

Contents

1	Course Introduction	2
1.1	Motivating Reinforcement Learning	2
2	Multi-armed Bandits	3
2.1	A k -armed Bandit Problem	3
2.2	Action-value Methods	4
2.3	The 10-armed Testbed	4
2.4	Incremental Implementation	5
2.5	Tracking a Nonstationary Problem using Exponentially Weighted Moving Averages	6
2.6	Optimistic Initial Values	7
2.6.1	Unbiased constant-step-size trick	7
2.7	Upper-confidence bound action selection	8
2.8	Sequential Decision Making with Evaluative Feedback	8
3	Finite Markov Decision Processes	9
3.1	Returns and Episodes	10
3.2	Examples of MDPs	11
3.3	The reward hypothesis	12

1 Course Introduction

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:

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

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 certainty, 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:

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

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.

$$A_t := \arg \max_a Q_t(a).$$

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.

¹To 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 $\sum_{t=1}^{\infty} \underbrace{\Pr(A_t = a)}_{\geq \epsilon}$ which diverges to ∞ .

²The 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 after 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 i th 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.

$$Q_n := \frac{R_1 + R_2 + \dots + R_{n-1}}{n - 1}.$$

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

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

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 then 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:

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

The expression $\text{Target} - \text{OldEstimate}$ is an *error* in our estimate which is reduced by taking a step towards the “target”, which is presumed to indicate a desirable 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  $a = 1, \dots, k$  do
   $Q(a) \leftarrow 0$ 
   $N(a) \leftarrow 0$ 
end
while True do
   $A \leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \epsilon \quad (*\text{breaking ties randomly}) \\ \text{random action} & \text{with probability } \epsilon \end{cases}$ 
   $R \leftarrow \text{Bandit}(A)$ 
   $N(A) \leftarrow N(A) + 1$ 
   $Q(A) \leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)]$ 
end

```

$\text{Bandit}(\cdot)$ 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

$$Q_{n+1} := Q_n + \alpha [R_n - Q_n] \quad (3)$$

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 :

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

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.

³If $1 - \alpha = 0$, 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

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

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 disappears 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 values 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:

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

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:

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

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 aspect 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. Whenever 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 it's 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 distribution exactly 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 horizon 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.

⁴In 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 $A_t \in \mathcal{A}(S_t)$.

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 formalizes 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. $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, 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}, \dots , 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:

$$G_t := R_{t+1} + R_{t+2} + \dots + R_T \tag{6}$$

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 standard 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. 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.

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.

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*:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad (7)$$

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. 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 agent becomes farsighted. Note that

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

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

3.2 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 1.

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 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.

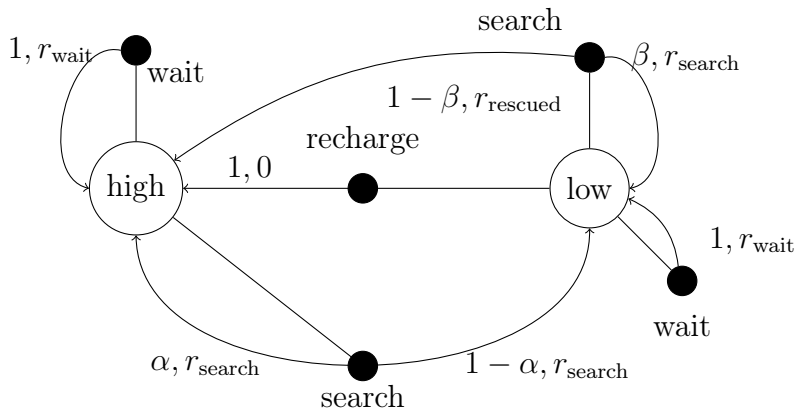


Figure 1: The *state* of the robot’s battery is indicated by open nodes in the graph, demarcated by “high” and “low”. Actions are denoted by coloured nodes, and an edge between a state-action pair indicates that it is possible to execute said action from the corresponding state. After taking an action 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.

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.

3.3 The reward hypothesis

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 it's 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".