Part V

Reinforcement Learning and Control

Chapter 15

Reinforcement learning

We now begin our study of reinforcement learning and adaptive control.

In supervised learning, we saw algorithms that tried to make their outputs mimic the labels y given in the training set. In that setting, the labels gave an unambiguous "right answer" for each of the inputs x. In contrast, for many sequential decision making and control problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the "correct" actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

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

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web-page indexing. Our study of reinforcement learning will begin with a definition of the **Markov decision processes (MDP)**, which provides the formalism in which RL problems are usually posed.

15.1 Markov decision processes

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

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

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

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$$

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

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

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

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

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

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

$$\mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots\right]$$

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

A **policy** is any function $\pi : S \mapsto A$ mapping from the states to the actions. We say that we are **executing** some policy π if, whenever we are in state s, we take action $a = \pi(s)$. We also define the **value function** for a policy π according to

$$V^\pi(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \,\middle|\, s_0 = s, \pi\right].$$

$V^\pi(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.[1]

Given a fixed policy π, its value function V^π satisfies the **Bellman equations**:

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

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

Bellman's equations can be used to efficiently solve for V^π. Specifically, in a finite-state MDP ($|S| < \infty$), we can write down one such equation for $V^\pi(s)$ for every state s. This gives us a set of $|S|$ linear equations in $|S|$ variables (the unknown $V^\pi(s)$'s, one for each state), which can be efficiently solved for the $V^\pi(s)$'s.

[1] This notation in which we condition on π isn't technically correct because π isn't a random variable, but this is quite standard in the literature.

We also define the **optimal value function** according to

$$V^*(s) = \max_\pi V^\pi(s). \tag{15.1}$$

In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy. There is also a version of Bellman's equations for the optimal value function:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.2}$$

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

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

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.3}$$

Note that $\pi^*(s)$ gives the action a that attains the maximum in the "max" in Equation (15.2).

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

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

The first equality says that the V^{π^*}, the value function for π^*, is equal to the optimal value function V^* for every state s. Further, the inequality above says that π^*'s value is at least a large as the value of any other other policy. In other words, π^* as defined in Equation (15.3) is the optimal policy.

Note that π^* has the interesting property that it is the optimal policy for *all* states s. Specifically, it is not the case that if we were starting in some state s then there'd be some optimal policy for that state, and if we were starting in some other state s' then there'd be some other policy that's optimal policy for s'. The same policy π^* attains the maximum in Equation (15.1) for *all* states s. This means that we can use the same policy π^* no matter what the initial state of our MDP is.

15.2 Value iteration and policy iteration

We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces ($|S| <$

∞, $|A| < \infty$). In this section, we will also assume that we know the state transition probabilities $\{P_{sa}\}$ and the reward function R.

The first algorithm, **value iteration**, is as follows:

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s')V(s'). \tag{15.4}$$

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations (15.2).

There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for $V(s)$ for every state s, and then overwrite all the old values with the new values. This is called a **synchronous** update. In this case, the algorithm can be viewed as implementing a "Bellman backup operator" that takes a current estimate of the value function, and maps it to a new estimate. (See homework problem for details.) Alternatively, we can also perform **asynchronous** updates. Here, we would loop over the states (in some order), updating the values one at a time.

Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can then use Equation (15.3) to find the optimal policy.

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

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

After at most a *finite* number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.[2]

[2]Note that value iteration cannot reach the exact V^* in a finite number of iterations,

Algorithm 5 Policy Iteration

1: Initialize π randomly.
2: **for** until convergence **do**
3: Let $V := V^\pi$. ▷ typically by linear system solver
4: For each state s, let

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

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn't currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fats and converges with very few iterations. However, for MDPs with large state spaces, solving for V^π explicitly would involve solving a large system of linear equations, and could be difficult (and note that one has to solve the linear system multiple times in policy iteration). In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration. For some more discussions on the comparison and connection of value iteration and policy iteration, please see Section 15.5.

15.3 Learning a model for an MDP

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

For example, suppose that, for the inverted pendulum problem (see prob-

whereas policy iteration with an exact linear system solver, can. This is because when the actions space and policy space are discrete and finite, and once the policy reaches the optimal policy in policy iteration, then it will not change at all. On the other hand, even though value iteration will converge to the V^*, but there is always some non-zero error in the learned value function.

lem set 4), we had a number of trials in the MDP, that proceeded as follows:

$$
\begin{aligned}
&s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \ldots \\
&s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \ldots \\
&\ldots
\end{aligned}
$$

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

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

$$P_{sa}(s') = \frac{\#\text{times took we action } a \text{ in state } s \text{ and got to } s'}{\#\text{times we took action a in state } s} \tag{15.5}$$

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

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of (15.5), then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of P_{sa}.

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

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

1. Initialize π randomly.
2. Repeat {
   (a) Execute π in the MDP for some number of trials.

(b) Using the accumulated experience in the MDP, update our estimates for P_{sa} (and R, if applicable).

(c) Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function V.

(d) Update π to be the greedy policy with respect to V.

}

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

15.4 Continuous state MDPs

So far, we've focused our attention on MDPs with a finite number of states. We now discuss algorithms for MDPs that may have an infinite number of states. For example, for a car, we might represent the state as $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$, comprising its position (x, y); orientation θ; velocity in the x and y directions $\dot{x}$ and $\dot{y}$; and angular velocity $\dot{\theta}$. Hence, $S = \mathbb{R}^6$ is an infinite set of states, because there is an infinite number of possible positions and orientations for the car.[3] Similarly, the inverted pendulum you saw in PS4 has states $(x, \theta, \dot{x}, \dot{\theta})$, where θ is the angle of the pole. And, a helicopter flying in 3d space has states of the form $(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$, where here the roll ϕ, pitch θ, and yaw ψ angles specify the 3d orientation of the helicopter.

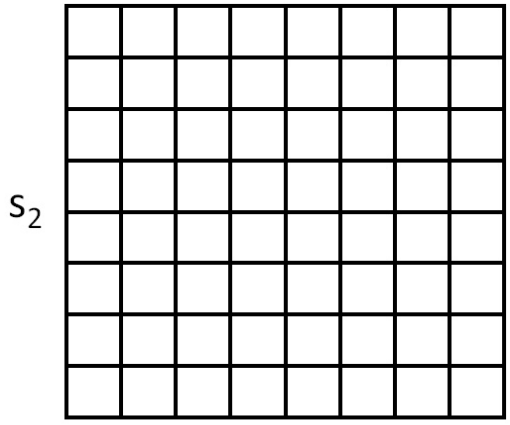
In this section, we will consider settings where the state space is $S = \mathbb{R}^d$, and describe ways for solving such MDPs.

15.4.1 Discretization

Perhaps the simplest way to solve a continuous-state MDP is to discretize the state space, and then to use an algorithm like value iteration or policy iteration, as described previously.

For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:

[3] Technically, θ is an orientation and so the range of θ is better written $\theta \in [-\pi, \pi)$ than $\theta \in \mathbb{R}$; but for our purposes, this distinction is not important.

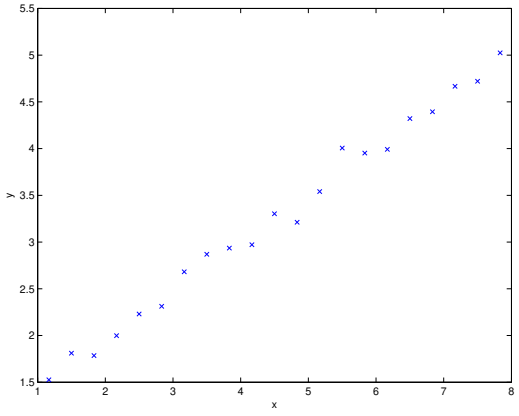


[t]

Here, each grid cell represents a separate discrete state $\bar{s}$. We can then approximate the continuous-state MDP via a discrete-state one $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$, where $\bar{S}$ is the set of discrete states, $\{P_{\bar{s}a}\}$ are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the $V^*(\bar{s})$ and $\pi^*(\bar{s})$ in the discrete state MDP $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$. When our actual system is in some continuous-valued state $s \in S$ and we need to pick an action to execute, we compute the corresponding discretized state $\bar{s}$, and execute action $\pi^*(\bar{s})$.

This discretization approach can work well for many problems. However, there are two downsides. First, it uses a fairly naive representation for V^* (and π^*). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the gridcells).

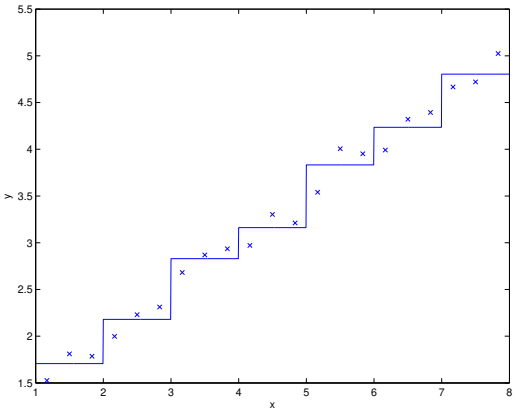
To better understand the limitations of such a representation, consider a *supervised learning* problem of fitting a function to this dataset:



[t]

Clearly, linear regression would do fine on this problem. However, if we instead discretize the x-axis, and then use a representation that is piecewise constant in each of the discretization intervals, then our fit to the data would look like this:

[t]



This piecewise constant representation just isn't a good representation for many smooth functions. It results in little smoothing over the inputs, and no generalization over the different grid cells. Using this sort of representation, we would also need a very fine discretization (very small grid cells) to get a good approximation.

A second downside of this representation is called the **curse of dimensionality**. Suppose $S = \mathbb{R}^d$, and we discretize each of the d dimensions of the state into k values. Then the total number of discrete states we have is k^d. This grows exponentially quickly in the dimension of the state space d, and thus does not scale well to large problems. For example, with a 10d state, if we discretize each state variable into 100 values, we would have $100^{10} = 10^{20}$ discrete states, which is far too many to represent even on a modern desktop computer.

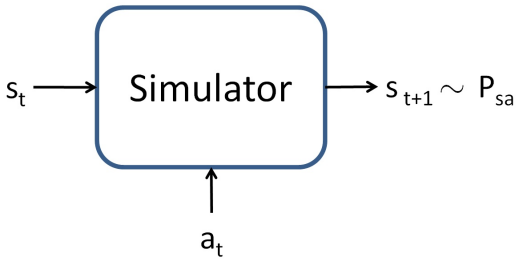
As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you're extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

15.4.2 Value function approximation

We now describe an alternative method for finding policies in continuous-state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, called value function approximation, has been successfully applied to many RL problems.

Using a model or simulator

To develop a value function approximation algorithm, we will assume that we have a **model**, or **simulator**, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state s_t and action a_t, and outputs a next-state s_{t+1} sampled according to the state transition probabilities $P_{s_t a_t}$:



[t]

There are several ways that one can get such a model. One is to use physics simulation. For example, the simulator for the inverted pendulum in PS4 was obtained by using the laws of physics to calculate what position and orientation the cart/pole will be in at time $t+1$, given the current state at time t and the action a taken, assuming that we know all the parameters of the system such as the length of the pole, the mass of the pole, and so on. Alternatively, one can also use an off-the-shelf physics simulation software package which takes as input a complete physical description of a mechanical system, the current state s_t and action a_t, and computes the state s_{t+1} of the system a small fraction of a second into the future.[4]

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute n **trials** in which we repeatedly take actions in an MDP, each trial for T timesteps. This can be done picking actions at random, executing some specific policy, or via some other way of

[4]Open Dynamics Engine (http://www.ode.com) is one example of a free/open-source physics simulator that can be used to simulate systems like the inverted pendulum, and that has been a reasonably popular choice among RL researchers.

choosing actions. We would then observe n state sequences like the following:

$$\begin{aligned}
&s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} \cdots \xrightarrow{a_{T-1}^{(1)}} s_T^{(1)} \\
&s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} \cdots \xrightarrow{a_{T-1}^{(2)}} s_T^{(2)} \\
&\cdots \\
&s_0^{(n)} \xrightarrow{a_0^{(n)}} s_1^{(n)} \xrightarrow{a_1^{(n)}} s_2^{(n)} \xrightarrow{a_2^{(n)}} \cdots \xrightarrow{a_{T-1}^{(n)}} s_T^{(n)}
\end{aligned}$$

We can then apply a learning algorithm to predict s_{t+1} as a function of s_t and a_t.

For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t, \tag{15.6}$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our n trials, by picking

$$\arg\min_{A,B} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \right\|_2^2.$$

We could also potentially use other loss functions for learning the model. For example, it has been found in recent work Luo et al. [2018] that using $\|\cdot\|_2$ norm (without the square) may be helpful in certain cases.

Having learned A and B, one option is to build a **deterministic** model, in which given an input s_t and a_t, the output s_{t+1} is exactly determined. Specifically, we always compute s_{t+1} according to Equation (15.6). Alternatively, we may also build a **stochastic** model, in which s_{t+1} is a random function of the inputs, by modeling it as

$$s_{t+1} = As_t + Ba_t + \epsilon_t,$$

where here ϵ_t is a noise term, usually modeled as $\epsilon_t \sim \mathcal{N}(0, \Sigma)$. (The covariance matrix Σ can also be estimated from data in a straightforward way.)

Here, we've written the next-state s_{t+1} as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model $s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$, where ϕ_s and ϕ_a are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms, such as locally weighted linear regression, to learn to estimate s_{t+1} as a function of s_t and a_t. These approaches can also be used to build either deterministic or stochastic simulators of an MDP.

Fitted value iteration

We now describe the **fitted value iteration** algorithm for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space $S = \mathbb{R}^d$, but that the action space A is small and discrete.[5]

Recall that in value iteration, we would like to perform the update

$$\begin{aligned} V(s) &:= R(s) + \gamma \max_a \int_{s'} P_{sa}(s')V(s')ds' && (15.7) \\ &= R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] && (15.8) \end{aligned}$$

(In Section 15.2, we had written the value iteration update with a summation $V(s) := R(s) + \gamma \max_a \sum_{s'} P_{sa}(s')V(s')$ rather than an integral over states; the new notation reflects that we are now working in continuous states rather than discrete states.)

The main idea of fitted value iteration is that we are going to approximately carry out this step, over a finite sample of states $s^{(1)}, \ldots, s^{(n)}$. Specifically, we will use a supervised learning algorithm—linear regression in our description below—to approximate the value function as a linear or non-linear function of the states:

$$V(s) = \theta^T \phi(s).$$

Here, ϕ is some appropriate feature mapping of the states.

For each state s in our finite sample of n states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (the right hand side of Equation 15.8). Then, it will apply a supervised learning algorithm to try to get $V(s)$ close to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (or, in other words, to try to get $V(s)$ close to $y^{(i)}$).

In detail, the algorithm is as follows:

1. Randomly sample n states $s^{(1)}, s^{(2)}, \ldots s^{(n)} \in S$.
2. Initialize $\theta := 0$.
3. Repeat {

   For $i = 1, \ldots, n$ {

[5] In practice, most MDPs have much smaller action spaces than state spaces. E.g., a car has a 6d state space, and a 2d action space (steering and velocity controls); the inverted pendulum has a 4d state space, and a 1d action space; a helicopter has a 12d state space, and a 4d action space. So, discretizing this set of actions is usually less of a problem than discretizing the state space would have been.

For each action $a \in A$ {
Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).
Set $q(a) = \frac{1}{k} \sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$
// Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
}
Set $y^{(i)} = \max_a q(a)$.
// Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
}
// In the original value iteration algorithm (over discrete states)
// we updated the value function according to $V(s^{(i)}) := y^{(i)}$.
// In this algorithm, we want $V(s^{(i)}) \approx y^{(i)}$, which we'll achieve
// using supervised learning (linear regression).
Set $\theta := \arg\min_\theta \frac{1}{2} \sum_{i=1}^{n} \left(\theta^T \phi(s^{(i)}) - y^{(i)}\right)^2$
}

Above, we had written out fitted value iteration using linear regression as the algorithm to try to make $V(s^{(i)})$ close to $y^{(i)}$. That step of the algorithm is completely analogous to a standard supervised learning (regression) problem in which we have a training set $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$, and want to learn a function mapping from x to y; the only difference is that here s plays the role of x. Even though our description above used linear regression, clearly other regression algorithms (such as locally weighted linear regression) can also be used.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems. Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting $k = 1$ in the algorithm. This is because the expectation in Equation (15.8) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above, we had to draw k samples, and average to try to approximate that expectation (see the definition of $q(a)$, in the algorithm pseudo-code).

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

$$\arg\max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] \tag{15.9}$$

The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample $s'_1, \ldots, s'_k \sim P_{sa}$ to approximate the expectation. (And again, if the simulator is deterministic, we can set $k = 1$.)

In practice, there are often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$, where f is some deterministic function of the states (such as $f(s_t, a_t) = As_t + Ba_t$), and ϵ is zero-mean Gaussian noise. In this case, we can pick the action given by

$$\arg\max_a V(f(s, a)).$$

In other words, here we are just setting $\epsilon_t = 0$ (i.e., ignoring the noise in the simulator), and setting $k = 1$. Equivalent, this can be derived from Equation (15.9) using the approximation

$$\begin{aligned} \mathrm{E}_{s'}[V(s')] &\approx V(\mathrm{E}_{s'}[s']) && (15.10) \\ &= V(f(s, a)), && (15.11) \end{aligned}$$

where here the expectation is over the random $s' \sim P_{sa}$. So long as the noise terms ϵ_t are small, this will usually be a reasonable approximation.

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above, can be computationally expensive.

15.5 Connections between Policy and Value Iteration (Optional)

In the policy iteration, line 3 of Algorithm 5, we typically use linear system solver to compute V^π. Alternatively, one can also the iterative Bellman updates, similarly to the value iteration, to evaluate V^π, as in the Procedure VE($\cdot$) in Line 1 of Algorithm 6 below. Here if we take option 1 in Line 2 of the Procedure VE, then the difference between the Procedure VE from the

Algorithm 6 Variant of Policy Iteration

procedure VE(π, k) ▷ To evaluate V^π
    Option 1: initialize $V(s) := 0$; Option 2: Initialize from the current V in the main algorithm.
    for $i = 0$ to $k - 1$ **do**
        For every state s, update

$$V(s) := R(s) + \gamma \sum_{s'} P_{s\pi(s)}(s')V(s'). \tag{15.12}$$

    return V

Require: hyperparameter k.
Initialize π randomly.
for until convergence **do**
    Let $V = \text{VE}(\pi, k)$.
    For each state s, let

$$\pi(s) := \arg\max_{a \in A} \sum_{s'} P_{sa}(s')V(s'). \tag{15.13}$$

value iteration (Algorithm 4) is that on line 4, the procedure is using the action from π instead of the greedy action.

Using the Procedure VE, we can build Algorithm 6, which is a variant of policy iteration that serves an intermediate algorithm that connects policy iteration and value iteration. Here we are going to use option 2 in VE to maximize the re-use of knowledge learned before. One can verify indeed that if we take $k = 1$ and use option 2 in Line 2 in Algorithm 6, then Algorithm 6 is semantically equivalent to value iteration (Algorithm 4). In other words, both Algorithm 6 and value iteration interleave the updates in (15.13) and (15.12). Algorithm 6 alternate between k steps of update (15.12) and one step of (15.13), whereas value iteration alternates between 1 steps of update (15.12) and one step of (15.13). Therefore generally Algorithm 6 should not be faster than value iteration, because assuming that update (15.12) and (15.13) are equally useful and time-consuming, then the optimal balance of the update frequencies could be just $k = 1$ or $k \approx 1$.

On the other hand, if k steps of update (15.12) can be done much faster than k times a single step of (15.12), then taking additional steps of equation (15.12) in group might be useful. This is what policy iteration is leveraging — the linear system solver can give us the result of Procedure VE with $k = \infty$ much faster than using the Procedure VE for a large k. On the flip side, when such a speeding-up effect no longer exists, e.g.,, when the state space is large and linear system solver is also not fast, then value iteration is more preferable.

Chapter 16

LQR, DDP and LQG

16.1 Finite-horizon MDPs

In Chapter 15, we defined Markov Decision Processes (MDPs) and covered Value Iteration / Policy Iteration in a simplified setting. More specifically we introduced the **optimal Bellman equation** that defines the optimal value function V^{π^*} of the optimal policy π^*.

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

Recall that from the optimal value function, we were able to recover the optimal policy π^* with

$$\pi^*(s) = \text{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^*(s')$$

In this chapter, we'll place ourselves in a more general setting:

1. We want to write equations that make sense for both the discrete and the continuous case. We'll therefore write

$$\mathbb{E}_{s' \sim P_{sa}} \left[V^{\pi^*}(s') \right] \qquad \text{instead of}$$
$$\sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s')$$

   meaning that we take the expectation of the value function at the next state. In the finite case, we can rewrite the expectation as a sum over

states. In the continuous case, we can rewrite the expectation as an integral. The notation $s' \sim P_{sa}$ means that the state s' is sampled from the distribution P_{sa}.

2. We'll assume that the rewards depend on **both states and actions**. In other words, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This implies that the previous mechanism for computing the optimal action is changed into

$$\pi^*(s) = \text{argmax}_{a \in \mathcal{A}} R(s, a) + \gamma \mathbb{E}_{s' \sim P_{sa}} \left[V^{\pi^*}(s') \right]$$

3. Instead of considering an infinite horizon MDP, we'll assume that we have a **finite horizon MDP** that will be defined as a tuple

$$(\mathcal{S}, \mathcal{A}, P_{sa}, T, R)$$

with $T > 0$ the **time horizon** (for instance $T = 100$). In this setting, our definition of payoff is going to be (slightly) different:

$$R(s_0, a_0) + R(s_1, a_1) + \cdots + R(s_T, a_T)$$

instead of (infinite horizon case)

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$
$$\sum_{t=0}^{\infty} R(s_t, a_t) \gamma^t$$

What happened to the discount factor γ? Remember that the introduction of γ was (partly) justified by the necessity of making sure that the infinite sum would be finite and well-defined. If the rewards are bounded by a constant $\bar{R}$, the payoff is indeed bounded by

$$|\sum_{t=0}^{\infty} R(s_t) \gamma^t| \leq \bar{R} \sum_{t=0}^{\infty} \gamma^t$$

and we recognize a geometric sum! Here, as the payoff is a finite sum, the discount factor γ is not necessary anymore.

In this new setting, things behave quite differently. First, the optimal policy π^* might be non-stationary, meaning that **it changes over time**. In other words, now we have

$$\pi^{(t)} : \mathcal{S} \to \mathcal{A}$$

where the superscript (t) denotes the policy at time step t. The dynamics of the finite horizon MDP following policy $\pi^{(t)}$ proceeds as follows: we start in some state s_0, take some action $a_0 := \pi^{(0)}(s_0)$ according to our policy at time step 0. The MDP transitions to a successor s_1, drawn according to $P_{s_0 a_0}$. Then, we get to pick another action $a_1 := \pi^{(1)}(s_1)$ following our new policy at time step 1 and so on...

Why does the optimal policy happen to be non-stationary in the finite-horizon setting? Intuitively, as we have a finite numbers of actions to take, we might want to adopt different strategies depending on where we are in the environment and how much time we have left. Imagine a grid with 2 goals with rewards +1 and +10. At the beginning, we might want to take actions to aim for the +10 goal. But if after some steps, dynamics somehow pushed us closer to the +1 goal and we don't have enough steps left to be able to reach the +10 goal, then a better strategy would be to aim for the +1 goal...

4. This observation allows us to use **time dependent dynamics**

$$s_{t+1} \sim P^{(t)}_{s_t, a_t}$$

meaning that the transition's distribution $P^{(t)}_{s_t,a_t}$ changes over time. The same thing can be said about $R^{(t)}$. Note that this setting is a better model for real life. In a car, the gas tank empties, traffic changes, etc. Combining the previous remarks, we'll use the following general formulation for our finite horizon MDP

$$\left(\mathcal{S}, \mathcal{A}, P^{(t)}_{sa}, T, R^{(t)}\right)$$

Remark: notice that the above formulation would be equivalent to adding the time into the state.

The value function at time t for a policy π is then defined in the same way as before, as an expectation over trajectories generated following policy π starting in state s.

$$V_t(s) = \mathbb{E}\left[R^{(t)}(s_t, a_t) + \cdots + R^{(T)}(s_T, a_T)|s_t = s, \pi\right]$$

Now, the question is

In this finite-horizon setting, how do we find the optimal value function

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

It turns out that Bellman's equation for Value Iteration is made for **Dynamic Programming**. This may come as no surprise as Bellman is one of the fathers of dynamic programming and the Bellman equation is strongly related to the field. To understand how we can simplify the problem by adopting an iteration-based approach, we make the following observations:

1. Notice that at the end of the game (for time step T), the optimal value is obvious

$$\forall s \in \mathcal{S}: \quad V_T^*(s) := \max_{a \in \mathcal{A}} R^{(T)}(s, a) \tag{16.1}$$

2. For another time step $0 \leq t < T$, if we suppose that we know the optimal value function for the next time step V_{t+1}^*, then we have

$$\forall t < T, s \in \mathcal{S}: \quad V_t^*(s) := \max_{a \in \mathcal{A}} \left[R^{(t)}(s, a) + \mathbb{E}_{s' \sim P_{sa}^{(t)}}\left[V_{t+1}^*(s')\right]\right] \tag{16.2}$$

With these observations in mind, we can come up with a clever algorithm to solve for the optimal value function:

1. compute V_T^* using equation (16.1).
2. for $t = T - 1, \ldots, 0$:

   compute V_t^* using V_{t+1}^* using equation (16.2)

Side note We can interpret standard value iteration as a special case of this general case, but without keeping track of time. It turns out that in the standard setting, if we run value iteration for T steps, we get a γ^T approximation of the optimal value iteration (geometric convergence). See problem set 4 for a proof of the following result:

Theorem Let B denote the Bellman update and $||f(x)||_\infty := \sup_x |f(x)|$. If V_t denotes the value function at the t-th step, then

$$\begin{aligned} ||V_{t+1} - V^*||_\infty &= ||B(V_t) - V^*||_\infty \\ &\leq \gamma ||V_t - V^*||_\infty \\ &\leq \gamma^t ||V_1 - V^*||_\infty \end{aligned}$$

In other words, the Bellman operator B is a γ-contracting operator.

16.2 Linear Quadratic Regulation (LQR)

In this section, we'll cover a special case of the finite-horizon setting described in Section 16.1, for which the **exact solution** is (easily) tractable. This model is widely used in robotics, and a common technique in many problems is to reduce the formulation to this framework.

First, let's describe the model's assumptions. We place ourselves in the continuous setting, with

$$\mathcal{S} = \mathbb{R}^d, \quad \mathcal{A} = \mathbb{R}^d$$

and we'll assume **linear transitions** (with noise)

$$s_{t+1} = A_t s_t + B_t a_t + w_t$$

where $A_t \in R^{d \times d}, B_t \in R^{d \times d}$ are matrices and $w_t \sim \mathcal{N}(0, \Sigma_t)$ is some gaussian noise (with **zero** mean). As we'll show in the following paragraphs, it turns out that the noise, as long as it has zero mean, does not impact the optimal policy!

We'll also assume **quadratic rewards**

$$R^{(t)}(s_t, a_t) = -s_t^\top U_t s_t - a_t^\top W_t a_t$$

where $U_t \in R^{d\times n}, W_t \in R^{d\times d}$ are positive definite matrices (meaning that the reward is always **negative**).

Remark Note that the quadratic formulation of the reward is equivalent to saying that we want our state to be close to the origin (where the reward is higher). For example, if $U_t = I_d$ (the identity matrix) and $W_t = I_d$, then $R_t = -||s_t||^2 - ||a_t||^2$, meaning that we want to take smooth actions (small norm of a_t) to go back to the origin (small norm of s_t). This could model a car trying to stay in the middle of lane without making impulsive moves...

Now that we have defined the assumptions of our LQR model, let's cover the 2 steps of the LQR algorithm

step 1 suppose that we don't know the matrices A, B, Σ. To estimate them, we can follow the ideas outlined in the Value Approximation section of the RL notes. First, collect transitions from an arbitrary policy. Then, use linear regression to find $\text{argmin}_{A,B} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(A s_t^{(i)} + B a_t^{(i)} \right) \right\|^2$. Finally, use a technique seen in Gaussian Discriminant Analysis to learn Σ.

step 2 assuming that the parameters of our model are known (given or estimated with step 1), we can derive the optimal policy using dynamic programming.

In other words, given

$$\begin{cases} s_{t+1} &= A_t s_t + B_t a_t + w_t \qquad A_t, B_t, U_t, W_t, \Sigma_t \text{ known} \\ R^{(t)}(s_t, a_t) &= -s_t^\top U_t s_t - a_t^\top W_t a_t \end{cases}$$

we want to compute V_t^*. If we go back to section 16.1, we can apply dynamic programming, which yields

1. **Initialization step**

   For the last time step T,

$$\begin{aligned} V_T^*(s_T) &= \max_{a_T \in \mathcal{A}} R_T(s_T, a_T) \\ &= \max_{a_T \in \mathcal{A}} -s_T^\top U_T s_T - a_T^\top W_t a_T \\ &= -s_T^\top U_t s_T \qquad\qquad \text{(maximized for } a_T = 0) \end{aligned}$$

2. **Recurrence step**

   Let $t < T$. Suppose we know V^*_{t+1}.

   Fact 1: It can be shown that if V^*_{t+1} is a quadratic function in s_t, then V^*_t is also a quadratic function. In other words, there exists some matrix Φ and some scalar Ψ such that

$$\begin{aligned} &\text{if } V^*_{t+1}(s_{t+1}) = s_{t+1}^\top \Phi_{t+1} s_{t+1} + \Psi_{t+1} \\ &\text{then } V^*_t(s_t) = s_t^\top \Phi_t s_t + \Psi_t \end{aligned}$$

   For time step $t = T$, we had $\Phi_t = -U_T$ and $\Psi_T = 0$.

   Fact 2: We can show that the optimal policy is just a linear function of the state.

   Knowing V^*_{t+1} is equivalent to knowing Φ_{t+1} and Ψ_{t+1}, so we just need to explain how we compute Φ_t and Ψ_t from Φ_{t+1} and Ψ_{t+1} and the other parameters of the problem.

$$\begin{aligned} V^*_t(s_t) &= s_t^\top \Phi_t s_t + \Psi_t \\ &= \max_{a_t} \left[R^{(t)}(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim P^{(t)}_{s_t, a_t}} \left[V^*_{t+1}(s_{t+1}) \right] \right] \\ &= \max_{a_t} \left[-s_t^\top U_t s_t - a_t^\top V_t a_t + \mathbb{E}_{s_{t+1} \sim \mathcal{N}(A_t s_t + B_t a_t, \Sigma_t)} \left[s_{t+1}^\top \Phi_{t+1} s_{t+1} + \Psi_{t+1} \right] \right] \end{aligned}$$

   where the second line is just the definition of the optimal value function and the third line is obtained by plugging in the dynamics of our model along with the quadratic assumption. Notice that the last expression is a quadratic function in a_t and can thus be (easily) optimized[1]. We get the optimal action a^*_t

$$\begin{aligned} a^*_t &= \left[(B_t^\top \Phi_{t+1} B_t - V_t)^{-1} B_t \Phi_{t+1} A_t \right] \cdot s_t \\ &= L_t \cdot s_t \end{aligned}$$

   where

$$L_t := \left[(B_t^\top \Phi_{t+1} B_t - W_t)^{-1} B_t \Phi_{t+1} A_t \right]$$

[1] Use the identity $\mathbb{E}\left[w_t^\top \Phi_{t+1} w_t \right] = \mathrm{Tr}(\Sigma_t \Phi_{t+1})$ with $w_t \sim \mathcal{N}(0, \Sigma_t)$

which is an impressive result: our optimal policy is **linear in** s_t. Given a_t^* we can solve for Φ_t and Ψ_t. We finally get the **Discrete Ricatti equations**

$$\Phi_t = A_t^\top \left(\Phi_{t+1} - \Phi_{t+1} B_t \left(B_t^\top \Phi_{t+1} B_t - W_t\right)^{-1} B_t \Phi_{t+1}\right) A_t - U_t$$
$$\Psi_t = -\operatorname{tr}\left(\Sigma_t \Phi_{t+1}\right) + \Psi_{t+1}$$

Fact 3: we notice that Φ_t depends on neither Ψ nor the noise Σ_t! As L_t is a function of A_t, B_t and Φ_{t+1}, it implies that the optimal policy also **does not depend on the noise**! (But Ψ_t does depend on Σ_t, which implies that V_t^* depends on Σ_t.)

Then, to summarize, the LQR algorithm works as follows

1. (if necessary) estimate parameters A_t, B_t, Σ_t
2. initialize $\Phi_T := -U_T$ and $\Psi_T := 0$.
3. iterate from $t = T - 1 \dots 0$ to update Φ_t and Ψ_t using Φ_{t+1} and Ψ_{t+1} using the discrete Ricatti equations. If there exists a policy that drives the state towards zero, then convergence is guaranteed!

Using Fact 3, we can be even more clever and make our algorithm run (slightly) faster! As the optimal policy does not depend on Ψ_t, and the update of Φ_t only depends on Φ_t, it is sufficient to update **only** Φ_t!

16.3 From non-linear dynamics to LQR

It turns out that a lot of problems can be reduced to LQR, even if dynamics are non-linear. While LQR is a nice formulation because we are able to come up with a nice exact solution, it is far from being general. Let's take for instance the case of the inverted pendulum. The transitions between states look like

$$\begin{pmatrix} x_{t+1} \\ \dot{x}_{t+1} \\ \theta_{t+1} \\ \dot{\theta}_{t+1} \end{pmatrix} = F\left(\begin{pmatrix} x_t \\ \dot{x}_t \\ \theta_t \\ \dot{\theta}_t \end{pmatrix}, a_t\right)$$

where the function F depends on the cos of the angle etc. Now, the question we may ask is

Can we linearize this system?

16.3.1 Linearization of dynamics

Let's suppose that at time t, the system spends most of its time in some state $\bar{s_t}$ and the actions we perform are around $\bar{a_t}$. For the inverted pendulum, if we reached some kind of optimal, this is true: our actions are small and we don't deviate much from the vertical.

We are going to use Taylor expansion to linearize the dynamics. In the simple case where the state is one-dimensional and the transition function F does not depend on the action, we would write something like

$$s_{t+1} = F(s_t) \approx F(\bar{s_t}) + F'(\bar{s_t}) \cdot (s_t - \bar{s_t})$$

In the more general setting, the formula looks the same, with gradients instead of simple derivatives

$$s_{t+1} \approx F(\bar{s_t}, \bar{a_t}) + \nabla_s F(\bar{s_t}, \bar{a_t}) \cdot (s_t - \bar{s_t}) + \nabla_a F(\bar{s_t}, \bar{a_t}) \cdot (a_t - \bar{a_t}) \qquad (16.3)$$

and now, s_{t+1} is linear in s_t and a_t, because we can rewrite equation (16.3) as

$$s_{t+1} \approx As_t + Bs_t + \kappa$$

where κ is some constant and A, B are matrices. Now, this writing looks awfully similar to the assumptions made for LQR. We just have to get rid of the constant term κ! It turns out that the constant term can be absorbed into s_t by artificially increasing the dimension by one. This is the same trick that we used at the beginning of the class for linear regression...

16.3.2 Differential Dynamic Programming (DDP)

The previous method works well for cases where the goal is to stay around some state s^* (think about the inverted pendulum, or a car having to stay in the middle of a lane). However, in some cases, the goal can be more complicated.

We'll cover a method that applies when our system has to follow some trajectory (think about a rocket). This method is going to discretize the trajectory into discrete time steps, and create intermediary goals around which we will be able to use the previous technique! This method is called **Differential Dynamic Programming**. The main steps are

step 1 come up with a nominal trajectory using a naive controller, that approximate the trajectory we want to follow. In other words, our controller is able to approximate the gold trajectory with

$$s_0^*, a_0^* \to s_1^*, a_1^* \to \dots$$

step 2 linearize the dynamics around each trajectory point s_t^*, in other words

$$s_{t+1} \approx F(s_t^*, a_t^*) + \nabla_s F(s_t^*, a_t^*)(s_t - s_t^*) + \nabla_a F(s_t^*, a_t^*)(a_t - a_t^*)$$

where s_t, a_t would be our current state and action. Now that we have a linear approximation around each of these points, we can use the previous section and rewrite

$$s_{t+1} = A_t \cdot s_t + B_t \cdot a_t$$

(notice that in that case, we use the non-stationary dynamics setting that we mentioned at the beginning of these lecture notes)

Note We can apply a similar derivation for the reward $R^{(t)}$, with a second-order Taylor expansion.

$$\begin{aligned} R(s_t, a_t) \approx\ & R(s_t^*, a_t^*) + \nabla_s R(s_t^*, a_t^*)(s_t - s_t^*) + \nabla_a R(s_t^*, a_t^*)(a_t - a_t^*) \\ & + \frac{1}{2}(s_t - s_t^*)^\top H_{ss}(s_t - s_t^*) + (s_t - s_t^*)^\top H_{sa}(a_t - a_t^*) \\ & + \frac{1}{2}(a_t - a_t^*)^\top H_{aa}(a_t - a_t^*) \end{aligned}$$

where H_{xy} refers to the entry of the Hessian of R with respect to x and y evaluated in (s_t^*, a_t^*) (omitted for readability). This expression can be re-written as

$$R_t(s_t, a_t) = -s_t^\top U_t s_t - a_t^\top W_t a_t$$

for some matrices U_t, W_t, with the same trick of adding an extra dimension of ones. To convince yourself, notice that

$$\begin{pmatrix} 1 & x \end{pmatrix} \cdot \begin{pmatrix} a & b \\ b & c \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \end{pmatrix} = a + 2bx + cx^2$$

step 3 Now, you can convince yourself that our problem is **strictly** re-written in the LQR framework. Let's just use LQR to find the optimal policy π_t. As a result, our new controller will (hopefully) be better!

Note: Some problems might arise if the LQR trajectory deviates too much from the linearized approximation of the trajectory, but that can be fixed with reward-shaping...

step 4 Now that we get a new controller (our new policy π_t), we use it to produce a new trajectory

$$s_0^*, \pi_0(s_0^*) \to s_1^*, \pi_1(s_1^*) \to \ldots \to s_T^*$$

note that when we generate this new trajectory, we use the real F and not its linear approximation to compute transitions, meaning that

$$s_{t+1}^* = F(s_t^*, a_t^*)$$

then, go back to step 2 and repeat until some stopping criterion.

16.4 Linear Quadratic Gaussian (LQG)

Often, in the real word, we don't get to observe the full state s_t. For example, an autonomous car could receive an image from a camera, which is merely an **observation**, and not the full state of the world. So far, we assumed that the state was available. As this might not hold true for most of the real-world problems, we need a new tool to model this situation: **Partially Observable MDPs**.

A POMDP is an MDP with an extra observation layer. In other words, we introduce a new variable o_t, that follows some conditional distribution given the current state s_t

$$o_t | s_t \sim O(o|s)$$

Formally, a finite-horizon POMDP is given by a tuple

$$(\mathcal{S}, \mathcal{O}, \mathcal{A}, P_{sa}, T, R)$$

Within this framework, the general strategy is to maintain a **belief state** (distribution over states) based on the observation $o_1, \ldots, o_t$. Then, a policy in a POMDP maps this belief states to actions.

In this section, we'll present a extension of LQR to this new setting. Assume that we observe $y_t \in \mathbb{R}^n$ with $m < n$ such that

$$\begin{cases} y_t & = C \cdot s_t + v_t \\ s_{t+1} & = A \cdot s_t + B \cdot a_t + w_t \end{cases}$$

where $C \in R^{n \times d}$ is a compression matrix and v_t is the sensor noise (also gaussian, like w_t). Note that the reward function $R^{(t)}$ is left unchanged, as a function of the state (not the observation) and action. Also, as distributions are gaussian, the belief state is also going to be gaussian. In this new framework, let's give an overview of the strategy we are going to adopt to find the optimal policy:

step 1 first, compute the distribution on the possible states (the belief state), based on the observations we have. In other words, we want to compute the mean $s_{t|t}$ and the covariance $\Sigma_{t|t}$ of

$$s_t | y_1, \dots, y_t \sim \mathcal{N}\left(s_{t|t}, \Sigma_{t|t}\right)$$

to perform the computation efficiently over time, we'll use the **Kalman Filter** algorithm (used on-board Apollo Lunar Module!).

step 2 now that we have the distribution, we'll use the mean $s_{t|t}$ as the best approximation for s_t

step 3 then set the action $a_t := L_t s_{t|t}$ where L_t comes from the regular LQR algorithm.

Intuitively, to understand why this works, notice that $s_{t|t}$ is a noisy approximation of s_t (equivalent to adding more noise to LQR) but we proved that LQR is independent of the noise!

Step 1 needs to be explicated. We'll cover a simple case where there is no action dependence in our dynamics (but the general case follows the same idea). Suppose that

$$\begin{cases} s_{t+1} & = A \cdot s_t + w_t, \quad w_t \sim N(0, \Sigma_s) \\ y_t & = C \cdot s_t + v_t, \quad v_t \sim N(0, \Sigma_y) \end{cases}$$

As noises are Gaussians, we can easily prove that the joint distribution is also Gaussian

$$\begin{pmatrix} s_1 \\ \vdots \\ s_t \\ y_1 \\ \vdots \\ y_t \end{pmatrix} \sim \mathcal{N}(\mu, \Sigma) \qquad \text{for some } \mu, \Sigma$$

then, using the marginal formulas of gaussians (see Factor Analysis notes), we would get

$$s_t | y_1, \ldots, y_t \sim \mathcal{N}\left(s_{t|t}, \Sigma_{t|t}\right)$$

However, computing the marginal distribution parameters using these formulas would be computationally expensive! It would require manipulating matrices of shape $t \times t$. Recall that inverting a matrix can be done in $O(t^3)$, and it would then have to be repeated over the time steps, yielding a cost in $O(t^4)$!

The **Kalman filter** algorithm provides a much better way of computing the mean and variance, by updating them over time in **constant time in** t! The kalman filter is based on two basics steps. Assume that we know the distribution of $s_t | y_1, \ldots, y_t$:

predict step compute $s_{t+1} | y_1, \ldots, y_t$

update step compute $s_{t+1} | y_1, \ldots, y_{t+1}$

and iterate over time steps! The combination of the predict and update steps updates our belief states. In other words, the process looks like

$$(s_t | y_1, \ldots, y_t) \xrightarrow{\text{predict}} (s_{t+1} | y_1, \ldots, y_t) \xrightarrow{\text{update}} (s_{t+1} | y_1, \ldots, y_{t+1}) \xrightarrow{\text{predict}} \ldots$$

predict step Suppose that we know the distribution of

$$s_t | y_1, \ldots, y_t \sim \mathcal{N}\left(s_{t|t}, \Sigma_{t|t}\right)$$

then, the distribution over the next state is also a gaussian distribution

$$s_{t+1} | y_1, \ldots, y_t \sim \mathcal{N}\left(s_{t+1|t}, \Sigma_{t+1|t}\right)$$

where

$$\begin{cases} s_{t+1|t} &= A \cdot s_{t|t} \\ \Sigma_{t+1|t} &= A \cdot \Sigma_{t|t} \cdot A^\top + \Sigma_s \end{cases}$$

update step given $s_{t+1|t}$ and $\Sigma_{t+1|t}$ such that

$$s_{t+1}|y_1, \dots, y_t \sim \mathcal{N}\left(s_{t+1|t}, \Sigma_{t+1|t}\right)$$

we can prove that

$$s_{t+1}|y_1, \dots, y_{t+1} \sim \mathcal{N}\left(s_{t+1|t+1}, \Sigma_{t+1|t+1}\right)$$

where

$$\begin{cases} s_{t+1|t+1} &= s_{t+1|t} + K_t(y_{t+1} - C s_{t+1|t}) \\ \Sigma_{t+1|t+1} &= \Sigma_{t+1|t} - K_t \cdot C \cdot \Sigma_{t+1|t} \end{cases}$$

with

$$K_t := \Sigma_{t+1|t} C^\top (C \Sigma_{t+1|t} C^\top + \Sigma_y)^{-1}$$

The matrix K_t is called the **Kalman gain**.

Now, if we have a closer look at the formulas, we notice that we don't need the observations prior to time step t! The update steps only depends on the previous distribution. Putting it all together, the algorithm first runs a forward pass to compute the K_t, $\Sigma_{t|t}$ and $s_{t|t}$ (sometimes referred to as $\hat{s}$ in the literature). Then, it runs a backward pass (the LQR updates) to compute the quantities Ψ_t, Ψ_t and L_t. Finally, we recover the optimal policy with $a_t^* = L_t s_{t|t}$.

Chapter 17

Policy Gradient (REINFORCE)

We will present a model-free algorithm called REINFORCE that does not require the notion of value functions and Q functions. It turns out to be more convenient to introduce REINFORCE in the finite horizon case, which will be assumed throughout this note: we use $\tau = (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T)$ to denote a trajectory, where $T < \infty$ is the length of the trajectory. Moreover, REINFORCE only applies to learning a **randomized policy**. We use $\pi_\theta(a|s)$ to denote the probability of the policy π_θ outputting the action a at state s. The other notations will be the same as in previous lecture notes.

The advantage of applying REINFORCE is that we only need to assume that we can sample from the transition probabilities $\{P_{sa}\}$ and can query the reward function $R(s, a)$ at state s and action a,[1] but we do not need to know the analytical form of the transition probabilities or the reward function. We do not explicitly learn the transition probabilities or the reward function either.

Let s_0 be sampled from some distribution μ. We consider optimizing the expected total payoff of the policy π_θ over the parameter θ defined as.

$$\eta(\theta) \triangleq \mathrm{E}\left[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)\right] \tag{17.1}$$

Recall that $s_t \sim P_{s_{t-1}a_{t-1}}$ and $a_t \sim \pi_\theta(\cdot|s_t)$. Also note that $\eta(\theta) = \mathrm{E}_{s_0 \sim P}\left[V^{\pi_\theta}(s_0)\right]$ if we ignore the difference between finite and infinite horizon.

[1] In this notes we will work with the general setting where the reward depends on both the state and the action.

We aim to use gradient ascent to maximize $\eta(\theta)$. The main challenge we face here is to compute (or estimate) the gradient of $\eta(\theta)$ without the knowledge of the form of the reward function and the transition probabilities.

Let $P_\theta(\tau)$ denote the distribution of τ (generated by the policy π_θ), and let $f(\tau) = \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)$. We can rewrite $\eta(\theta)$ as

$$\eta(\theta) = \mathrm{E}_{\tau \sim P_\theta}[f(\tau)] \tag{17.2}$$

We face a similar situations in the variational auto-encoder (VAE) setting covered in the previous lectures, where the we need to take the gradient w.r.t to a variable that shows up under the expectation — the distribution P_θ depends on θ. Recall that in VAE, we used the re-parametrization techniques to address this problem. However it does not apply here because we do know not how to compute the gradient of the function f. (We only have an efficient way to evaluate the function f by taking a weighted sum of the observed rewards, but we do not necessarily know the reward function itself to compute the gradient.)

The REINFORCE algorithm uses an another approach to estimate the gradient of $\eta(\theta)$. We start with the following derivation:

$$\begin{aligned}
\nabla_\theta \mathrm{E}_{\tau \sim P_\theta}[f(\tau)] &= \nabla_\theta \int P_\theta(\tau) f(\tau) d\tau \\
&= \int \nabla_\theta (P_\theta(\tau) f(\tau)) d\tau \quad \text{(swap integration with gradient)} \\
&= \int (\nabla_\theta P_\theta(\tau)) f(\tau) d\tau \quad \text{(becaue } f \text{ does not depend on } \theta) \\
&= \int P_\theta(\tau) (\nabla_\theta \log P_\theta(\tau)) f(\tau) d\tau \\
&\qquad\qquad \text{(because } \nabla \log P_\theta(\tau) = \tfrac{\nabla P_\theta(\tau)}{P_\theta(\tau)}\text{)} \\
&= \mathrm{E}_{\tau \sim P_\theta}[(\nabla_\theta \log P_\theta(\tau)) f(\tau)] \qquad (17.3)
\end{aligned}$$

Now we have a sample-based estimator for $\nabla_\theta \mathrm{E}_{\tau \sim P_\theta}[f(\tau)]$. Let $\tau^{(1)}, \ldots, \tau^{(n)}$ be n empirical samples from P_θ (which are obtained by running the policy π_θ for n times, with T steps for each run). We can estimate the gradient of $\eta(\theta)$ by

$$\begin{aligned}
\nabla_\theta \mathrm{E}_{\tau \sim P_\theta}[f(\tau)] &= \mathrm{E}_{\tau \sim P_\theta}[(\nabla_\theta \log P_\theta(\tau)) f(\tau)] \qquad (17.4) \\
&\approx \frac{1}{n} \sum_{i=1}^{n} (\nabla_\theta \log P_\theta(\tau^{(i)})) f(\tau^{(i)}) \qquad (17.5)
\end{aligned}$$

The next question is how to compute $\log P_\theta(\tau)$. We derive an analytical formula for $\log P_\theta(\tau)$ and compute its gradient w.r.t θ (using auto-differentiation). Using the definition of τ, we have

$$P_\theta(\tau) = \mu(s_0)\pi_\theta(a_0|s_0)P_{s_0a_0}(s_1)\pi_\theta(a_1|s_1)P_{s_1a_1}(s_2)\cdots P_{s_{T-1}a_{T-1}}(s_T) \quad (17.6)$$

Here recall that μ to used to denote the density of the distribution of s_0. It follows that

$$\begin{aligned}\log P_\theta(\tau) = \log \mu(s_0) + \log \pi_\theta(a_0|s_0) + \log P_{s_0a_0}(s_1) + \log \pi_\theta(a_1|s_1) \\ + \log P_{s_1a_1}(s_2) + \cdots + \log P_{s_{T-1}a_{T-1}}(s_T)\end{aligned} \quad (17.7)$$

Taking gradient w.r.t to θ, we obtain

$$\nabla_\theta \log P_\theta(\tau) = \nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1) + \cdots + \nabla_\theta \log \pi_\theta(a_{T-1}|s_{T-1})$$

Note that many of the terms disappear because they don't depend on θ and thus have zero gradients. (This is somewhat important — we don't know how to evaluate those terms such as $\log P_{s_0a_0}(s_1)$ because we don't have access to the transition probabilities, but luckily those terms have zero gradients!)

Plugging the equation above into equation (17.4), we conclude that

$$\begin{aligned}\nabla_\theta \eta(\theta) = \nabla_\theta \mathrm{E}_{\tau\sim P_\theta}\left[f(\tau)\right] &= \mathrm{E}_{\tau\sim P_\theta}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\cdot f(\tau)\right] \\ &= \mathrm{E}_{\tau\sim P_\theta}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\cdot\left(\sum_{t=0}^{T-1}\gamma^t R(s_t,a_t)\right)\right]\end{aligned} \quad (17.8)$$

We estimate the RHS of the equation above by empirical sample trajectories, and the estimate is unbiased. The vanilla REINFORCE algorithm iteratively updates the parameter by gradient ascent using the estimated gradients.

Interpretation of the policy gradient formula (17.8)**.** The quantity $\nabla_\theta P_\theta(\tau) = \sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)$ is intuitively the direction of the change of θ that will make the trajectory τ more likely to occur (or increase the probability of choosing action $a_0, \ldots, a_{t-1}$), and $f(\tau)$ is the total payoff of this trajectory. Thus, by taking a gradient step, intuitively we are trying to improve the likelihood of all the trajectories, but with a different emphasis or weight for each τ (or for each set of actions $a_0, a_1, \ldots, a_{t-1}$). If τ is very rewarding (that is, $f(\tau)$ is large), we try very hard to move in the direction

that can increase the probability of the trajectory τ (or the direction that increases the probability of choosing $a_0, \ldots, a_{t-1}$), and if τ has low payoff, we try less hard with a smaller weight.

An interesting fact that follows from formula (17.3) is that

$$\mathrm{E}_{\tau \sim P_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0 \tag{17.9}$$

To see this, we take $f(\tau) = 1$ (that is, the reward is always a constant), then the LHS of (17.8) is zero because the payoff is always a fixed constant $\sum_{t=0}^{T} \gamma^t$. Thus the RHS of (17.8) is also zero, which implies (17.9).

In fact, one can verify that $\mathrm{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t) = 0$ for any fixed t and s_t.[2] This fact has two consequences. First, we can simplify formula (17.8) to

$$\begin{aligned}
\nabla_\theta \eta(\theta) &= \sum_{t=0}^{T-1} \mathrm{E}_{\tau \sim P_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(\sum_{j=0}^{T-1} \gamma^j R(s_j, a_j)\right)\right] \\
&= \sum_{t=0}^{T-1} \mathrm{E}_{\tau \sim P_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(\sum_{j \geq t}^{T-1} \gamma^j R(s_j, a_j)\right)\right]
\end{aligned} \tag{17.10}$$

where the second equality follows from

$$\begin{aligned}
&\mathrm{E}_{\tau \sim P_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(\sum_{0 \leq j < t} \gamma^j R(s_j, a_j)\right)\right] \\
&= \mathrm{E}\left[\mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)|s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t\right] \cdot \left(\sum_{0 \leq j < t} \gamma^j R(s_j, a_j)\right)\right] \\
&= 0 \qquad \text{(because } \mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)|s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t\right] = 0\text{)}
\end{aligned}$$

Note that here we used the law of total expectation. The outer expectation in the second line above is over the randomness of $s_0, a_0, \ldots, a_{t-1}, s_t$, whereas the inner expectation is over the randomness of a_t (conditioned on $s_0, a_0, \ldots, a_{t-1}, s_t$.) We see that we've made the estimator slightly simpler. The second consequence of $\mathrm{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t) = 0$ is the following: for any value $B(s_t)$ that only depends on s_t, it holds that

$$\begin{aligned}
&\mathrm{E}_{\tau \sim P_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot B(s_t)\right] \\
&= \mathrm{E}\left[\mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)|s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t\right] B(s_t)\right] \\
&= 0 \qquad \text{(because } \mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)|s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t\right] = 0\text{)}
\end{aligned}$$

[2] In general, it's true that $\mathrm{E}_{x \sim p_\theta}[\nabla \log p_\theta(x)] = 0$.

Again here we used the law of total expectation. The outer expectation in the second line above is over the randomness of $s_0, a_0, \ldots, a_{t-1}, s_t$, whereas the inner expectation is over the randomness of a_t (conditioned on $s_0, a_0, \ldots, a_{t-1}, s_t$.) It follows from equation (17.10) and the equation above that

$$\begin{aligned}\nabla_\theta \eta(\theta) &= \sum_{t=0}^{T-1} \mathrm{E}_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(\sum_{j \geq t}^{T-1} \gamma^j R(s_j, a_j) - \gamma^t B(s_t) \right) \right] \\ &= \sum_{t=0}^{T-1} \mathrm{E}_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \gamma^t \left(\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j) - B(s_t) \right) \right] \end{aligned} \tag{17.11}$$

Therefore, we will get a different estimator for estimating the $\nabla \eta(\theta)$ with a difference choice of $B(\cdot)$. The benefit of introducing a proper $B(\cdot)$ — which is often referred to as a **baseline** — is that it helps reduce the variance of the estimator.[3] It turns out that a near optimal estimator would be the expected future payoff $\mathrm{E}\left[\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j) | s_t\right]$, which is pretty much the same as the value function $V^{\pi_\theta}(s_t)$ (if we ignore the difference between finite and infinite horizon.) Here one could estimate the value function $V^{\pi_\theta}(\cdot)$ in a crude way, because its precise value doesn't influence the mean of the estimator but only the variance. This leads to a policy gradient algorithm with baselines stated in Algorithm 7.[4]

[3] As a heuristic but illustrating example, suppose for a fixed t, the future reward $\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j)$ randomly takes two values $1000 + 1$ and $1000 - 2$ with equal probability, and the corresponding values for $\nabla_\theta \log \pi_\theta(a_t|s_t)$ are vector z and $-z$. (Note that because $\mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0$, if $\nabla_\theta \log \pi_\theta(a_t|s_t)$ can only take two values uniformly, then the two values have to two vectors in an opposite direction.) In this case, without subtracting the baseline, the estimators take two values $(1000 + 1)z$ and $-(1000 - 2)z$, whereas after subtracting a baseline of 1000, the estimator has two values z and $2z$. The latter estimator has much lower variance compared to the original estimator.

[4] We note that the estimator of the gradient in the algorithm does not exactly match the equation 17.11. If we multiply γ^t in the summand of equation (17.13), then they will exactly match. Removing such discount factors empirically works well because it gives a large update.

Algorithm 7 Vanilla policy gradient with baseline

for $i = 1, \cdots$ **do**

Collect a set of trajectories by executing the current policy. Use $R_{\geq t}$ as a shorthand for $\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j)$

Fit the baseline by finding a function B that minimizes

$$\sum_{\tau} \sum_{t} (R_{\geq t} - B(s_t))^2 \tag{17.12}$$

Update the policy parameter θ with the gradient estimator

$$\sum_{\tau} \sum_{t} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_{\geq t} - B(s_t)) \tag{17.13}$$

Bibliography

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

Mikhail Belkin, Daniel Hsu, and Ji Xu. Two models of double descent for weak features. *SIAM Journal on Mathematics of Data Science*, 2(4):1167–1180, 2020.

David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International Conference on Machine Learning*, pages 1597–1607. PMLR, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

Jeff Z HaoChen, Colin Wei, Jason D Lee, and Tengyu Ma. Shape matters: Understanding the implicit bias of the noise covariance. *arXiv preprint arXiv:2006.08680*, 2020.

Trevor Hastie, Andrea Montanari, Saharon Rosset, and Ryan J Tibshirani. Surprises in high-dimensional ridgeless least squares interpolation. 2019.

Trevor Hastie, Andrea Montanari, Saharon Rosset, and Ryan J Tibshirani. Surprises in high-dimensional ridgeless least squares interpolation. *The Annals of Statistics*, 50(2):949–986, 2022.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015. URL `http://jmlr.org/proceedings/papers/v37/ioffe15.html`.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning, second edition*, volume 112. Springer, 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Yuping Luo, Huazhe Xu, Yuanzhi Li, Yuandong Tian, Trevor Darrell, and Tengyu Ma. Algorithmic framework for model-based deep reinforcement learning with theoretical guarantees. In *International Conference on Learning Representations*, 2018.

Song Mei and Andrea Montanari. The generalization error of random features regression: Precise asymptotics and the double descent curve. *Communications on Pure and Applied Mathematics*, 75(4):667–766, 2022.

Preetum Nakkiran. More data can hurt for linear regression: Sample-wise double descent. 2019.

Preetum Nakkiran, Prayaag Venkat, Sham Kakade, and Tengyu Ma. Optimal regularization can mitigate double descent. 2020.

Manfred Opper. Statistical mechanics of learning: Generalization. *The handbook of brain theory and neural networks*, pages 922–925, 1995.

Manfred Opper. Learning to generalize. *Frontiers of Life*, 3(part 2):763–775, 2001.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Blake Woodworth, Suriya Gunasekar, Jason D Lee, Edward Moroshko, Pedro Savarese, Itay Golan, Daniel Soudry, and Nathan Srebro. Kernel and rich regimes in overparametrized models. *arXiv preprint arXiv:2002.09277*, 2020.

Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.