaneously in all its components by a single deterministic policy.

There is also a version of Bellman's equations for the optimal value function:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s').$$

The first term above is the immediate reward as before. The second term is the maximum over all actions a of the expected future sum of discounted rewards well get upon after action a. You should make sure you understand this equation and see why it makes sense.

We also define a policy $\pi^* : S \to A$ as follows:

$$\pi^*(s) = arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s').$$

Note that $\pi^*(s)$ gives the action a that attains the maximum in the "max" in the bellman equations for $V^*(s)$.

It is a fact that for every state s and every policy π, we have

$$V^*(s) = V^{\pi^*}(s) \geq V^\pi(s).$$

The first equality says that the V^{π^*}, the value function for π^*, is equal to the optimal value function V^* for every state s. Further, the inequality above says that π^*'s value is at least as large as the value of any other policy. In other words, π^* as defined above is the optimal policy.

3 Solving MDPs

Policy evaluation: Given an MDP and a fixed policy π, policy evaluation means computing the value function V^π for the policy V^π. One way to do this efficiently is by using Bellman's equations. Specifically, in a finite-state MDP ($|S| < \infty$), we can write down one such equation for $V^\pi(s)$ for every state s. This gives us a set of $|S|$ linear equations in $|S|$ variables (the unknown $V^\pi(s)$'s, one for each state), which can be efficiently solved for the $V^\pi(s)$'s as a linear system. With $V^\pi \in \mathbb{R}^{|S|}$ vector containing value functions for each state, $R \in \mathbb{R}^{|S|}$ containing rewards for each state and P^π as the matrix with its (i, j)-th entry as $P^\pi(i, j) = P_{s_i\pi(s_i)}(s_j)$:

$$V^\pi = R + \gamma P^\pi V^\pi \quad \text{(Bellman's equations)}$$
$$V^\pi = (I - \gamma P^\pi)^{-1} R$$

We can also evaluate a policy using value iteration (described shortly).

Solving an MDP: An MDP is said to be solved when the optimal value V^* and the optimal policy π^* have been computed for the MDP. We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces ($|S| < \infty, |A| < \infty$).

3.1 Value Iteration

The first algorithm, value iteration, can be used for both policy evaluation and for solving MDPs.
Policy evaluation: For policy evaluation it is applied as follows:

1. For each state s, initialize $V^\pi(s) := 0$.
2. Repeat until convergence {
   For every state, update $V^\pi(s) := R(s) + \gamma \sum_{s'} P_{s\pi(s)}(s')V^\pi(s')$.
   }

Solving MDP: For computing the optimal value function and policy, it is applied as follows:

1. For each state s, initialize $V^*(s) := 0$.
2. Repeat until convergence {
   For every state, update $V^*(s) := R(s) + \max_{a\in A} \gamma \sum_{s'} P_{sa}(s')V^*(s')$.
   }
3. $\pi^*(s) = arg\max_{a\in A} \sum_{s'\in S} P_{sa}(s')V^*(s')$ for all $s \in S$.

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman's equations. The updates can be written in matrix-vector notations for faster and parallelized implementation using numerical libraries.

There are two possible ways of performing the updates in the inner loop of each version of the algorithm. In the first, we can first compute the new values for $V^\pi(s)$ (or $V^*(s)$) for every state s, and then overwrite all the old values with the new values. This is called a **synchronous** update. In this case, the algorithm can be viewed as implementing a "Bellman backup operator" that takes a current estimate of the value function, and maps it to a new estimate. Alternatively, we can also

perform **asynchronous** updates. Here, we would loop over the states (in some order), updating the values one at a time. Under either synchronous or asynchronous updates, it can be shown that value iteration converges to V^π (or V^*). The asynchronous version converges slightly faster than the synchronous version, but the synchronous counterpart is easier to analyze theoretically.

3.2 Policy Iteration

Apart from value iteration, there is a second standard algorithm for finding an optimal policy for an MDP. The policy iteration algorithm proceeds as follows:

1. Initialize π randomly.
2. Repeat until convergence {
   (a) **Evaluate**: Assign $V := V^\pi$ (i.e. evaluate policy π).
   (b) **Improve**: For each state s, $\pi(s) := arg\max_{a \in A} \sum_{s'} P_{sa}(s')V(s')$.
   }

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is greedy with respect to V.) Note that step (a) can be done via solving Bellman's equations as described earlier, which in the case of a fixed policy, is just a set of $|S|$ linear equations in $|S|$ variables. After at most a finite number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isnt currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V^π explicitly would involve solving a large system of linear equations, and could be difficult. In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.

4 Model-based Learning

So far, we have discussed MDPs and algorithms for MDPs assuming that the state transition probabilities and rewards are known. In many realistic problems, we are not given state transition probabilities and rewards explicitly, but must instead estimate them from data. (Usually, S, A and γ are known.)

For example, suppose that, we had a number of trials in the MDP, that proceeded as follows:

$$s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \dots$$
$$s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \dots$$
$$\dots$$

Here, $s_i^{(j)}$ is the state we were at time i of trial j, and $a_i^{(j)}$ is the corresponding action that was taken from that state. In practice, each of the trials above might be run until the MDP terminates, or it might be run for some large but finite number of timesteps.

Given this "experience" in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:

$$P_{sa}(s') = \frac{\#\text{times we took action } a \text{ in state } s \text{ and reached } s'}{\#\text{times we took action } a \text{ in state } s}$$

Or, if the ratio above is "0/0" – corresponding to the case of never having taken action a in state s before – then we might simply estimate $P_{sa}(s')$ to be $1/|S|$, i.e. estimate P_{sa} to be the uniform distribution over all states.

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms, then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of P_{sa}.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilities and rewards. For example, putting together model learning and value iteration, here is one possible algorithm for learning in an MDP with unknown state transition probabilities:

1. Initialize π randomly.
2. Repeat {
   (a) Execute π in the MDP for some number of trials.
   (b) Using the accumulated experience in the MDP, update our estimates for P_{sa} (and R, if applicable).
   (c) Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function V.
   (d) Update π to be the greedy policy with respect to V.

   }

We note that, for this particular algorithm, there is one simple optimization that can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with $V = 0$, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

5 Model-free Learning

Suppose we do not know the transition probabilities P_{sa} and rewards R for an MDP. One way to find the optimal policy π^* for the MDP is to learn the MDP model from experience. But learning

a huge MDP model to just compute the optimal policy can be too computationally expensive. Instead, we can also develop model-free methods which try to learn the optimal value function and optimal policy without explicitly learning the model. We will cover such a method called Q-learning briefly in this section. It is inspired from value iteration and follows a similar iterative procedure in the basence of a model.

Before we develop a model-free counterpart of value iteration, it is important to note that after doing value iteration retreiving the optimal policy π^* requires the knowledge of the distribution P_{sa} since we compute $\pi^*(s) = arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s')V^*(s')$ for all $s \in S$. This happens because the value function V is only defined over states, but does not convey the value of taking a particular action a in a state s.

So we will first define the **action value function** (also called **Q-function**) $Q^\pi : S \times A \to \mathbb{R}$ for a policy π as:

$$Q^\pi(s,a) = \mathbb{E}[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \cdots | s_0 = s, a_0 = a; \pi]$$

Note that $Q^\pi(s,a)$ is the expected total payoff for taking action a in state s and then following the policy π thereafter. This new value function makes the value of taking action a in state s more explicit. The optimal Q-value function is defined as:

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

There is also a version of Bellman's equations for the Q-function (Q^π) and for Q^*:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P_{sa}(s') Q^\pi(s', \pi(s'))$$

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P_{sa}(s') \left\{ \max_{a'} Q^*(s',a') \right\}$$

Also note that the value function and Q-function are related as follows:

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

Finally the optimal policy is directly computable from Q^* as:

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

Hence computing the optimal Q-function Q^* results in direct computation of the policy π^* without having knowledge of the transition distribution P_{sa}.

5.1 Q-learning

Q-learning learns the optimal Q-value function (Q^*) and consequently the optimal policy (π^*) as follows:

1. Initialize $Q(s,a) = 0 \ \forall (s,a) \in S \times A$.
2. Repeat until convergence {

(a) In state s_t, choose action a_t as a random action $a \in A$ with probability ϵ, or $arg\max_a Q(s_t, a)$ with probability $1 - \epsilon$. Record the transition (s_t, a_t, r_t, s_{t+1}).

(b) Update

$$Q(s_t, a_t) := \left\{ \begin{array}{ll} (1-\alpha_t)Q(s_t,a_t) + \alpha_t R_t, & \text{for absorbing } s_{t+1} \\ (1-\alpha_t)Q(s_t,a_t) + \alpha_t(R_t + \gamma \max_{a'} Q(s_{t+1}, a')), & \text{for non-absorbing } s_{t+1} \end{array} \right\}$$

3. $\pi(s) = arg\max_a Q(s, a)$. }

Q-learning uses $R_t + \gamma \max_{a'} Q(s_{t+1}, a')$ as an unbiased sample-based estimate of the optimal Q-value at each time step t. Note that α_t plays the role of a step-size and helps in maintaing a running average to replace the true average/expectation as required by the Bellman operator.

Also, the algorithm is completely **model-free** in the sense that it does not make use of P_{sa} anywhere and at any timestep it only uses the reward obtained at that timestep.

Exploration: Since the algorithm works using a greedy action selection principle, it might get stuck into a local minimum, hence we often use an **ϵ-greedy** strategy instead. So in step (a), instead of always choosing the currently believed best action $arg\max_a Q(s_t, a)$, we choose it with a high probability $1 - \epsilon$. sometimes, we choose random actions with a small probability $\epsilon \approx 0.1$ or 0.05 in order to promote exploration.

More improvements to Q-learning: In the last decade, Q-learning has been improved further by adding many other features and has been applied to many state-of-the-art RL tasks. Some of the improvements are described below briefly:

- **Action-replay memory**: Previous transitions (s_t, a_t, r_t, s_{t+1}) should be reused for better sample efficiency. Hence they can be recorded and stored in an action-replay memory. This also prevents divergence due to heavily correlated consecutive transition samples.
- **Function approximation**: For large MDPs, it is impossible to store the Q-function in a tabular form. Mostly the Q-function is stored in the weights of a function approximator like a Deep neural network.
- **Target and Working Q-functions**: To prevent divergence due to heavily correlated consecutive transition samples and dynamically changing targets, two copies of Q-function are maintained. The target copy stays stable for many timesteps and the working copy is regularly trained. After every fixed (large) number of timesteps the target copy is reinitialized with the working copy of Q-function.

For more details, we encourage you to read the DQN paper from DeepMind [3].

Acknowledgements

This handout is an adaptation of [1], with some additional topics from [2] and [3].

References

[1] Stanford CS229 lecture notes on reinforcement learning, URL: `http://cs229.stanford.edu/notes/cs229-notes12.pdf`

[2] David Silver's class on reinforcement learning at UCL, URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[3] Mnih, V. et al. Human-level control through deep reinforcement learning. Nature 518, 529533 (2015).