Lecture 1

Reinforcement learning

Reinforcement learning is learning what to do-how to map situations to actions-so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. Reinforcement learning, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task.

All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. By a complete, interactive, goal-seeking agent we do not always mean something like a complete organism or robot. These are clearly examples, but a complete, interactive, goal-seeking agent can also be a component of a larger behaving system.

Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a policy, a reward signal, a value function, and,

optionally, a model of the environment.

A *policy* defines the learning agent?s way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus?response rules or associations. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action.

A *reward signal* defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the *reward*. The agent?s sole objective is to maximize the total reward it receives over the long run. In a biological system, we might think of rewards as analogous to the experiences of pleasure or pain. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

A *value* function specifies what is good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. Unfortunately, it is much harder

to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime

The fourth and final element of some reinforcement learning systems is a model of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners?viewed as almost the opposite of planning.

Multi-armed Bandits

Consider the following learning problem. You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps. This is the original form of the k-armed bandit problem, so named by analogy to a slot machine, or "one-armed bandit," except that it has k levers instead of one. In our k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the value of that action. We denote the action selected on time step t as At, and the corresponding reward as Rt. The value then of an arbitrary action a, denoted is the expected reward given that a is selected. If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the greedy actions.

Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit them many times. Because it is not possible both to explore and to exploit with any single action selection, one often refers to the "conflict" between exploration and exploitation.

Action-value Methods

We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call action-value methods. Recall that the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

Greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ϵ, instead select randomly from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule ϵ-greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Qt(a)$ converge to. This of course implies that the probability of selecting the optimal action converges to greater than that is, to near certainty.

Incremental Implementation

The action-value methods we have discussed so far all estimate action values as sample averages of observed rewards. We now turn to the question of how these

averages can be computed in a computationally efficient manner, in particular, with constant memory and constant per-time-step computation.

To simplify notation we concentrate on a single action. Let R_i now denote the reward received after the ith selection of this action, and let Qn denote the estimate of its action value after it has been selected n-1 times, which we can now write simply as

$$Q_n = \frac{R_1 + R_2 + ... + T_{n-1}}{n-1}$$

The obvious implementation would be to maintain a record of all the rewards and then perform this computation whenever the estimated value was needed. However, if this is done, then the memory and computational requirements would grow over time as more rewards are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator.

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Gradient Bandit Algorithms

So far in this chapter we have considered methods that estimate action values and use those estimates to select actions. This is often a good approach, but it is not the only one possible. In this section we consider learning a numerical preference

for each action a, which we denote Ht(a). The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important; if we add 1000 to all the action preferences there is no effect on the action probabilities, which are determined according to a soft-max distribution (i.e., Gibbs or Boltzmann distribution) as follows

$$Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{k}}$$

where here we have also introduced a useful new notation, ?t(a), for the probability of taking action a at time t. Initially all action preferences are the same (e.g., H1(a) = 0, for all a) so that all actions have an equal probability of being selected.

Lecture 2

Markov decision process (MDP)

MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward.

The Agent-Environment Interface

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment?s *state*, $S_t \in S$, and on that basis selects an action, $A_t \in A(s)$. One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in R \subset R$, and finds itself in a new state, S_{t+1}. The MDP and agent together thereby give rise to a sequence or trajectory that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

In a finite MDP, the sets of states, actions, and rewards (S, A, and R) all have

a finite number of elements. In this case, the random variables Rt and St have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s\prime \in S$ and $r \in R$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p\left(s', r|s, a\right) \doteq \Pr\left\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right\}$$

for all $s', s \in S, r \in R$, and $a \in A(s)$. The function p defines the dynamics of the MDP. The dot over the equals sign in the equation reminds us that it is a definition (in this case of the function p) rather than a fact that follows from previous definitions. The dynamics function $p : S \times R \times S \times A \rightarrow [0, 1]$ is an ordinary deterministic function of four arguments. The ?|? in the middle of it comes from the notation for conditional probability, but here it just reminds us that p specifies a probability distribution for each choice of s and a, that is, that

$$\sum_{x' \in S} \sum_{r \in \mathcal{R}} p\left(s', r|s, a\right) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s)$$

In a Markov decision process, the probabilities given by p completely characterize the environment?s dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, $St - 1$ and A_{t-1}, and, given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state. The state must include information about all aspects of the past agent?environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property.*

From the four-argument dynamics function, p, one can compute anything else one might want to know about the environment, such as the state-transition probabilities (which we denote, with a slight abuse of notation, as a three-argument function

$p : S \times S \times \mathcal{A} \to [0, 1]$),

$$p\left(s'|s,a\right) \doteq \Pr\left\{S_t = s'|S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r\in\mathcal{R}} p\left(s', r|s, a\right)$$

We can also compute the expected rewards for state?action pairs as a two-argument function $r : S \times A \to R$:

$$r(s,a) \doteq \mathbb{E}\left[R_t|S_{t-1} = s, A_{t-1} = a\right] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{B}} p\left(s', r|s, a\right)$$

and the expected rewards for state?action?next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$,

$$r\left(s, a, s'\right) \doteq \mathbb{E}\left[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'\right] = \sum_{r\in\mathcal{R}} r\frac{p\left(s', r|s, a\right)}{p\left(s'|s, a\right)}$$

Goals and Reward

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in \mathbb{R}$. Informally, the agent?s goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run. We can clearly state this informal idea as the *reward hypothesis*:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.

Although formulating goals in terms of reward signals might at first appear limiting, in practice it has proved to be flexible and widely applicable. The best

way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot?s forward motion. In making a robot learn how to escape from a maze, the reward is often .1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of +1 for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are +1 for winning, -1 for losing, and 0 for drawing and for all nonterminal positions.

Returns and Episodes

In general, we seek to maximize the expected return, where the return, denoted G_t, is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

where T is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent?environment interaction breaks naturally into subsequences, which we call *episodes*, such as plays of a game, trips through a maze, or any sort of repeated interaction. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Even if you think of episodes as ending in different ways, such as winning and losing a game, the next episode begins independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different

rewards for the different outcomes. Tasks with episodes of this kind are called *episodic tasks*. In episodic tasks we sometimes need to distinguish the set of all nonterminal states, denoted S, from the set of all states plus the terminal state, denoted S+. The time of termination, T , is a random variable that normally varies from episode to episode.

The additional concept that we need is that of discounting. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses At to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is a parameter, $0 \leq \gamma \leq 1$ called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum in eq has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose At so as to maximize only R_{t+1}. If each of the agent?s actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize above eq by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Returns at successive time steps are related to each other in a way that is im-

portant for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned}$$

Note that although the return is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant-if $\gamma < 1$. For example, if the reward is a constant $+1$, then the return is

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}.$$

Lecture 3

Bellman equations

Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating value functions-functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Like p, π is an ordinary function; the "|" in the middle of $\pi(a|s)$ merely reminds that it defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. Reinforcement learning methods specify how the agent?s policy is changed as a result of its experience.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right], \text{ for all } s \in \mathcal{S}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy* π.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi\left[G_t|S_t=s,A_t=a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s,A_t=a\right]$$

We call q_π the action-value function for policy π.

The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state?s value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s,a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return . For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t|S_t=s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1}+\gamma G_{t+1}|S_t=s\right] \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p\left(s',r|s,a\right)\left[r+\gamma\mathbb{E}_\pi\left[G_{t+1}|S_{t+1}=s'\right]\right] \\
&= \sum_a \pi(a|s)\sum_{s',r} p\left(s',r|s,a\right)\left[r+\gamma v_\pi\left(s'\right)\right], \qquad \text{for all } s\in\mathcal{S}
\end{aligned}$$

where it is implicit that the actions, a, are taken from the set $\mathcal{A}(s)$, that the next states, s', are taken from the set $\mathcal{S}$ (or from $\mathcal{S}^+$ in the case of an episodic problem),

and that the rewards, r, are taken from the set $\mathcal{R}$. Note also how in the last equation we have merged the two sums, one over all the values of s' and the other over all the values of r, into one sum over all the possible values of both. We use this kind of merged sum often to simplify formulas. Note how the final expression can be read easily as an expected value. It is really a sum over all values of the three variables, a, s', and r. For each triple, we compute its probability, $\pi(a|s)p(s', r|s, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

Above equation is the *Bellman equation* for v_π. It expresses a relationship between the value of a state and the values of its successor states.

Optimal Policies and Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if *eq* for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy.* Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the *optimal state-value function*, denoted v_*, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_*, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a)$$

for all $s \in S$ and $a \in \mathcal{A}(s)$. For the state?action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) | S_t = s, A_t = a\right]$$

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values eq. Because it is the optimal value function, however, v_*'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_*, or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\
&= \max_a \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_*\left(s'\right)\right]
\end{aligned}$$

The last two equations are two forms of the Bellman optimality equation for v_*. The Bellman optimality equation for q_* is

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right) | S_t = s, A_t = a\right] \\
&= \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]
\end{aligned}$$

The backup diagrams in the figure below show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_*.

Lecture 4

Policy evaluation using Monte Carlo

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. we can easily obtain optimal policies once we have found the optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_*\left(s'\right)\right], \text{ or} \\ q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right) | S_t = s, A_t = a\right] \\ &= \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right] \end{aligned}$$

for all $s \in \mathcal{S}, a \in \mathcal{A}$ and $s' \in \mathcal{S}^+$. DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

Policy Evaluation

First we consider how to compute the state-value function v? for an arbitrary policy ?. This is called policy evaluation in the DP literature. We also refer to it as the prediction problem.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi \left(S_{t+1} \right) | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p \left(s', r | s, a \right) \left[r + \gamma v_\pi \left(s' \right) \right] \end{aligned}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

If the environment?s dynamics are completely known, then is a system of $|S|$ simultaneous linear equations in $|S|$ unknowns , s 2 S). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions $v_0, v_1, v_2, ...$, each mapping $S+$ to R (the real numbers). The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k \left(S_{t+1} \right) | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p \left(s', r | s, a \right) \left[r + \gamma v_k \left(s' \right) \right] \end{aligned}$$

for all $s \in S$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v assures us of equality in this case. Indeed, the sequence

$\{v_k\}$ can be shown in general to converge to v as k_1 under the same conditions that guarantee the existence of v. This algorithm is called iterative policy evaluation.

To produce each successive approximation, vk+1 from vk, iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation an expected update. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function

There are several different kinds of expected updates, depending on whether a state (as here) or a state?action pair is being updated, and depending on the precise way the estimated values of the successor states are combined. All the updates done in DP algorithms are called expected updates because they are based on an expectation over all possible next states rather than on a sample next state.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v? for an arbitrary deter-

ministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action a $6 = \pi(s)$. We know how good it is to follow the current policy from s?that is $v_\pi(s)$?but would it be better or worse to change to the new policy One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π. The value of this way of behaving is

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\ &= \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right] \end{aligned}$$

The key criterion is whether this is greater than or less than v?(s). If it is greater?that is, if it is better to select a once in s and thereafter follow ? than it would be to follow ? all the time?then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

$$q_\pi\left(s, \pi'(s)\right) \geq v_\pi(s)$$

That this is true is a special case of a general result called the policy improvement theorem. Let π and pi_0 be any pair of deterministic policies such that, for all s 2 S

$$v'_\pi(s) \geq v_\pi(s)$$

Then the policy π_0 must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in S$:

$$\begin{aligned} \pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\ &= \arg\max_a \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right] \end{aligned}$$

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action. It is a natural extension

Lecture 5

Policy iteration and Value iteration

Policy Iteration

Once a policy, π, has been improved using v to yield a better policy, π_0, we can then compute v_0 and improve it again to yield an even better ?00. We can thus obtain a sequence of monotonically improving policies and value functions

where E denotes a policy evaluation and .!I denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called policy iteration. A complete algorithm is given in the box below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure certainly suggests that it may be

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called value iteration. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k\left(S_{t+1}\right) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_k\left(s'\right)\right] \end{aligned}$$

for all $s \in S$. For arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v under the same conditions that guarantee the existence of v.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Generalized policy iteration

Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement. In asynchronous DP methods, the evaluation and improvement processes are

interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same?convergence to the optimal value function and an optimal policy

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policyimprovement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal.

Lecture 6

StateAction-Reward-State-Action (SARSA)

Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment?s dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning; this chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways.

TD Prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V\left(S_t\right) \leftarrow V\left(S_t\right) + \alpha\left[G_t - V\left(S_t\right)\right]$$

where Gt is the actual return following time t, and α is a constant step-size parameter . Let us call this method constant-α MC. Whereas Monte Carlo methods

must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_t+1)$. The simplest TD method makes the update

$$V\left(S_t\right) \leftarrow V\left(S_t\right)+\alpha\left[R_{t+1}+\gamma V\left(S_{t+1}\right)-V\left(S_t\right)\right]$$

immediately on transition to S_t+1 and receiving R_t+1. In effect, the target for the Monte Carlo update is Gt, whereas the target for the TD update is $R_t+1+\gamma V(S_t+1)$. This TD method is called $TD(0)$, or one-step TD, because it is a special case of the $TD(\lambda)$ and n-step TD methods developed in . The box below specifies $TD(0)$ completely in procedural form.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0,1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Because TD(0) bases its update in part on an existing estimate, we say that it is a bootstrapping method, like DP. We know from Chapter 3 that

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t | S_t=s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1}+\gamma G_{t+1} | S_t=s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1}+\gamma v_\pi\left(S_{t+1}\right) | S_t=s\right] \end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of as a target, whereas DP methods use an estimate of as a target. The Monte Carlo target is an estimate

because the expected value in is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v(S_t + 1)$ is not known and the current estimate, $V(S_t + 1)$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in and it uses the current estimate V instead of the true v. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Finally, note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of St and the better estimate $Rt + 1 + \gamma V(St + 1)$. This quantity, called the TD error, arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V\left(S_{t+1}\right) - V\left(S_t\right)$$

Notice that the TD error at each time is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, is the error in $V(S_t)$, available at time $t + 1$. Also note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned}
&G_t - V\left(S_t\right) = R_{t+1} + \gamma G_{t+1} - V\left(S_t\right) + \gamma V\left(S_{t+1}\right) - \gamma V\left(S_{t+1}\right) \\
&= \delta_t + \gamma\left(G_{t+1} - V\left(S_{t+1}\right)\right) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\left(G_{t+2} - V\left(S_{t+2}\right)\right) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\left(G_T - V\left(S_T\right)\right) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0) \\
&= \textstyle\sum_{k=t}^{T-1} \gamma^{k-t}\delta_k
\end{aligned}$$

This identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately. Generalizations of this identity play an important role in the theory and algorithms of temporal-difference learning.

Lecture 7

Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right]$$

In this case, the learned action-value function, Q, directly approximates q, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state?action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q. The Q-learning algorithm is shown below in procedural form.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

What is the backup diagram for Q-learning? The rule (6.8) updates a state?action pair, so the top node, the root of the update, must be a small, filled action node. The update is also from action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these ?next action? nodes with an arc across them

Expected Sarsa

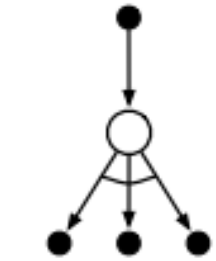
Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state?action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$Q(S_t, A_t) \leftarrowtail Q(S_t, A_t) + \alpha$$

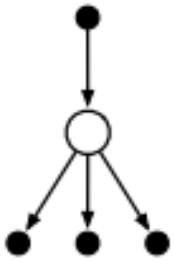
but that otherwise follows the schema of Q-learning. Given the next state, $St + 1$, this algorithm moves deterministically in the same direction as Sarsa moves in expectation, and accordingly it is called Expected Sarsa.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_t+1. Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. shows summary results on the cliff-walking task with Expected Sarsa compared to Sarsa and Q-learning. Expected Sarsa retains the significant advantage of Sarsa over Q-learning on this problem. In addition, Expected Sarsa shows a significant improvement

over Sarsa over a wide range of values for the step-size parameter α. In cliff walking the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can safely set $\alpha = 1$ without suffering



Q-learning



Expected Sarsa

any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of ?, at which short-term performance is poor. In this and other examples there is a consistent empirical advantage of Expected Sarsa over Sarsa.

Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often ϵ-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s, a)$, are all zero but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this maximization bias.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-

learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads,

Are there algorithms that avoid maximization bias? To start, consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action. As we discussed above, there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them Q1(a) and Q2(a), each an estimate of the true value q(a), for all a 2 A. We could then use one estimate, say Q1, to determine the maximizing action A? = argmaxa Q1(a), and the other, Q2, to provide the estimate of its value, $Q2 = Q2$(argmaxa Q1(a)). This estimate will then be unbiased in the sense that $E[Q2(A)] = q(A)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate Q1(argmaxa Q2(a)). This is the idea of double learning. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrowtail +\alpha$$

If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For ex-

ample, an ϵ-greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates. A complete algorithm for Double Q-learning is given in the box below. This is the algorithm used to produce the results in . In that example, double learning seems to eliminate the harm caused by maximization bias. Of course there are also double versions of Sarsa and Expected Sarsa.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Lecture 8

Model-based Reinforcement Learning

Reinforcement learning algorithms are divided into two classes?model-free methods and model-based methods. These two classes differ by the assumption made about the model of the environment. Model-free algorithms learn a policy from mere interactions with the environment without knowing anything about it, whereas model-based algorithms already have a deep understanding of the environment and use this knowledge to take the next actions according to the dynamics of the model.

Model-free algorithms are a formidable kind of algorithm that have the ability to learn very complex policies and accomplish objectives in complicated and composite environments.

Trained agents have been able to beat top professional players. However, the biggest downside is in the huge number of games that need to be played in order to train agents to master these games. In fact, to achieve these results, the algorithms have been scaled massively to let the agents play hundreds of years' worth of games against themselves.

Well, until you are training an agent for a simulator, you can gather as much experience as you want. The problem arises when you are running the agents in an environment as slow and complex as the world you live in. In this case, you cannot wait hundreds of years before seeing some interesting capabilities. So, can we develop an algorithm that uses fewer interactions with the real environment? Yes. And, as you probably remember, we already tackled this question in model-free algorithms. The solution was to use off-policy algorithms. However, the gains were relatively marginal and not substantial enough for many real-world problems.

Let's first remember what a model is. A model consists of the transition dynamics and rewards of an environment. Transition dynamics are a mapping from a state,

s, and an action, a, to the next state, s'. Having this information, the environment is fully represented by the model that can be used in its place. And if an agent has access to it, then the agent has the ability to predict its own future.

In the following sections, we'll see that a model can be either known or unknown. In the former case, the model is used as it is to exploit the dynamics of the environment; that is, the model provides a representation that is used in place of the environment. In the latter case, where the model of the environment is unknown, it can be learned by direct interaction with the environment. But since, in most cases, only an approximation of the environment is learned, additional factors have to be taken into account when using it. Now that we have explained what a model is, we can see how can we use one and how it can help us to reduce the number of interactions with the environment. The way in which a model is used depends on two very important factors?the model itself and the way in which actions are chosen.

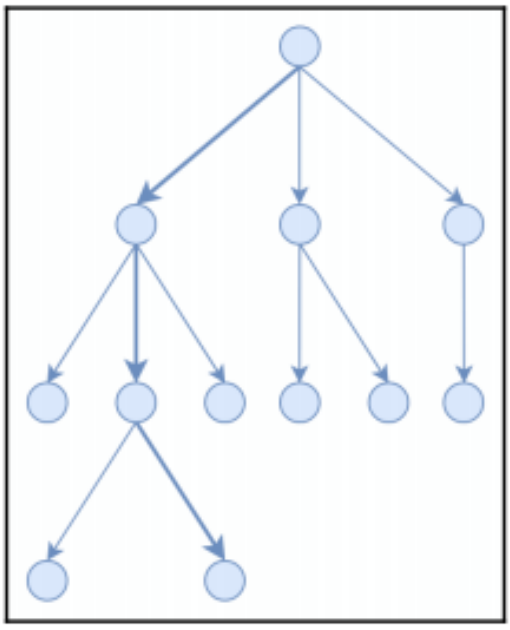
A known model

When a model is known, it can be used to simulate complete trajectories and compute the return for each of them. Then, the actions that yield the highest reward are chosen. This process is called planning, and the model of the environment is indispensable as it provides the information required to produce the next state (given a state and an action) and reward.

Planning algorithms are used everywhere, but the ones we are interested in differ from the type of action space on which they operate. Some of them work with discrete actions, others with continuous actions.

Planning algorithms for discrete actions are usually search algorithms that build a decision tree, such as the one illustrated in the following diagram:

The current state is the root, the possible actions are represented by the arrows,



and the other nodes are the states that are reached following a sequence of actions. You can see that by trying every possible sequence of actions, you'll eventually find the optimal one. Unfortunately, in most problems, this procedure is intractable as the number of possible actions increases exponentially. Planning algorithms used for complex problems adopt strategies that allow planning by relying on a limited number of trajectories.

An algorithm of these, adopted also in AlphaGo, is called Monte Carlo Tree Search (MCTS). MCTS iteratively builds a decision tree by generating a finite series of simulated games, while sufficiently exploring parts of the tree that haven't been visited yet. Once a simulated game or trajectory reaches a leaf (that is, it ends the game), it backpropagates the results on the states visited and updates the information of win/loss or reward held by the nodes. Then, the action that yields to the next state with the higher win/loss ratio or reward is taken.

On the opposite side, planning algorithms that operate with continuous actions involve trajectory optimization techniques. These are much more difficult

to solve than their counterpart with discrete actions, as they deal with an infinite-dimensional optimization problem.

Furthermore, many of them require the gradient of the model. An example is Model Predictive Control (MPC), which optimizes for a finite time horizon, but instead of executing the trajectory found, it only executes the first action. Doing so, MPC has a faster response compared to other methods with infinite time horizon planning.

Unknown model

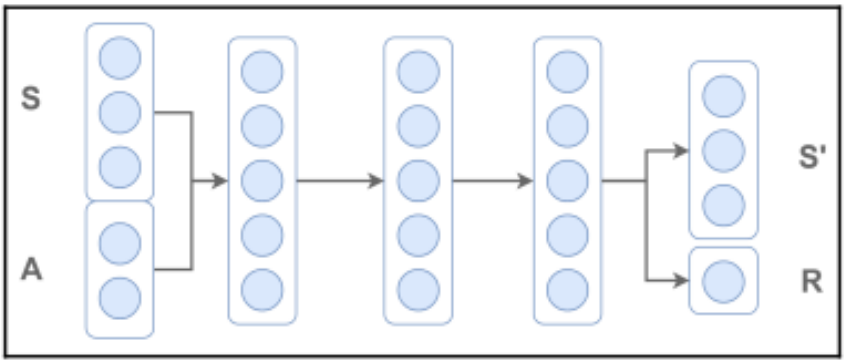
What should you do when the model of the environment is unknown? Learn it! Almost everything we have seen so far involves learning. So, is it the best approach? Well, if you actually want to use a model-based approach, the answer is yes, and soon we'll see how to do it. However, this isn't always the best way to proceed.

In reinforcement learning, the end goal is to learn an optimal policy for a given task. Previously in this chapter, we said that the model-based approach is primarily used to reduce the number of interactions with the environment, but is this always true? Imagine your goal is to prepare an omelet. Knowing the exact breaking point of the egg isn't useful at all; you just need to know approximately how to break it. Thus, in this situation, a model-free algorithm that doesn't deal with the exact structure of the egg is more appropriate.

However, this shouldn't lead you to think that model-based algorithms are not worth it. For example, model-based approaches outweigh model-free approaches in situations where the model is much easier to learn than the policy

The only way to learn a model is (unfortunately) through interactions with the environment. This is an obligatory step, as it allows us to acquire and create a dataset about the environment. Usually, the learning process takes place in a supervised fashion, where a function approximator (such as a deep neural network)

is trained to minimize a loss function, such as the mean squared error loss between the transitions obtained from the environment and the prediction. An example of this is shown in the following diagram, where a deep neural network is trained to model the environment by predicting the next state, s', and the reward, r, from a state, s and an action, a:

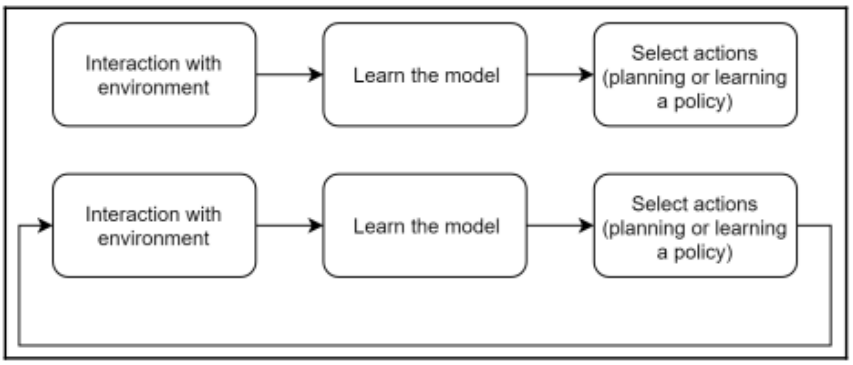


There are other options besides neural networks, such as Gaussian processes, and Gaussian mixture models. In particular, Gaussian processes have the particularity of taking into account the uncertainty of the model and are regarded as being very data efficient. In fact, until the advent of deep neural networks, they were the most popular choice.

However, the main drawback of Gaussian processes is that they are slow with large datasets. Indeed, to learn more complex environments (thereby requiring bigger datasets), deep neural networks are preferred. Furthermore, deep neural networks can learn models of environments that have images as observations.

There are two main ways to learn a model of the environment; one in which the model is learned once and then kept fixed, and one in which the model is learned at the beginning but retrained once the plan or policy has changed. The two options are illustrated in the following diagram:

In the top half of the diagram, a sequential model-based algorithm is shown,



where the agent interacts with the environment only before learning the model. In the bottom half, a cyclic approach to model-based learning is shown, where the model is refined with additional data from a different policy.

To understand how an algorithm can benefit from the second option, we have to define a key concept. In order to collect the dataset for learning the dynamics of the environment, you need a policy that lets you navigate it. But in the beginning, the policy may be deterministic or completely random. Thus, with a limited number of interactions, the space explored will be very restricted.