Contents

1 Course Introduction

"The essence of reinforcement learning is memoized (context-sensitive) search" -Barto and Sutton.

1.1 Motivating Reinforcement Learning

Suppose you want to program a robot to collect cans that are laying around a messy lab. To do this naively, we'd first need to build a computer vision system that recognizes cans, obstacles, and people. Then, we'd need to construct a map of the environment and figure out how we can instruct the robot to learn *where* it is in the map, i.e. the robot will need to localize itself. There's additional requirements as well, but let's consider an alternative. What if we were to use *reinforcement learning*? In this example, the reward can be the number of cans the robot collects, and the agent could simply learn to collect as many cans as possible through trial and error. In principle, it wouldn't even need a map of the lab. In the event that a new type or color of can starts appearing in the lab, a reinforcement learning system would still learn to pick these up, whereas a pre-learned perception system would fail in this scenario. In order to make reinforcement learning work, we'll need (i) a good *function approximation* if we want to learn from an on-board camera, as well as (ii) *planning* so that the agent can revisit parts of the lab it hasn't been to in a while to check for new cans. The *reward function* can simply be the total count of the number of cans collected at the *end* of each day; this requires a TD-based algorithm to handle the delayed feedback.

Course Introduction The promise of reinforcement learning is that an agent can figure out how the world works simply by trying things and seeing what happens. What's the difference between supervised learning, reinforcement learning, and unsupervised learning?

- In *supervised learning*, we assume the learner has access to labeled examples giving the correct answer.
- In *reinforcement learning*, the reward gives the agent an idea of how good or bad its recent actions were. While supervised learning is kind of like having a teacher that tells you what the correct *answer* is, reinforcement learning is kind of like having someone tell you what good *behavior* is, but they can't tell you exactly how to do it.
- In *unsupervised learning*, we try to extract the underlying structure of the data, i.e. data representation. Note that it's possible to use unsupervised learning to construct representations that make a supervised or RL system more performant.

Online Learning In RL, we focus on the problem of learning while interacting with an ever changing world. We do not expect our agents to compute a good behavior and then execute that behavior in an open-loop fashion. Instead, we expect our agents to sometimes make mistakes and refine their understanding as they go. The world is not a static place: we get injured, the weather changes, and we encounter new situations in which our goals change. An agent that immediately integrates its most recent experience should do well especially compared with ones that attempt to simply perfectly memorize how the world works. The idea of learning *online* is an extremely powerful if not defining feature of RL. Even the way that this course introduces concepts tries to reflect this fact. For example, bandits and exploration will be covered before we derive inspiration from supervised learning. Getting comfortable learning *online* requires a new perspective. Today, RL is evolving at what feels like breakneck pace: search companies, online retailers, and hardware manufacturers are exploring RL solutions for their day to day operations. There are convincing arguments to be made that such systems can be more efficient, save money, and keep humans out of risky situations. As the field evolves, it's important to focus on the fundamentals. E.g. DQN combines Q-learning, neural networks, and experienced replay. This course covers the fundamentals used in modern RL systems. By the end of the course, you'll implement a neural network learning system to solve an infinite state control task. We'll start with the multi-armed bandit problem: this introduces us to estimating values, incremental learning, exploration, non-stationarity, and parameter tuning.

2 Multi-armed Bandits

What distinguishes RL from other types of learning is that it uses training information that *evaluates* the actions rather than *instructs* by giving correct actions. Because we do not know what the *correct* actions are, this creates the need for active exploration to search for good behavior.

- Purely evaluative feedback indicates how good the action was, but not whether it was the best or the worst action possible.
- Purely instructive feedback indicates the correct action to take, independently of the action actually taken.

To emphasize: evaluative feedback depends *entirely* on the action taken, whereas instructive feedback is *independent* of the action taken. To start, we study the evaluative aspect of reinforcement learning in a simplified setting: one that does not involve learning to act in more than one situation. This is known as a *non-associative* setting. We can then take one-step closer to the full RL problem by discussing what happens when the bandit problem becomes associative, i.e. when actions are taken in more than one situation.

2.1 A k-armed Bandit Problem

SUppose 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. The objective is to maximize the expected total reward over some time-period e.g. $1,000$ action selections or *time-steps*. Through repeated actions, we aim to maximize reward by concentrating actions on the best levers.

The expected reward of an action In the k-armed bandit problem, each of the k actions has an expected reward given that the action is selected; we refer to this quantity as the *value* of an action. Denote the action selected at time step t as A_t, and the corresponding reward as R_t. The value of an arbitrary action a, denoted by $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) = \mathbb{E}\left[R_t | A_t = a\right] = \sum_r \Pr(r|a) r$$

If we knew the value of each action, the solution to the k-armed bandit problem is trivial: simply always select the action with highest value. But, we don't know the action values with certainy, instead we only have estimates: denote by $Q_t(a)$ the estimated value of action a at time step t. We hope that $Q_t(a) \approx q_*(a)$.

Greedy actions Suppose we maintain estimates of the action values. At any arbitrary time step, there is at least one action whose estimated value is greatest: these are called our *greedy* actions. Selecting one of these is akin to *exploiting* our current knowledge of the values of the actions. If we instead select a non-greedy action, then it is said we are *exploring*, because this enables us to improve our estimate of the non-greedy action's value. Although exploitation is the obvious thing to do in a one-shot game, exploration may produce a greater total reward in the long run. Since it's not possible to both explore and to exploit with any single action selection, we often refer to a "conflict" between these strategies.

Balancing exploit-explore In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, their respective uncertainties, and the number of remaining time steps. There are sophisticated methods for balancing exploit-explore for particular formulations of the k-armed bandit and related problems, but most of these methods make strong assumptions

about stationarity and prior knowledge that is often violated in practice. In this course, we focus not on an optimal balance between exploration and exploitation, but instead we worry about simply balancing them at all.

2.2 Action-value Methods

Let's consider methods for estimating the values of actions and for using these estimates to make subsequent action selection decisions; we collectively call these *action-value* methods. We defined the true value of a reward as the mean reward when that action is selected, so it is natural to estimate this by averaging rewards actually received:

$$Q_t(a) \coloneqq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}. \tag{1}$$

Note that if the denominator is zero, we may instead define $Q_t(a)$ as some default value e.g. zero. Note that as the denominator tends to infinity, by the weak law of large numbers, $Q_t(a) \xrightarrow{p} q_*(a)$, i.e. the sample mean converges in probability to the population mean. Equation 1 defines a *sample-average* method for estimate action values.

An action-selection rule The simplest action-select rule is to chose among the set of actions with the highest estimated value. If there is more than one, we may select among them in an arbitrary way.

$$A_t \coloneqq \arg\max_a Q_t(a).$$

Greedy action selection exploits our current knowledge to maximize immediate reward, and doesn't spend any time sampling seemingly inferior actions to see if they indeed may be better.

ϵ-greedy Methods One alternative is to behave greedily most of the time, but every once in a while, say with probability ϵ, to select randomly from among all the actions with equal probability, independently of the action-value estimates. This forms a class of ϵ-greedy methods. An obvious advantage here is that in the limit as the number of steps increases, all actions will be sampled an infinite number of times,[1] and this ensures that $Q_t(a) \xrightarrow{p} q_*(a)$ for each action. Whence, the probability of selecting the optimal action converges to greater than $1 - \epsilon$, i.e. to near certainty.[2]

2.3 The 10-armed Testbed

We propose a testbest on which to assess the relative effectiveness of the greedy and ϵ-greedy action-value methods. Here, our k-armed bandit problem is for $k = 10$. We replicate across B bandit problems (e.g. $2,000$): for each bandit problem, the action values $q_*(a)$ are selected from a standard normal distribution; then, the reward for each action is drawn from a normal distribution with unit variance and mean $q_*(a)$. For any learning method, we can measure its performance and behavior as it improves with experience for e.g. 1,000 time steps when applied to one instance of the bandit problem. Repeating for B runs, each with a different bandit problem, we can obtain a measure of the learning algorithms average behavior.

[1] To see this, realize that with an ϵ-greedy method each action has probability at least $\epsilon > 0$ of being chosen at each time-step. For an arbitrary action a, the expected number of times we sample the action is given by $\sum_{t=1}^{\infty} \underbrace{\Pr(A_i = a)}_{\geq \epsilon}$ which diverges to ∞.

[2] The reason for this last statement is quite obvious. Our strategy is to choose $A_t = \arg\max_a Q_t(a)$ with probability $1 - \epsilon$ and to sample among all possible actions with probability ϵ, i.e. we choose A_t with probability $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} > 1 - \epsilon$. By weak law of large numbers, in the limit we will have estimated each value precisely and so we have estimated the ordering of $Q_t(a)$ over all possible a, and so we are indeed selecting the optimal action with probability strictly greater than $1 - \epsilon$.

Advantages of ϵ-greedy methods It depends on the task. If we have noisy rewards (i.e. they are drawn from a distribution with high variance), then it will take *more* exploration to find the optimal action, and so we would expect the ϵ-greedy method to fare better relative to the greedy method. If the reward variances are zero, then the greedy method would know the true value of each action aftery trying it once; in this case the greedy algorithm can quickly find the optimal action and then never explore. However, suppose we relax the problem a bit: if the bandit task is non-stationary, i.e. the true values of the actions evolve over time, then exploration is *required* even in the deterministic case to ensure that one of the non-greedy actions has not changed to become better than the greedy one.

2.4 Incremental Implementation

The action-value methods discussed so far all estimate action values as sample averages of observed rewards. How can we compute these efficiently? In particular, we want a running average that gets updated with constant memory and constant work required per update. To simplify notation let us concentrate on a single action. Let R_i denote the reward received after the ith selection *of this action*, and let Q_n denote the estimate of its action value after it has been selected $n-1$ times, i.e.

$$Q_n \coloneqq \frac{R_1 + R_2 + \ldots + R_{n-1}}{n-1}.$$

Note that a naive implementation would require more and more memory as more rewards are encountered. Instead, realize that given Q_n and the n-th reward R_n, the new average of all n rewards can be computed by

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i && \text{definition of } Q_{n+1} \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) && \text{breaking apart summation} \\
&= \frac{1}{n}\left(R_n + \frac{n-1}{n-1}\sum_{i=1}^{n-1} R_i\right) && \text{multiplying by 1 to make an average appear} \\
&= \frac{1}{n}\left(R_n + (n-1)\,Q_n\right) && \text{By definition of } Q_n \\
&= \frac{1}{n}\left(R_n + nQ_n - Q_n\right) && \text{factoring out terms} \\
&= Q_n + \frac{1}{n}\left[R_n - Q_n\right] && \text{algebraic simplification} \qquad (2)
\end{aligned}$$

What's nice is that 2 holds even for $n = 1$, in which case we get that $Q_2 = R_1$ for arbitrary Q_1. Notice that we only have to store Q_n and n, and the perform the work required by 2 at each time step. What's interesting about this update rule is that it actually reoccurs frequently throughout this book:

$$\texttt{NewEstimate} \leftarrow \texttt{OldEstimate} + \texttt{StepSize}\left[\underbrace{\texttt{Target} - \texttt{OldEstimate}}_{error}\right]$$

The expression $\texttt{Target} - \texttt{OldEstimate}$ is an *error* in our estimate which is reduced by taking a step towards the "target", which is presumed to indicate a desireable direction in which to move albeit may be subject to noise. In the case of our sample-average action-value method, the target is the n-th reward.

Step-size parameter Note that there is a parameter `StepSize` that appears in 2 which can depend upon the time step. In processing the n-th reward for action a, the sample-average action-value method uses the step size parameter $\frac{1}{n}$. In general, we denote the step size parameter by $\alpha_t(a)$.

A simple bandit algorithm We now write out pseudo-code for a complete bandit algorithm using incremental updates for our sample-averages and an ϵ-greedy action selection strategy. Here, the function

Algorithm 1: A simple ϵ-greedy k-armed bandit algorithm

for $a = 1, \ldots, k$ **do**
  $Q(a) \leftarrow 0$
  $N(a) \leftarrow 0$
end
while *True* **do**
  $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\epsilon \quad (*\text{breaking ties randomly}) \\ \text{random action} & \text{with probability } \epsilon \end{cases}$
  $R \leftarrow \texttt{Bandit}(A)$
  $N(A) \leftarrow N(A) + 1$
  $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$
end

`Bandit`$(\cdot)$ accepts an action as argument and returns a corresponding reward.

2.5 Tracking a Nonstationary Problem using Exponentially Weighted Moving Averages

The sample-average action-value methods described so far are only appropriate for stationary bandit problems, i.e. ones in which the reward probabilities don't change over time. In cases of non-stationarity, it makes sense to give more weight to recent rewards than to long-past rewards. One of the ways to accomplish this is by using a *constant* step size parameter. For example we can change 2 to something like

$$Q_{n+1} \coloneqq Q_n + \alpha\left[R_n - Q_n\right] \tag{3}$$

where the step size $\alpha \in (0, 1]$ is constant. Realize that this results in Q_{n+1} being a weighted average of past rewards, given an initial estimate Q_1:

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha\left[R_n - Q_n\right] && \text{by equation 3} \\
&= \alpha R_n + (1-\alpha)\, Q_n && \text{re-arranging terms} \\
&= \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] && \text{substituting for } Q_n \text{ based on pattern observed above} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} && \text{distributing } (1-\alpha) \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + \ldots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 && \text{iteratively plugging in for } Q_i \\
&= \underbrace{(1-\alpha)^n Q_1}_{f(\text{initial value})} + \underbrace{\sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i}_{\text{weighted average of rewards}} \ . && (4)
\end{aligned}$$

Note that the sum of the weights is given by $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$. Note that crucially, the weight $\alpha(1-\alpha)^{n-i}$ given to the reward R_i depends on how many rewards ago, $n-i$, that it was observed. Since $\alpha \in (0, 1]$, the quantity $1-\alpha < 1$, thus the weight given to R_i decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on $1-\alpha$.[3] Further, our initial value matters less and less the more data we obtain.

[3] If $1-\alpha = 0$, then all the weight goes to the very last reward R_n, because of the convention that $0^0 = 1$.

Varying Step-Size Parameter from step-to-step Let $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the n-th selection of action a. The choice of $\alpha_n(a) = \frac{1}{n}$ results in the sample-average action-value strategy, which is guaranteed to converge to the true action-values by the weak law of large numbers. But, convergence isn't guaranteed for all choices of the sequence $\{\alpha_n(a)\}$. To ensure convergence with probability 1, we require that

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty. \tag{5}$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions (i.e. that we revisit actions infinitely many times), and the second condition guarantees that eventually, the step sizes become small enough to assure convergence. Note that both conditions are met for the sample-average case of $\alpha_n(a) = \frac{1}{n}$, but *not* for the case of the constant step size parameter $\alpha_n(a) = \alpha$. In the latter case, the second condition isn't met, indicating that the estimates may never completely converge but instead continue to vary in response to the most recently received rewards; this is actually *desireable* in the nonstationary environment.

2.6 Optimistic Initial Values

Our methods discussed thus far depend to some extent on initial action-value estimates $Q_1(a)$, i.e. we are *biased* by our initial estimates. For sample-average methods, the bias dissappears once all actions have been selected at least once. But, for methods with constant α, the bias is permanent albeit decreasing over time as given by equation 4. Although this kind of bias can sometimes be helpful, the downside is that the initial estimates become a set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to encode prior knowledge about what levels of rewards can be expected.

Encouraging early exploration through large initial action values Suppose that instead of setting initial action valus to zero in the 10-arm testbed, that we instead set them to +5. Note that based on our simulation we drew $q_*(a)$ from a standard normal, and so an initial estimate of +5 is wildly optimistic. What happens is that whichever action(s) are initially selected, the reward will always be less than the initial estimates and the algorithm will be encouraged to continue exploring all possibilities at least once. The system does a fair bit of exploration even if the greedy action-value strategy is employed! This technique can be particularly effective on stationary problems: initially the optimistic method performs worse than ϵ-greedy because it explores more often, but eventually it performs *better* because its exploration decreases with time. Note that the method is ill-suited to nonstationary problems since the drive for exploration is temporary. Note that in practice, it may not be obvious what an "optimistic" initial value is, since we may not know what is a maximal reward.

2.6.1 Unbiased constant-step-size trick

Sample averages are not satisfactory because they do not perform well on nonstationary problems. Consider a step size of

$$\beta_n = \alpha / \bar{o}_n,$$

to process the n-th reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and $\bar{o}_n$ is a trace of one that starts at 0:

$$\bar{o}_n \coloneqq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}) \qquad \text{for } n \geq 0, \quad \text{with } \bar{0}_0 = 0.$$

2.7 Upper-confidence bound action selection

Optimism in the face of uncertainty We require exploration since there is always uncertainty about the accuracy of our action-value estimates. Greedy actions have been defined as the set of actions which look best at present, but it's possible that other actions could be better. The ϵ-greedy action selection method forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are near greedy or particularly uncertain. Wouldn't it be nice if we could select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates? One effective method is upper-confidence bound action selection:

$$A_t \coloneqq \arg\max_a \left[\underbrace{Q_t(a)}_{\text{exploit}} + c \underbrace{\sqrt{\frac{\ln t}{N_t(a)}}}_{\text{explore}} \right]$$

where $c > 0$ controls the degree of exploration. If $N_t(a) = 0$ we consider a to be a maximizing action. The fundamental idea is that the square root term is a measure of uncertainty or variance in the estimate of a's value. The quantity being maximized over is an upper bound on the true value of action a, with c determining the confidence level. Each time some a is selected, its uncertainty is presumably reduced: $N_t(a)$ increments, and as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases, but $N_t(a)$ doesn't, and because t appears in the numerator the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but they are unbounded; i.e. all actions will eventually be selected, but actions with lower value estimates, or those that have already been selected frequently, will be selected with decreasing frequency over time.

2.8 Sequential Decision Making with Evaluative Feedback

In RL, the agent generates its own training data by interacting with the world. The agent must learn the consequences of their actions through trial and error, rather than being told what correct actions are. Let's focus on the evaluative asepect of RL; in particular we will formalize the problem of decision making under uncertainty using k-armed bandits, and we will use this as a foundation to discover fundamental RL concepts such as rewards, timesteps, and values.

Motivating Example: Medical Treatments Imagine a medical trial where a doctor wants to measure the effectiveness of three different treatments. Whenver a patient comes into the office, the doctor prescribes a treatment at random. After a while, the doctor notices that one treatment seems to be working better than the others. The doctor is now faced with a decision: should he/she stick with the best-performing treatment, or continue with the randomized study? If the doctor only prescribes one treatment, they can no longer collect data on the other two, and what if its the case that one of the other treatments is actually better (it only happens to appear worse due to random chance)?

Making RL work in Real Life You want to shift your priorities.

Important to RL, but should be Less Important in Real Life	Important in Real Life
Temporal credit assignment	Generalization across observations
Control environment	Environment controls
Computational efficiency	Statistical efficiency
State	Features
Learning	Evaluation
Last policy	Every policy

In RL, computational efficiency is paramount, since we are running simulations and we have as many observations as we can compute; but in real life we are often more focused on statistical efficiency since the number of observations are fixed. As another example, we may have a 1 megapixel camera, but do we really need to capture all of that information as part of our state? Perhaps a higher representation suffices. Lastly, in RL we only care about the last policy since we're running our simulations for a very long time; but in the real world every observation involves some interaction with the world, so we want the performance to always be pretty good, i.e. we care about the entire trajectory of policies.

Contextual Bandits Suppose that there are several different k-armed bandit tasks, and that on each step you confront one of these chosen at random. The bandit task is changing randomly from step to step. This would appear to the agent as a single, non-stationary k-armed bandit task whose true action values change randomly from step to step. Unless the true action values change very slowly, the methods described so far won't work. But now imagine that when a bandit task is selected for you on each round, you are given a distinctive clue about its identity (but not the action values), e.g. you are facing an actual slot machine that changes color of its display as it changes the action values.[4] Now, you can learn a policy associating each task, signaled by the color you see, with the best action to take when faced with that task. This is an example of an *associative search* task, so called because it involves both trial-and-error learning to *search* for the best actions as well as the *association* of these actions with the situations in which they are best. Associative search tasks are often called *contextual bandits* in the literature.

Thompson Sampling There is a family of Bayesian methods which assume an initial distribution over the action values and then update the distributionexactly after each step (assuming that the true action values are stationary). In general, the update computations are complex, but for conjugate priors they are straightforward. One possibility is to then select actions at each step according to their *posterior* probability of being the best action. This method, sometimes called *posterior sampling* or *Thompson sampling*, often performs similarly to the best of the distribution-free methods we've gone over so far. In the Bayesian setting, it's conceivable to compute the *optimal* balance between explore/exploit. For any possible action, one can compute the probability of each possible immediate reward and the resultant posterior distributions over action values. This *evolving* distribution becomes the *information state* of the problem. Given a horizen of T time-steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for T time steps. Given the assumptions, the rewards and the probabilities of each possible chain of events can be determined, and one can simply pick the best. The problem is computationally intractable, however, as the tree of possibilities grows exponentially.

3 Finite Markov Decision Processes

Suppose we have a rabbit who prefers carrots over broccoli, each having reward +10 and +3 respectively. In the first time step, the rabbit can move *right* to get a carrot or left to get broccoli: clearly it prefers to move right to get the highest reward. But at a later time step the situation is reversed, where the rabbit can now move *left* to get a carrot or right to get broccoli. Clearly the rabbit should take the action of moving left. Or, consider another example where on the other side of the carrot is a predator: now the rabbit should prefer to take the broccoli since it'll have a better chance of survival in the long run (what looks good in the short-term isn't always what's best in the long-term). Note that the k-armed bandit does not account for the sequential decision making process as described here.

[4] In contextual bandits, we observe some features x. Perhaps its the geolocation of the user, the profile based on the way the user has behaved in the past, maybe it's features of possible actions which are available as well.

MDPs The problem involves evaluative feedback, as in the case of bandits, but additionally there is an associative aspect of choosing different actions in different situations. MDPs are a classical formalization of sequential decision making, where actions influence not only immediate rewards but also subsequent situations, or states, and through those future rewards. In MDPs we estimate the value $q_*(s, a)$ of each action a in each state s, or we estimate the value $v_*(s)$ of each state given optimal action selections. In this new framework, the agent and environment interact with each other. At each time step, there is a set of possible states the agent can be in $S_t \in \mathcal{S}$, and there is correspondingly a (subset of) actions they make take $A_t \in \mathcal{A}(S_t)$.

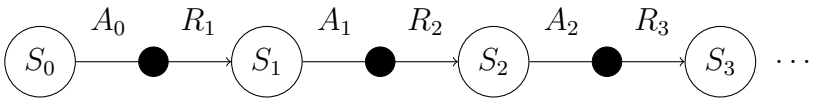


Figure 1: Here we represent the sequential nature of an MDP. At each time step, the agent finds itself in a given state. It takes an action which results in some reward. The process then repeats itself (perhaps indefinitely).

Dynamics of an MDP As in bandits, the outcomes of our action-state pairs are stochastic and so we may use the language of probability. When the agent takes an action in a state, there are many possible next states and rewards. The transition dynamics function P formalizeds this notion. Given a state s and action a, p instructs us of the joint probability of the next state s' and reward are, i.e. $\Pr(s', r|s, a)$. In this course, we assume the set of states, actions, and rewards are finite, although we will learn about algorithms that can handle infinite and uncountable sets. Since p is a probability distribution, it must be non-negative and its sum over all possible next states and rewards must equal one. I.e. $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, and

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} \Pr(s', r|s, a) = 1, \quad \forall s \in \mathcal{S} \ a \in \mathcal{A}(s)$$

Note that the future state and reward depend only on the current state and action. In writing our probability in this way, we have defined a Markov process, and it intuitively means that the present state is sufficient and that remembering earlier states or trajectories would not improve predictions about the future.

3.1 Returns and Episodes

An agents goal is to maximize the cumulative reward it receives in the long run. If the sequence of rewards received after time step t is denoted by $R_{t+1}, R_{t+2}, \ldots$, each a random variable, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize *expected* return, where the return at time step t, denoted by G_t, is some specific function of the reward sequence. In the simplest case the return is the sum of rewards:

$$G_t \coloneqq R_{t+1} + R_{t+2} + \ldots + R_T \tag{6}$$

where T is a final time step. This approach really only makes sense in applications where there is a notion of a final time step, i.e. when the agent-environment interaction naturally breaks into subsequences, which we call *episodes*. Example episodes are plays of a game, trips through a maze, or any sort of repeated interaction; episodes end in a special *terminal state*, followed by a reset to a standart starting state or to a sample from a standard distribution of starting states. No matter how the last episode finished, e.g. win or lose, the next episode begins independently of how the previous one ended. Tasks with episodes of this kind are called *episodic* tasks; in such tasks we sometimes distinguish the set of all non-terminal states $\mathcal{S}$ from the set of all states plus the terminal state, denoted by $\mathcal{S}^+$. Note that the time of termination T is a random variable that varies from episode to episode.

Effect of adding a constant to all rewards in an episodic task Note that if we add a constant to all rewards in an episodic task, then this may change the set of optimal policies. To see this, realize that adding a constant to the reward signal can make *longer* episodes more or less advantageous, depending on whether the constant is positive or negative.

Examples of episodic tasks One example of an episodic task is a game of chess; it always ends in either a checkmate, draw, or resignation, and a single game constitutes an episode. Each game starts from the same start state with all pieces reset. The state is given by the position of all the pieces on the board, and the actions are all the legal moves. We could define the reward as +1 for winning and zero for all other moves. Another example is the case of an agent learning to play a video game, in which the player gets a point for collecting treasure blocks and dies when they touch an enemy. This game is clearly an episodic MDP: the agent tries to get a high score, collecting as many points as possible before the game ends. The state is an array of pixel values corresponding to the current screen. There are four actions: up, left, down, right. The agent gets a reward of +1 every time they collect a treasure block. An episode ends when the agent touches one of the enemies. Regardless of how the episode ends, the next episode begins in the same way, with the agent at the center of the screen with no enemies present.

3.2 Continuing Tasks

In many cases, the agent-environment interaction doesn't break naturally into identifiable episodes, but instead goes on continually without limit; e.g. a process-control task, or an application to a robot with a long life span. We call these *continuing tasks*. The definition of return as in equation 6 is problematic because the final time step would be $T = \infty$, and the return (which we are trying to maximize) could itself be divergent. For example, suppose an agent receives a reward of +1 at each time step. To address this issue, we use discounting.

Effect of adding a constant to all rewards in continuing tasks on the set of optimal policies Note that if we add a constant to all rewards in a continuing task, then the agent will accumulate the same amount of extra reward independent of its behavior, and so the set of optimal policies will not change.

Discounting Returns In this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. I.e. it selects A_t to maximize the expected *discounted return*:

$$G_t \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{7}$$

where $\gamma \in [0, 1]$ is called the *discount rate*. This 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.[5] Note that if $\gamma < 1$, then our discounted reward in equation 7 is convergent so long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is myopic and only focuses on maximizing immediate reward: its objective becomes choose A_t so as to maximize only R_{t+1}. If it happened to be the case that the agent's actions influenced *only* the immediate reward, not future rewards as well, then a myopic agent could maximize 7 by separately maximizing each immediate reward. But, in general acting to maximize the immediate reward can reduce access to future rewards so that the return is reduced. On the other hand, as $\gamma \rightsquigarrow 1$, the return objective takes future rewards into account more strongly and the

[5] Intuitively, this makes sense: a dollar today is worth more to you than a dollar in a year.

agent becomes farsighted. Note that

$$\begin{aligned} G_t &\coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots \right) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{8}$$

This works for all time steps $t < T$, even if termination occurs at $t+1$, so long as we define $G_T = 0$. Even though our return 7 is an infinite sequence, it is finite if the reward is non-zero and constant and if $\gamma < 1$. E.g. if the reward is a constant $+1$ then $G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$.

Proof of convergence Let's examine why it's the case that $G_t < \infty$ if $\gamma < 1$. Assume that $R_{\max}$ is the maximum reward our agent can achieve at any time step. We can upper bound the return G_t by replacing every reward with $R_{\max}$, and since it's just a constant we can then pull it out of the sum. We're left with a (scaled) geometric series which converges since $\gamma < 1$:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k R_{\max} = R_{\max} \sum_{k=0}^{\infty} \gamma^k = R_{\max} \frac{1}{1-\gamma}.$$

Examples of Continuing tasks Consider a smart thermostat which regulates the temperature of a building. This can be formulated as a continuing task since the thermostat never stops interacting with the environment. The state could be the current temperature, along with details of the situation like the time of day and the number of people in the building. There are just two actions: turn on the heater or turn it off. The reward can be -1 every time someone has to manually adjust the temperature and zero otherwise. To avoid negative reward, the thermostat would learn to anticipate the user's preferences. Another example is an agent scheduling jobs on a set of servers. Suppose we have three servers used by reinforcement learning researchers to run experiments. Researchers submit jobs with different priorities to a single queue. The state is a number of free servers, and the priority to the job at the top of the queue. The actions are to reject or accept the job at the top of the queue if a server is free. Accepting the job runs it and yields a reward equal to the job's priority. Rejecting a job yields a negative reward proportional to the priority, and sends the job to the back of the queue. Note that the agent must be careful about scheduling low-priority jobs, since this could prevent high priority jobs from being scheduled later. The servers become available as they finish their jobs. The researchers continually add jobs to the queue, and the agent accepts or rejects them; the process never stops! It's well-described as a continuing task.

3.3 Examples of MDPs

The MDP framework can be used to formalize a wide variety of sequential decision-making problems. Consider our recycling robot which collects empty soda cans in an office environment: it can detect soda cans, pick them up using a gripper, and then drop them off in a recycling bin. The robot runs on a rechargeable battery, and its objective is to collect as many cans as possible. To formulate this problem as an MDP, we start with the states, actions, and rewards. Suppose the sensors can only distinguish between two levels of battery charge: `high` and `low`, and these represent the robot's state. In each state, the robot has three choices: it can search for cans for a fixed amount of time, it can remain stationary and wait for someone to bring it a can, or it can go to the charging station to recharge its battery. We'll only allow charging from low battery state since charging otherwise is pointless. Now, let us consider transition dynamics as per figure 2.

Extensibility of MDPs The MDP framework is highly extensible, and can be used in many applications. States can be low-level sensory readings, for example, in the pixel values of the video frame. They could

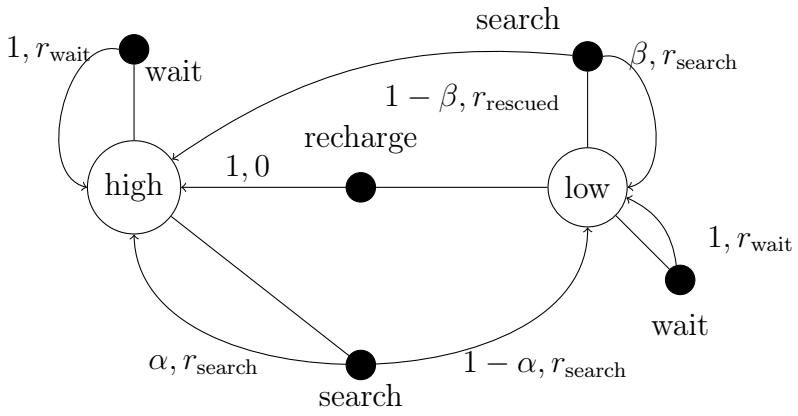


Figure 2: The *state* of the robot's battery is indicated by open nodes in the graph, demarcated by "high" and "low". Actions are denoted by coloured nodes, and an edge between a state-action pair indicates that it is possible to execute said action from the corresponding state. After taking an action we observe a reward, and also a state change with some specified probability; for each transition we indicate the probability of transition and also the reward as a comma separated tuple alongside the corresponding edge. Hypothetical reward values: $r_{\text{wait}} = 1$, $r_{\text{search}} = 10$, $r_{\text{rescued}} = -20$. In our problem, $\alpha, \beta \in [0, 1]$ describe transition probabilities when searching from a high or low battery state respectively.

also be high-level such as object descriptions. Similarly, actions could be low level, such as the wheel speed of the robot, or they could be high level, such as "go to the charging station". Time steps can be very small or very large, for example they can be one millisecond or one month; they can either be fixed intervals of time or successive stages of decision making.

Pick-and-place task Suppose we want to use RL to control a robot arm in which the goal of the robot is to pick up objects and place them in a particular location. There are many ways we could formalize this task, but here's one. The *state* could be the readings of the joint-angles and velocities. The *actions* could be the voltages applied to each motor. The *reward* could be +100 for successfully placing each object. But, we also want the robot to use as little energy as possible, so let's include a small negative reward corresponding to energy used, e.g. -1 for each unit of energy consumed.

Reward functions Consider a robot learning to walk. The reward could be proportional to the forward motion. Lurching forward clearly maximizes immediate reward, however this action causes the robot to fall over. If the robot instead maximizes total forward motion instead, it would walk quickly but carefully. To teach an agent how to escape from a maze, the reward might be -1 for each time step that passes prior to escape, encouraging the agent to escape as quickly as possible. For an agent learning to play checkers or chess, the natural rewards might be $+1$ for winning, -1 for losing, and 0 for drawing and for all non-terminal positions. In a stock market, we can use monetary rewards to optimize: since buying actions cost dollars, selling actions generate dollars, and the trade-offs are quite simple.

- **Goal rewarding encoding:** define a state where the goal is achieved as having $+1$ reward and all other states yield zero reward.
- **Action penalty representation:** penalize the agent with a -1 each step in which the goal has not been achieved.

Both these formulations achieve optimal behavior by reaching the goal, eventually. But, what the agent does along the way is subtly different. Goal rewarding encoding doesn't really encourage the agent to get to the goal with any sense of urgency, and action penalty representation runs into serious problems if there is some small probability of getting stuck and never reaching the goal. Both schemes can lead to big problems for goals with *really* long horizons, e.g. consider encouraging an agent to win a Nobel Prize. Just

giving an agent a reward for attaining the ultimate goal is quite a tough sell. Some intermediate rewards like doing well on a science test or earning tenure could make a difference in orienting the agent in the right direction.

3.4 The reward hypothesis

The hypothesis states "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)". There are three ways to think about creating intelligent behavior.

- Give a man a fish. If we want a machine to be smart, we program it with the behavior we want it to have. But, as new problems arise, the machine won't be able to adapt to new circumstances. It requires us to always be there providing new programs.
- Teach a man to fish. This is supervised learning: provide training examples and the machine writes its own program to match those examples. It learns, so long as we have a way to provide training examples, our machine will be able to write its own programs, and that's progress. But situations change.
- Give a man a taste for fish. This is reinforcement learning. The idea is that we don't have to specify the mechanism for achieving the goal. We can just encode the goal and the machine can design its own strategy for achieving it.

Rewards are easy to define if there is an underlying common currency The reward hypothesis states that goals can be thought of as maximization of the expected value of the cumulative sum of a received scalar reward signal. This suggests two main branches of research: (i) figure out what rewards agents should optimize, and (ii) design algorithms to maximize the expected cumulative reward. How can we define rewards? Sometimes its not natural: consider designing a reinforcement learning agent to control a thermostat. Turning on the heat or air conditioning costs energy, but not turning it on causes discomfort in the occupants, so there is no common currency. We could try translating both into dollars, but that's not so natural, e.g. "how much are you willing to pay to move the temperature a little closer to your comfort zone?".

Ways of defining rewards Programming is one common way of defining rewards for a learning agent: a person sits down and does the work of translating goals and behaviors into reward values. We can write a program that takes in states and outputs rewards. We can also specify rewards by example; one interesting version of this is inverse reinforcement learning. In IRL, a trainer demonstrates desired behavior, and the learner tries to figure out what rewards the trainer must have been maximizing that makes said behavior optimal. So whereas RL is about going from rewards to behaviors, IRL is about going from behaviors to rewards. Once identified by IRL, these rewards can be maximized in other settings, resulting in powerful *generalization* between environments. One shortcoming of the reward hypothesis is that it's hard to use it to explain risk-averse behavior, in which an agent chooses actions that might not be best on average but instead minimize the chance of a worst case outcome.

What if we want a balance of actions? What about when the desired behavior isn't to do the best thing all the time but instead to do a bunch of things in some balance? Imagine a pure reward maximizing music recommendation system: it would figure out your favorite song and then play it for you all the time, and that's not what we want. Perhaps there are ways to expand the state space so that the reward for playing a song is scaled back if that song has been played recently. "An animal gets a lot of value from drinking, but only if it's thirsty".

3.5 Policies and Value Functions

Almost all RL algorithms involve estimating *value functions* - functions of states (or 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). Here, "how good" means expected cumulative future rewards. Because the rewards that the agent can expect to receive in the future depend on what actions it will take, value functions are defined with respect to particular ways of acting called policies. A *policy* is a mapping from states to probabilities of selecting each possible action; critically, policies can only depend on the current state.

Deterministic Policies In their simplest form, a policy could be a deterministic mapping from states to actions; note that the same action could be selected from multiple states, and some actions may not be selected in any state. It's important that policies depend only on the state, it defines all the things relevant to take an action, and not other things like time.[6] It's best to think of this as a requirement on the state, and not as a limitation on the agent. In MDPs, we assume that the state encodes all the information required for decision-making.

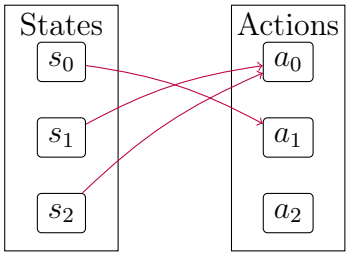


Figure 3: We draw out a deterministic policy, which we denote by $\pi(s) = a$. In this example, the policy selects action a_1 in state zero, and action a_1 in states one and two. Notice that some actions are selected from more than one state, and that other actions are not selected at all.

State	Action
s_0	a_1
s_1	a_0
s_2	a_0

Table 1: A deterministic policy can also be represented by a table. Each row describes the action chosen by the policy π in each state.

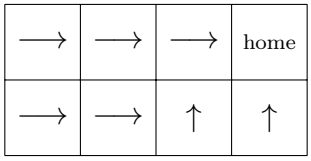


Figure 4: Consider a second deterministic policy in which an agent moves toward its home on a grid. The states correspond to the locations on the grid, and the actions move the agent up, down, left, and right. We've demarcated one possible policy using arrows. Each arrow tells the agent which direction to move in each state.

[6]For example, suppose we can take two actions: left or right. A valid policy could be: at each time step go left with probability 1/2 and right with probability 1/2. An alternative strategy that is *not* a policy is to alternate between left and right steps; critically, the action chosen at each time step depends on something other than the state (in particular, the past action), and so this is *not* a policy. If we believed that alternating actions between left and right would yield a higher return, then we could encode the last action as part of our state variable!

Stochastic Policies In general, a policy assigns probabilities to each action in each state; i.e. multiple actions may be selected each with some non-zero probability. 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 $|$ reminds us that it defines a conditional probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. Note that because $\pi(a|s)$ defines a probability distribution, we require that (i) $\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1$ and (ii) $\pi(a|s) \geq 0$.

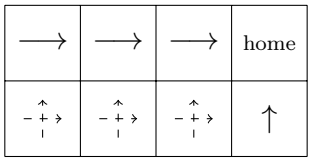


Figure 5: Here, we have a stochastic policy. We've indicated the randomness by drawing two possible actions with dashed lines for some of our states. To be concrete, the lower-left states each select an action of "up" or "right" each with probability 1/2. Notice that the stochastic policy will require the *same* number of steps to reach the house as the deterministic policy previously shown.

3.5.1 (Action/State) Value Functions

Many problems involve some sort of delayed reward; e.g. a store manager could lower their prices and sell off their entire inventory to maximize short-term gain. But, they might do better in the long-run by maintaining inventory to sell when demand is high. In RL, rewards capture the notion of short-term gains, but our objective is to learn a policy that achieves the most rewards in the long run; a value function formalizes what this means.

State value function The *state-value function* of a state s under a policy π, denoted by $v_\pi(s)$, is the expected return when starting in s and following π thereafter.[7] For MDPs, we can define this formally by

$$v_\pi(s) \coloneqq \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \qquad \text{for all } s \in \mathcal{S}. \tag{9}$$

We define the value of a terminal state (if there are any) to be zero. v_π is the *state-value function for policy* π. Crucially, value functions allow an agent to query the quality of its current situation as opposed to waiting to observe the long-term outcome; they summarize all possible futures by averaging over returns. Since ultimately we care about learning a good policy, the value function enables us to compare the quality of different policies!

Intuition on state-value functions Suppose there is an agent playing a game of chess, which is clearly an episodic MDP. The state is given by the positions of all the pieces on the board, the actions are the set of legal moves, and termination occurs when the game ends in either a win, loss, or draw. We could define a reward as $+1$ when our agent wins and zero for all other moves. Realize that the reward doesn't tell us much about how well the agent is playing during the match: we have to wait till the end of the episode/game to realize the non-zero reward. However, the *value function* lets us see much more; since the state value is equal to the expected sum of future rewards. In this case, we effectively defined reward as an indicator for winning, whence taking an expectation over an indicator variable we realize a probability, i.e. the value function in this instance encodes the probability of winning the game if we follow the policy π. Note that in this game, the opponent's move is part of the state transition. E.g. the environment moves both the agent's piece, and the opponent's piece, and this puts the board into a new state s'. An *action-value function* would allow us to assess the probability of winning for each possible move given we follow the policy π for the rest of the game.

[7] Note that because long-run outcomes depend on repeated actions and outcomes, value functions only make sense in the context of policies.

Action value function We can also define the value of taking action a in state s under a policy π, denoted by $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s,a) \coloneqq \mathbb{E}_\pi\left[G_t|S_t=s,A_t=a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s,A_t=a\right]. \tag{10}$$

We refer to this as the *action-value function for policy* π. Note that by the law of total probability, $v_\pi(s) = \mathbb{E}_\pi\left[G_t|S_t=s\right] = \sum_a \pi(a|s)q_\pi(s,a)$. We could also equivalently characterize the state-action function for a given policy π by

$$q_\pi(s,a) = \sum_{s',r}\Pr(s',r|s,a)\left[r+\gamma v_\pi(s')\right].$$

Intuition for action value functions Let's consider a continuing MDP.

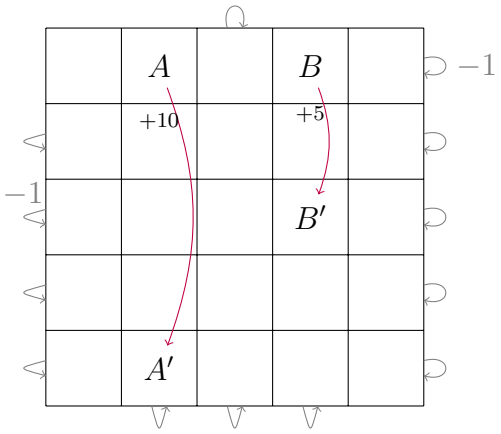


Figure 6: The states are defined by the locations on the grid. The actions move the agent up, down, left, or right. The agent cannot move off the grid and bumping generates a reward of minus one. Most other actions yield no reward, but there are two special states labeled A and B. Every action in state A yields $+10$ reward and $+5$ in state B. Correspondingly, every action in A and B transition the agent with probability one to A' and B' respectively.

We must first specify the policy before we can figure out what the value function is. Suppose we start with a uniform random policy. Since this is a continuing task, we need to choose $\gamma < 1$. Let's try $\gamma = 0.9$. We will later learn how to estimate the value function but for now let's take it as given below.

"The essence of reinforcement learning is combining search with memory" -Rich Sutton. "RL at its root is memoized (context-sensitive) search" -Andy Barto.

Monte Carlo methods We can estimate v_π and q_π from experience. 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 at each state, then these averages will similarly converge to the action values $q_\pi(s,a)$. Methods of estimating in this way are known as Monte Carlo methods because they involve averaging over many random samples of actual returns. If there are very many states, it many not be practical to keep separate averages for each state individually. Instead, we could have our agent maintain v_π and q_π as parameterized functions (with fewer parameters than states), and adjust the parameters to match the observed returns; these are known as approximate solution methods.

Bellman equation for state-value function In everyday life, we learn a lot without getting explicit positive or negative feedback. E.g. you could be riding your bike and hit a rock that sends you off balance.

$\gamma = 0.9$

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	−0.4
−1.0	−0.4	−0.4	−0.6	−1.2
−1.9	−1.3	−1.2	−1.4	−2.0

Figure 7: Recall that we have simulated a value function for an agent under a random action policy. Notice the negative values near the bottom, these values are low because the agent is likely to bump into the wall before reaching the distant states A or B. Remember that A and B are both the only sources of positive reward in this MDP. It's interesting to note that the state-value function at A is < 10, even though the immediate reward is $+10$: because every transition from A moves the agent closer to the lower wall in which the random policy is likely to bump and get a negative reward. I.e. in expectation the reward is slightly less than $+10$ because we expect to also run into a wall. On the other hand, the state-value at B is slightly greater than $+5$, because the next step involves moving the agent to the middle of the grid where they are unlikely to bump into any edges and further its reasonably close to states A and B. The value function compactly summarizes all these possibilities.

In spite of not getting injured, you learn to avoid rocks in the future and perhaps react more quickly if you do it one. How do we know that hitting a rock is bad even when nothing bad happened *this* time? The answer is that we recognize that losing our balance is bad even without falling and hurting ourselves, and perhaps we've had similar experiences in the past when things didn't work out so nicely. In RL, there's a similar idea that allows us to relate the value of the current state to the value of future states without waiting to observe all future rewards: we use the Bellman equations to formalize this connection between the value of a state and its possible successors. 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:

$$
\begin{aligned}
v_\pi(s) &\coloneqq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] && \text{by equation 8} \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma \underbrace{\mathbb{E}_\pi\left[G_{t+1} | S_{t+1} = s'\right]}_{v_*(s')} \right] \\
&= \sum_a \pi(a|s) \sum_{s', r} \Pr(s', r|s, a) \left[r + \gamma v_\pi(s') \right], \qquad \text{for all } s \in \mathcal{S}. && (11)
\end{aligned}
$$

The penultimate equality follows from the definition of expectation in two steps (the first step is to integrate out all possible actions, and the second is to integrate over all possible rewards and next states), and all we're doing is writing out our expectation explicitly as a weighted sum of possible outcomes, where the weights are the probabilities the event occurs. Note that we could continue to recursively expand our expression, but this would only make it more complicated. Further, realize that embedded in our expression is the value function for state s', i.e. $v_*(s')$, with the only difference being that the time index is time step $t + 1$ instead of t. But, realize that neither the policy nor p depends on time. Making this replacement, we realize the Bellman equation for the state-value function. I.e. equation 11 is known as the Bellman equation for v_π: it expresses a relationship between the value of a state and the values of its successor states. The magic value is that we can use them as a stand-in for the average of an infinite number of possible futures.

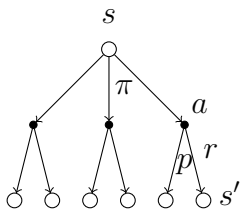


Figure 8: Think of looking ahead from a state to its possible successor states. Each open circle represents a state, and each solid circle represents a state-action pair. Starting from state s, the root node, the agent can take ay of some set of actions (three of which are shown in the diagram) based on its policy π. From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward r, depending on the dynamics given by the function p. The Bellman equation 11 averages over all possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the discounted value of the expected next state, plus the reward expected along the way.

If $r(s,a)$ is the expected reward for taking action a in state s, then we can re-express the Bellman equations using this expected reward function as follows.

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s',a'} \Pr(s'|s,a)\pi(a'|s')q_\pi(s',a')$$

$$v_\pi(s) = \sum_a \pi(a|s) \left[r(s,a) + \gamma \sum_{s'} \Pr(s'|s,a) v_\pi(s') \right]$$

$$v_*(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} \Pr(s'|s,a) v_*(s') \right].$$

The value function v_π is the unique solution to its Bellman equation. Diagrams like 8 are called *backup diagrams* because they diagram relationships that form the basis of the *update* operations; these operations transfer value information *back* to a state (or state-action pair) from its successor states (or state-action pairs). Unlike transition graphs, the state nodes of backup diagrams do not necessarily represent distinct states, e.g. a state might be its own successor.

Action-value Bellman equation We can derive a similar equation for the action-value function. It will be a recursive equation for the value of a state-action pair in terms of its possible successor state-action pairs. In this case, the equation does not begin with the policy selecting an action: this is because the action is already fixed as part of the input argument state-action pair. Instead, we simply skip to the dynamics function p which selects the immediate reward and next state s'. Again, we have a weighted sum over terms consisting of immediate reward plus expected future return given a specific next state little s'. Unlike the Bellman equation for the state value function, we can't stop here. We want a recursive equation for the value of one state-action pair in terms of the *next* state-action pair.

$$q_\pi(s,a) \coloneqq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{12}$$

$$= \sum_{s'} \sum_{r} \Pr(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \tag{13}$$

At the moment we have the expected return given only the next state. To change this, we can express the expected return from the next state as a sum of the agent's possible action choices, via law of total probability. In particular, we can change the expectation to be conditioned on both the next state and the next action and then sum over all possible actions.

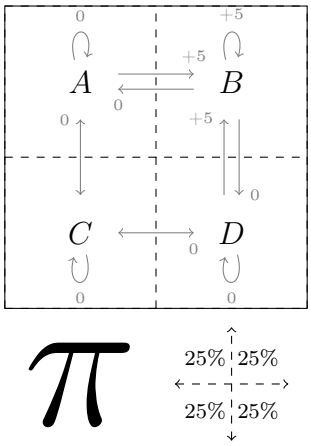
$$= \sum_{s'} \sum_{r} \Pr(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s', A_{t+1} = a' \right] \right]$$

Each term is weighted by the probability under our policy π of selecting a' in the state s'. This expected return is the same as the definition of the action-value function for s' and a'. Making this replacement, we get the Bellman equation for the action-value function.

$$= \sum_{s'} \sum_{r} \Pr(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right]$$

Bellman equations for both state and action-value functions provide relationships between the values of a state or state-action pair and the possible next states or next state-action pairs. These equations capture an important structure of the RL problem, and we will discuss how we can use this to design algorithms which efficiently estimate value functions.

Using Bellman equations to compute value-functions Let's illustrate a simple example consisting of four states, labeled A, B, C, and D on a grid. The action space consists of moving up, down, left, and right. Actions which would move off the grid, instead keep the agent in place. E.g. if we start in state C and move "up" we end in state A. If we then try to move "left" we would hit a wall and stay in state A. MOve to the right next would take us to state B, etc. The reward is 0 everywhere except for any time the agent lands in state B, in which case it gets a reward of +5; this includes starting in state B and hitting a wall to remain there. Suppose we consider a uniform random policy, which moves in each direction 1/4



of the time. Since this is a continuing task, we choose a discount factor of $\gamma = 0.7$. How can we actually work out the value of each of these states under this policy? Recall that the value function is the expected return under policy π, i.e. an average over the return obtained by each sequence of actions an agent could possibly choose (infinitely many possible futures). Fortunately, the Bellman equation for the state-value function provides an elegant solution: using the Bellman equation, we can write down an expression for the value of state A in terms of the sum of the four possible actions the resulting possible successor states.

$$v_\pi(s) = \sum_{a} \pi(a|s) \sum_{r} \sum_{s'} \Pr((s', r|s, a) \left[r + \gamma v_\pi(s') \right]$$

We can simplify the expression further *in this case*, because for each action there's only one possible associated next state and reward, i.e. the sum over s' and r reduces to a single value.

$$v_\pi(A) = \sum_{a} \pi(a|A) \left(r + 0.7 v_\pi(s') \right)$$

To go even further, if we go right from state A, we land in state B and receive a reward of +5. This happens 1/4 of the time under the random policy. If we go down, we land in state C, and receive no immediate reward, and this occurs 1/4 of the time. If we go either up or left, we will land back in state

A again, since each of the actions occur with probability 1/4, and since both actions land back in state A and receive no reward, we can combine them into a single term with a coefficient of 1/2.

$$v_\pi(A) = \frac{1}{4}\left(5 + 0.7v_\pi(B)\right) + \frac{1}{4}\left(0.7v_\pi(C)\right) + \frac{1}{2}0.7v_\pi(A).$$

Using the same methodology, we can write down a similar equation for each of the other states.

$$\begin{aligned}
v_\pi(B) &= \frac{1}{2}\left(5 + 0.7v_\pi(B)\right) + \frac{1}{4}0.8v_\pi(A) + \frac{1}{4}0.7v_\pi(D) \\
v_\pi(C) &= \frac{1}{4}0.8v_\pi(A) + \frac{1}{4}0.8v_\pi(D) + \frac{1}{2}0.7v_\pi(C) \\
v_\pi(D) &= \frac{1}{4}\left(5 + 0.7V_\pi(B)\right) + \frac{1}{4}0.7v_\pi(C) + \frac{1}{2}0.7v_\pi(D).
\end{aligned}$$

We now have a system of four equations in four unknowns, which can be solved for. The unique solution for this example happens to work out to

$$v_\pi(A) = 4.2, \qquad v_\pi(B) = 6.1, \qquad v_\pi(C) = 2.2, \qquad v_\pi(D) = 4.2.$$

What's important to note is that the Bellman equation reduced an unmanageable infinite sum over possible futures to a simple linear algebra problem. The Bellman equation provides a powerful general relationship for MDPs. Note that for more interesting problems like chess, we won't even be able to list all possible states, since there are around 10^{45} of them, let alone construct and solve the resulting system of Bellman equations.

3.6 Optimal Policies and Optimal Value Functions

Up to this point, we've generally talked about a policy as something that is given; it specifies how our agent behaves. Given such a way of behaving, we've discussed how to find the value function. But, solving an RL task means *finding* a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely find an optimal policy as follows. Value functions define a partial ordering over policies: a policy π is defined to be better than or equal to another policy π' if its expected return is greater than or equal to that of π' for all states. I.e. $\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. If a policy is better than another in some states but not others, we cannot say that one policy is better than the other. Clearly, there is always at least one policy that is better than or equal to all other policies; this is an *optimal policy*.[8] Although there may be more than one, we denote all optimal policies by π_*.

Example optimal policy Let's consider a two-choice MDP, where the only decision to be made is in the top state labeled X. In the above MDP, there are only two deterministic policies which are completely defined by the agent's choice in state X: take action A_1 or action A_2. Let's call these π_1 and π_2 respectively.

$$\pi_1(X) = A_1, \qquad \pi_2(X) = A_2.$$

The optimal policy is the one for which the value of X is highest, but the answer depends on the discount factor γ (since this is a continuing task). If $\gamma = 0$, then the value function is defined using only the

[8] To see this, suppose that we have two policies π_1 and π_2, where π_1 sees some states with higher value relative to π_2, but not for all states. We can create a new policy π_3 which is defined by combining the two aforementioned policies such that for each state we select the policy that performs best. This new policy π_3 will have a value greater than or equal to both π_1 and π_2 in *every* state. Therefore, we will never have a situation where doing well in one state requires a sacrifice at another. Due to this (informal) reasoning, there will always exist some policy which is best in every state. Note that it's possible that π_3 achieves values that strictly exceed both of π_1 and π_2, and this is due to the fact that by playing other states more optimally that are closeby, we could be setting ourselves up for a higher long-term reward.

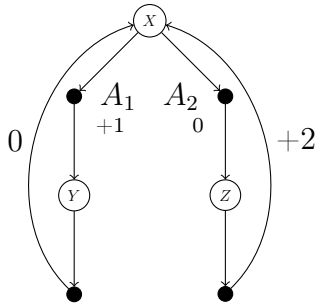


Figure 9: The agent can take either action A_1 or A_2. From state X, action A_1 takes the agent to state Y. In state Y, only action A_1 is available and it takes the agent back to X. On the other hand, action A_2 (from X) takes the agent to state Z. From state Z, action A_1 is again the only action available and it takes the agent back to state X. Realize that although action A_1 gives an immediate reward of $+1$, action A_2 offers a larger (albeit delayed) reward of $+2$ after a single step delay.

immediate reward. In this case, the value of X under π_1 is $+1$, while the value under π_2 is zero because the positive reward is only incurred after a delay, and this doesn't affect the return when $\gamma = 0$. In this case of $\gamma = 0$, then π_1 is the optimal policy. What if instead, $\gamma = 0.9$? In this case, the value of X under each policy is an infinite sum. The state-value functions are

$$v_{\pi_1}(X) = 1 + 0.9 * 0 + (0.9)^2 * 1 + \ldots = \sum_{k=0}^{\infty} 0.9^{2k} = \frac{1}{1 - 0.9^2} \approx 5.3$$

$$v_{\pi_2}(X) = 0 + 0.9 * 2 + (0.9)^2 * 0 + \ldots = \sum_{k=0}^{\infty} (0.9)^{2k+1} * 2 = \frac{0.9}{1 - 0.9^2} * 2 \approx 9.5$$

We've used the fact that we can compactly represent each state-value function by a geometric series. Note that in the case of π_1, we are incurring a positive unit valued reward on each even time step, whereas for π_2 we are incurring a slightly higher reward on each odd time step. In this example, π_2 is the more optimal policy, and we could find it simply by computing the state-value function for each deterministic policy under consideration.[9] In general, it won't be so easy. Even if we limited ourselves to deterministic policies, the number of possible policies is equal to the number of possible actions raised to the number of possible states, $|\mathcal{A}|^{|\mathcal{S}|}$. This means we cannot use a brute force search for even moderately sized problems. Fortunately, the *Bellman optimality equations* can help us organize our search of the policy space.

Optimal value functions Optimal policies achieve the goal of RL by achieving as much reward as possible in the long run, but the exponential number of possible policies make searching for the optimal policy by brute-force intractable. Recall that

$$\pi_1 \geq \pi_2 \iff v_{\pi_1}(s) \geq v_{\pi_2}(s) \quad \forall \quad s \in \mathcal{S}.$$

An optimal policy is one that is as good or better than every other policy. The value function for the optimal policy thus has the greatest value possible in every state.

$$v_{\pi_*}(s) \coloneqq \mathbb{E}_{\pi_*}\left[G_t | S_t = s\right] = \max_{\pi} v_{\pi}(s) \quad \forall \ s \in \mathcal{S}.$$

What does it mean to take a maximum over different policies? Imagine we were to consider every possible policy and compute each of their values for the state s: the value of an optimal policy is defined to be the largest of all the computed values. We could repeat this for every state and the value of an optimal policy would always be largest.

[9] As a final example consider the case of $\gamma = 0.5$.In this case, $v_{\pi_1} = \frac{1}{1-0.5^2} = \frac{4}{3}$ and $v_{\pi_2} = \frac{0.5}{1-0.5^2} \times 2 = \frac{4}{3}$ and both policies are optimal.

Optimal Policies An optimal policy is defined as the policy with the highest possible value function in all states. At least one optimal policy always exists, but there may be more than one. They share the same state-value function, called the *optimal state-value function*, denoted v_*, and defined as

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

for all $s \in \mathcal{S}$. Note that optimal policies also share the same *optimal action-value function*, denoted q_*, and defined as

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

for all $s \in \mathcal{S}$ 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 may write q_* in terms of v_* as follows:

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

Bellman optimality equation Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation 11 for state values. I.e. we have that

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_\pi(s')\right].$$

If we simply substitute the optimal policy π_* into this equation, we get the Bellman equation for v_*,

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right].$$

So far, nothing special, we've simply substituted an optimal policy into the Bellman equation. However, because this is an optimal policy, v_* can be written in a special form without referencing the policy itself. Remember that there always exists an optimal deterministic policy, one that selects an optimal action in every state. Such a determinstic policy would assign probability 1 for an action that achieves the highest value and probability zero for all other actions. We can express this another way by replacing the sum over π_* with a maximum over the action-space, i.e. using $\max_a$:

$$v_*(s) = \max_a \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right]. \tag{14}$$

Notice that π_* no longer appears in the equation! We have derived a relationship that applies directly to v_* itself. This special form of the equation is known as the *Bellman optimality equation* for v_*. Intuitively, this 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; re-writing it all together:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) = \max_a \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] = \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \qquad (15) \\ &= \max_a \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right]. \qquad (16) \end{aligned}$$

The last two equations are just two different forms of the Bellman optimality equation for v_*.

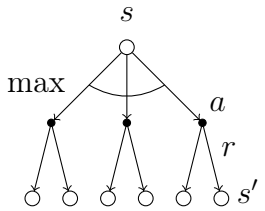


Figure 10: We draw a backup diagram for the Bellman optimality equation 16 for v_*. Notice that we've added an arc to indicate that the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy.

Bellman *optimality* equation for action-value function The Bellman optimality equation for q_* can be derived in a similar way, where we replace a summation over actions by a maximum, using a similar argument as above. Recall that

$$q_\pi(s,a) = \sum_{s'}\sum_{r} \Pr(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a',s') q_\pi(s',a')\right]$$

and we can substitute in an optimal policy to yield

$$q_*(s,a) = \sum_{s'}\sum_{r} \Pr(s',r|s,a)\left[r + \gamma \sum_{a'} \pi_*(a'|s') q_*(s',a')\right].$$

Making a similar argument as above, we can replace the use of a probability distribution $\pi_*(a',s')$ with a call to $\max_{a'}$, since our policy is optimal:

$$q_*(s,a) = \sum_{s'}\sum_{r} \Pr(s',r|s,a)\left[r + \gamma \underbrace{\max_{a'} q_*(s',a')}_{=v_*(s')}\right].$$

This yields the Bellman optimality equation for q_*. Writing it all out again in slightly different notation:

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

We discussed previously that the Bellman equations allowed us to form a linear system of equations for which we could solve for the value-state functions. The Bellman optimality equation gives us a similar system of equations for the *optimal* value. One natural question is: can we solve this system in a similar

$$\pi, p, \gamma \rightarrow \boxed{\text{Linear System Solver}} \longrightarrow v_\pi$$

Figure 11: A simple diagram showing how we can use a policy to determine a value-function.

way to find the optimal state-value function? Unfortunately, no, since the max operation over actions is not linear! Standard techniques from linear algebra no longer apply. In this course, we'll use other techniques based on the Bellman equations to compute value functions and policies. We might also wonder why we can't simply use π_* in the ordinary Bellman equation to get a system of linear equations for v_*, but unfortunately we don't know π_* (learning *it* is the fundamental goal of RL).

Uniqueness of solution for value function For finite MDPs, the Bellman optimality equation for v_* has a unique solution: it's actually a system of equations (one for each state), so if there are n states or equations then there are also n unknowns. If the dynamics p of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of *nonlinear* equations. We can also solve a related set of equations for q_*.

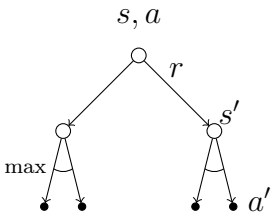
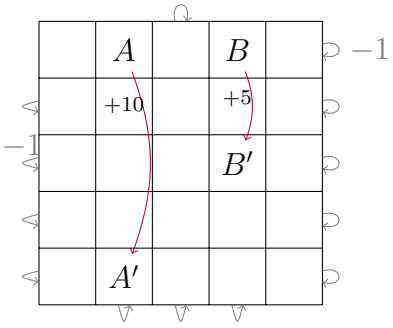


Figure 12: We draw a backup diagram for the Bellman optimality equation for q_*.

Using v_* to determine an optimal policy Having learned about optimal value functions and the Bellman optimality equations, we might wonder why this all matters when our ultimate goal is not to find the *value function* of an optimal policy but instead just the optimal policy itself! It turns out it's quite easy to go from an optimal value function to the associated optimal policy; the two goals are almost the same. Let us return to our grid-world problem from before.



Let's hold off on the question of how to find an optimal value function, but instead take the associated optimal values for each state as given.

$\gamma = 0.9$

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Figure 13: Notice that unlike before, the values along the bottom row are not negative. Unlike the uniform random policy, the optimal policy won't ever choose to bump into the walls. As a consequence, the value of state A is also much higher than the immediate reward of $+10$.

In general, having v_* makes it relatively easy to work out the optimal policy as long as we also have access to the dynamics function p. For any state, we can look at each available action and evaluate the

following:

$$v_*(s) = \max_a \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right]$$
$$\pi_*(s) = \arg\max_a \underbrace{\sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right]} \tag{17}$$

For any state we can look at each available action and evaluate the expression in braces; there will be some action for which this term attains a maximum. A deterministic policy which selects this maximizing action for each state will neccessarily be optimal, since it obtains the highest possible value. Thus, the equation shown here for π_* is thus almost the same as the Bellman optimality equation for v_*. To evaluate this term for a given action, we need only perform a one-step look ahead at the possible next states and rewards that follow.

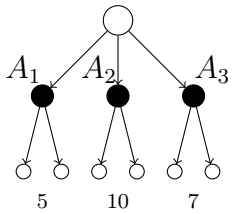


Figure 14: v_* is equal to the maximum of the underbraced term in equation 17, whereas π_* is the arg max. Critically, to evaluate the underbraced term for a given action, we need only perform a one step look ahead at the possible next states and rewards that follow. Imagine doing so for a particular action, labeled A_1, and suppose that we look at each state and reward which may follow from state s after taking action A_1; since we have access to v_* and our probability mechanics function p, we can evaluate each term in the summations over s' and r. Suppose that for A_1 the underbraced term evaluates to 5. We can repeat this procedure for all the actions, in this case A_2 and A_3, in each case the computation only requires a one step look ahead thanks to having access to v_* and p. Of the three actions in this example, it happens to be that A_2 is the optimal action; if there were multiple maximizing actions, we could define a stochastic optimal policy that chooses between each of them with some probability.

For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability *only* to these actions is an optimal policy. This is akin to a one-step search: if you have the optimal value function v_*, then the actions that appear best after a one-step search will be optimal actions. Put differently, any policy that is *greedy* with respect to the optimal evaluation function v_* is an optimal policy. The beauty of v_* is that if one uses it to evaluate the short-term consequences of actions, specifically the one-step consequences, then a greedy policy is actually optimal in the long-term sense in which we are interested because v_* already takes into account the reward consequences of all possible future behavior. I.e. v_*, the expected long run return, is turned into a quantity that is available locally and immediately available for each state. Whence, a one-step-ahead search yields the long-term optimal actions.

Let's continue with our grid-world example, building off of figure 13, which we reprint below for clarity. We will use the optimal value function to learn an optimal policy.

As a last example, consider state A itself. Realize that regardless of the action we pick in state A, the transition dynamics dictate that we move to state A' with an immediate reward of $+10$. This means that in state A, all actions are optimal since the transitions are equivalent. In particular $v_*(A) = 10 + \gamma * v_*(A') = 10 + 0.9 * 16 = 24.4$ which is indeed the recorded value for v_* at this state.

Note that in the grid-world example, working out the optimal policy from v_* was especially easy since each action leads us to determinstically arrive at some next state with some specified reward; so, we only had to evaluate one transition per action. But, in general the dynamics function p can be stochastic, so it may not be so simple. But, as long as we have access to p, we can always find the optimal action from v_* by computing the right-hand side of the Bellman optimality equation for each action and finding the largest value.

γ=0.9

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Figure 15: Consider the location or state defined by the cell that is in the second from the top row and all the way to the right; one-step look ahead considers each action and the potential next states and rewards. It's especially simple in this case because the actions leads us to deterministically pick a next state and reward. Suppose we consider the `up` action, there's no immediate reward and the next state has value 17.5, so using $\gamma = 0.9$ and by our formula $r + \gamma v_*(s')$, this evaluates to $0 + 0.9 * 17.5 = 14$. If we take the `right` action we bump into a wall, yielding an immediate -1 reward and leaving us in the same state which has a value of 16, so $r + \gamma v_*(s')$ for this action evaluates to $-1 + 0.9 * 16 = 13.4$. The `down` action yields no immediate reward and leads us to a state with value 14.4, and so our expression of interest evaluates to $0 + 0.9 * 14.4 = 13$. Finally, the action `left` action leads to no immediate reward but a next state value of 17.8, and after we discount by γ this evaluates to $0 + 0.9 * 17.8 = 16$. Of all these choices, the highest value attained is 16, therefore, moving `left` corresponds to the optimal action in this state and *must* be selected by any optimal policy. Note that for the maximizing action, the value attained is 16 and this is indeed equal to v_* for the state itself.

γ=0.9

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Figure 16: Let's consider a second example, this time we're considering starting from the state in the middle row and middle column. Notice that there are two actions, `up` and `left` which give the same optimal value of $0.9 * 19.8 = 17.8$. In this state, there are two different optimal actions, and an optimal policy is free to pick either (with some probability).

Using q_* to determine optimal actions With q_*, the agent does not even have to do a one-step-ahead search: for any state s it can simply find any action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step-ahead searches. Hence, at the cost of representing a function of state-action pairs instead of just states, the optimal action value function allows optimal actions to be selected without having to know anything about possible successor states and their values, i.e. without having to know anything about the environment's dynamics. In this sense, the problem of finding an optimal action-value function corresponds directly to the goal of finding an optimal policy!

Downsides of explicitly solving Bellman optimality equation The approach is rarely directly useful, since it's akin to an exhaustive search which looks ahead at all possibilities, computes their probabilities of occurrence and their desireabilities in terms of expected rewards. This solution relies on three assumptions that are rarely true in practice:

1. We accurately know the dynamics of the environment,
2. We have enough computational resources to complete the computation of the solution, and
3. The Markov property.

The second limitation often means that we typically have to settle for approximate solutions.

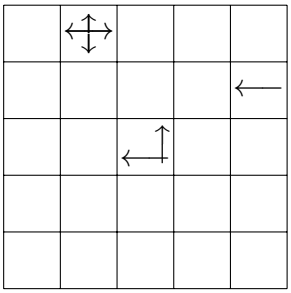


Figure 17: We draw in a few of the optimal action(s) for various states that we've gone over above. The key idea is that we can use the Bellman optimality equation to derive an optimal policy.

3.6.1 Summarizing policies, value functions, Bellman equations, and optimality

Policies Policies tell an agent how to behave. A deterministic policy maps each state to an action; each time a state is visited, a deterministic policy selects the associated action $\pi(s)$. Stochastic policies map each state to a distribution over all possible actions. Each time a state is visited, a stochastic policy randomly draws an action from the associated distribution with probability $\pi(a|s)$. A policy by definition depends only on the current state: it cannot depend on things like time or previous states. This is best thought of as a limitation on the state, not the agent: the state should provide the agent with all the information it needs to make a good decision.

Value functions Value functions capture the future total reward under a particular policy. There are two kinds of value functions: state value functions and state-action value functions. The state value function gives hte expected return from the current state under a given policy, $v_\pi(s) \coloneqq \mathbb{E}_\pi\left[G_t|S_t = s\right]$. The action-value function gives the expected return from state s if the agent first selects action a and follows π after that, i.e. $q_\pi(s,a) \coloneqq \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right]$. Value functions simplify things by aggregating many possible future returns into a single number.

Bellman equations The Bellman equations define a relationship between the value of a state (or state-action pair) and its successor states. The Bellman equation for the state value function gives the value of the current state as a sum over the values of all the successor states, and immediate rewards,

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a)\left[r + \gamma v_\pi(s')\right].$$

The Bellman equation for the action-value function gives the value of a particular state-action pair as the sum over all values of all possible next state-action pairs and rewards.

$$q_\pi(s,a) = \sum_{s'} \sum_r \Pr(s', r|s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right].$$

The Bellman equations can be directly solved to find the value function. These Bellman equations help us evaluate policies, but they don't achieve our ultimate goal of finding the policy that attains as much reward as possible in the long term.

Optimal policies An optimal policy is one which achieves the highest possible value in every state.

$$v_* = v_{\pi_*}(s) = \max_\pi v_\pi(s) \qquad \text{for all } s \in \mathcal{S}.$$

There's always at least one optimal policy, but there may be more. The optimal state value function is equal to the highest possible value in every state. Every optimal policy shares the same optimal state value

function. The same is true for optimal action-value functions and optimal policies.

$$q_* = q_{\pi_*}(s,a) = \max_\pi q_\pi(s,a) \qquad \text{for all } s \in \mathcal{S} \quad \text{and } a \in \mathcal{A}.$$

Like all value functions, the optimal value functions have Bellman equations

$$v_{\pi_*}(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_{\pi_*}(s')\right]$$

$$q_{\pi_*}(s,a) = \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma \sum_{a'} \pi_*(a'|s') q_{\pi_*}(s', a')\right]$$

but in this special case they do not reference any specific policy. This amounts to replacing the policy in the Bellman equation with a `max` over all actions, since the optimal policy must always select the best available action.

$$v_*(s) = \max_a \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_*(s')\right]$$

$$q_*(s,a) = \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right]$$

We can extract an optimal policy from the optimal state value function, but to do so we need the one-step dynamics of the MDP, i.e. we need our transition probabilities fully specified. We can get the optimal policy with much less work if we have the optimal action-value function: we simply select the action with the highest value in each state, i.e. $v_*(s) = \max_a q_*(s,a)$.

3.7 Optimality and Approximation

We've defined optimal value functions and optimal policies. Although in theory an agent that learns an optimal policy has succeeded, in practice this rarely happens. Optimal policies can only be generated with extreme computational cost. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step. Memory is also an important constraint, since a large amount of it is often required to built approximations of value functions, policies, and models. For tasks with small, finite state sets, we can use arrays or tables with a single entry per state (or state-action pair); this is what we call the *tabular* case. Unfortunately, in many cases there are far too many states than we can store in memory. In these cases we must approximate the functions, using some sort of more compact parameterized function representation. Note that in approximating optimal behavior, there may be some states that the agent faces with such a low probability that selecting suboptimal actions has little impact on the reward the agent receives. The online nature of RL makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states.

3.8 Summary of Finite MDPs

RL is about learning from interaction how to behave well in order to achieve a goal. The RL *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* ar ethe basis for evaluating the choices. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time. When RL is formulated with well-defined transition probabilities, we realize an MDP. A finite MDP has finite state, action, and reward sets. The *return* is the function of future rewards that

the agent seeks to maximize in expected value. There are different definitions depending on the nature of the task, i.e. for *continuing* tasks we use a *discounted* delayed reward whereas in *episodic* tasks we use the undiscounted formulation.

A policy's *value functions* assign to each state (or state-action pair) the expected return from that state (or state-action pair), given that the agent uses the policy. The *optimal value functions* assign to each state (or state-action pair) the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined.

Even if the agent has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is also an important constraint, since memory may be required to build up accurate approximations of value functions, policies, and models. In most cases of interest, there are far more states than could possibly be entries in a table, and approximations must be made.

4 Dynamic Programming

The key idea of reinforcement learning is the use of value functions to organize and structure the search for good policies. This section is about demonstrating how dynamic programming (DP) can be used to obtain optimal policies once we have found optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations. DP algorithms are obtained by turning Bellman equations into update rules for improving approximations of the desired value functions.

Policy evaluation vs. control We often talk about two distinct tasks: policy evaluation and control. Policy evaluation is the task of determining the value function for a specific policy. Control is the task of finding a policy that obtains as much reward as possible, i.e. finding a policy which maximizes the value function. Although control is the ultimate goal of RL, the task of policy evaluation is a necessary first step, since it's hard to improve our policy if we don't have a way to assess how good it is. DP algorithms use the Bellman equations to define iterative algorithms for both policy evaluation and control. Schematically,

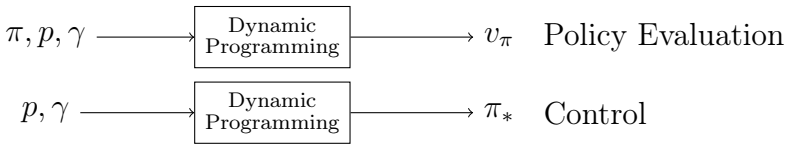


Figure 18: DP uses the various Bellman equations along with knowledge of p to work out value functions and optimal policies. Classical DP does not involve interaction with the environment at all. It turns out that most RL algorithms can be seen as an approximation to DP programming without the model of the MDP specified?

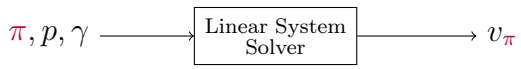
4.1 Policy Evaluation (Prediction)

Let's consider how to compute the state-value function v_π for arbitrary policy π; in the literature this is known as *policy evaluation*, but we also refer to it as the *prediction problem*. I.e our task is to learn a

mapping $\pi \longrightarrow v_\pi$. Recall that

$$\begin{aligned} v_\pi(s) &\coloneqq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a)\left[r + \gamma v_\pi(s')\right], \end{aligned} \tag{18}$$

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 denote that they are conditioned on π being followed. Recall that the Bellman equation reduces the problem of finding v_π to a system of linear equations, one equation for each state. So, the problem of policy evaluation reduces to solving this system of linear equations, and in principle we can take a variety of methods from linear algebra to do so.



Existence and uniqueness of v_π are guaranteed as long as $\gamma < 1$ or eventual termination is guaranteed from all states under policy π. If the environment dynamics are completely known, then equation 18 is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns, i.e. the $v_\pi(s)$ for each $s \in \mathcal{S}$.

Iterative solutions In practice, iterative solution methods of DP are more suitable for general MDPs. I.e. instead of using direct solvers, lets consider a sequence of *approximate* value functions, $v_0, v_1, v_2, \ldots$ which each map $\mathcal{S}^+$ to $\mathbb{R}$ (the real numbers). The initial approximation v_0 may be chosen arbitrarily, but each successive update may be obtained by using the Bellman equation for v_π 18 as an update rule:

$$\begin{aligned} v_{k+1}(s) &\coloneqq \mathbb{E}_\pi\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a)\left[r + \gamma v_k(s')\right] \end{aligned} \tag{19}$$

for all $s \in \mathcal{S}$. Realize that $v_k = v_\pi$ is a fixed-point for this update rule because the Bellman equation for v_π assures us of equality in this case. The sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. In practice, once $v_{k+1} = v_k$ for all states, then $v_k = v_\pi$ and we have found the value function.

Expected updates To produce each successive approximation, v_{k+1} from v_k, the 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. Such an operation is referred to as an *expected update*. Each iteration of policy evaluation updates the value of every state once to produce a new approximate value function v_{k+1}.

In-place updates To write the sequential compute program to implement iterative policy evaluation as given by update rule 19, you would naively have to use two arrays: one for the old values $v_k(s)$ and another for the new values $v_{k+1}(s)$. The length of these arrays would be equal to the number of states in the MDP. With these two arrays, the new values can be computed one by one from the old values without the old values being changed. It's instead easier to implement "in place", with each new value immediately overwriting the old one. Depending on the order in which the states are updated, sometimes new values are used instead of old ones on the right hand side of 19. This in-place algorithm is also guaranteed to converge to v_π and is usually faster than the two-array version, since we use new data as soon as they are available.

Algorithm 2: Iterative Policy Evaluation, for estimating v_π

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 $V(\text{terminal}) = 0$.

while $\Delta \geq \theta$ **do**

  $\Delta \leftarrow 0$

  for $s \in \mathcal{S}$ **do**

    $v \leftarrow V(s)$

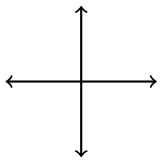
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma V(s')\right]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ `// Track largest update to state-value function.`

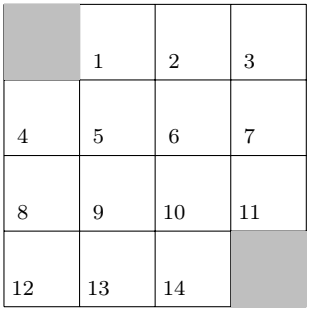
  end

end

Grid-world example We consider the 4×4 gridworld shown below.



actions



$R_1 = -1$ on all transitions

Figure 19: The non-terminal states are $\{1, 2, \ldots, 14\}$, and there are four possible actions in each state $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cuase the corresponding state transitions, except that actions which would take the agent off the grid in fact leave the state unchanged. In the notation of $\Pr(s', r|s, a)$ we have examples like $\Pr(6, -1|5, \texttt{right}) = 1$, or $\Pr(7, -1|7, \texttt{right}) = 1$, as well as $\Pr(10, r|5, \texttt{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached, which is denoted by a gray shaded region(s); although shown in two places, it is formally one state. The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a. Suppose the agent chooses the equiprobable random policy (where all actions are equally likely). The function v_π gives for each state the negated expected number of steps from that state until termination.

4.2 Policy Improvement

The reason we care to compute the value function for a policy is such that we can use it to help find better policies. Suppose we've determined the value function v_π for an *arbitrary* deterministic policy π. For some state s, we'd like to know whether we should change the policy to deterministically choose an action $a \neq \pi(s)$. Of course, we know how good it is to follow the current policy from s, i.e. $v_\pi(s)$, but would it be better or worse to switch to a new policy? Consider selecting a in s and thereafter following the existing policy π. The value of *this* way of behaving is given by

$$q_\pi(s, a) \coloneqq \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] = \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma v_\pi(s')\right]. \tag{20}$$

We'd like to know how this quantity compares to $v_\pi(s)$. We would intuitively expect that if it is better to select a once in s and thereafter follow π relative to just following π all the time, that we could improve our policy further still by just taking action a *every* time we encounter s. It turns out this is a special case of a general result called the *policy improvement theorem*.

4.2.1 Policy Improvement Theorem

Let π and π' be any pair of deterministic policies such that for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{21}$$

then policy π' must be as good as or better than π; i.e. it must obtain greater than or equal expected return for all states $s \in \mathcal{S}$: $v_{\pi'}(s) \geq v_\pi(s)$. If equation 21 holds with *strict* equality then it follows that $v_{\pi'}(s) > v_\pi(s)$. The proof behind the policy improvement theorem is easy to understand. Starting from 21, we continually expand out using equation 20 and reapply 21 until we get $v_{\pi'}(s)$.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi\left(s, \pi'(s)\right) \\
&= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)\right] && \text{by 20} \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s\right] && \text{by 21} \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma \mathbb{E}_{\pi'}\left[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})\right] | S_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s\right] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots | S_t = s\right] \\
&= v_{\pi'}(s).
\end{aligned}$$

So far we've seen how, given a policy and its value function, how to easily evaluate a change in the policy at a single state to a particular action. It's natural extension to consider changes at *all* states and to *all* possible actions, selecting at each state the action that appears to be best according to $q_\pi(s, a)$. I.e, define a *greedy* policy π' given by

$$\pi'(s) \coloneqq \arg\max_a q_\pi(s, a) = \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] = \arg\max_a \sum_{s', r} \Pr(s', r | s, a)\left[r + \gamma v_\pi(s')\right]. \tag{22}$$

Policy improvement In the even that there are multiple actions attaining a maximal value, we may break ties arbitrarily. By construction, the greedy policy meets the condition of the policy improvement theorem 21 and so we know it's at least as good as the original policy. This process of making a new policy that improves upon an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

4.2.2 Why policy improvement gives us a strictly better policy

Suppose toward contradiction that the new greedy policy π' is as good as, but not better than, the old policy π. Then, $v_\pi = v_{\pi'}$, and from equation 22 it follows that for all $s \in \mathcal{S}$:

$$v_{\pi'}(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s | A_t = a\right] = \max_a \sum_{s', r} \Pr(s', r | s, a)\left[r + \gamma v_{\pi'}(s)\right].$$

But, realize this is the same as the Bellman optimality equation! Therefore, $v_{\pi'}$ must be v_*, and both π and π' must be optimal policies. I.e. policy improvement therefore must give a strictly better policy except when the original policy is already optimal.

Stochastic optimal policies The policy improvement theorem carries through as stated for the stochastic case where a policy π specifies probabilities $\pi(a|s)$ for taking each action a in each state s. If there are ties in the policy improvement steps such as 22, i.e. if there are several actions at which the maximum is achieved, then in the stochastic case we can allocate a portion of the probability of being selected in the new greedy policy to each of these maximizing actions; any portioning scheme is allowed so long as all submaximal actions are given zero probability.

4.3 Policy Iteration (Control)

We can iteratively improve upon our policies to obtain a sequence of monotonically improving policies and value functions.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Figure 20: In this figure, $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*. Each policy is deterministic, and is guaranteed to be strictly better than the last, unless it's already optimal. Because a finite MDP has only a finite number of deterministic 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 known as *policy iteration*. We provide a complete sequence of instructions in algorithm 3. Notice that in the step of Policy Evaluation, that we are using the value functions learned from our previous policy; this typically results in a large increase in the speed of convergence of our algorithm, presumably because the value function is changing slowly from one policy to the next.

Algorithm 3: Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$. Note that this algorithm can oscillate between two equally good policies, and so in practice we should keep track of whether our value function is changing (i.e. still improving) in addition to checking for `policy_stable`.

Input: θ, a small positive number determining the accuracy of estimation.
Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$ `// Initialization.`
`// Policy Evaluation (make` v `reflect` π`).`
do
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} \Pr(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
  end
while $\Delta \geq \theta$;
`// Policy improvement (make` π `greedy).`
`policy_stable` $\leftarrow$ `true`
for $s \in \mathcal{S}$ **do**
  `old_action` $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma V(s')\right]$
  if *old_action* $\neq \pi(s)$ **then** `policy_stable` $\leftarrow$ `false` ;
end
if *policy_stable* **then return** $V \approx v_*$ *and* $\pi \approx \pi_*$;
else Evaluate policy, going back to second step in our algorithm. ;

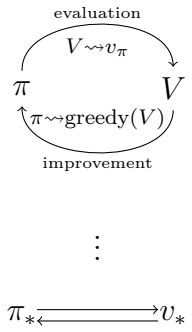


Figure 21: We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Almost all RL learning methods can be described as GPI: i.e. all 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 in our diagram. If both the evaluation process and improvement process stabilize, then the value function and policy must be optimal: the value function only stabilizes when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies the Bellman optimality equation holds and thus that the policy and value function are optimal!

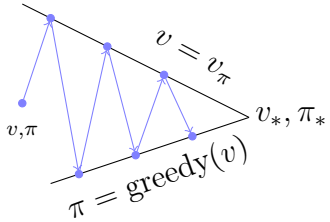


Figure 22: We can think of the nteraction between the evaluation and improvement processes in GPI in terms of two constraints or goals – for example as two lines in two-dimensional space as suggested by the diagram. Although the real geometry is much more complicated, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals *interact* because the lines are not orthogonal: driving directly toward one goal causes some movement away from the other goal; inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in the diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI, one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly!

4.4 Value Iteration

A downside to policy iteration is that each of its iterations involves policy evaluation, which may itself be an iterative computation involving multiple sweeps through the state set. If policy evaluation is done iteratively, it is only in the limit that we get convergence to v_π. The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration! One important case is when the algorithm is stopped after just one sweep (or one update of each state). This algorithm is known as *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$v_{k+1} \coloneqq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\right] = \max_a \sum_{s',r} \Pr(s', r|s, a)\left[r + \gamma v_k(s')\right], \tag{23}$$

for all $s \in \mathcal{S}$. For any arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_*. Realize that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also, note that the value iteration update is identical to the policy evaluation update 19 except it requires the maximum to be taken over all actions.

How does value iteration terminate? Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*. In practice, we stop once the value function changes by only a small amount in a sweep.

Algorithm 4: Value iteration, for estimating $\pi \approx \pi_*$

Input: A small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$.
do
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    `// In the next step, we don't refer to any policy, hence name value iteration`
    $V(s) \leftarrow \max_a \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
  end
while $\Delta < \theta$;
Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg\max_a \sum_{s',r} \Pr(s', r|s, a) \left[r + \gamma V(s')\right].$$

Value iteration effectively combines, each each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

Asynchronous DP Methods that perform systematic sweeps like this are called synchronous; this can be problematic if the state space is large since the sweeps can take a long time. Asynchronous DP is a family of methods in which we update the states in any order; in fact, multiple updates may be performed to a single state before another state is updated once. In order to guarantee convergence, asynchronous algorithms must update the values of all states.

4.5 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, but this isn't necessary as we saw in value iteration only a single iteration of policy evaluation is performed in between each policy improvement. 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.

Evaluation and Improvement as both competing and cooperating processes The evaluation and improvement processes in GPI are both competing and cooperating processes. They compete in the sense that making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes the policy no longer to be greedy. In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.

4.6 Efficiency of Dynamic Programming

For very large problems, DP is not practical. But, compared with other methods for solving MDPs, DP methods are actually quite efficient. The worst case work required by DP methods to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k. Note that we find an optimal policy in polynomial time even tohugh the total number of (deterministic) policies is k^n; in this sense, DP is exponentially faster than any direct search in policy space could be. DP is sometimes thought to be of limited applicability because of the *curse of dimensionality*, the fact that the number of states often grows exponentially with the number of state variables. Although it's true that large state sets create difficulties, these are inherent difficulties in the problem not of DP as a solution method. DP is in fact comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

4.6.1 Monte Carlo sampling as an alternative method for learning a value function

The value of each state can be treated as a totally independent estimation problem. First, recall that the value of a state is given by the expected reward starting from that state when acting under a particular policy, i.e.

$$v_\pi(s) \coloneqq \mathbb{E}_\pi \left[G_t | S_t = s \right].$$

First, let's gather a large number of returns under π and then take their average for each state; this method is called the Monte Carlo method. However, if we do it this way we may need a large number of returns from each state. To see this, realize that each return depends on many random actions, selected by π, as well as many random state transitions due to the dynamics of the MDP. We could be dealing with a lot of randomness here, and so each return might be very different than the true state value. So, we may need to average many returns before the estimate converges, and we have to do this for every single state.

DP bootstraps value estimates from successor states to improve current value estimate The key insight to DP is that we don't have to treat the evaluation of each state as a separate problem: we can use the other value estimates that we've worked hard to compute. The process of using the value estimates of successor states to improve our current value estimate is known as *bootstrapping*. This can be much more efficient than a Monte-Carlo method which estimates each value independently.

4.6.2 Brute-force search as an alternative method for finding an optimal policy

Here, we simply evaluate every possible deterministic policy one at a time. We then select the one with the highest value. Since there are a finite number of deterministic policies, there always exists an optimal deterministic policy. So, brute-force search will find the answer eventually. However, the number of deterministic policies can be huge. Because a deterministic policy consists of one action choice per state, the total number of deterministic policies is exponential in the number of states; i.e. there are $|\mathcal{S}|$ states, and for each we have to consider from $|\mathcal{A}|$ different actions, this leads to multiplying $|\mathcal{A}|$ by itself $|\mathcal{S}|$ times: $|\mathcal{A}|^{|\mathcal{S}|}$. On the other hand, the policy improvement theorem guarantees that policy iteration will find a sequence of better and better policies: this is a significant improvement over exhaustively searching each and every policy.

4.7 Summary

Policy evaluation is the task of determining the state-value function v_π for a particular policy π. Iterative policy evaluation takes the Bellman equation for v_π,

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

and turns it into an update rule, producing better and better approximations for v_π:

$$v_{k+1} \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_k(s')\right].$$

Control refers to the task of improving a policy. The policy improvement theorem tells us how to construct a better policy from a given policy:

$$\pi'(s) \coloneqq \arg\max_a \sum_{s'} \sum_r \Pr(s', r|s, a) \left[r + \gamma v_\pi(s')\right].$$

The new policy π' is produced by "greedifying" with respect to current values. We are guaranteed to yield a better policy unless what we started with was already optimal. The DP algorithm for control is built upon the policy improvement theorem; this algorithm is called policy iteration. It consists of two steps: policy evaluation and improvement. In policy evaluation, we find the value function for the current policy. Policy improvement or "greedification" makes the policy greedy with respect to the current value functions. These steps are repeated over and over until the policy doesn't change, in which case we are guaranteed to have found an optimal policy. We discussed the framework of generalized policy iteration, in which the evaluation and improvement steps need not be run to completion; one example is the value iteration algorithm. Generalized policy iteration also includes asynchronous methods, which can more efficiently propagate value information This can be particularly helpful if the state space is very large.

5 Sample-based Learning Methods

Sample-based learned methods mean that we learn solely from trial and error, i.e. from experience. All agents can learn how the world works on their own, and we do not need to specify a model of the world any longer. If we do have a model, we can still use it by simulating experience from it. Sample-based learning algorithms are based on value functions and ideas from DP, including one of the most fundamental algorithms in RL called temporal difference learning. We first start with methods to learn value functions for prediction, then we'll see how we can learn action values for control, and finally we'll talk about planning with our learned model. We will discuss Monte-Carlo methods, and how TD methods can be used for prediction and control. We'll also see how TD control methods like SARSA and Q-learning are connected to generalized policy iteration and Bellman operators. Along the way, we'll discuss expiration, which is critical when learning from trial and error interaction. Further, we'll learn about off-policy learning, which is essential when learning from experience.

5.1 Monte Carlo methods

Here, we do not assumpe complete knowledge of the environment; instead we only require *experience* – sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no knowledge of the environment's dynamics, and yet an optimal policy can still be obtained. Learning from *simulated* experiences is powerful, and although a model is required it need only generate sample transitions as opposed to complete probability distributions of all possible transitions that is required for DP. Monte Carlo methods are a way of solving

RL problems by averaging sample (complete) returns; we focus only on episodic tasks such that returns are well defined. In particular, Monte Carlo methods sample and average *returns* for each state-action pair, and average *rewards* for each action. Whereas we previously computed value functions from knowledge of the MDP, here we *learn* value functions from sample returns with the MDP. Value functions and corresponding policies still interact to attain optimality in the same way (GPI).

An example where computing transition probabilities is tedious, and how MC methods can help Think about predicting the outcome of 12 rolls of dice, the sheer number and complexity of calculations makes them tedious and error-prone both in terms of coding and numerical precision. A Monte Carlo method, on the other hand, would simply estimate values by averaging over a large number of random samples; we just roll 12 dice a few times and see the result. Note that the Monte Carlo method doesn't require knowledge of any of the probabilities, and we don't need to exhaustively sweep through all possible outcomes in order to get an estimate for some states. Further, realize that with a Monte Carlo method, the computation required to update the value of each state does not depend on the size of the MDP; it instead depends on the length of the episode.

Computing returns efficiently In Monte Carlo methods, for each state in the episode, we'll want to compute the return and store it in a list of returns to be averaged over. How can we do that efficiently? Realize that if we were to write out our returns for, e.g. an episodic task which ends after five actions,

$$
\begin{aligned}
G_0 &= R_1 + \gamma G_1 \\
G_1 &= R_2 + \gamma G_2 \\
G_2 &= R_3 + \gamma G_3 \\
G_3 &= R_4 + \gamma G_4 \\
G_4 &= R_5 + \gamma G_5 \\
G_5 &= 0
\end{aligned}
$$

Notice that each return is included in the equation for writing the previous time step's return. That means we can avoid duplicating computations by starting at the terminal state G_5 and working our way backwards.

Why Monte Carlo methods are defined on episodic MDPs We'll see that in the algorithms that follow, we wait until the *end* of an episode beforing averaging the returns witnessed from varios states or state-action pairs. Because of this, we limit our discussion in this section to episodic MDPs which have a termination state.

5.1.1 Monte Carlo Prediction

Recall that the value of a state is the expected return – expected cumulative future discounted reward– starting from that state. An obvious way to estimate it from experience is by averaging returns observed after visits to that state; as more returns are observed, weak law of large numbers guarantees the sample average converges in probability to the population mean. This idea underlies all Monte Carlo methods. In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Note that s may be visited multiple times in the same episode; let us define the first time it is visited as the *first visit* to s. The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every visit MC method* averages the returns following all visits to s.

Both first-visit and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to ∞. It's particularly easy to see in the case of first-visit MC: each return is an independent, identically

Algorithm 5: First-visit MC prediction, for estimating $V \approx v_\pi$

Input: A policy π to be evaluated
// Initialization.
$V(s) \in \mathbb{R}$ arbitrarily, for all $s \in \mathcal{S}$
$\texttt{Returns}(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
while *True* **do**
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t = T-1, T-2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if S_t *not in* $S_0, S_1, \ldots, S_{t-1}$ **then**
      Append G to $\texttt{Returns}(S_t)$
      $V(S_t) \leftarrow \texttt{average}(\texttt{Returns}(S_t))$
    end
  end
end

distributed estimate of $v_\pi(s)$ with finite variance; by weak law of large numbers the sequence of averages of these estimates converges to their true expected value; each average itself is an unbiased estimate and the standard deviation of its error decreases like $\frac{1}{\sqrt{n}}$, where n is the number of returns averaged.

Blackjack Example The goal is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, an ace can count as either 1 or 11. We consider a game where we only play against the dealer. The game starts with two cards dealt to both dealer and player. *One of the dealer's cards is face up and the other is face down.* If the player has 21 immediately (an ace and a 10-card), then it is called a natural and he/she wins unless the dealer also has a natural in which case we pronounce a draw. If the player doesn't have a natural, he/she can either request an additional card (hit), stop requesting cards (stick), or go bust (if the value of cards exceeds 21). If he/she goes bust, the agent loses. If the agent sticks, it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy: stick on any sum 17 or greater, and hits otherwise. If the dealer goes bust the agent wins; otherwise, the outcome is determined by whosever final sum is closer to 21. This problem is naturally formulated as an episodic MDP, where each game is an episode. Rewards of +1, -1, and 0 are given for winning, losing, and drawing respectively; all rewards within a game are zero, and we don't discount. Whence, the terminal rewards are also the returns. The agent's *actions* are to hit or to stick. The states depend on the player's cards and the dealer's showing card. We assume the cards are dealt from an infinite deck (i.e. with replacement) so that there is no advantage to keeping track of which cards are already dealt, and further this will make our state respect the Markov property. If the player holds an ace that he/she could count as 11 without going bust, then the ace is said to be *useable*. In this case, it is always counted as 11, because counting it as a 1 would make the sum 11 or less, in which case there is no decision to be made because obviously the player should always hit (since in this case a hit is impossible to cause a bust, since the max valued card is 10 ignoring aces which can be treated as ones). Thus, the player makes their decision based on three variables: their current sum (12-21), the dealer's one showing card (ace-10), and whether or not he/she holds a useable ace. This makes for a total of 200 states. We consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach, we simulate many blackjack games using the policy and average the returns following each state.

Comparing why DP not suitable for task of blackjack Although we have complete knowledge of the environment in the blackjack task, it wouldn't be easy to apply DP methods to compute the value function. DP methods requrie the distribution of next events – in particular, they require the environments dynamics as given by the four-argument function $\Pr(s', r|s, a)$ – and it is not easy to determine this for blackjack. E.g. suppose the player's sum is 14 and they choose to stick. What is the probability of the agent terminating with a reward of +1 as a function of the dealer's showing card? All of the probabilities must be computed *before* DP can be applied, and these computations are complex and error-prone for this example. In contrast, generating the sample games required by MC methods is easy, and this is the case surprisingly often.

Backup diagrams and Monte Carlo algorithms The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo evaluation of v_π, the root is a state node, and below it is an entire trajectory of transitions along a particular single episode, ending at the terminal state.

Whereas the DP diagram shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

Monte Carlo methods imply estimates for each state are independent The estimate for one state does not depend or build upon the estimate of any other state, as in the sense of DP. In other words, Monte Carlo methods do not *bootstrap* as defined previously.

Computational expense of estimating the value of a single state is independent of the number of states This can make Monte Carlo methods particularly attractice when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. This is a third advantage of Monte Carlo methods can have over DP method, after the ability to learn from (simulated) experience.

5.1.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate *action* values (the value of state-action pairs) rather than *state* values. With a model, state values alone are sufficient to determine a policy: one simply looks one step ahead and chooses whichever action leads to the best combination of reward and next state.

Without a model, however, state values alone are not sufficient: one must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, a primary goal for Monte Carlo methods is to estimate q_*. To achieve this, we consider the policy evaluation problem for action values.

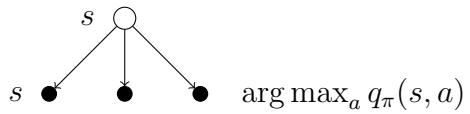


Figure 23: Action values lend themselves toward learning a policy improvement: we simply greedily select the action that leads to the maximum return from a given state. But, realize that in order to make the comparison, i.e. to make the call to $\arg\max_a$ we must have estimates for $q_\pi(s, a)$ for each $a \in \mathcal{A}(s)$. This is why RL methods must maintain exploration.

Policy evaluation problem for action values The problem is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π. The Monte Carlo methods for this are essentially the same as just presented for state values, except now we talk about visits to a state-action pair rather than to a state. A state-action pair s, a is said to be visited in an episode if ever the state s is visited and action a is taken in it. The every-visit MC method estimates the value of a state-action pair as the average of the returns that have followed all the visits to it, whereas the first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected. These method both converge to the true expected values as the number of visits to each state-action pair approaches infinity.

The general problem of maintaining exploration There is a complication in that many state-action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. This is problematic because the purpose of learning action values is to help in choosing among the actions available in each state. To compare alternatives, we need to estimate the value of *all* the actions from each state, not just the one we currently favor. This is the problem of *maintaining exploration*.

The method of exploring starts For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes *start in a state-action pair*, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. This strategy is known as *exploring starts*. Although this method is sometimes useful, it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment. In that case, the starting conditions are unlikely to be so helpful. The most common alternative approach to assuring that all state-action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state.

5.1.3 Monte Carlo Control

We can now consider how MC estimation can be used in control, i.e. to approximate better policies. The overall idea is to proceed according to the idea of generalized policy iteration; we will maintain both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. Although the two changes work against each other to some extent, as each creates a moving target for the other, they cause both policy and value function to approach optimality.

Monte Carlo version of classical policy iteration In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function.

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*$$

Figure 24: In this figure, $\xrightarrow{\text{E}}$ denotes a complete policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a complete policy *improvement*.

For the moment, we assume that we do indeed observe an infinite number of episodes and additionally that the episodes are generated with exploring starts. Under these assumptions the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k.

Policy Improvement Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function q, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value, i.e.

$$\pi_{k+1}(s) \coloneqq \arg\max_a q_{\pi_k}(s, a).$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}. The policy improvement theorem then applies to π_k and π_{k+1} because for all $s \in \mathcal{S}$:

$$q_{\pi_k}(s, \pi_{k+1}(s)) = q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s, a)) = \max_a q_{\pi_k}(s, a) \geq q_{\pi_k}(s, \pi_k(s)) \geq v_{\pi_k}(s).$$

Whence our policy improvement assures us that eacy π_{k+1} is uniformly better than π_k, or just as good as it, in which case they are both optimal policies. This ensures that the overall process converges to the optimal policy and optimal value function; in this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

Removing the assumption that policy evaluation operates on an infinite number of episodes There are two approaches. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation; measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to ensure that these bounds are sufficiently small. Although this approach can guarantee convergence, it doesn't do so fast enough. The second approach is to give up trying to complete policy evaluation before returning to policy improvement; on each evaluation step we move the value function *toward* q_{π_k}, but we don't expect to get close except over many steps.[10] For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states *visited* in the episode.

Note that the pseudocode in algorithm 6 is inefficient because it maintains a listing of all returns for each state-action pair and repeatedly calculates their mean; it would be more efficient to maintain just the running average and a count (for each state-action pair) and update them incrementally.

Convergence of Monte Carlo with Exploring Starts Suppose toward contradiction that our algorithm converges to a suboptimal policy. Realize that the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal.

[10] One extreme form of this idea is value iteration, in which only one iteration of iterative policy evaluation is performed before each step of policy improvement. The in-place version of value iteration is even more extreme in that we alternate between improvement and evaluation steps for single states.

Algorithm 6: Monte Carlo Exploring Starts, for estimating $\pi \approx \pi_*$

```
// Initialize.
π(s) ∈ A(s) (arbitrarily), for all s ∈ S
Q(s, a) ∈ ℝ (arbitrarily), for all sinS, a ∈ A(s)
Returns(s, a) ← empty list, for all sinS, a ∈ A(s)
for each episode (without termination) do
    Choose S_0 ∈ S, A_0 ∈ A(S_0) randomly such that all pairs have probability > 0.
    Generate an episode from S_0, A_0 following π: S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T.
    G ← 0.
    for each step of the episode t = T − 1, T − 2, ..., 0 do
        G ← γG + R_{t+1}
        if S_t, A_t does not appear in S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1} then
            Append G to Returns(S_t, A_t)
            Q(S_t, A_t) ← average(Returns(S_t, A_t)) // Policy Evaluation
            π(S_t) ← arg max_a Q(S_t, a)             // Policy Improvement
        end
    end
end
```

Solving Blackjack using Monte Carlo with Exploring Starts Because all episodes are simulated games, we can arrange for exploring starts that include all possibilities. We simply pick the dealer's cards, the payer's sum, and whether or not the player has a usable ace, all at random and with equal probabilty from their respective domains; our previous formulation of blackjack already suits these environment conditions, all we have to do is take a random action at the start of each episode. As the initial policy, we can arbitrarily use the policy that sticks only on 20 or 21. The initial action-value function can be zero for all state-action pairs.

5.1.4 Monte Carlo Control without exploring starts

The only way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches for this: (i) *on-policy* methods which attempt to evaluate or improve the policy that is used to make decisions, e.g. Monte Carlo with exploring starts, and (ii) *off-policy* methods which evaluate or improve a policy different from that used to generate the data. We first consider on-policy methods.

On-policy methods In on-policy control methods, the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, but gradually shifted closer and closer to a deterministic optimal policy. We will use an ϵ-greedy policy, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ϵ they instead select a random action. All non-greedy actions are given the minimal probability of selection of $\frac{\epsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$ is given to the greedy action. Note that ϵ-greedy policies are examples of ϵ-*soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\epsilon > 0$; among the family of ϵ-soft policies, ϵ-greedy policies are in some sense those that are closest to greedy. Note that in defining our policy in this way, we ensure the agent has a positive probability of visiting each state-action pair indefinitely. Further, realize that all ϵ-soft policies are stochastic.

Example of when exploring starts is non-feasible The assumption of exploring starts is not always reasonable. For example, what's the initial state-action pair for a self-driving car? How can we ensure

the agent has a chance of starting in all possible states? We would need to put the car in many different configurations, e.g. in the middle of a busy freeway.

Overall idea of on-policy Monte Carlo methods is GPI We will use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of nongreedy actions. In our on-policy method we will move toward an ϵ-greedy policy; for any ϵ-soft policy, π, any ϵ-greedy policy with respect to q_π is guaranteed to perform better than or equal to π.

Algorithm 7: On-policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

Input: $\epsilon > 0$.
Output: An optimal ϵ-soft policy.
`// Initialization.`
$\pi \leftarrow$ an arbitrary ϵ-soft policy `// Note we require initialization with an` ϵ`-soft policy.`
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$\texttt{Returns}(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for *each episode, without termination* **do**
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ `// Just follow` π`, no ES.`
  $G \leftarrow 0$
  for *each step of the episode* $t = T-1, T-2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if S_t, A_t *not in* $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ **then**
      Append G to $\texttt{Returns}(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \texttt{average}(\texttt{Returns}(S_t, A_t))$
      `// Policy improvement`
      $A^* \leftarrow \arg\max_a Q(S_t, a)$ `// With ties broken arbitrarily.`
      for *each* $a \in \mathcal{A}(s)$ **do**

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

      end
    end
  end
end

Policy improvement theorem implies improving policies at each iteration We are guaranteed by the policy improvement theorem that any ϵ-greedy policy with respect to q_π is an improvement over any ϵ-soft policy π. To see this, let π' be the ϵ-greedy policy. The conditions of the policy improvement theorem still apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \max_a q_\pi(s,a) \qquad (24) \\ &\geq \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \sum_a \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} q_\pi(s,a) \end{aligned}$$

where the sum is a weighted average with non-negative weights summing to 1, and as such must be less than or equal to the largest number averaged.

$$= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \tag{25}$$

$$= v_\pi(s). \tag{26}$$

Whence by the policy improvement theorem $\pi' \geq \pi$ (i.e. $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$; equality only holds when both π' and π are optimal among the ϵ-soft policies.

Proving that we get strictly better policies at each iteration Consider a new environment that is just like the original, except with the requirement that policies must be ϵ-soft "moved inside" the environment; The new environment has the same action and state set as the original and behaves as follows. If in state s and taking action a, then with probability $1-\epsilon$ the new environment behaves exactly like the old environment, but with probability ϵ it repicks the action at random with equal probabilities and then behaves like the old environment with the new, random action. Realize that the best one can do in this new environment with general policies is the same as the best one could do in the original environment with ϵ-soft policies. Let $\tilde{v}_*$ and $\tilde{q}_*$ denote the optimal value functions for the new environment. Then, a policy π is optimal among ϵ-soft policies $\iff$ $v_\pi = \tilde{v}_*$. From the definition of $\tilde{v}_*$, we know that is the unique solution to

$$\begin{aligned}
\tilde{v}_*(s) &= (1-\epsilon) \max_a \tilde{q}_*(s,a) + \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a \tilde{q}_*(s,a) \\
&= (1-\epsilon) \max_a \sum_{s',r} \Pr(s',r|s,a) \left[r + \gamma \tilde{v}_*(s') \right] + \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s',r} \Pr(s',r|s,a) \left[r + \gamma \tilde{v}_*(s') \right].
\end{aligned}$$

When equality holds and the ϵ-solf policy is no longer improved, then we also know from equation 24 that

$$\begin{aligned}
v_\pi(s) &= (1-\epsilon) \max_a q_\pi(s,a) + \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) \\
&= (1-\epsilon) \max_a \sum_{s',r} \Pr(s',r|s,a) \left[r + \gamma v_\pi(s') \right] + \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s',r} \Pr(s',r|s,a) \left[r + \gamma v_\pi(s') \right].
\end{aligned}$$

Realize that this equation is the same as the previous one, except for the substitution of v_π for $\tilde{v}_*$, and because $\tilde{v}_*$ is the unique solution is must be the case that $v_\pi = \tilde{v}_*$.

Policy iteration works for ϵ-soft policies Effectively, this shows that policy iteration works for ϵ-soft policies; i.e. using the natural notion of greedy policy for ϵ-soft policies, one is assured of improvement at each step except when the best policy has been found among the family of ϵ-soft policies. Although we *only* achieve the best policy among ϵ-soft policies, we have removed the assumption of exploring starts.

5.1.5 Off-policy Prediction via Importance Sampling

Target and Behavior policies All learning control methods face a dilemma in that they wish to learn action-values condtional on subsequent *optimal* behavior, but in order to explore all actions and find which are optimal they need to behave non-optimally. How can we learn an optimal policy whilst still behaving according to an exploratory policy? On-policy approaches are one method – they learn action values not for the optimal policy but for a near-optimal policy that still explores. An arguably more straightforward approach would be to use two policies: one that is learned and becomes the optimal policy, i.e. the *target policy*, and another that is more exploratory used to generate behavior, i.e. the *behavior policy*. In this case we say that learning is from data "off" the target policy, and the overall process is coined *off-policy learning*.

Comparison of on-policy vs. off-policy methods Although off-policy learning is arguably the key to learning multi-step predictive models of the world's dynamics, they don't come for free.

- On-policy methods are generally simpler, whereas off-policy methods require additional concepts and notation.
- Off-policy methods are often of greater variance and slower to converge, but at the same time they are more powerful and general; they include on-policy methods as a special case in which the target and behavior policies are the same.

The prediction problem Here, both target and behavior policies are considered fixed and given, and our goal is to estimate one of v_π or q_π for our target policy π. Unfortunately, all we have are episodes following another policy b, where $b \neq \pi$; in this case π is the target policy and b is the behavior policy.

The assumption of *coverage* In order to use episodes from b to estimate values for π, we *require* that every action taken under π is also taken, at least occassionally, under b; i.e. we require that $\pi(a|s) > 0 \implies b(a|s) > 0$, often referred to as the assumption of *coverage*. It follows that b must be stochastic in states where it is not identical to π. On the other hand, the target policy π could be deterministic, e.g. a deterministic greedy policy with respect to the current estimate of the action-value function. The target policy becomes a deterministic optimal policy while the behavior policy remains stochastic and more exploratory, for example, an ϵ-greedy policy.

Deriving importance sampling We start with the definition of expectation:

$$\begin{aligned}
\mathbb{E}_\pi[X] &\coloneqq \sum_{x \in X} x\pi(x) \\
&= \sum_{x \in X} x\pi(x)\frac{b(x)}{b(x)} && \text{Multiplying by 1.} \\
&= \sum_{x \in X} x \underbrace{\frac{\pi(x)}{b(x)}}_{\substack{\text{importance sampling} \\ \text{ratio}} \coloneqq \rho(x)} b(x) && \text{Algebraic re-arrangement.} \\
&= \mathbb{E}_b[X\rho(X)].
\end{aligned}$$

To compute an importance-sample expectation in practice, it amounts to taking a weighted sum where the weights are given by $\rho(\cdot)$. So,

$$\mathbb{E}_\pi[X] \approx \frac{1}{n}\sum_{i=1}^{n} x_i \rho(x_i) \qquad x_i \sim b$$

Example computation with importance sampling Suppose we are given the following probability mass functions: Further, suppose we draw three realizations $x_i \sim b$, where in particular we yield values

x	$b(x)$	$\pi(x)$
1	0.85	0.30
2	0.05	0.40
3	0.05	0.10
4	0.05	0.20

$x_b = [1, 3, 1]$. In this example, the importance-sample average is given by

$$\frac{1}{n}\sum_{i=1}^{n} x\rho(x) = \frac{\left(1 \times \frac{0.3}{0.85}\right) + \left(3 \times \frac{0.1}{0.05}\right) + \left(1 \times \frac{0.3}{0.85}\right)}{2} = 2.24.$$

In this example, it happens to be that $\mathbb{E}_\pi[X] = 2.2$, so we were not too far off.

Importance sampling as a way of estimating expected values under target distribution given observations drawn from behavior policy Off-policy methods utilizie *importance sampling* in order to estimate expected values under the target distribution given samples where our agent acts according to the behavior policy. The strategy is to weight returns according to their relative probability of their trajectories occurring under the target and behavior policies using an *importance-sampling ratio*. Given a starting state S_t, the probability of the subsequent state-action trajectory $A_t, S_{t+1}, A_{t+1}, \ldots, S_T$, occurring under any policy π is

$$\begin{aligned}\Pr(\{A_t, S_{t+1}, A_{t+1}, \ldots, S_T | S_t, A_{t:T-1} \sim \pi\} &= \pi(A_t|S_t)\Pr(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1})\Pr(S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)\Pr(S_{k+1}|S_k, A_k),\end{aligned}$$

where our probabilities are the state-transition probabilities, i.e. $\Pr(s'|s,a) \coloneqq \Pr(S_t = s'|S_{t-1} = s, A_{t-1} = a) = \sum_{r\in\mathcal{R}} \Pr(s', r|s, a)$. The relative probability of the trajectory under the target and behavior policies (i.e. the importance sampling ratio) is

$$\rho_{t:T-1} \coloneqq \frac{\Pr(\text{trajectory under } \pi)}{\Pr(\text{trajectory under } b)} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)\Pr(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)\Pr(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

In moving to the last equality, there was a cancellation of terms; this is critical as these trajectory probabilities depend on the MDP transition probabilities which are generally unknown. Whence, this last equality shows us that the importance sampling ratio depends only on the two policies and the sequence, not on the MDP. Recall that we wish to estimate the expected returns (values) under the target policy, but we only have access to returns G_t due to the behavior policy. These returns have the wrong expectation, namely $\mathbb{E}[G_t|S_t = s] = v_b(s)$ and cannot be averaged to obtain v_π. Instead, we will use importance sampling, where the ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value,

$$\mathbb{E}_b[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s). \tag{27}$$

Monte Carlo algorithm for Importance Sampling It is at this point that we are ready to provide our Monte Carlo algorithm for averaging returns from of observed episodes following policy b to estimate $v_\pi(s)$. It will be convenient to define time steps in a way that increases across episode boundaries, e.g. if the first episode ends in a terminal state at time $t = 100$, then the next episode is said to begin at time $t = 101$; in this way, we can refer to particular steps in particular episodes. For an every-visit method, define the *set* of all time steps in which state s is visited by $\mathcal{T}(s)$, whereas for a first-vsit method $\mathcal{T}(s)$ would only include time steps that were first visits to s within their episodes. Further, denote by $T(t)$ the first time of termination following time t, and G_t the return after t up through $T(t)$. Then $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertain to state s, and $\{\rho_{t:T-1}\}_{t\in\mathcal{T}(s)}$ are the corresponding importance-sampling ratios. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and then average the results

$$V(s) \coloneqq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{|\mathcal{T}(s)|}. \tag{28}$$

When importance sampling is done as a simple-average in this way we call it *ordinary importance sampling*. An alternative is *weighted* importance sampling, which uses a weighted average defined as

$$V(s) \coloneqq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}, \tag{29}$$

which we define to be zero if the denominator is zero.

Comparing ordinary vs. weighted importance sampling Let's consider the estimates of each method under the first-visit regime after observing a single return from a state s. In the weighted-average estimate, the ratio $\rho_{t:T(t)-1}$ for the single return will cancel in both the numerator and denominator, so that the estimate is equal to the observed return independent of the ratio (assuming the ratio is non-zero)! Given that this return was the only one observed, this is not unreasonable however the expectation of this quantity is $v_b(s)$ rather than $v_\pi(s)$, i.e. in the statistical sense we have a biased estimator. In contrast, the first-visit version of the ordinary importance sampling estimator in equation 28 is *always* $v_\pi(s)$ in expectation, i.e. it is unbiased; however, its value can be extreme. E.g. suppose the ratio were ten, indicating trajectory observed is ten times more likely under the target policy than it is under the behavior policy: in this case the ordinary importance sampling estimate would be *ten times greater* than the observed return, and this is peculiar since the importance sampling ratio instructed us that the episode's trajectory is considered "more representative" of the target policy.

Bias variance tradeoff The differences between the first-vist methods are expressed in terms of their bias-variance tradeoffs.

- Formally, ordinary importance sampling is unbiased but has unbounded variance due to the fact that the variance of the ratios can be unbounded.
- The weighted importance sampling scheme on the other hand is biased (although the bias converges asymptotically to zero), but has the benefit that the largest weight on any single return is one, whence if we assume bounded returns the variance of the weighted importance-sampling estimator converges to zero, even if the ratios themselves are infinite!

In practice the weighted estimator has lower variance and is strongly preferred.

Every visit methods for ordinary and weighted importance sampling Both of these every-visit methods are biased, although this bias drops to zero asymptotically as the number of samples increases. In practice, we prefer every-visit methods because they remove the need to keep track of which states have been visited and which haven't, and we can more easily extend them to approximations.

An example of infinite variance Ordinary importance sampling estimates will typically have infinite variance whenever the returns have infinite variance, and this can happen easily in off-policy learning when trajectories contain loops. Suppose we have only one non-terminal state s and there are only two actions `left` and `right`. Taking the `right` action leads us to the terminal state with no reward, whereas taking the `left` action transitions the agent with probability 0.9 back to state s and with probability 0.1 to the terminal state where in the latter case the agent picks up a reward of $+1$. Realize that the target policy is to always select `left`, since this is the only way to (possibly) pick up a reward: all episodes under this policy consist of some number of (possibly zero) transitions back to s followed by termination with a return of unit value. Since our task is episodic, we don't discount and the value of s under the target policy is one. Now, suppose that we are attempting to estimate this state-value from off-policy data where the behavior policy selects between `left` and `right` with equal probability. To see that the variance

of the ordinary importance-sampling technique is infinite, first recall that $\texttt{Var}(X) \coloneqq \mathbb{E}\left[(X - \bar{X})^2\right] = \mathbb{E}\left[X^2 - 2X\bar{X} + \bar{X}^2\right] = \mathbb{E}\left[X^2\right] - \bar{X}^2$, whence if our mean is finite (which it is in our example) then the variance is infinite $\iff$ the expectation of the square of the random variable is infinite. Thus, we need to show that

$$\mathbb{E}_b\left[\left(\prod_{t=0}^{T-1}\frac{\pi(A_t|S_t)}{b(A_t|S_t)}G_0\right)^2\right] = \infty.$$

To compute this expectation, we break it down into cases defined based on episode length and termination. Realize that for any episode observed under our behavior policy that ends with the `right` action, the importance sampling ratio is zero because the target policy never selects this action. Therefore we are left with considering episodes that involve some number (possibly zero) of `left` actions that transition back to the only state s, followed by a `left` action that transitions to termination. All of these episodes have a return of exactly unit value, and so we can ignore G_0. To get the expected square of the random variable we now need only consider the length of each episode, where we will multiply the probability of the episode's occurrence by the square of its importance sampling ratio and then add all these terms up.

$$\begin{aligned}
&= \underbrace{\frac{1}{2}\cdot 0.1}_{\substack{\text{probability of observing}\\ \text{episode under } b}} \left(\frac{1}{0.5}\right)^2 && \text{the length 1 episode}\\
&\quad + \frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.1\left(\frac{1}{0.5}\frac{1}{0.5}\right)^2 && \text{the length 2 episode}\\
&\quad + \frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.1\left(\frac{1}{0.5}\frac{1}{0.5}\frac{1}{0.5}\right)^2 && \text{the length 3 episode}\\
&\quad + \dots\\
&= 0.1\sum_{k=0}^{\infty} 0.9^k\cdot 2^k\cdot 2 = 0.2\sum_{k=0}^{\infty} 1.8^k = \infty.
\end{aligned}$$

A note on finding optimal deterministic policies Realize that off-policy learning with an ϵ-soft behavior policy and a deterministic target policy is capable of finding an optimal deterministic policy.

Algorithm for off-policy Monte Carlo prediction We provide an algorithm here.

Realize that it is valid to compute ρ incrementally as we have done in algorithm 8. To see why realize that our algorithm iterates from time step $T-1$ backward to time step t,

$$\begin{aligned}
\rho_{t:T-1} &\coloneqq \prod_{k=t}^{T-1}\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\\
&= \rho_t\rho_{t+1}\rho_{t+2}\underbrace{\cdots}_{\longleftarrow}\rho_{T-2}\rho_{T-1}
\end{aligned}$$

Whence we see that $W_1 \leftarrow \rho_{T-1}$, $W_2 \leftarrow \rho_{T-1}\rho_{T-2}$, $W_3 \leftarrow \rho_{T-1}\rho_{T-2}\rho_{T-3}$, etc., and that each time step adds one additional term to the product and reuses all previous terms; we compute this recursively without having to store all past values of ρ.

5.1.6 Summary

Advantages of MC over DP methods Monte Carlo methods present four advantages over DP methods.

Algorithm 8: Every-visit MC prediction, for estimating $V \approx v_\pi$

Input: A policy π to be evaluated, a policy b which dictates behavior
`// Initialization`
$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
`Returns(s)` $\leftarrow$ an empty list, for all $s \in \mathcal{S}$
for *each episode, without termination* **do**
  Generate an episode following $b : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$, $W \leftarrow 1$
  for *each step of the episode* $t = T-1, T = -2, \ldots, 0$ **do**
    $G \leftarrow \gamma W G + R_{t+1}$
    Append G to `Returns`(S_t)
    $V(S_t) \leftarrow$ `average(Returns`(S_t)`)`
    $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
  end
end

1. They can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics.

2. They can be used with simulation or *sample models*; for surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods.

3. It is easy and efficient to *focus* MC methods on a small subset of the states; a region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set.

4. They may be less harmed by violations of the Markov property, because they do not update their value estimates on the basis of the value estimates of successor states, i.e. because they do not bootstrap.

MC methods adhere to GPI Rather than use a model to compute the value of each state, MC methods simply average many returns that start in that state. Because a state's value is the expected return, this average can become a good approximation to the value (function). In control methods we are interested in approximating the action-value function because they can be used directly to optimize the policy without requiring a model of the environment's transition dynamics. MC methods interweave policy evaluation and policy improvement on an episode-by-episode basis, and they can be incrementally implemented on an episode-by-episode basis as well.

The problem of maintaining exploration It's not enough to simply act upon an existing optimal policy deterministically, because we will not observe returns for all states in this case, and without being able to evaluate against alternatives we may never learn the optimal action. One approach is to assume exploring starts, i.e. that each episode starts in an arbitrary state-action pair with strictly positive probability; exploring starts aren't always feasible e.g. autonomous cars. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Off-policy prediction This is the idea of learning the value function of a *target policy* when data was generated by following a different *behavior policy*. These learning methods rely on *importance sampling* to

reweight returns by the ratio of the probabilities of the trajectory, i.e. taking the observed action(s) and landing in the observed state(s), under the two policies; this has the effect of transforming the expectation of the behavior policy to match the expectation under the target policy. *Ordinary* importance sampling uses a simple average of weighted returns, is unbiased, but has large(r) possibly infinite variance, whereas on the other hand *weighted* importance sampling uses a weighted average, is a biased estimator, but always has finite variance (if returns are bounded) and is preferred in practice.

Contrasting MC with DP There are two major differences.

1. MC methods operate on sample experience, and thus can be used for direct learning without a model.
2. MC methods do not bootstrap, i.e. they do not update their value estimates based on the value estimates for other states.

It's interesting to note that these two differences are not tightly linked, and can be de-coupled. In the next section, we will learn about methods that learn from experience, similar to MC, but also bootstrap, similar to DP.

5.2 Temporal Difference Learning

Temporal Difference (TD) methods are a combination of Monte Carlo (MC) and Dynamic Programming (DP). Like MC 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, i.e. they bootstrap; the combination is more powerful than the sum of its parts.

5.2.1 TD Prediction

TD and MC methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v_π for non-terminal states S_t occurring in that experience. Roughly speaking, MC methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$; this happens at the end of an episode.

Constant α-MC A simple every-visit MC method appropriate for non-stationary environments is given by the update rule

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

where here G_t is the actual return following time t, and α is a constant step-size parameter. We call this method constant-α MC. Realize that this incremental update uses a constant step size, and that we can form a Monte Carlo estimate without saving lists of returns. However, also not that the update rule depends on G_t and therefore we have to take a sample of full trajectories; this means that we don't learn *during* an episode. To learn incrementally before the end of an episode, we need a new strategy.

One-step TD Whereas a constant-α MC requires we wait until the end of the episode to determine the increment to $V(S_t)$, in contrast TD methods need only wait 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 $V(S_{t+1})$. We've seen how we can define returns recursively:

$$G_t \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = R_{t+1} + \gamma G_{t+1}$$

and if we substitute this expression into our value-function for a state, we end up with a new update rule.

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

This gives us a recursive expression for the value function; the simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{30}$$

immediately upon transition to S_{t+1} and receiving R_{t+1}. We can think of $\gamma V(S_{t+1})$ as an approximation for the return we would get if we waited until the end of the episode. In effect, the target for the MC update is G_t whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. The method we've outlined here is referred to as TD(0) or one-step TD, and it is a special case of a family of algorithms called $TD(\lambda)$ and n-step TD.

Algorithm 9: Tabular TD(0) for estimating v_π

Input: The policy π to be evaluated; step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\texttt{terminal}) = 0$
for *each episode* **do**
  Initialize S
  for *each step in the episode, until state S is terminal* **do**
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S') - V(S)\right]$
    $S \leftarrow S'$
  end
end

How TD(0) combines sampling of MC with bootstrapping of DP Realize that TD(0) bases its updates in part on an existing estimate, therefore we say that it is a bootstrapping method, similar to DP. Recall from before that

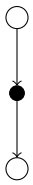
$$\begin{aligned} v_\pi(s) &\coloneqq \mathbb{E}_\pi \left[G_t | S_t = s\right] && && (31)\\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] && \text{from equation 8} && (32)\\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right] && && (33) \end{aligned}$$

Roughly speaking, MC methods use an estimate of 31 as a target, whereas DP methods use as estimate of 33 as a target. To be clear, the MC target is an *estimate* because the population expectation is unknown, and in place of this we use a sample average of returns. The DP target is also an *estimate*, where in this case we know the environment dynamics and so the expectation is well defined and available to compute, but instead it's an estimate because $v_\pi(S_{t+1})$ is not known and instead we use $V(S_{t+1})$. Finally, the TD target is an estimate for both reasons: it samples the expected returns as seen in equation 33 *and* it uses the current estimate of V instead of the true v_π. This is how TD combines the sampling of MC with the bootstrapping of DP.

Emphasizing differences between DP and TD In DP, we use an expectation over all possible next states, and this expectation requires a model of the environment. In TD, we only need the next state, and we can get that directly from the environment without a model.

TD error Finally, realize that the expression $[R + \gamma V(S') - V(s)]$ is sort of an error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity is called the *TD error*, and it arises in various forms in reinforcement learning:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{34}$$



TD(0)

Figure 25: Here we show the backup diagram for tabular TD(0). The value estimate for the root note is updated on the basis of one sample transition from it to the immediately following state. We refer to TD and MC updates as *sample updates* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compue a backed-up value, and then updating the value of the original state (or state-action pair) accordingly. On the other hand, DP updates are *expected* in the sense that they are based on a complete distribution of all possible successors.

Observe that the TD error at each time step is the error in the estimate *made at that time*, but in order to compute it realize that it depends on the next state and next reward, so it's not actually available until one time step later. I.e. δ_t is the error in $V(S_t)$ and it is available to us at time step $t+1$.

Rich Sutton on the importance of TD learning TD learning is specialized toward prediction learning, which makes it perhaps the most important thing for AI of this century. Prediction learning is scalable because it's effectively unsupervised supervised learning, i.e. we have a target (just by waiting) yet no human labeling is needed! Therefore, prediction learning is scalable model-free learning; we don't even need things like objective functions. TD is a kind of learning that's specialized for learning to predict: as time passes by we make a sequence of predictions and after enough time passes we learn what the correct answer was for all of those predictions. TD learning can be thought of as supervised learning where we backpropagate the error. We might question why we can't simply wait until the target is known and then use a one step method, e.g. wait until the end of the game and then regress to the outcome. In reality, there are significant computational costs to this: memory scales with the # of steps in the prediction, and computation is poorly distributed over time. We can avoid these pitfalls if we're willing to use learning methods that are specialized for multi-step, but sometimes it's also the case that the target is never known e.g. off-policy learning. These aren't nuissances that come up by chance, they are *clues* or hints from nature as to how AI really works.

An identity: MC error as a sum of TD errors If during the execution of our algorithm it is the case that the array V does not change during the episode (similar to how it doesn't change in MC methods), then the MC error can be written as a sum of TD errors:

$$
\begin{aligned}
G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{from equation 8} && (35)\\
&= \delta_t + \gamma\left(G_{t+1} - V(S_{t+1})\right) && && (36)\\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\left(G_{t+2} - V(S_{t+2})\right) && && (37)\\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \ldots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\underbrace{\left(G_t - V(S_T)\right)}_{=0-0} && && (38)\\
&= \sum_{k=1}^{T-1}\gamma^{k-t}\delta_k. && && (39)
\end{aligned}
$$

Of course, this identity is not exact if V gets updated during the episode, like it does in TD(0), but if the step size is small enough this identity approximately holds. Generalizations of this identity play an important role in the theory of RL.

Example: driving home Suppose that each day you leave from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. You have a good estimate of how long it takes to get home from various situations you might encounter along the way. Along a single journey home, you might observe the following states alongside predictions of when we think we'll arrive at home.

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
Leaving office, Friday at 6	0	30	30
Reach car, raining	5	35	40
Exiting highway	20	15	35
2ndary road, behind truck	30	10	40
Entering home street	40	3	43
Arrive home	43	0	43

Table 2: The rewards in this example are the elapsed times on each leg of the journey; if we wished to minimize travel time we'd use the negative of travel times but to keep things simple let's stick with positive valued rewards. We are not discounting, since our task is episodic (each time we arrive home, the episode terminates), so $\gamma = 1$, and thus the return for each state is simply the actual time to go from that state home. It follows that the value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.

One way to view the operation of MC methods is to plot the predicted total time over the sequence. If we set $\alpha = 1$, the recommended changes in predictions as given by the constant-α MC method will be exactly equal to the error between the estimated value (predicted time to go) in each state and the actual return (actual time to go). E.g. when we exited the highway, we thought it would take only 15 minutes more to get home, but it in fact took 23 minutes; the error $G_t - V(S_t)$ at this time is eight minutes, and so if we had a step size of $\alpha = 1/2$ the predicted time to go after exiting the highway would be revised upward by four minutes as a result of experiencing this episode! The critical part, however, is to realize that this update can only be made once you get home, at least that's what the MC method suggests; it is only at this point that you know any of the actual returns. On the other hand, is it really necessary to wait until the final outcome is known before learning can begin? Perhaps if we get stuck in traffic along our way home, we can increase our estimate for our initial state before waiting to actually get home. It's easy to see how it works when $\alpha = 1$, in this case the terms $V(S_t)$ on the right hand side of the TD(0) update rule cancel and our update reduces to $V(S_t) \leftarrow R_{t+1} + \gamma V(S_{t+1})$, where here $\gamma = 1$ since our task is episodic. So, our update rule with $\alpha = 1$ is to simply add our commute time between states (or legs of our journey) to our best estimate for how long it would take to get home from the next state; as we make our way home, we can learn online without waiting to know the final outcome. The TD approach suggests that each error is proportional to the change over time of the prediction, i.e. to the *temporal differences* in predictions.

TD and generalization: a scenario where TD update is better on average than an MC update Suppose you have lots of experience driving home from work. Then, you move to a new building and a new parking lot, but you still enter the highway at the same place! We now start to learn predictions for the new building; in this case, TD updates would allow us to leverage our existing knowledge of traffic patterns in order to yield more efficient updates initially as opposed to MC methods.

5.2.2 Advantages of TD Prediction Methods

Questioning TD Methodology TD methods effectively update their estimates based in part on other estimates. They learn a guess from another guess – i.e. they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over MC and DP methods? Answering these questions is not an easy task, but we develop some intuition here.

Advantages of TD over MC or DP

- TD methods have the advantage over DP methods that they do not require a model of the environment i.e. of its reward and next-state transition probabilities.
- TD methods can be implemented online in a natural fashion with only one time step delay, as opposed to MC methods which require the episode to terminate before updating estimate(s) (since it is only at the end of the episode that the true return is known).
- TD methods can be applied to continuing tasks which have no episodes at all, making them useful across a wider range of tasks when compared with MC methods. Further, some applications have extremely long episodes, so long that delaying all learning until the end of the episode is too slow.
- TD methods learn from each transition regardless of what subsequent actions are taken, whereas MC methods ignore or discount episodes in which experimental actions are taken (which greatly slows learning).

We might wonder whether TD is guaranteed to converge. The answer is yes! For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases accordingly to the conditions in equation 5. As of this writing, it's an open research question which of TD or MC methods converge "faster" to the correct predictions.

5.2.3 Optimality of TD(0)

Batch updating Suppose there is available only a finite amount of experience, e.g. 10 episodes or 100 time steps. A common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. I.e. given an approximated value function V, then we increment according to one of

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

or

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

for every time step t at which a non-terminal state is visited, but the value function is changed only once by the sum of all the increments. Then, all available experience is processed again with the new value function to produce a new overall increment, and so on until the process converges. This process is called *batch updating* because the updates are made only after processing one complete *batch* of training data.

TD(0) converges determinstically to a single answer independent of step-size parameter Note that under batch updating, as long as α is chosen sufficiently small, then no matter its value TD(0) will converge deterministically to a single answer. This is in contrast to using a constant α with a MC method, which would also converge deterministically under the same conditions but to a different answer. We need to understand why these two methods produce two different answers. Let's take a look at a couple of examples first.

Random walk Consider the following *Markox reward process* depicted in figure 26, which is a Markov decision process without actions. Let's compare and contrast TD(0) updates with MC updates. Suppose we initialize all state-values with value 1/2 initially. On the first episode, the agent hops around states C, D, and E before eventually hitting the right most terminating state and picking up a positive unit-valued reward. Since the episode has ended, the MC method can update the state-values for states C, D, and

E (i.e. all the states that were encountered along the last journey), whereas it is interesting to note that the TD(0) method would *only* have updated the state-value for state E. To see this, consider the first transition from C to D. The TD error is equal to the reward on the transition plus the value of the next state D minus the value of state C: the reward is zero because this is a non-terminal transition, and the values of both C and D are 1/2, and so the TD error is zero and we make no change to the estimate for state C. The same thing happens on every time step except the transition to the terminal state from state E. If we allow our agent to experience more episodes, eventually the TD agent akes updates to the values on *every* step, whereas in contrast the MC method requires the agent reach the end of the episode before making any updates.

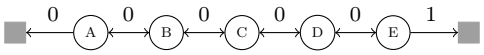


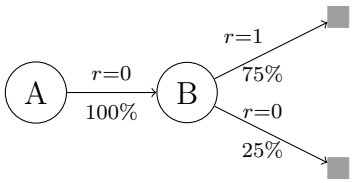
Figure 26: In this Markov Reward Process, all episodes start in the center state C, then proceed either left or right exactly one state each time step with equal probability. The termination states are on the extreme left and extreme right, but notice that it is only by ending the episode by moving to the terminal right node that we incur a +1 reward; all other rewards are zero. Realize that the true value of a state is the probability of terminating in the right most node. It is easy to see that $v_\pi(C) = \frac{1}{2}$. For the true values of all states A through E that they are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$ respectively.

Random walk under batch updating Let's now consider batch-updating versions of TD(0) and constant-α MC as applied to the above problem. After each new episode, all episodes seen so far are treated as a batch and they are then repeatedly presented to the prediction algorithm (one of TD(0) or constant-α MC). We choose α sufficiently small so that we are guaranteed to converge to the value function v_π. Since we can work out the value of each state analytically, we know what the true answer is for each state-value and we can compare the average root mean-squared error across the five states when running the respective learning algorithms across 100 independent repetitions of the experiment. It can be seen that the RMS error is significantly less for the TD(0) method when compared with constant-α MC across all episodes. Under batch training, constant-α MC converges to values $V(s)$ that are sample averages of the actual returns experienced after visiting each state s. From statistics, we know that the sample average minimizes mean squared error at least on our training set, and it is in this sense surprising that TD(0) can outperform constant-α when using RMS-error as the measuring stick. It turns out that MC method is optimal only in a limited way, and that the TD method is optimal in a way that is more relevant to predicting returns.

You are the Predictor Suppose you observe the following eight episodes for an unknown Markov reward process:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

The way to read this is that the first episode started in state A, transitioned to B with a reward of zero, then terminated from B with a reweard of zero. The next seven episodes start from B and terminate immediately, with six of them earning unit reward and the last earning zero reward. Given this batch of data, it is reasonable to assert that $V(B) = \frac{3}{4}$, since the state B was encountered in all eight episodes and six out of these times we incurred unit reward. What is more controversial is the state value of $V(A)$, given this data. ONe answer is that 100% of the time that the process was in state A, it traversed to state B, albeit with an immediate reward of zero, but because state B has value $\frac{3}{4}$ then so should state A (we are not discounting, the task is episodic). One way to view this answer is according to the following diagram



If we compute the correct estimates given this model, indeed it is the case that $V(A) = \frac{3}{4}$, and this is the answer that TD(0) yields. However, there is another reasonable answer! And that is to observe that we have only observed state A once and the return that followed it was zero; we therefore could estimate that $V(A) = 0$. This is the answer that batch MC gives. Realize that this answer gives the minimum mean-squared error on the training data, but we would intuitively expect that the TD(0) answer is better.

Batch MC as minimizing MSE, TD(0) as maximizing likelihood Batch MC methods find estimates that minimize the mean-squared error on the training data set. On the other hand, TD(0) always finds the estimates that would be correct for the maximum likelihood model of the Markov process, i.e. the estimate whose probability of generating the data is greatest.

Certainty-equivalence estimate For our example above "You are the Predictor", the MLE es the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from state i to state j is the fraction of observed transitions that went from i to j, and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct; this is known as the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

Computing certainty-equivalence estimate is not practical Although the certainty-equivalence estimate is in some sense an optimal solution, it is not feasible to compute directly. Denote by $n = |\mathcal{S}|$ the number of states, then simply forming the MLE requires $O(n^2)$ memory, and computing the corresponding value function requires $O(n^3)$ work. It is in this sense quite striking that TD methods can approximate the same solution using memory requirements bounded above by $O(n)$ and with repeated computations over the training set. On tasks with large state spaces, TD methods may be the only viable way of approximating the certainty-equivalence function.

5.2.4 Sarsa: On-policy TD Control

Recall Generalized Policy Iteration Generalized policy iteration combines two parts: policy evaluation and policy improvement. The first algorithm we saw for this was policy iteration, which runs policy evaluation until convergence before greedifying the policy. Then, we saw GPI with Monte Carlo, which performs a cycle of policy iteration and improvement every episode. To better remember GPI with MC, we go back to a simple example, as depicted in figure 27.

Realize that GPI with MC does not perform a full policy evaluation step before improvement, but instead it evaluates and improves after each episode. If we go even further, we could choose to improve the policy after only one policy evaluation step! This is what we will do with TD control.

Estimating the value of state-action pairs The first step in using GPI with TD methods for the control method is to learn an action-value function rather than a state value function. We must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. Recall that an episode consists of alternating sequences of states and state-action pairs as in figure 28.

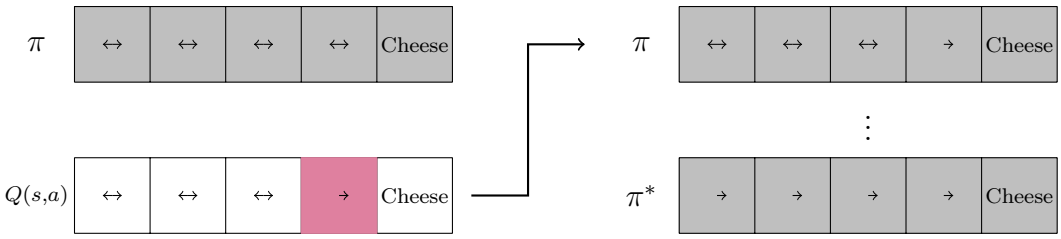


Figure 27: Suppose that we have a four-state corridor with cheese at the end. A mouse, our agent, starts out knowing nothing and follows a random policy. Eventually, the mouse will stumble into the cheese simply by moving randomly, and it is as that point that the episode concludes, and with Monte Carlo GPI we update action values $Q(s, a)$. Then, it improves its policy by greedifying with respect to the action values. This process then repeats, eventually learning the optimal policy.

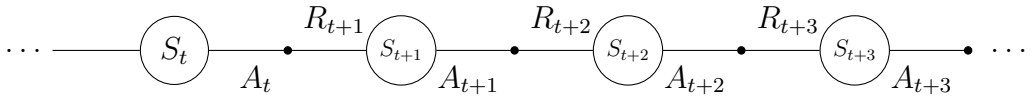


Figure 28: Whereas we previously considered transitions from state to state and learned the values of states, we now consider transitions from state-action pair to state-action pair, and we learn the values of state-action pairs. Mathematically, the same theorems assuring convergence of state values under TD(0) also apply to the corresponding algorithm for action-values.

Sarsa Our update becomes:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]. \tag{40}$$

We perform this update for each transition from a nonterminal state S_t, and if S_{t+1} is terminal then we define $Q(S_{t+1}, A_{t+1}) = 0$. Since each updates involves the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next we call this method *Sarsa*.

On-policy control method We can easily design an on-policy control algorithm based on the Sarsa prediction method: we continually estimate q_π for the behavior policy π and at the same time make π greedy with respect to q_π. The convergence properties of the algorithm depend on the nature of the policies dependence on Q, e.g. we could use ϵ-greedy or ϵ-soft policies. Sarsa converges almost surely to an optimal policy and action-value function so long as all state-action pairs are visited infinitely many times, and the policy converges asymptotically to the greedy policy; one idea is to set $\epsilon = 1/t$ so that in the limit we recover an optimal policy.

Algorithm 10: Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\texttt{terminal}, \cdot) = 0$. **for** *each episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g. ϵ-greedy)
  for *each step in the episode, until state S is terminal* **do**
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g. ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A) \right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  end
end

Sarsa on the Windy gridworld Let's see how Sarsa behaves on an example.

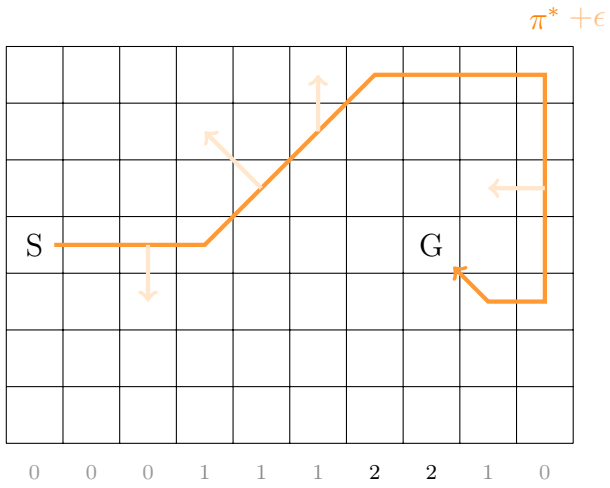


Figure 29: In this episodic task, an agent seeks to learn how to get from the starting state, denoted by S, to the goal state, denoted by G. In addition to the agent being able to move left, right, up, or down, there is also an upward wind that is indicated below each column: the units indicate how many spaces upward the agent will move when taking an action from a state within a particular column. Moving into a boundary does nothing. The agent receives a reward of -1 per step, incentivizing it to want to reach the goal quickly. Since it's an episodic task, we set $\gamma = 1$. Additionally, we choose $\alpha = 1/2$ as our learning rate, and we choose an ϵ-greedy action selection policy with $\epsilon = 0.1$. We set our initial values optimistically to encourage systematic exploration. Realize that a MC method would not be a great fit here, because many policies do not lead to termination. MC methods only learn when an episode terminates, and so a deterministic policy might get trapped and never learn a good policy in this windy grid world; e.g. if the agent takes the left action from the starting state, it would never terminate. Sarsa avoids this trap because it would learn such policies are bad *during* the episode, and so it would switch policies and not get stuck.

5.2.5 Q-learning: Off-policy TD Control

What is Q-learning? Recent applications like learning to play Atari games, controlling traffic signals, or even automatically configuring web systems have all been built atop a single algorithm called Q-learning; there is a special relationship between Q-learning and the Bellman optimality equations. Q-learning is an *online* algorithm. This algorithm was one of the early breakthroughs in RL, defined by the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{41}$$

We call the learned action-value function Q and it is meant to directly approximate q_*, the optimal action-value function, independent of the policy being followed. All that is required for correct convergence is that all state-action pairs continue to be *updated*. Under this assumption of continual updating and a variant of the usual stochastic approximation conditions on the sequence of the step-size parameters, Q learning converges almost surely to q_*. The Q-learning procedure is presented in algorithm 11.

The connection between Sarsa, Q-learning, and the Bellman Optimality Equations Whereas Sarsa uses the value of the next state-action pair in its target, Q-learning on the other hand uses a *max*. The connection goes way back to Dynamic Programming. Realize that the update for Sarsa looks suspiciously similar to the Bellman equation for the action-values; in fact it turns out Sarsa is a sample-based algorithm to *solve* the Bellman equation for action-values.

Sarsa:
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right)$$

Bellman equation:
$$q_\pi(s, a) = \sum_{s', r} \Pr(s', r | s, a) \left(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right).$$

Algorithm 11: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Input: Step size $\alpha \in (0, 1]$, and a small $\epsilon > 0$.
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\texttt{terminal}, \cdot) = 0$
for *each episode* **do**
  Initialize S **for** *each step of the episode, until S is terminal* **do**
    Choose A from S using policy derived from Q (e.g. ϵ-greedy)
    Take action A, observe R, S'
    `// Key difference from Sarsa is in this next step.`
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  end
end

Q-learning also solves the Bellman equation using samples from the environment, but instead of using the standard Bellman equation, Q-learning uses the Bellman's Optimality Equation for action-values.

Q-learning: $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\right)$$

Bellman optimality equation: $$q_*(s, a) = \sum_{s', r} \Pr(s', r | s, a) \left(r + \gamma \max_{a'} q_\pi(s', a')\right).$$

This effectively means that Q-learning can directly learn q_* instead of switching between policy improvement and policy evaluation steps! Sarsa is a sample-based version of policy iteration which uses the Bellman equations for action-values, that each depend on a fixed policy. Q-learning is a sample-based version of value iteration which iteratively applies the Bellman optimality equation. Applying the Bellman optimality equation strictly improves the value function, unless it's already optimal, and so value iteration continually improves as value function estimate, which eventually converges to the optimal solution. For the same reason, Q-learning also converges to the optimal value function, so long as the agent continues to explore and samples all areas in the state-action space.

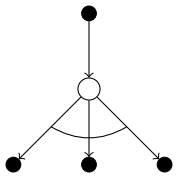


Figure 30: Realize that our update rule 41 updates a state-action pair, and so if we were to inspect a backup diagram for this procedure the root node would be a small, filled action node. The update is also *from* action nodes, maximizing over all possible actions in the next state transition. Thus, the bottom nodes of the backup diagram should be of all these possible action nodes. We may indicate the maximum over these actions by using an arc across them.

Comparing the variance of our target in Sarsa vs. Q-learning Q-learning takes the max over next action values, and so it only changes its action-value function when the agent learns that one action is better than another. In contrast, Sarsa uses the estimate of the next action value in its target, which changes every time the agent takes an exploratory action.

How is Q-learning considered off-policy? So far, all off-policy algorithms we've presented use importance sampling. How can Q-learning be off-policy without using importance sampling? Recall that an agent estimates its value function according to the expected returns under their target policy, but they actually behave according to their target policy. When the target and behavior policy are the same, the

agent is learning on-policy, otherwise the agent is learning off-policy. Sarsa is an on-policy algorithm, because the agent bootstraps off of the value of the action it's going to take next, which is sampled from its behavior policy!

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, \underbrace{A_{t+1}}_{\sim\pi}) - Q(S_t, A_t) \right)$$

On the other hand, Q-learning bootstrapps off of the largest action-value in the next state, which is like sampling an action under an estimate of the optimal policy *rather* than the behavior policy.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

Since Q-learning learns about the best action it could possibly take rather than the actions it actually takes, it is learning off-policy. Q-learning's target policy is always greedy with respect to its current values. However, its behavior policy can be anything that continues to visit all state-action pairs during learning; one possible behavior policy is ϵ-greedy. The difference here between the target and behavior policies confirms that Q-learning is greedy.

Why are there no importance sampling ratios in Q-learning? Since the agent is estimating action values with a known policy, it does not need importance sampling ratios to correct for the difference in action selection. The action-value function represents the returns following each action in a given state, and the agent's target policy represents the probability of taking each action in a given state. Putting these two elements together, the agent can calculate the expected return under its target policy from any given state, in particular, the next state S_{t+1}. Since the agent's target policy is greedy with respect to its action values, all non-maximum actions have probability zero. As a result, the expected return from that state is equal to a maximal action value from that state.

Cliff walking example We can compare Sarsa with Q-learning on a modified gridworld example, as shown in figure 31.

5.2.6 Expected Sarsa

Motivating Expected Sarsa Recall the Bellman equation for action-values. Here we see that the expectation is over values of possible next state-action pairs. In particular, if we break the summation apart we see that there is a sum over all possible next states as well as possible next actions.

$$q_\pi(s, a) = \sum_{s', r} \Pr(s', r | s, a) \left(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right).$$

Sarsa estimates this expectation by sampling the next state from the environment and the next action from its policy. I.e.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(\underbrace{S_{t+1}}_{\sim \Pr(s', r|s, a)}, \underbrace{A_{t+1}}_{\sim \pi(a'|s')}) - Q(S_t, A_t) \right).$$

But, the agent already knows this policy $\pi(a'|s')$, why should it have to sample its next action? Instead, it should just compute the expectation directly. In this case, we can take a weighted sum of the values

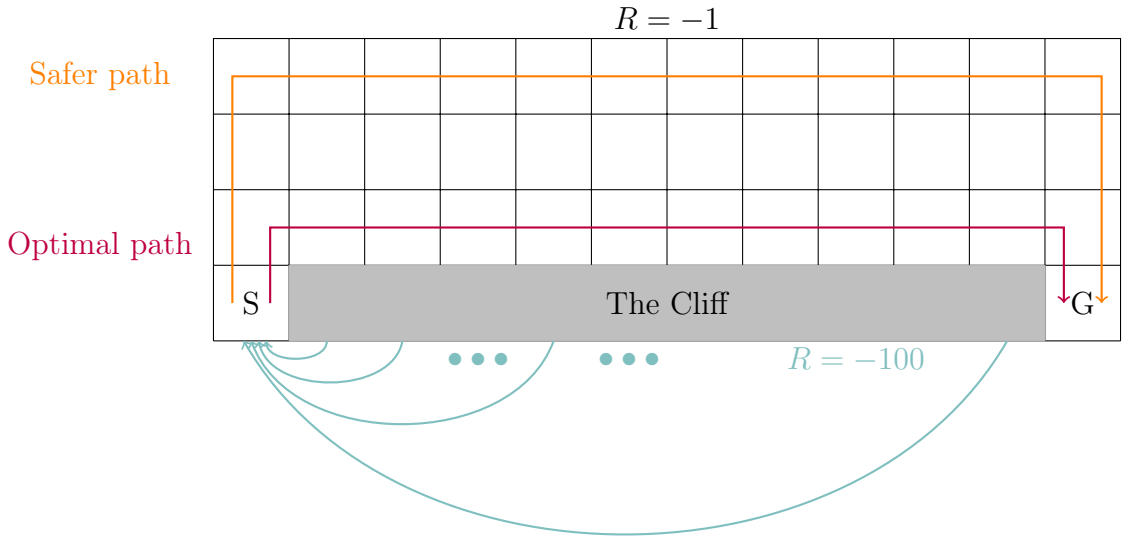


Figure 31: We have a standard undiscounted episodic task with start and goal states and four actions causing movement up, down, left, and right. The reward is -1 on all transitions except into the region marked "The Cliff", in which case the agent incurs a reward of -100 and is reset back to the start (note the episode is not yet terminated, in this case). A Sarsa algorithm can be compared against a Q-learning algorithm with an ϵ-greedy action selection setting $\epsilon = 0.1$. It happens to be that Sarsa learns the safer path and earns a higher sum of rewards during the episode, whereas on the other hand Q-learning is able to recover the optimal path however its exploration strategy forces it to sometimes veer off into the cliff causing it to earn less return over the long run when compared with Sarsa. Sarsa, on the other hand, still uses an exploration algorithm but it takes into account the effect of ϵ-greedy action-selection when determining the best policy, and so that's why it learns the safest path: it protects itself from random actions that could cause it to veer toward the cliff. It's noteworthy that if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy.

of all possible next-actions, where the weights are the probability of taking each action under the agent's policy. I.e. we may replace $Q(S_{t+1}, A_{t+1})$ with

$$\sum_{s'} \pi(a'|S_{t+1}) Q(S_{t+1}, a').$$

This is the core idea behind Expected Sarsa: instead of taking the maximum over next state-action pairs we could instead use the expected value, taking into account how likely each action is under the current policy.

Expected Sarsa has a more stable target than Sarsa Realize that Expected Sarsa has a more stable update target than Sarsa: Sarsa realies on the fact that in expectation across multiple updates it will move in the right direction, whereas in contrast expected Sarsa's update targets are exactly correct and do not change estimated values away from the true values. I.e. Expected Sarsa target has much lower variance than Sarsa. Such an update rule for Expected Sarsa would look like

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi \left[Q(S_{t+1}, A_{t+1}) | S_{t+1} \right] - Q(S_t, A_t) \right] \\ &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \end{aligned} \tag{42}$$

With a more stable target, Expected Sarsa can take advantage of higher learning rates more effectively when compared with Sarsa. Expected Sarsa's updates are deterministic for a given state and action, whereas Sarsas updates can vary significantly depending on the next action. With exception to the modification of the update rule, we follow the schema of Q-learning. Given a next state S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, whence its name.

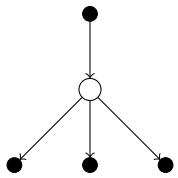


Figure 32: Expected Sarsa backup diagram.

Computational cost of Expected Sarsa Realize that Expected Sarsa is more computationally complex than Sarsa, and further that computing the average over next actions becomes more expensive as the number of actions increases. But, in return we eliminate the variance due to the *random* selection of A_{t+1}. In general, given the same amount of experience as Sarsa it generally outperforms Sarsa.

Relating Expected Sarsa with Q-learning We introduced Expected Sarsa as an on-policy algorithm, i.e. in the update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

where $A_{t+1} \sim \pi(a'|S_{t+1})$, i.e. the next action is sampled from policy π. But in general it might use a different policy from the target policy π to generate behavior, i.e. the expectation over actions is computed independently of the action actually selected in the next state. In such cases, it becomes an off-policy algorithm, and we don't even need importance sampling, since the expectation is taken over the target policy in the update target! Another interesting note is that we could take π to be the greedy policy and a behavior policy that is more exploratory, for example; in this case, Expected Sarsa is exactly Q-learning. It is in this sense that Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa. Its true the Expected Sarsa incurs a higher computational expense, but it in general dominates both of the other well-known TD control algorithms.

5.2.7 Summary of TD

The TD control algorithms are based on Bellman equations; we learned about three of them. Sarsa uses a sample-based version of the Bellman equation, and it learns q_π. Q-learning uses the Bellman optimality equation, whence it learns q_*. Expected Sarsa uses the same Bellman equation as Sarsa, but samples it differently: it takes an expectation over next action values. Whereas Sarsa is an on-policy algorithm that learns the action values for the policy its currently following, Q-learning is an off-policy algorithm that learns the optimal action values. Expected Sarsa is both an on-policy and off-policy algorithm that can learn the action values for *any* policy!. Sarsa can do better than Q-learning when performance is measured online, because on-policy control methods account for their own exploration; e.g. in cliff-world we saw that Q-learning was susceptible to falling off the cliff as it attempted to maintain continual exploration whilst generally following an optimal policy right along the cliff edge. On the other hand, Sarsa learned the longer but safer path that rarely fell off the cliff, which resulted in higher returns. We can improve Sarsa using Expected Sarsa, where the latter in general outperforms the former: this is because Expected Sarsa mitigates the variance due to its own policy. Expected Sarsa takes its expectation over the next action.

5.3 Tabular Methods

What is a model? Sometimes, we make decisions without thinking too much about them: such as driving to work. Other times, we sit and imagine many possible scenarios based on our understanding

of the world; we can use these imagined outcomes to decide what to do or not to do. E.g. if we carry a fragile object with one hand, we can imagine scenarios where it falls and breaks and we make choices accordingly (use two hands!). We've seen sample-based methods like TD which learn only from sampled experience, and we've also seen methods like DP which plan by using complete information about how the environment dynamics work, without having to make decisions. It would be even better if we could obtain an intermediate method that can leverage the best of both extremes. This section introduces the Dyna architecture. In particular, models are used to sore knowledge about the dynamics.

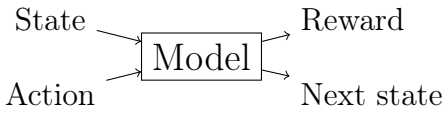


Figure 33: From a particular state and action, the model should produce a possible next state and reward; the model stores transition and reward dynamics. Critically, this allows the agent to "see" an outcome of an action without actually having to take it, which in turn allows for *planning*, which is the process of using a model to improve a policy.

$$\text{Model} \xrightarrow{\text{creates}} \underset{\text{Experience}}{\text{Simulated}} \xrightarrow{\text{updates}} \underset{\text{Function}}{\text{Value}} \xrightarrow{\text{modifies}} \text{Policy}$$

Figure 34: One of the ways to plan with a model is to use simulated experience and perform value-action updates as if those experiences actually happened. By improving the value-estimates, we can make more informed decisions. Simulating experience improves the sample efficiency: the addition of simulated experience means the agent needs fewer interactions with the world to come up with the same policy.

Types of models Several are useful. One is a *sample* model, which produces an actual outcome drawn from some underlying probabilities; e.g. a sample model for flipping a coin can generate a random sequence of heads and tails. Another type of model is a *distribution* model, which completely specifies the likelihood or probability of each outcome. In a fair-coin flipping example, it would specify that heads come up with probability 1/2 and that tails can occur with probability 1/2; realize that it can also produce the odds of any sequence of heads and tails using this information.

- Sample models are computationally inexpensive because random outcomes can be produced according to a set of rules: e.g. to flip five coins, a sample model can randomly pick zero or one independently five times. It only needs to produce a single outcome for each flip.
- Distribution models contain more information, but can be difficult to specify and can be very large. In the example of flipping five coins, a distribution model would need to enumerate each possible sequence of heads and tails possible across five coins and assign a probability to each sequence; in this problem that would consist of $2^5 = 32$ possible outcomes. Note that distribution models can be used as sample models by trying outcomes according to the explicit probabilities of each outcome.

As an example, consider a problem involving 12 dice rolls. Realize that we can simulate the dynamics by using a single die 12 times with enough patience; this is an example of a sample model. Programatically, we would generate a random number between 1-6 inclusive, 12 times. This is very compact and does not consider the joint probabilities: modeling the complete joint distribution for 12 dice is much more work. Here, we'd need to consider every possible outcome of 12 dice and assign a probability to each outcome. Realize that there are $6^{12} \approx 2$ billion possible outcomes to consider.

- Sample models require less memory.

- Distribution models can be used to compute the exact expected outcome by summing over all outcomes weighted by their probabilities. On the other hand, sample models can only approximate the expected outcome by averaging many samples together.
- Knowing the exact probabilities also allows us to assess risk, e.g. if a Doctor is prescribing medicine they can consider the possible side effects and how likely they are to occur.

5.3.1 Models and Planning

A *model* of the environment is anything an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model will produce a prediction of the resulting next state and next reward. Models can be *stochastic*, in which case there can be several possible next states and next rewards, each with some probability of occurring. There is a particular family of models that produce descriptions of all possibilities and their probabilities, and we call these *distribution models*; other *sample models* simply produce one of the possibilities, sampled according to the probabilities. Whereas a distribution model would produce all possible sums and their probabilities of occurring, a sample model would produce an individual sum drawn according to the probability distribution. Realize that DP assumes a distribution model in using $\Pr(s', r|s, a)$, the four-argument transition function describing the MDP's dynamics. On the other hand, when we discussed a Blackjack example previously, this was a sample based model. Note that there are many applications where writing a compute program to procure a sample is much easier than computing the transition dynamics, e.g. the sum of a dozen dice rolls.

Using models to simulate the environment We can mimic or simulate experience if we have a model. Given a starting state and a policy, a sample model could produce an entire episode. On the other hand, given the same inputs a distribution model could generate all possible episodes *and* their probabilities. Either way, the model *simulates* the environment and produces *simulated experience*.

Planning Planning refers to any computational process that takes a model as input and produces or improves upon a policy for interacting with the modeled environment, i.e.

$$\text{model} \xrightarrow{\text{planning}} \text{policy}$$

State-space planning In AI, there are two distinct approaches to planning according to the aforementioned definition. *State-space planning* is a search through the state space for an optimal policy or an optimal path to a goal; actions cause transitions from state-to-state, and value functions are computed over states. *Plan-space planning* is instead a search through the space of plans, where operators transform one plan into another, and value functions, if any are defined, are defined over the space of plans. Note that plan-space planning includes evolutionary methods. Since plan-space methods are difficult to apply efficiently to stochatic sequential decision problems which are at the root of RL, we don't focus on them further.

A unified view All state-space planning methods share a common structure: (i) all state-space planning methods involve computing value functions as a key intermediary step toward improving the policy, and (ii) they compute value functions by updates or backup operations applied to simulated experience. Realize

$$\text{model} \longrightarrow \begin{matrix}\text{simulated}\\ \text{experience}\end{matrix} \xrightarrow{\text{backups}} \text{values} \longrightarrow \text{policy}$$

that DP clearly fits this structure: in each iteration we make a sweep through the spaces of states,

generating for each state the distribution of possible transitions. Then, each distribution is used to compute a backed-up value (update target) and update the state's estimated value.

Comparing planning methods with learning methods At the heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The key difference is that whereas planning uses *simulated* experience generated from a model, learning methods use real experience generated by the environment. In many cases, a learning algorithm can be substituted for the key update step of a planning method! This follows from the fact that learning methods only require experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. Consider a *random-sample one-step tabular Q-planning* procedure below, which converges to the optimal policy for the model under the conditions that (i) each state-action pair must be selected an infinite number of times in the first step and (ii) α must decrease appropriately over time.

Algorithm 12: Random-sample one-step tabular Q-planning

for *ever, without termination* **do**
  Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
  Send S, A to a sample model, and obtain a sample next reward, R, and a sample next state S'
  Apply one-step tabular Q-learning to S, A, R, S':

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_a Q(S',a) - Q(S,A) \right].$$

end

The benefit of planning in small, incremental steps This enables planning to be interrupted or redirected at any time with little wasted computation, and this appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. In fact, planning in very small steps may be the most efficient approach even on pure planning problems, if the problem is too large to be solved exactly.

5.3.2 Dyna: Integrated Planning, Acting, and Learning

What happens when planning is done online While planning online and interacting with the environment, a number of interesting issues arise. For example, new information gained from the interaction may change the model and thereby interact with planning: it may be desireable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. Another issue is that if decision making and model learning are both computationally-intensive processes, then the available computational resources may need to be divided among them. Dyna-Q begins to explore these issues; it is a simple architecture which integrates the majore functions needed in an online planning agent.

Direct and indirect RL Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment), i.e. *model learning*, and it can be used to directly improve the value function and policy using the kinds of RL methods we've previously discussed, i.e. *direct reinforcement learning*. We can diagram the possible relationships between experience, model, values, and policy as follows.

Dyna-Q inclues all of the processes shown in the diagram above - planning, acting, model-free learning, and direct RL - all occurring continually. In particular: the planning method is the random-sample one-step tabular planning method presented previously; the direct RL method is one-step tabular Q-learning;

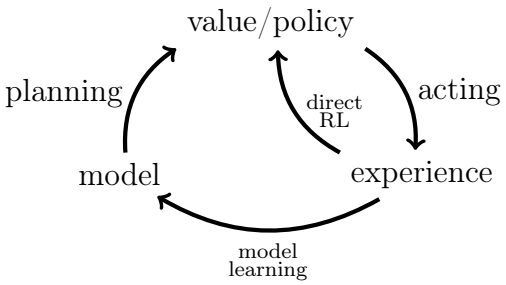


Figure 35: Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter which is sometimes known as *indirect reinforcement learning*.

the model-learning method is also table-based and assumes the environment is deterministic, where after each transition $S_t, A_t \longrightarrow R_{t+1}, S_{t+1}$ the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state-action pairs that have previously been experienced, so the model is never queried with a pair about which it has no information.

Tradeoffs of direct vs. indirect methods Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model.

Architecture of Dyna agents We present the architecture of Dyna agents, of which the Dyna-Q algorithm is just one example.

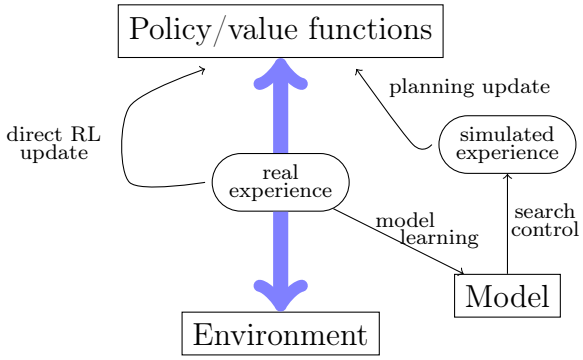


Figure 36: The central column represents the basic interaction between agent and environment, giving rise to the trajectory of real experience. The arrow on the left represents direct RL operating on real experience to improve the value function and the policy. On the right are model-based processes; the model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Lastly, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. It is typical that the same RL method is used for both learning from real experience and planning from simulated experience; the RL method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

Theoretically, planning, acting, model-learning, and direct RL can all occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, we specify an ordering of computations: we save planning for last and assume that in each time step there is sufficient time in each step after acting, model-learning, and direct RL to complete n-iterations of the Q-planning algorithm. Below, `Model`(s, a) denotes the contents of the predicted next state and reward for state action pair (s, a).

Algorithm 13: Tabular Dyna-Q

Initialize $Q(s,a)$ and `Model`(s,a) for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.
for *ever* **do**
  $S \leftarrow$ current (non-terminal) state
  $A \leftarrow \epsilon - \text{greedy}(S,Q)$
  Take action A; observe resultant reward R, and state, S'
  $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ `// Direct RL`
  `Model`$(S,A) \leftarrow R, S'$ (assuming deterministic environment) `// Model-learning`
  for $i = 1, \ldots, n$ **do**
    `// Planning...`
    $S \leftarrow$ random previously observed state `// Search...`
    $A \leftarrow$ random action previously taken in S `// ...Control`
    $R, S' \leftarrow$ `Model`(S,A) `// Model Query`
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ `// Value Update`
  end
end

If we omit the model-learning and planning steps, we recover one-step tabular Q-learning.

Dyna Maze Consider the simple maze shown below.

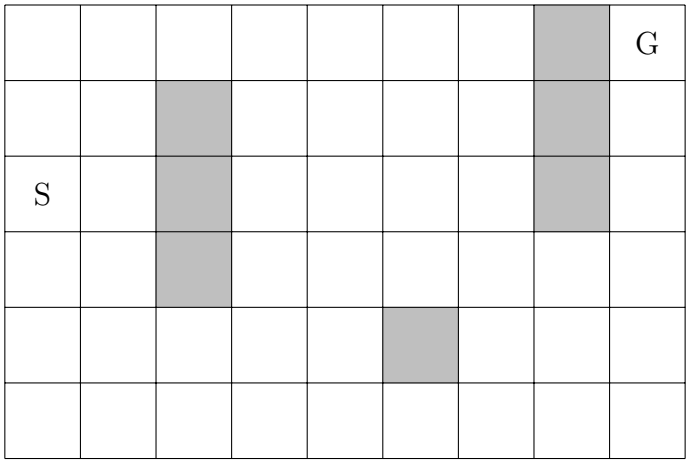


Figure 37: In each of the 47 states there are four actions: `up, down, right,` and `left`, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains stationary. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state G, the agent returns to the start state S to begin a new episode. The task is episodic, and we will use $\gamma = 0.95$.

We can apply various Dyna-Q agents to this maze task, setting all initial action-values to zero, with step-size parameter $\alpha = 0.1$ and exploration parameter $\epsilon = 0.1$, and further breaking ties arbitrarily when selecting among greedy actions. The agents vary in the number of planning steps n they performed at each real step. We see that on this problem the agents that use more planning are able to reach ϵ-optimal performance faster than those that do not utilize planning. Without planning, each episode only adds one additional step to the policy, whereas with planning although only one step is learned during the first episode but an extensive policy can be learned during the second episode, reaching almost as far back as the start state.

5.3.3 When the Model is Wrong

A model is inaccurate when transitions stored are different from transitions that actually happen in the environment. When an agent begins learning, they haven't tried a majority of the actions is nearly every state; the transitions associated with trying those actions in those states are simply missing from the model. A model with missing transitions is called an *incomplete* model. Another way for models to become inaccurate is if the environment changes: taking an action in a state could result in a different next state and reward than what the agent observed before the change to the environment. We say such a model is inaccurate because what actually happens is different from what the model states. Model accuracy produces a new variant of the exploration-exploitation trade-off, since the agent has to explore in order to make sure its model remains accurate.

Planning with inaccurate models The effect of planning with inaccurate models depends on exactly how the model is inaccurate. When an agent starts to learn from an incomplete model, realize that the model can't produce a next step or reward, and so it cannot be used for planning. However, as the agent interacts with the environment, the model stores more transitions; this allows the agent to perform updates by simulating transitions it has seen before. This means that as long as the agent has seen *some* transitions, it can plan with the model. What happens when the environment changes? Suppose that an agent has already visited every state-action pair and stored its experience in the model. Then, a sudden environment change occurs. A transition in the model now no longer reflects the transition in the environment. If the agent tries to use the inaccurate model to perform a planning update, the value function or policy that the agent learns might change in the wrong direction: remember that planning improves the policy with respect to what the model thinks will happen, rather than what will really happen in the environment. When the environment changes, the model becomes outdated. Planning with an outdated model will likely make the agent's performance worse with respect to the environment. To summarize,

- Models can be inaccurate if:
  - They are incomplete.
  - The environment changes.
- Planning with an inaccurate model improves the policy or value function with respect to the model, and not the environment.
- Dyna-Q can plan with an incomplete model by only sampling state-action pairs that have been previously visited.

Changing environments and Dyna-Q+ Because planning with an inaccurate model can lead to making the policy or value-function worse with respect to the environment, agents have a vested interest in making sure their model stays accurate. In general, an agent might want to double-check that all its models transitions are correct: however, double-checking transitions with low-valued actions will often lead to low reward. In changing environments, the agent's model might become inaccurate at any time, and so the agent has to make a choice: explore to make sure the model is accurate, or exploit the model to compte the optimal policy, assuming the model is correct. Of course, when the environment changes the model will be incorrect, and it will remain incorrect until the agent revisits that part of the environment that has changed and updates the model. This suggests that an agent should explore places it has not recently visited. To encourage the agent to revisit states periodically, we can add a bonus to the reward used in planning.

$$\texttt{new reward} = r + \kappa\sqrt{\tau}$$

where r is the reward from the model, τ is the amount of time it's been since the state-action pair was last visited in the environment, and κ is a small positive constant that controls the influence of the bonus on the

planning update. E.g. if $\kappa = 0$, we would ignore the bonus completely. By adding this exploration bonus to the planning updates, we get a new algorithm called Dyna-Q+. By artificially increasing the rewards used in planning, we increase the value of state-action pairs that haven't been visited recently. This encourages exploration because it a state-action pair hasn't been visited in a long time, then τ becomes large, and as τ grows the bonus becomes bigger and bigger; eventually, planning will change the policy to go directly to this state due to the large bonus. When the agent finally visits the state, it might see a big reward or it might be disappointed, but either way the model will be updated to reflect the true environment dynamics.

5.4 Summary

Planning requires a *model* of the environment, and there are in general two types of models: a *distribution* model specifies completely the probabilities of all possible next states and rewards for all possible actions, and *sample* models produce single transitions and rewards generated according to those probabilities. DP requires a distribution model because it inherently uses *expected updates*, which involve computing expecations over all possible next states and rewards. On the other hand, a *sample model* can be used to simulate interactions with the environment and corresponding *sample updates* can be made. Sample models are in general easier to obtain than distribution models.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion, each producing what the other needs to improve. The most natural approach is to allow all processes to procede asynchronously and in parallel. One dimension in variation among state-space planning methods is the size of the update. At one end of the spectrum, we have one-step sample updates as in Dyna.

From state-values to state-action values:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[\hat{G}_t - V(S_t)\right] \rightsquigarrow Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\hat{G}_t - Q(S_t, A_t)\right]$$

where we can define a variety of update rules:

Sarsa	$\hat{G}_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$
Q-Learning	$\hat{G}_t = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$
Expected Sarsa	$\hat{G}_t = R_{t+1} + \gamma \sum_{a'} \pi(a'\|S_{t+1}) Q(S_{t+1}, a')$

Sarsa is an on-policy algorithm, whereas Q-learning is off-policy: it estimates the value function for the optimal policy. Expected Sarsa can be an on-or-off policy algorithm, because the expectation can be conditioned on any policy; Q-learning is a special case of expected Sarsa.