# CMPSCI 687: Reinforcement Learning

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Syllabus</td>
</tr>
<tr>
<td>1.1</td>
<td>Class</td>
</tr>
<tr>
<td>1.2</td>
<td>Website</td>
</tr>
<tr>
<td>1.3</td>
<td>Book</td>
</tr>
<tr>
<td>1.4</td>
<td>Office Hours</td>
</tr>
<tr>
<td>1.5</td>
<td>Teaching Assistants and Office Hours</td>
</tr>
<tr>
<td>1.6</td>
<td>Piazza</td>
</tr>
<tr>
<td>1.7</td>
<td>Support Summary</td>
</tr>
<tr>
<td>1.8</td>
<td>Grading</td>
</tr>
<tr>
<td>1.9</td>
<td>Late Policy</td>
</tr>
<tr>
<td>1.10</td>
<td>Missing Class / Assignments</td>
</tr>
<tr>
<td>1.11</td>
<td>Disability Services</td>
</tr>
<tr>
<td>1.12</td>
<td>Cheating</td>
</tr>
<tr>
<td>1.13</td>
<td>\LaTeX</td>
</tr>
<tr>
<td>2</td>
<td>Introduction</td>
</tr>
<tr>
<td>2.1</td>
<td>Notation</td>
</tr>
<tr>
<td>2.2</td>
<td>What is Reinforcement Learning (RL)?</td>
</tr>
<tr>
<td>2.3</td>
<td>687-Gridworld: A Simple Environment</td>
</tr>
<tr>
<td>2.4</td>
<td>Describing the Agent and Environment Mathematically</td>
</tr>
<tr>
<td>Homework 1</td>
<td></td>
</tr>
<tr>
<td>Pop Quiz 1</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>Additional Terminology, Notation, and Assumptions</td>
</tr>
<tr>
<td>3</td>
<td>Black-Box Optimization for RL</td>
</tr>
<tr>
<td>3.1</td>
<td>Hello Environment!</td>
</tr>
<tr>
<td>3.2</td>
<td>Black-Box Optimization (BBO) for Policy Search</td>
</tr>
<tr>
<td>3.3</td>
<td>Cross-Entropy Method</td>
</tr>
<tr>
<td>3.4</td>
<td>First-Choice Hill-Climbing</td>
</tr>
</tbody>
</table>
1 Syllabus

1.1 Class

Class will be held on Tuesdays and Thursdays from 4:00pm–5:15pm in Engineering Lab II, Room 119. Lectures will be given primarily on the whiteboard, with typed notes provided below and updated throughout the semester as additional material is covered. These notes are not a complete summary of all material that students are responsible for—you are responsible for all material covered in class, even if it is not present in these notes.

1.2 Website

The class website is https://people.cs.umass.edu/~pthomas/courses/CMPSCI_687_Fall2019.html. All homework assignments, due dates, and notes will be posted there.

1.3 Book

The start of the course will be roughly based on the first edition of Sutton and Barto’s book, Reinforcement Learning: An Introduction. It can be found on Amazon here. It is also available for free online here. Although the book is a fantastic introduction to the topic (and I encourage purchasing a copy if you plan to study reinforcement learning), owning the book is not a requirement.

1.4 Office Hours

Prof. Thomas’ office hours will be Mondays from 1:30pm–3:00pm in his office, room 346 of the computer science building. Office hours will follow the academic calendar: they will be offered from 1:30-3:00pm on (and only on) all days that are a Monday schedule.

1.5 Teaching Assistants and Office Hours

The teaching assistants (TAs) this semester will be Blossom Metevier (bmetevier@umass.edu) and Scott Jordan (sjordan@cs.umass.edu). Blossom will have office hours on Fridays from 10:00am–11:00am in CS Building room 207. Scott will have office hours on Tuesdays from 2:45pm–4pm in CS Building room 207.

1.6 Piazza

The course will use Piazza as a forum where you can ask questions. Every afternoon (at some time between 1pm and 6pm), Prof. Thomas or one of the TAs will go through and answer all of the questions on Piazza. If you asked
a question before 1pm on a weekday that is not a holiday, and did not get a
response by 6pm, please e-mail Prof. Thomas directly (pthomas@cs.umass.edu),
as this should not occur.

1.7 Support Summary
Office hours:

- **Mondays**: 1:30pm-3:00pm, Prof. Thomas’ office hours, CS room 346.
- **Tuesdays**: 2:45pm–4:00pm, Scott Jordan’s office hours, CS room 207.
- **Friday**: 10:00am–11:00am, Blossom Metevier’s office hours, CS room 207.

Piazza question answering (some time between 1pm and 6pm):

- **Mondays**: Blossom Metevier
- **Tuesdays**: Scott Jordan
- **Wednesdays**: Blossom Metevier
- **Thursdays**: Prof. Thomas
- **Fridays**: Scott Jordan

1.8 Grading

Your grade will have four components:

1. **Homework Assignments** (50%): There will be *roughly* seven homework
assignments. Each problem in an assignment will specify its point value,
and not all homework assignments will necessarily have the same point
value (i.e., some homework assignments may be smaller and worth less
than others). All assignments will have total point values of at most 100
(thus, the last assignment will not have a point value so high that previous
assignments are irrelevant).

2. **Pop Quizzes** (15%): There will be pop-quizzes given in class without
prior announcement. They will typically take about 10 minutes to complete
and will be given at the start of class. If you know in advance that you
will miss class, please e-mail both TAs, and you may be excused from any
quizzes that occur that day.

3. **Midterm Exam** (15%): There will be a midterm exam in class (the
regular room and time) on Tuesday November 19.

4. **Project** (20%): There will be a course project. The details of the project
will be announced later in the semester, and may depend on how much
content is covered.
A cumulative grade in $[90\% - 100\%]$ will be an A- or A, $[80\%, 90\%)$ will be a B-, B, or B+, and $[70\%, 80\%)$ will be a C-, C, or C+. Course grades will be curved only in students’ favor (that is, these thresholds may be lowered, but a grade of 90% will not be lower than an A-).

1.9 Late Policy

Late homework assignments will not be accepted. An assignment submitted one minute late is late, and will not be accepted. I recommend submitting homework well in advance of the due date and time.

1.10 Missing Class / Assignments

If you are going to miss class, e-mail the TAs (not Prof. Thomas) before the start of class letting them know. You will then be excused from any pop-quizzes that occur on that day (your grade will be computed as though that quiz did not occur).

Sometimes things come up that prevent you from completing an assignment well or at all. To handle this, your homework assignment with the lowest score will be dropped. To avoid encouraging skipping the final assignment, if you perform consistently on all assignments without any clearly low outliers, Prof. Thomas will consider this when assigning grades (it may bump you up if you’re near a boundary).

1.11 Disability Services

If you have a disability and require accommodations, please let me know as soon as possible. You will need to register with Disability Services (161 Whitmore Administration Building; phone (413) 545–0892). Information on services and materials for registering are also available on their website: www.umass.edu/disability.

1.12 Cheating

Cheating will not be tolerated. Each assignment includes instructions about what forms of collaboration are allowed. Copying answers or code from online sources or from solutions to assignments from previous years is always considered cheating. All instances of cheating will be reported to the university’s Academic Honesty Board, and will result in a failing grade letter grade for the course.

1.13 \LaTeX

Your homework submissions must be typed using \LaTeX. If you have not used \LaTeX before, you may want to complete an online tutorial now. Also, the instructor and TAs are prepared to help you learn about \LaTeX during their office hours. Note: The formatting of math using editors like Microsoft Word
is not as clear as \LaTeX. Assignments created using other editors will not be accepted.
2 Introduction

2.1 Notation

When possible, sets will be denoted by calligraphic capital letters (e.g., $\mathcal{X}$), elements of sets by lowercase letters (e.g., $x \in \mathcal{X}$), random variables by capital letters (e.g., $X$), and functions by lowercase letters (e.g., $f$). This will not always be possible, so keep an eye out for exceptions (e.g., later $P$ will be a function).

We write $f : \mathcal{X} \to \mathcal{Y}$ to denote that $f$ is a function with domain $\mathcal{X}$ and range $\mathcal{Y}$. That is, it takes as input an element of the set $\mathcal{X}$ and produces as output an element of $\mathcal{Y}$. We write $|\mathcal{X}|$ to denote the cardinality of the set $\mathcal{X}$—the number of elements in $\mathcal{X}$, and $|x|$ to denote the absolute value of $x$ (thus the meaning of $|\cdot|$ depends on context).

We typically use capital letters for matrices (e.g., $A$) and lowercase letters for vectors (e.g., $b$). We write $A^\top$ to denote the transpose of $A$. Vectors are assumed to be column vectors. Unless otherwise specified, $\|b\|$ denotes the $l^2$-norm (Euclidean norm) of the vector $v$.

We write $\mathbb{N}_{>0}$ to denote the natural numbers not including zero, and $\mathbb{N}_{\geq 0}$ to denote the natural numbers including zero.

We write $:= \triangleq$ to denote is defined to be. In lecture we may write $\triangleleft$ rather than $\triangleq$ since the triangle is easier to see when reading handwriting from the back of the room.

If $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ for any sets $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$, then we write $f(\cdot, y)$ to denote a function, $g : \mathcal{X} \to \mathcal{Z}$, such that $g(x) = f(x, y)$ for all $x \in \mathcal{X}$.

We denote sets using brackets, e.g., $\{1, 2, 3\}$, and sequences and tuples using parentheses, e.g., $(x_1, x_2, \ldots)$.

The notation that we use is not the same as that of the book or other sources (papers and books often use different notations, and there is no agreed-upon standard). Our notation is a mix between the notations of the first and second editions of Sutton and Barto’s book.

2.2 What is Reinforcement Learning (RL)?

Reinforcement learning is an area of machine learning, inspired by behaviorist psychology, concerned with how an agent can learn from interactions with an environment.
Agent: Child, dog, robot, program, etc.

Environment: World, lab, software environment, etc.

Evaluative Feedback: Rewards convey how “good” an agent’s actions are, not what the best actions would have been. If the agent was given instructive feedback (what action it should have taken) this would be a supervised learning problem, not a reinforcement learning problem.

Sequential: The entire sequence of actions must be optimized to maximize the “total” reward the agent obtains. This might require forgoing immediate rewards to obtain larger rewards later. Also, the way that the agent makes decisions (selects actions) changes the distribution of states that it sees. This means that RL problems aren’t provided as fixed data sets like in supervised learning, but instead as code or descriptions of the entire environment.

**Question 1.** If the agent-environment diagram describes a child learning to walk, what exactly is the “Agent” block? Is it the child’s brain, and its body is part of the environment? Is the agent the entire physical child? If the diagram describes a robot, are its sensors part of the environment or the agent?

Neuroscience and psychology ask how animals learn. It is the study of some examples of learning and intelligence. Reinforcement learning asks how we can make an agent that learns. It is the study of learning and intelligence in general (animal, computer, match-boxes, purely theoretical, etc.). In this course we may discuss the relationship between RL and computational neuroscience in one lecture, but in general will not concern ourselves with how animals learn (other than, perhaps, for intuition and motivation).

There are many other fields that are similar and related to RL. Separate research fields often do not communicate much, resulting in different language and approaches. Other notable fields related to RL include operations research.
and control (classical, adaptive, etc.). Although these fields are similar to RL, there are often subtle but impactful differences between the problems studied in these other fields and in RL. Examples include whether the dynamics of the environment are known to the agent a priori (they are not in RL), and whether the dynamics of the environment will be estimated by the agent (many, but not all, RL agents do not directly estimate the dynamics of the environment). There are also many less-impactful differences, like differences in notation (in control, the environment is called the plant, the agent the controller, the reward the (negative) cost, the state the feedback, etc.).

A common misconception is that RL is an alternative to supervised learning—that one might take a supervised learning problem and convert it into an RL problem in order to apply sophisticated RL methods. For example, one might treat the state as the input to a classifier, the action as a label, and the reward as $-1$ if the label is correct and 1 otherwise. Although this is technically possible and a valid use of RL, it should not be done. In a sense, RL should be a last resort—the tool that you use when supervised learning algorithms cannot solve the problem you are interested in. If you have labels for your data, do not discard them and convert the feedback from instructive feedback (telling the agent what label it should have given) to evaluative feedback (telling the agent if it was right or wrong). The RL methods will likely be far worse than standard supervised learning algorithms. However, if you have a sequential problem or a problem where only evaluative feedback is available (or both!), then you cannot apply supervised learning methods and you should use RL.

**Question 2. [Puzzle]** There are 100 pirates. They have 10,000 gold pieces. These pirates are ranked from most fearsome (1) to least fearsome (100). To divide the gold, the most fearsome pirate comes up with a method (e.g., split it evenly, or I get half and the second most fearsome gets the other half). The pirates then vote on this plan. If 50% or more vote in favor of the plan, then that is how the gold is divided. If > 50% vote against the plan, the most fearsome pirate is thrown off the boat and the next most fearsome comes up with a plan, etc. The pirates are perfectly rational. You are the most fearsome pirate. How much of the gold can you get? How?

**Answer 2.** You should be able to keep 9,951 pieces of gold.

If you solved the above puzzle, you very likely did so by first solving easier versions. What if there were only two pirates? What if there were three? This is what we will do in this course. We will study and understand an easier version of the problem and then will build up to more complex and interesting cases over the semester.
### 2.3 687-Gridworld: A Simple Environment

<table>
<thead>
<tr>
<th>State</th>
<th>State 2</th>
<th>State 3</th>
<th>State 4</th>
<th>State 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>State 6</td>
<td>State 8</td>
<td>State 9</td>
<td>State 10</td>
</tr>
<tr>
<td>State 11</td>
<td>State 12</td>
<td>Obstacle</td>
<td>State 13</td>
<td>State 14</td>
</tr>
<tr>
<td>State 15</td>
<td>State 16</td>
<td>Obstacle</td>
<td>State 17</td>
<td>State 18</td>
</tr>
<tr>
<td>State 19</td>
<td>State 20</td>
<td>State 22</td>
<td>Goal</td>
<td>State 23</td>
</tr>
</tbody>
</table>

Figure 2: 687-Gridworld, a simple example environment we will reference often.

**State:** Position of robot. The robot does not have a direction that it is facing.

**Actions:** Attempt Up, Attempt Down, Attempt Left, Attempt Right. We abbreviate these as: AU, AD, AL, AR.

**Environment Dynamics:** With probability 0.8 the robot moves in the specified direction. With probability 0.05 it gets confused and veers to the right—moves $+90^\circ$ from where it attempted to move (that is, AU results in the robot moving right, AL results in the robot moving up, etc.). With probability 0.05 it gets confused and veers to the left—moves $-90^\circ$ from where it attempted to move (that is, AU results in the robot moving left, AL results in the robot moving down, etc.). With probability 0.1 the robot temporarily breaks and does not move at all. If the movement defined by these dynamics would cause the agent to exit the grid (e.g., move up from state 2) or hit an obstacle (e.g., move right from state 12), then the agent does not move. The robot starts in state 1, and the process ends when the robot reaches state 23.

**Rewards:** The agent receives a reward of $-10$ for entering the state with the water and a reward of $+10$ for entering the goal state. Entering any other state results in a reward of zero. If the agent is in the state with the water (state 21) and stays in state 21 for any reason (hitting a wall, temporarily breaking), it counts as “entering” the water state again and results in an additional reward of $-10$. We use a reward discount parameter (the purpose of which is described later) of $\gamma = 0.9$. 


2.4 Describing the Agent and Environment Mathematically

In order to reason about learning, we will describe the environment (and soon the agent) using math. Of the many different mathematical models that can be used to describe the environment (POMDPs, DEC-POMDPs, SMDPs, etc.), we will initially focus on Markov decision processes (MDPs). Despite their apparent simplicity, we will see that they capture a wide range of real and interesting problems, including problems that might at first appear to be outside their scope (e.g., problems where the agent makes observations about the state using sensors that might be incomplete and noisy descriptions of the state). Also, a common misconception is that RL is only about MDPs. This is not the case: MDPs are just one way of formalizing the environment of an RL problem.

- An MDP is a mathematical specification of both the environment and what we want the agent to learn.
- Let \( t \in \mathbb{N}_{\geq 0} \) be the time step (iteration of the agent-environment loop).
- Let \( S_t \) be the state of the environment at time \( t \).
- Let \( A_t \) be the action taken by the agent at time \( t \).
- Let \( R_t \in \mathbb{R} \) be the reward received by the agent at time \( t \). That is, when the state of the environment is \( S_t \), the agent takes action \( A_t \), and the environment transitions to state \( S_{t+1} \), the agent receives the reward \( R_t \). This differs from some other sources wherein this reward is called \( R_{t+1} \).

Formally, a finite MDP is a tuple, \( (\mathcal{S}, \mathcal{A}, P, d_0, \gamma) \), where:

- \( \mathcal{S} \) is the set of all possible states of the environment. The state at time \( t \), \( S_t \), always takes values in \( \mathcal{S} \). For now we will assume that \( |\mathcal{S}| < \infty \)—that the set of states is finite.
- \( \mathcal{A} \) is the set of all possible actions the agent can take. The action at time \( t \), \( A_t \), always takes values in \( \mathcal{A} \). For now we will assume that \( |\mathcal{A}| < \infty \).
- \( P \) is called the transition function, and it describes how the state of the environment changes.

\[
P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1].
\]

For all \( s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{S}, \) and \( t \in \mathbb{N}_{\geq 0} \):

\[
P(s, a, s') := \Pr(S_{t+1} = s'|S_t = s, A_t = a).
\]

Hereafter we suppress the sets when writing quantifiers (like \( \exists \) and \( \forall \))—these should be clear from context. We say that the transition function is deterministic if \( P(s, a, s') \in \{0, 1\} \) for all \( s, a, \) and \( s' \). Recall that we will use lower-case letters to denote functions when possible—notice that \( P \) is an exception to this rule (due to historical usage and to avoid using \( p \), a commonly used symbol otherwise).
• \(d_R\) describes how rewards are generated. Intuitively, it is a conditional distribution over \(R_t\) given \(S_t, A_t\), and \(S_{t+1}\). For now we assume that the rewards are bounded—that \(|R_t| \leq R_{\text{max}}\) always, for all \(t \in \mathbb{N}_{\geq 0}\) and some constant \(R_{\text{max}} \in \mathbb{R}\).\(^1\)

• \(R\) is a function called the reward function, which is implicitly defined by \(d_R\). Other sources often define an MDP to contain \(R\) rather than \(d_R\). Formally

\[
R : \mathcal{S} \times \mathcal{A} \to \mathbb{R},
\]

and

\[
R(s, a) := \mathbb{E}[R_t | S_t = s, A_t = a],
\]

for all \(s, a,\) and \(t\). Although the reward function, \(R\), does not precisely define how the rewards, \(R_t\), are generated (and thus a definition of an MDP with \(R\) in place of \(d_R\) would in a way be incomplete), it is often all that is necessary to reason about how an agent should act. Like \(P\), notice that \(R\) is a function despite being a capital letter. This is also due to a long history of this notation, and also because we will use \(r\) to denote a particular reward, e.g., when writing \((s, a, r, s', a')\) later.

• \(d_0\) is the initial state distribution:

\[
d_0 : \mathcal{S} \to [0, 1],
\]

and for all \(s\):

\[
d_0(s) = \Pr(S_0 = s).
\]

• \(\gamma \in [0, 1]\) is a parameter called the reward discount parameter, and which we discuss later.

---

End of Lecture 2, September 5, 2019

Just as we have defined the environment mathematically, we now define the agent mathematically. A policy is a decision rule—a way that the agent can select actions. Formally, a policy, \(\pi\), is a function:

\[
\pi : \mathcal{S} \times \mathcal{A} \to [0, 1],
\]

and for all \(s \in \mathcal{S}, a \in \mathcal{A},\) and \(t \in \mathbb{N}_{\geq 0}\),

\[
\pi(s, a) := \Pr(A_t = a | S_t = s).
\]

Thus, a policy is the conditional distribution over actions given the state. That is, \(\pi\) is not a distribution, but a collection of distributions over the action set—one per state. There are an infinite number of possible policies, but a finite number of deterministic policies (policies for which \(\pi(s, a) \in \{0, 1\}\) for all \(s\) and \(a\)). We
<table>
<thead>
<tr>
<th>AU</th>
<th>AD</th>
<th>AL</th>
<th>AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>5</td>
<td>0.25</td>
<td>0.25</td>
<td>0.5</td>
</tr>
<tr>
<td>6</td>
<td>0.2</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Example of a tabular policy. Each cell denotes the probability of the action (specified by the column) in each state (specified by the row). In this format, $\Pi$ is the set of all $|S| \times |A|$ matrices with non-negative entries and rows that all sum to one.

denote the set of all policies by $\Pi$. Figure 3 presents an example of a policy for 687-Gridworld.

To summarize so far, the interaction between the agent and environment proceeds as follows (where $R_t \sim d_R(S_t, A_t, S_{t+1}, \cdot)$ denotes that $R_t$ is sampled according to $d_R$):

\[
\begin{align*}
S_0 & \sim d_0 & (9) \\
A_0 & \sim \pi(S_0, \cdot) & (10) \\
S_1 & \sim P(S_0, A_0, \cdot) & (11) \\
R_0 & \sim d_R(S_0, A_0, S_{t+1}, \cdot) & (12) \\
A_1 & \sim \pi(S_1, \cdot) & (13) \\
S_2 & \sim P(S_1, A_1, \cdot) & (14) \\
\ldots & & (15)
\end{align*}
\]

In pseudocode:

<table>
<thead>
<tr>
<th>Algorithm 1: General flow of agent-environment interaction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  \ $S_0 \sim d_0$;</td>
</tr>
<tr>
<td>2  \ for $t = 0$ to $\infty$ do</td>
</tr>
<tr>
<td>3   \ $A_t \sim \pi(S_t, \cdot)$;</td>
</tr>
<tr>
<td>4   \ $S_{t+1} \sim P(S_t, A_t, \cdot)$;</td>
</tr>
<tr>
<td>5   \ $R_t \sim d_R(S_t, A_t, S_{t+1}, \cdot)$;</td>
</tr>
</tbody>
</table>

The running of an MDP is also presented as a Bayesian network in Figure 4.

Notice that we have defined rewards so that $R_0$ is the first reward, while Sutton and Barto (1998) define rewards such that $R_1$ is the first reward. We do this because $S_0$, $A_0$, and $t = 0$ are the first state, action, and time, and so

\footnote{Sometimes $d_R$ will place probabilities on a small number of rewards (often the reward may be a deterministic function of $S_t$, $A_t$, and $S_{t+1}$). Sometimes $d_R$ will characterize a continuous distribution. Defining $d_R$ properly therefore requires the use of measure theory for probability. To keep things simple, we will not do this—we will not define $d_R$ more formally.}
having $R_1$ be the first reward would be inconsistent. Furthermore, this causes indices to align better later on. However, when comparing notes from the course to the book, be sure to account for this notational discrepancy.

**Agent’s goal**: Find a policy, $\pi^*$, called an **optimal policy**. Intuitively, an optimal policy maximizes the expected total amount of reward that the agent will obtain.

**Objective function**: $J : \Pi \to \mathbb{R}$, where for all $\pi \in \Pi$,

$$J(\pi) := \mathbb{E} \left[ \sum_{t=0}^{\infty} R_t \bigg| \pi \right]. \quad (16)$$

*Note*: Later we will revise this definition—if you are skimming looking for the correct definition of $J$, it is in (18).

*Note*: Expectations and probabilities can be conditioned on events. A policy, $\pi$, is not an event. Conditioning on $\pi$, e.g., when we wrote $|\pi$ in the definition of $J$ above, denotes that all actions (the distributions or values of which are not otherwise explicitly specified) are sampled according to $\pi$. That is, for all $t \in \mathbb{N}_{\geq 0}$, $A_t \sim \pi(S_t, \cdot)$.

**Optimal Policy**: An optimal policy, $\pi^*$, is any policy that satisfies:

$$\pi^* \in \arg \max_{\pi \in \Pi} J(\pi). \quad (17)$$

*Note*: Much later we will define an optimal policy in a different and more strict way.

**Property 1** (Existence of an optimal policy). If $|S| < \infty$, $|A| < \infty$, $R_{\text{max}} < \infty$, and $\gamma < 1$, then an optimal policy exists.\(^2\)

We will prove Property 1 later.

\(^2\)The restriction of $\gamma$ is not necessary, but is present because our proof of this property will rely on the assumption that $\gamma < 1$. 

---

Figure 4: Bayesian network depicted the running of an MDP.
Reward Discounting: If you could have one cookie today or two cookies on the last day of class, which would you pick? Many people pick one cookie today when actually presented with these options. This suggests that rewards that are obtained in the distant future are worth less to us than rewards in the near future. The reward discount parameter, $\gamma$, allows us to encode, within the objective function, this discounting of rewards based on how distant in the future they occur.

Recall that $\gamma \in [0,1]$. We redefine the objective function, $J$, as:

$$J(\pi) := \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_t \Bigg| \pi \right], \quad (18)$$

for all $\pi \in \Pi$. So, $\gamma < 1$ means that rewards that occur later are worth less to the agent—the utility of a reward, $r$, $t$ time steps in the future is $\gamma^t r$. Including $\gamma$ also ensures that $J(\pi)$ is bounded, and later we will see that smaller values of $\gamma$ make the MDP easier to solve (solving an MDP refers to finding or approximating an optimal policy).

To summarize, the agent’s goal is to find (or approximate) an optimal policy, $\pi^*$, as defined in (17), using the definition of $J$ that includes reward discounting—(18).

**Question 4.** What is an optimal policy for 687-Gridworld? Is it unique? How does the optimal action in state 20 change if we were to change the value of $\gamma$?

**Question 5.** Consider two MDPs that are identical, except for their initial state distributions, $d_0$. Let $\pi^*$ and $\mu^*$ be optimal policies for the first and second MDP, respectively. Let $s^* \in S$ be a state that has a non-zero probability of occurring when using $\pi^*$ on the first MDP and a non-zero probability of occurring when using $\mu^*$ on the second MDP. Consider a new policy, $\pi'$ such that $\pi'(s,a) = \pi^*(s,a)$ for all $s \in S \setminus \{s^*\}$ and $a \in A$ and $\pi'(s^*,a) = \mu^*(s^*,a)$ for all $a \in A$. Is $\pi'$ an optimal policy for the first MDP?
Answer 5.

Yes! Later we will have the mathematical tools to discuss this more formally. For now, notice that how the agent entered state $s^{\star}$ does not impact what action it should take in state $s^{\star}$ to get as much reward as possible in the future. In this way, optimal behavior in a state is independent of how the agent got to the state, and thus independent of the initial state distribution, $d_0$. And thus independent of the initial state distribution, $d_0$, if every state is reachable under every policy. Later we will present a different definition of an optimal policy that is completely independent of $d_0$ without this reachability condition.

When we introduced 687-Gridworld, we said that the agent-environment interactions terminate when the agent reaches state 23, which we called the goal. This notion of a terminal state can be encoded using our definition of an MDP above. Specifically, we define a terminal state to be any state that always transitions to a special state, $s_\infty$, called the terminal absorbing state. Once in $s_\infty$, the agent can never leave ($s_\infty$ is absorbing)—the agent will forever continue to transition from $s_\infty$ back into $s_\infty$. Transitioning from $s_\infty$ to $s_\infty$ always results in a reward of zero. Effectively, when the agent enters a terminal state the process ends. There are no more decisions to make (since all actions have the same outcome) or rewards to collect. Thus, an episode terminates when the agent enters $s_\infty$. Notice that terminal states are optional—MDPs need not have any terminal states. Also, there may be states that only sometimes transition to $s_\infty$, and we do not call these terminal states. Notice also that $s_\infty$ is an element of $S$. Lastly, although terminal states are defined, goal states are not defined—the notion of a goal in 687-Gridworld is simply for our own intuition.

When the agent reaches $s_\infty$, the current trial, called an episode ends and a new one begins. This means that $t$ is reset to zero, the initial state, $S_0$, is sampled from $d_0$, and the next episode begins (the agent selects $A_0$, gets reward $R_0$, and transitions to state $S_1$). The agent is notified that this has occurred, since this reset may change its behavior (e.g., it might clear some sort of short-term memory).

End of Lecture 3, September 10, 2019
Instructions: This homework assignment consists of a written portion and a programming portion. While you may discuss problems with your peers (e.g., to discuss high-level approaches), you must answer the questions on your own. Submissions must be typed (hand written and scanned submissions will not be accepted). You must use \LaTeX. The assignment should be submitted on Gradescope as PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code. You must use Python 3 for your homework code. You may not use any reinforcement learning or machine learning specific libraries in your code, e.g., TensorFlow or PyTorch (you may use libraries like numpy and matplotlib though). The automated system will not accept assignments after 11:55pm on September 19. The tex file for this homework can be found here.

Part One: Written (62 Points Total)

1. (Your grade will be a zero on this assignment if this question is not answered correctly) Read the class syllabus carefully, including the academic honesty policy. To affirm that you have read the syllabus, type your name as the answer to this problem.

2. (15 Points) Given an MDP \( M = (S, A, P, d_R, d_0, \gamma) \) and a fixed policy, \( \pi \), the probability that the action at time \( t = 0 \) is \( a \in A \) is:

\[
\text{Pr}(A_0 = a) = \sum_{s \in S} d_0(s) \pi(s, a).
\]  

Write similar expressions (using only \( S, A, P, R, d_0, \gamma, \) and \( \pi \)) for the following problems.

Hints and Probability Review:

- **Write Probabilities of Events**: In some of the probability hints below that are not specific to RL, we use expressions like \( \text{Pr}(a|b) \), where \( a \) and \( b \) are events. Remember that in the RL notation used for this class, the values of \( \text{Pr}(s_0), \text{Pr}(a_0), \text{Pr}(A_0), \) or \( \text{Pr}(A_0|S_0) \) are all undefined, since those are simply states, actions, or random variables (not events). Instead, we must write about the probabilities of events. For example: \( \text{Pr}(A_0 = a_0) \) or \( \text{Pr}(A_0 = a_0|S_0 = s_0) \).

- **Bayes’ Theorem**: \( \text{Pr}(a|b) = \frac{\text{Pr}(b|a) \text{Pr}(a)}{\text{Pr}(b)} \). This is useful for dealing with conditional probabilities \( \text{Pr}(a|b) \), where event \( a \) occurs before event \( b \). For example, it is often difficult to work with an expression like \( \text{Pr}(S_0 = s_0, A_0 = a_0) \), but much easier to deal with the 3 terms in \( \frac{\text{Pr}(A_0 = a_0|S_0 = s_0) \text{Pr}(S_0 = s_0)}{\text{Pr}(A_0 = a_0)} \).
• **The law of total probability:** For event \( a \), and a set of events \( B \),

\[
\Pr(a) = \sum_{b \in B} \Pr(b) \Pr(a|b)
\]

See the example below for several useful applications of this property.

• **“Extra” given terms:** Remember that when applying laws of probability, any “extra” given terms stay in the result. For example, applying the law of total probability:

\[
\Pr(a|c, d) = \sum_{b \in B} \Pr(b|c, d) \Pr(a|b, c, d)
\]

• **Example problem:** The probability that the state at time \( t = 1 \) is \( s \in S \).

\[
\Pr(S_1 = s) = \sum_{s_0 \in S} \Pr(S_0 = s_0) \Pr(S_1 = s|S_0 = s_0) \tag{20}
= \sum_{s_0 \in S} d_0(s_0) \Pr(S_1 = s|S_0 = s_0) \tag{21}
= \sum_{s_0 \in S} d_0(s_0) \sum_{a_0 \in A} \Pr(A_0 = a_0|S_0 = s_0)
\times \Pr(S_1 = s|S_0 = s_0, A_0 = a_0) \tag{22}
= \sum_{s_0 \in S} d_0(s_0) \sum_{a_0 \in A} \pi(s_0, a_0) P(s_0, a_0, s). \tag{23}
\]

Problems:

A The probability that the action at time \( t = 3 \) is either \( a \in A \) or \( a' \in A \), with \( a \neq a' \).
\[ \Pr(A_3 = a \cup A_3 = a') = \sum_{s_3} \Pr(A_3 = a \cup A_3 = a'|S_3 = s_3) \Pr(S_3 = s_3) \]  

(25)

\[ = \sum_{s_3} \Pr(S_3 = s_3)(\pi(s_3, a) + \pi(s_3, a')) \]  

(26)

\[ = \sum_{s_2, a_2} \Pr(S_2 = s_2) \Pr(A_2 = a_2|S_2 = s_2) \times \sum_{s_1} \Pr(S_1 = s_1|S_2 = s_2, A_2 = a_2)(\pi(s_3, a) + \pi(s_3, a')) \]  

(27)

\[ = \sum_{s_2} \Pr(S_2 = s_2) \sum_{a_2} \pi(s_2, a_2) \times \sum_{s_3} P(s_2, a_2, s_3)(\pi(s_3, a) + \pi(s_3, a')) \]  

(28)

\[ = \sum_{s_0} \Pr(S_0 = s_0) \sum_{a_0} \Pr(A_0 = a_0|S_0 = s_0) \sum_{s_1} \Pr(S_1 = s_1|S_0 = s_0, A_0 = a_0) \times \sum_{a_1} \Pr(A_1 = a_1|S_1 = s_1) \sum_{s_2} \Pr(S_2 = s_2|S_1 = s_1, A_1 = a_1) \]  

(30)

\[ \times \sum_{a_2} \pi(s_2, a_2) \sum_{s_3} P(s_2, a_2, s_3)(\pi(s_3, a) + \pi(s_3, a')) \]  

(31)

\[ = \sum_{s_0} d_0(s_0) \sum_{a_0} \pi(s_0, a_0) \sum_{s_1} P(s_0, a_0, s_1) \times \sum_{a_1} \pi(s_1, a_1) \sum_{s_2} P(s_1, a_1, s_2) \times \sum_{a_2} \pi(s_2, a_2) \sum_{s_3} P(s_2, a_2, s_3)(\pi(s_3, a) + \pi(s_3, a')) \]  

(32)

(33)

(34)

(35)

B The expected reward at time \( t = 6 \) given that the action at time \( t = 5 \)
is \( a \in \mathcal{A} \) and the state at time \( t = 4 \) is \( s \in \mathcal{S} \).

\[
E[R_6|A_5 = a, S_4 = s] = \sum_{s_6, a_6} R(s_6, a_6) \Pr(S_6 = s_6, A_6 = a_6|S_4 = s, A_5 = a)
\]

\[
= \sum_{s_6, a_6} R(s_6, a_6) \frac{\Pr(S_6 = s_6, A_6 = a_6, A_5 = a|S_4 = s)}{\Pr(A_5 = a|S_4 = s)} 
\]  

\[
(36)
\]

\[
= \sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a) \sum_{s_6, a_6} R(s_6, a_6) \frac{P(s_5, a_4, s_5) \pi(s_5, a_6)}{\Pr(A_5 = a|S_4 = s)} 
\]

\[
(37)
\]

\[
= \sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a) \sum_{s_6, a_6} R(s_6, a_6) \sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a) 
\]

\[
(38)
\]

\[
= \frac{\sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a) \sum_{s_6, a_6} P(s_5, a_4, s_5) \pi(s_5, a_6) R(s_6, a_6)}{\sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a)} 
\]

\[
(39)
\]

\[
= \sum_{a_4} \pi(s, a_4) \sum_{s_5} P(s, a_4, s_5) \pi(s_5, a) \sum_{s_6, a_6} P(s_5, a_4, s_5) \pi(s_5, a_6) R(s_6, a_6) 
\]

\[
(40)
\]

C The probability that the action at time \( t = 14 \) is \( a' \in \mathcal{A} \) given that

\[
\Pr(A_{16} = a'|S_{15} = s, A_{14} = a) = \Pr(A_{16} = a'|S_{15} = s)
\]

\[
= \sum_{a_{15}} \Pr(A_{15} = a_{15}|S_{15} = s) \sum_{s_{16}} \Pr(S_{16} = s_{16}|S_{15} = s, A_{15} = a_{15}) \Pr(A_{16} = a'|S_{16} = s_{16})
\]

\[
(41)
\]

\[
= \sum_{a_{15}} \pi(s, a_{15}) \sum_{s_{16}} P(s, a_{15}, s_{16}) \pi(s_{16}, a')
\]

\[
(42)
\]

D The probability that the initial state was \( s \in \mathcal{S} \) given that the action

\[
\Pr(S_0 = s|A_1 = a') = \frac{\Pr(S_0 = s, A_1 = a')}{\Pr(A_1 = a')}
\]

\[
= \frac{\sum_{a_0} \Pr(A_0 = a_0|S_0 = s) \sum_{s_1} \Pr(S_1 = s_1|S_0 = s, A_0 = a_0)}{\Pr(A_1 = a')}
\]

\[
= \frac{d_0(s) \sum_{a_0} \pi(s, a_0) \sum_{s_1} P(s, a_0, s_1) \pi(s_1, a')}{\sum_{a_0} d_0(s_0) \sum_{a_0} \pi(s_0, a_0) \sum_{s_1} P(s_0, a_0, s_1) \pi(s_1, a')}
\]

\[
(43)
\]

\[
= \frac{d_0(s) \sum_{a_0} \pi(s, a_0) \sum_{s_1} P(s, a_0, s_1) \pi(s_1, a')}{\sum_{a_0} d_0(s_0) \sum_{a_0} \pi(s_0, a_0) \sum_{s_1} P(s_0, a_0, s_1) \pi(s_1, a')}
\]

\[
(44)
\]

E This question has been removed. You do not need to submit an answer.
3. (3 Points) In 687-Gridworld, if we changed how rewards are generated so that hitting a wall (i.e., when the agent would enter an obstacle state, and is placed back where it started) results in a reward of \(-10\), then what is \(E[R_t|S_t = 17, A_t = AL, S_{t+1} = 17]\)?

Consider the event HitWall, which occurs if the agent hits the obstacle to the left of state 17, and does not occur if the agent does not hit the obstacle to the left of state 17. For brevity, we write 17AL17 to denote \(S_t = 17, A_t = AL, S_{t+1} = 17\).

\[
E[R_t|S_t = 17, A_t = AL, S_{t+1} = 17] = Pr(\text{HitWall}|17AL17)E[R_t|17AL17, \text{HitWall}] + Pr(\neg\text{HitWall}|17AL17)E[R_t|17AL17, \neg\text{HitWall}]
\]

\[
= Pr(\text{HitWall}|17AL17) \times -10 + Pr(\neg\text{HitWall}|17AL17) \times 0
\]

\[
= -10 Pr(\text{HitWall}|17AL17)
\]

To conclude, we must compute the probability that the agent hit the wall if it started in state 17, took action AL, and ended in state 17.

\[
Pr(\text{HitWall}|17AL17) = \frac{Pr(\text{HitWall}, S_{t+1} = 17|S_t = 17, A_t = AL)}{Pr(S_{t+1} = 17|S_t = 17, A_t = AL)}
\]

\[
= \frac{Pr(\text{HitWall}|S_t = 17, A_t = AL)}{Pr(S_{t+1} = 17|S_t = 17, A_t = AL)}
\]

\[
= \frac{0.8}{0.8 + 0.1}
\]

\[
= \frac{0.8}{0.9}
\]

So,

\[
E[R_t|S_t = 17, A_t = AL, S_{t+1} = 17] = -10 \frac{0.8}{0.9}
\]

4. (2 Points) How many deterministic policies are there for an MDP with \(|S| < \infty\) and \(|A| < \infty\)? (You may write your answer in terms of \(|S|\) and \(|A|\)).

\(|A|^{|S|}\)

5. (2 Points) Give an example of an MDP with \(|S| < \infty, |A| = \infty, \text{ and } \gamma < 1\) such that an optimal policy does not exist. Give an example of an MDP with \(|S| = \infty, |A| < \infty, \text{ and } \gamma < 1\) such that an optimal policy exists.

Student answers may vary. Example of nonoptimal: An MDP with one state that always transitions to \(s_\infty, A = [0, 1]\), and \(R_t = A_t\). Since the action set is open, the maximum element in \(A\) does not exist, and so no optimal policy exists. Example of optimal: mountain car domain.
6. (3 Points) Read about the Pendulum domain, described in Section 5.1 of this paper (Reinforcement Learning in Continuous Time and Space by Kenji Doya). Consider a variant where the initial state has the pendulum hanging down with zero angular velocity always (a deterministic initial state where the pendulum is hanging straight down with no velocity) and a variant where the initial angle is chosen uniformly randomly in \([-\pi, \pi]\) and the initial velocity is zero. Which variant do you expect an agent to require more episodes to solve? Why? Note: We did not talk about the complexity of solving MDPs in class yet—we want you to provide your best guess here.

Any reasonable answer is ok, but it must have an explanation that makes some sense.

7. (1 Point) How many episodes do you expect an agent should need in order to find near-optimal policies for the gridworld and pendulum domains? Note: We did not talk about the complexity of solving MDPs in class yet—we want you to provide your best guess here.

Any reasonable answer is ok, but it must have an explanation that makes some sense.

8. (5 Points) Select a problem that we have not talked about in class, where the agent does not make Markovian observations about the world around it. Describe how the environment for this problem can be formulated as an MDP by specifying \((S, A, P, \{d, r\} or R), d_0, \gamma)\) (your specifications of these terms may use English rather than math, but be precise).

Answers may vary. The key property should be that the states are Markovian and correspond to states, not observations.

9. (5 Points) We refer to the discounted sum of rewards, \(\sum_{t=0}^{\infty} \gamma^t R_t\), as the return. Let an MDP exist such that it has two optimal policies. Can the expected value of their returns differ? If so, give an example. If not, explain why. Can the variance of their returns differ? If so, give an example. If not, explain why.

By the definition of an optimal policy described in class, the expected value of the returns of the policies cannot differ. However, the variance of the policies can differ. Answers may vary for the example. One example is an MDP with one state that always transitions to \(s_\infty\) and which has two actions. The first actions results in \(R_t \sim \mathcal{N}(0, 1)\) and the second results in \(R_t \sim \mathcal{N}(0, 2)\).

10. (2 Points) Consider the one state MDP wherein \(s_0\) there are three actions, \(a_0, a_1, a_2\), and all actions transition to \(s_\infty\) with probability 0.5 and stay in \(s_0\) otherwise. The reward for taking actions \(a_0, a_1\) are drawn from the uniform distribution on \([0, 1]\) and the normal distribution \(\mathcal{N}(0.5, 1)\), respectively. The reward for \(a_3\) is always 0.25. What are all the optimal policies of this MDP?

The optimal policies are all policies such that \(\pi(s_0, a_0) + \pi(s_0, a_1) = 1\)
11. (2 Points) Read the Wikipedia page on Markov chains. A state in a Markov chain is irreducible if it is possible to get to any state from any state. An MDP is irreducible if the Markov chain associated with every policy is irreducible. A state in a Markov chain has period k if every return to the state must occur in multiples of k time steps. More formally,

\[ k = \gcd\{t > 0 : \Pr(S_t = s|S_0 = s) > 0\} \]

A Markov chain is aperiodic if the period of every state is \( k = 1 \). An MDP is aperiodic if the Markov chain associated with every deterministic policy is aperiodic (or equivalently, if the Markov chain associated with every policy is aperiodic). Can an MDP be aperiodic and not irreducible? If so, give an example. If not, explain your reasoning.

Yes, an MDP can be aperiodic and not irreducible. Student answers may vary for the example. Imagine an MDP with two states that always transition to themselves.

12. (5 Points) The state of a Markov chain is positive recurrent if the expected time until the state recurs is finite. A Markov chain is positive recurrent if all states are positive recurrent. Give an example of an MDP with \(|S| > 2\) states that is positive recurrent and aperiodic. For any number of states \(|S|\), can you think of a simple way of defining state transitions such that the MDP is positive recurrent and aperiodic? Explain your methodology (a picture might be useful).

Student answers may vary. An example is an MDP with states that always transition to themselves will be positive recurrent and aperiodic.

13. (1 point) Let a tabular policy representation be used to represent stochastic policies for an MDP with \( n \) states and \( m \) actions. What is the sum of every element in the matrix representation of this policy? Why?

If we represent a tabular policy as a matrix with rows being the states and columns being the actions, each row should sum to one. If there are \( n \) states, then the sum of each element is \( 1 \times n = n \)

14. (2 Points) Describe a real-world problem and how it can be reasonably modeled as an MDP where \( R_t \) is not a deterministic function of \( S_t, A_t \), and \( S_{t+1} \), and is not a bandit problem.

Student answers may vary. Example: Consider scanning space for extraterrestrial or other interesting radio signals. The state might be a vector representing the current spatial location and frequency the radio is scanning. The action space is the direction to point the radio and what frequency to listen to next. The reward would be +1 for finding an interesting signal and 0 otherwise. The signal may not always be received even when the radio focuses in the right direction/frequency due to noise in the signal, so it is not a deterministic function of the state, action, or next state.
15. (2 Points) If you know \((S, A, P, R, d_0, \gamma)\), can you derive \(d_R\)? Prove your answer is correct.

You cannot derive \(d_R\) given \(R\). To prove this, we provide two MDPs that have the same \(S, A, P, R, d_0, \gamma\), but different \(d_R\). Let \(d_R^1\) be \(U(0,1)\) for all \(s,a,s'\) in the first MDP, and let \(d_R^2\) be \(N(0.5, 1.0)\) for all \(s,a,s'\) in the second MDP. Then for any state \(s\) and action \(a\), \(R^1(s,a) = E[R_t|S_t = s, A_t = a, d_R^1]\) and \(R^2(s,a) = R^2(s,a)\). Therefore, given only \(R\) such that \(R = R^1\) or \(R = R^2\) it is not possible to know if the rewards come from a distribution corresponding to \(d_R^1\) or \(d_R^2\).

16. (2 Points) If you know \((S, A, P, R, d_0, \gamma)\), can you derive \(R\)? Prove your answer is correct.

Yes, you can derive \(R\). If the expectation \(E[R_t|S_t = s, A_t = a, S_{t+1} = s', d_R]\) exists for all \(s,s' \in S\) and \(a \in A\), then \(R\) can be derived.

\[
R(s,a) = E[R_t|S_t = s, A_t = a]
= \sum_{s' \in S} P(s,a,s') E[R_t|S_t = s, A_t = a, S_{t+1} = s']
= \sum_{s' \in S} P(s,a,s') E[R_t|R_t \sim d_R(s,a,s',\cdot)]
\]

where \(d_R(s,a,s',\cdot)\) is the probability distribution of \(R_t\) given \(S_t = s, A_t = a,\) and \(S_{t+1} = s'\).

17. (2 Points) Describe a real-world problem and how it can be reasonably modeled as an MDP where \(R_t\) is a deterministic function of \(S_t\).

Student answers may vary. One example is a robot learning to balance. The reward might be +1 for every time step where the robot has not fallen over (and zero when the robot falls over, entering \(s_\infty\)). Here we could define the reward to be +1 if \(S_t\) is a state that is not considered to be “fallen over”.

18. (2 Points) Describe a real-world problem and how it can be reasonably modeled as an MDP where the reward function, \(R\), would not be known.

Consider the problem of a robot cooking for a human. The reward could be +1 for the human being satisfied, −1 for disgusted, and 0 otherwise. Human preferences are dependent on many non-observable factors and change over time. This means that given \(s,a\) the expectation, \(R(s,a)\), would still be unknown.

19. (2 Points) Describe a real-world problem and how it can be reasonably modeled as an MDP where the transition function, \(P\), would be known.

Student answers may vary. Robotics applications sometimes involve physical systems, the dynamics of which are well-understood. We can view these
as being problems where the transition dynamics are known. Another example is digital marketing, where the state is a description of a person’s interests (perhaps based only on external information, not information gleaned from an episode) along with a count of how many times each advertisement was shown. The action is a choice about which ad to show, and the reward is +1 if the user clicks the ad and zero otherwise. In this case, the transition function is known (the counter for the shown ad will be incremented), but the reward function is unknown.

20. (2 Points) Describe a real-world problem and how it can be reasonably modeled as an MDP where the transition function, \( P \), would not be known. Student answers may vary. Consider a tutoring system, wherein the states correspond to what the student currently understands, and the actions correspond to decisions about what topic to teach next. The transition function for this problem includes human behavior—how a student learns about a topic. Modeling human learning is challenging, and an active research topic. Current models are not reliable enough to be viewed as known transition functions.

Part Two: Programming (25 Points Total)

More-Watery 687-Gridworld. For this assignment, we will be working with a slightly modified version of the 687-Gridworld domain described in class and in the class notes. In this new Gridworld, called More-Watery Gridworld, there are two extra water states located in state 7 and state 17, as shown in Figure 5. Implement More-Watery Gridworld.

Codebase. We have provided a template for programming the homework on the github repository for this class located here. You do not need to use this template for the assignment. After the due date for this assignment, an example will be posted on this site.

A (5 Points) Have the agent uniformly randomly select actions. Run 10,000 episodes. Report the mean, standard deviation, maximum, and minimum of the observed discounted returns.

These values can vary, e.g., the actual maximum value is the maximum return for any policy.

<table>
<thead>
<tr>
<th>policy</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform</td>
<td>-9.2600</td>
<td>7.2887</td>
<td>-45.984</td>
<td>4.3047</td>
</tr>
</tbody>
</table>

B (5 Points) Find an optimal policy (you may do this any way you choose, including by reasoning through the problem yourself). Report the optimal policy here. Comment on whether it is unique.

The optimal policy is not unique. The action in the terminal state, \( s_{23} \), can be anything. If the terminal state is not represented in the policy,
then the optimal policy is unique. However, since this can be challenging to ascertain without the tools that we will teach you later, a reasonable argument for why two different policies would be optimal (particularly in states where the expected return is very similar when the two actions are taken) may be accepted for full credit.

C (5 Points) Run the optimal policy that you found in the previous question for 10,000 episodes. Report the mean, standard deviation, maximum, and minimum of the observed discounted returns.

D (5 Points) The distribution of returns is often not normal, thus it cannot
be fully characterized by its mean and standard deviation. To provide more information about the performance, the empirical distribution of returns can be reported.

For a random variable, $X$, its cumulative distribution function (CDF), $F_X$, is defined as $F_X(x) := \Pr(X \leq x)$. The empirical CDF, $\hat{F}$, for a sequence of $n$ samples of $X$, $X_1, \ldots, X_n$ is given by the function

$$\hat{F}_n(x) := \frac{1}{n} \sum_{i=1}^{n} 1_{X_i \leq x},$$

where $X_i$ is the $i^{th}$ sample of $X$ and $1_A$ is the indicator function of an event $A$, i.e., $1_A = 1$ if $A$ is true and 0 otherwise.

The quantile function, also referred to as the inverse CDF, is the function $Q(\tau) := \inf \{ x \in \mathbb{R} : \tau \leq F_X(x) \}$ for $\tau \in (0, 1)$. The empirical quantile function, $\hat{Q}$, can be constructed by considering the order statistics, $Z_i$, the sorted samples of $X_i$ such that $Z_1 \leq Z_2 \leq \ldots \leq Z_n$. The empirical quantile function is given by

$$\hat{Q}(\tau) := Z_{\lfloor (n+1)\tau \rfloor}.$$

Both the CDF and quantile functions capture all the information about a random variable, but for plotting purposes the quantile function is often preferred. This is because we are interested in maximizing returns, so the quantile function has a more natural interpretation as higher is better.

Plot the distribution of returns for both the random policy and the optimal policy using 10,000 trials each. You must clearly label each line and axis. Additionally, report the random seed used for the experiments.

Random seed used was 4321 and was initialized before all experiments.
E (5 Points) Using simulations, empirically estimate the probability that \( S_{19} = 21 \) given that \( S_8 = 18 \) (the state above the goal) when running the uniform random policy. Describe how you estimated this quantity (there is not a typo in this problem, nor an oversight).

After using 100,000 trials, \( \Pr(S_{19} = s_{21}|S_8 = s_{18}) \approx 0.01858 \) and is in \([0.01776, 0.01944]\) with 95% confidence according to Wilson Score interval. Each trial started with \( S_0 = s_{18} \) and was executed until \( t = 11 \) or the terminal absorbing state was reached.
Instructions: You have 5 minutes to complete this quiz. This quiz is closed notes—do not use your notes or a laptop. Do not discuss problems with your neighbors until after everyone has handed in their quiz.

Use the notation from class. Don’t forget to capitalize your random variables. Presenting equalities that are true is not enough—you must provide the definitions of the symbols on the left.

Fill in the definitions for the following terms (your answer to 5 should be a real number, not math expressions):

[Answers in blue.]

1. \( P(s, a, s') = \Pr(S_{t+1} = s'|S_t = s, A_t = a) \)

2. \( R(s, a) = \mathbf{E}[R_t|S_t = s, A_t = a] \)

3. \( \pi(s, a) = \Pr(A_t = a|S_t = s) \)

4. \( J(\pi) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t|\pi\right] \)

5. \( \min_{a \in A} R(s_{\infty}, a) = 0 \)
Consider again the definition of reinforcement learning. Notice the segment “learn from interactions with the environment.” If \( P \) and \( R \) (or \( d_R \)) are known, then the agent does not need to interact with the environment. E.g., an agent solving 687-Gridworld can plan in its head, work out an optimal policy and execute this optimal policy from this start. This is not reinforcement learning—this is planning. More concretely, in planning problems \( P \) and \( R \) are known, while in reinforcement learning problems at least \( P \) (and usually \( R \)) is not known by the agent. Instead, the agent must learn by interacting with the environment—taking different actions and seeing what happens. Most reinforcement learning algorithms will not estimate \( P \). The environment is often too complex to model well, and small errors in an estimate of \( P \) compound over multiple time steps making plans built from estimates of \( P \) unreliable. We will discuss this more later.

2.5 Additional Terminology, Notation, and Assumptions

- A **history**, \( H_t \), is a recording of what has happened up to time \( t \) in an episode:
  \[
  H_t := (S_0, A_0, R_0, S_1, A_1, R_1, S_2, \ldots, S_t, A_t, R_t). \tag{62}
  \]

- A **trajectory** is the history of an entire episode: \( H_\infty \).

- The **return** or **discounted return** of a trajectory is the discounted sum of rewards: \( G := \sum_{i=0}^{\infty} \gamma^i R_t \). So, the objective, \( J \), is the expected return or expected discounted return, and can be written as \( J(\pi) := \mathbb{E}[G|\pi] \).

- The **return from time \( t \)** or **discounted return from time \( t \)**, \( G_t \), is the discounted sum of rewards starting from time \( t \):
  \[
  G_t := \sum_{k=0}^{\infty} \gamma^k R_{t+k}.
  \]

2.5.1 Example Domain: Mountain Car

Environments studied in RL are often called **domains**. One of the most common domains is **mountain car**, wherein the agent is driving a crude approximation of a car. The car is stuck in a valley, and the agent wants to get to the top of the hill in front of the car. However, the car does not have enough power to drive straight up the hill in front, and so it must learn to reverse up the hill behind it before accelerating forwards to climb the hill in front. A diagram of the mountain car environment is depicted in Figure 6.

- **State**: \( s = (x, v) \), where \( x \in \mathbb{R} \) is the position of the car and \( v \in \mathbb{R} \) is the velocity.

- **Actions**: \( a \in \{\text{forward, neutral, reverse}\} \). These actions can be renamed to be less unwieldy: \( a \in \{0, 1, 2\} \).
• **Dynamics**: The dynamics are *deterministic*—taking action \(a\) in state \(s\) always produces the same state, \(s'\). Thus, \(P(s, a, s') \in \{0, 1\}\). The dynamics are characterized by:

\[
v_{t+1} = v_t + 0.001a - 0.0025\cos(3x)x_{t+1} = x_t + v_{t+1}. \tag{63}
\]

If these equations would cause \(x_{t+1} < -1.2\) then instead \(x_{t+1} = -1.2\) and the velocity is set to zero: \(v_{t+1} = 0\). Similarly, if these equations would cause \(x_{t+1} > 0.5\), then \(x_{t+1} = 0.5\) and the velocity is set to zero: \(v_{t+1} = 0\). This simulates inelastic collisions with walls at \(-1.2\) and \(0.5\).

• **Terminal States**: If \(x_t = 0.5\), then the state is terminal (it always transitions to \(s_\infty\)).

• **Rewards**: \(R_t = -1\) always, except when transitioning to \(s_\infty\) (from \(s_\infty\) or from a terminal state), in which case \(R_t = 0\).

• **Discount**: \(\gamma = 1.0\).

• **Initial State**: \(S_0 = (-0.5, 0)\) deterministically (i.e., \(\Pr(S_0 = (-0.5, 0)) = 1\)).

**Question 6.** For this problem, what is an English description of the meaning behind a return? What is an episode? What is an optimal policy? How long can an episode be? What is the English meaning of \(J(\pi)\)?
The return is negative the number of time steps for the car to reach the goal. An episode corresponds to the car starting near the bottom of the valley and the agent driving it until it reaches the top of the hill in front of the car. In optimal policy training, the agent drives the car starting near the bottom of the valley and the agent tries to reach the top of the hill in front of the car. An episode corresponds to the car starting near the bottom of the valley and the agent drives the car until it reaches the goal. There is no limit on how long an episode can be.

\( J(\pi) \) is the expected number of time steps for the agent to reach the goal when it uses policy \( \pi \).

2.5.2 Markov Property

A seemingly more general non-Markovian formulation for the transition function might be:

\[
P(h, s, a, s') := \Pr(S_{t+1} = s' | H_{t-1} = h, S_t = s, A_t = a). \tag{64}
\]

The Markov assumption is the assumption that \( S_{t+1} \) is conditionally independent of \( H_{t-1} \) given \( S_t \). That is, for all \( h, s, a, s', t \):

\[
\Pr(S_{t+1} = s' | H_{t-1} = h, S_t = s, A_t = a) = \Pr(S_{t+1} = s' | S_t = s, A_t = a) \tag{65}
\]

Since we make this Markov assumption, \( P \) as defined earlier completely captures the transition dynamics of the environment, and there is no need for the alternate definition in (64). The Markov assumption is sometimes referred to as the Markov property (for example one would usually say that a domain has the Markov property, not that the domain satisfies the Markov assumption). It can also be stated colloquially as: the future is independent of the past given the present.

We also assume that the rewards are Markovian—\( R_t \) is conditionally independent of \( H_{t-1} \) given \( S_t \) (since \( A_t \) depends only on \( S_t \), this is equivalent to assuming that \( R_t \) is conditionally independent of \( H_{t-1} \) given both \( S_t \) and \( A_t \)). While the previous Markov assumptions apply to the environment (and are inherent assumptions in the MDP formulation of the environment), we make an additional Markov assumption about the agent: the agent’s policy is Markovian. That is, \( A_t \) is conditionally independent of \( H_{t-1} \) given \( S_t \).

**Question 7.** Can you give examples of Markovian and non-Markovian environments?

**Question 8.** Is the Markov property a property of the problem being formulated as an MDP or a property of the state representation used when...
To answer this second question, consider whether state transitions are Markovian in mountain car. It should be clear that they are as the domain has been described. What about if the state was $s = (x)$ rather than $s = (x, v)$? You could deduce $v_t$ from the previous state, $x_{t-1}$ and current state, $x_t$, but that would require part of the history before $s_t$. Thus, using $s = (x)$ mountain car is not Markovian. So, the Markov property is really a property of the state representation, not the problem being formulated as an MDP.

Notice that one can always define a Markovian state representation. Let $S_t$ be a non-Markovian state representation. Then $(S_t, H_{t-1})$ is a Markovian state representation. That is, we can include the history within the states in order to enforce the Markov property. This is typically undesirable because the size of the state set grows exponentially with the maximum episode length (a term discussed more later). This trick of adding information into the state is called state augmentation.

There is often confusion about terminology surrounding states, state representations, and the Markov property. The state of an MDP (and every other similar formulation, like POMDPs, DEC-POMDPs, SMDPs, etc.) should always be defined so that the Markov property is satisfied. Later we will reason about state representations that are not Markovian, in order to model situations where the agent might only be able to make partial or noisy observations about the state of the environment.

2.5.3 Stationary vs. Nonstationary

We assume that the dynamics of the environment are stationary. This means that the dynamics of the environment do not change between episodes, and also that the transition function does not change within episodes. That is, $\Pr(S_0 = s)$ is the same for all episodes, and also for all $s, a, t$, and $i$:

$$\Pr(S_{t+1} = s'|S_t = s, A_t = a) = \Pr(S_{t+1} = s'|S_i = s, A_i = a).$$  \hspace{1cm} (66)$$

Importantly, here $t$ and $i$ can be time steps from different episodes.

This is one of the assumptions that is most often not true for real problems. For example, when using RL to control a car, we might not account for how ware on the parts (e.g., tires) causes the dynamics of the car to change across drives. When using RL to optimize the selection of advertisements, the day of the week, time of day, and even season can have a large impact on how advertisements are received by people. Depending on how the problem is modeled as an RL problem, this may manifest as nonstationarity.

Although this assumption is almost always made, and is almost always false, we usually justify it by saying that some assumption of this sort is necessary.
This assumption is what allows us to use data from the past to inform how we make decisions about the future. Without some assumption saying that the future resembles the past, we would be unable to leverage data to improve future decision making. Still, there exist weaker assumptions than requiring stationarity (e.g., a small amount of work focuses on how to handle a small finite number of jumps in system dynamics or slow and continuous shifts in dynamics).

We also assume that the rewards are stationary (that the distribution over rewards that result from taking action \( a \) in state \( s \) and transitioning to state \( s' \) does not depend on the time step or episode number). However, usually we assume that \( \pi \) is nonstationary. This is because learning corresponds to changing (ideally, improving) the policy both within an episode and across episodes. A stationary policy is sometimes called a fixed policy.

2.5.4 Cart-Pole Balancing

Also called pole balancing, cart-pole, and inverted pendulum.

This domain models a pole balancing on a cart, as depicted in Figure 7. The agent must learn to move the cart forwards and backwards to keep the pole from falling.

- State: \( s = (x, v, \theta, \dot{\theta}) \), where \( x \) is the horizontal position of the cart, \( \theta \) is the angle of the pole, and \( \dot{\theta} \) is the angular velocity of the pole.
- Actions: \( \mathcal{A} = \{\text{left}, \text{right}\} \).
- \( R_t = 1 \) always.
- \( \gamma = 1 \).
• \( S_0 = (0, 0, 0, 0) \) always.

• Dynamics = physics of the system. See the work of Florian (2007) for the derivation of the correct dynamics. The domain was originally presented by Barto et al. (1983). However, this original work presents the dynamics with gravity reversed—pulling the pole up rather than down. Andy says that they did use the correct direction for gravity in their code though.

• Episodes terminate after 20 seconds, or when the pole falls down. Time is simulated with time steps of \( \Delta t = 0.02 \) seconds.

**Question 9.** Is the optimal policy for cart-pole unique?

**Answer 9.** No. An action might cause the pole to move away from vertical, but as long as it does not fall this is not penalized by the reward function.

**Question 10.** Is the state representation Markovian?

**Answer 10.** No. In order for the transition function to cause a transition to \( S_\infty \) after twenty seconds, the state must encode the current time step.

### 2.5.5 Finite-Horizon MDPs

The horizon, \( L \), of an MDP is the smallest integer such that

\[
\forall t \geq L, \quad \Pr(S_t = s_\infty) = 1.
\] (67)

If \( L < \infty \) for all policies, then we say that the MDP is finite horizon. If \( L = \infty \) then the domain may be indefinite horizon or infinite horizon. An MDP with indefinite horizon is one for which \( L = \infty \), but where the agent will always enter \( s_\infty \). One example of an indefinite horizon MDP is one where the agent transitions to \( s_\infty \) with probability 0.5 from every state. An infinite horizon MDP is an MDP where the agent may never enter \( s_\infty \).

For the cart-pole domain, how can we implement within \( P \) that a transition to \( s_\infty \) must occur after 20 seconds? We achieve this by augmenting the state to include the current time. That is, the state is \((s, t)\), where \( s \) is what we previously defined to be the state for cart-pole and \( t \) is the current time step. \( P \) increments \( t \) at each time step and causes transitions to \( S_\infty \) when \( t \) is incremented to \( 20/\Delta t = 1000 \). So, the state for cart-pole is really \( s = (x, v, \theta, \dot{\theta}, t) \). Often the dependence on \( t \) is implicit—we write \( s = (x, v, \theta, \dot{\theta}) \) and say that the domain is finite horizon.
2.5.6 Partial Observability

For many problems of interest, the agent does not know the state—it only makes observations about the state. These observations may be noisy and/or incomplete. We will discuss this later in the course.

2.5.7 Other Domains

See if you can model the following problems as MDPs:

- Type 1 Diabetes treatment, where the goal is to determine how much insulin an insulin pump should inject in order to keep a person’s blood glucose levels near optimum levels (Bastani, 2014).

- Digital marketing, where the goal is to present ads on a website that are likely to be clicked by the user. Assume that we have some knowledge about the user, like their age and gender.

- Playing video games to maximize score.

- Automation of methods for determining the composition of rocks using spectroscopy. For this task the agent observes spectra collected from a rock and makes a decision about what minerals it believes are present. There are data sets containing examples of spectra of rocks with known compositions, but these data sets are small.

- Functional electrical stimulation, where the goal is to stimulate the muscles in a paralyzed person's arm in order to cause it to move to a desired position (Thomas et al., 2009).

- Garbage collection systems, where the agent must optimize the choice of when to garbage collect.

- Intelligent tutoring systems, where the goal is to decide how to order topics presented to a student in order to maximize their score on a quiz after the tutorial (Woolf, 2010; Mandel et al., 2014).

- Elevator scheduling, where the agent must decide where to send an elevator to minimize wait times.

**Question 11.** All but one of the above problems are cases where supervision is not available. After the agent takes an action, we do not know what the optimal action would have been. One example is of a supervised learning problem that likely should not be solved using RL methods. Which one?

---

End of Lecture 5, September 17, 2019
3 Black-Box Optimization for RL

3.1 Hello Environment!

In this lecture we will describe how you can create your first RL agent. Agents can be viewed as objects in an object-oriented programming language that have functions for getting an action from the agent, telling an agent about what states and rewards it obtains, and for telling the agent when a new episode has occurs. High level pseudocode for an agent interacting with an environment may look like:

<table>
<thead>
<tr>
<th>Algorithm 2: Pseudocode for an agent interacting with an environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for episode = 0, 1, 2, ... do</td>
</tr>
<tr>
<td>2 \hspace{1em} s \sim d_0;</td>
</tr>
<tr>
<td>3 \hspace{1em} for t = 0 to \infty do</td>
</tr>
<tr>
<td>4 \hspace{2em} a = agent.getAction(s);</td>
</tr>
<tr>
<td>5 \hspace{2em} s' \sim P(s, a, \cdot);</td>
</tr>
<tr>
<td>6 \hspace{2em} r \sim d_R(s, a, s', \cdot);</td>
</tr>
<tr>
<td>7 \hspace{2em} agent.train(s, a, r, s');</td>
</tr>
<tr>
<td>8 \hspace{2em} if s' == s_\infty then</td>
</tr>
<tr>
<td>9 \hspace{3em} break; // Exit out of loop over time, t</td>
</tr>
<tr>
<td>10 \hspace{2em} s = s';</td>
</tr>
<tr>
<td>11 \hspace{2em} agent.newEpisode();</td>
</tr>
</tbody>
</table>

Here the agent has three functions. The first, \texttt{getAction}, which samples an action, \( a \), given the current state \( s \), and using the agent’s current policy. The second function, \texttt{train}, alerts the agent to the transition that just occurred, from \( s \) to \( s' \), due to action \( a \), resulting in reward \( r \). This function might update the agent’s current policy. The third function, \texttt{newEpisode}, alerts the agent that the episode has ended and it should prepare for the next episode. Notice that the agent object might have memory. This allows it to, for example, store the states, actions, and rewards from an entire episode during calls to \texttt{train}, and then update its policy when \texttt{newEpisode} is called using all of the data from the episode (or from multiple episodes).

**Question 12.** How might you create this agent object so that it improves its policy? This might be your last chance to think about this problem in your own creative way, before we corrupt your mind with the standard solutions used by the RL community.
3.2 Black-Box Optimization (BBO) for Policy Search

*Black-box optimization* (BBO) algorithms are generic optimization algorithms (i.e., not specific to RL) that solve problems of the form:

$$\arg \max_{x \in \mathbb{R}^n} f(x), \quad (68)$$

where $f : \mathbb{R}^n \to \mathbb{R}$. Different BBO algorithms make different assumptions about $f$, like that it is smooth or bounded. Furthermore, different BBO algorithms make different assumptions about what information is known about $f$. Some algorithm might assume that the optimization algorithm can compute $f(x)$ for any $x \in \mathbb{R}^n$, while others assume that the agent can compute the gradient, $\nabla f(x)$, at any point $x$. BBO algorithms are called *black-box* because they treat $f$ as a black-box—they do not look “inside” of $f$ to leverage knowledge about its structure (or knowledge of an analytic form) in order to speed up the optimization process. For RL, this means that BBO algorithms will not leverage the knowledge that the environment can be modeled as an MDP.

Here, we will consider BBO algorithms that assume the estimates of $f(x)$ can be produced for any $x \in \mathbb{R}^n$, but that the precise value of $f(x)$ is not known, and the gradient, $\nabla f(x)$, is also not known. Examples of BBO algorithms that can be used for problems of this sort include (first-choice) hill-climbing search, simulated annealing, and genetic algorithms (Russell et al., 2003). To apply these algorithms to RL, we will use them to optimize the objective function. That is, we will use them to solve the problem:

$$\arg \max_{\pi \in \Pi} J(\pi). \quad (69)$$

In order to apply these algorithms to the above problem, we must determine how we can estimate $J(\pi)$, and also how we can represent each policy, $\pi$, as a vector in $\mathbb{R}^n$.

3.2.1 How to estimate the objective function?

We can estimate $J(\pi)$ by running the policy $\pi$ for $N$ episodes and then averaging the observed returns. That is, we can use the following estimator, $\hat{J}$, of $J$:

$$\hat{J}(\pi) := \frac{1}{N} \sum_{i=1}^{N} G^i \quad (70)$$

$$= \frac{1}{n} \sum_{i=1}^{N} \sum_{t=0}^{\infty} \gamma^t R^i_t, \quad (71)$$

where $G^i$ denotes the return of the $i$th episode and $R^i_t$ denotes the reward at time $t$ during episode $i$. Hereafter, we will use superscripts on random variables to denote the episode during which they occurred (but will omit these superscripts when the relevant episode should be clear from context).
3.2.2 Parameterized Policies

Recall from Figure 3 that we can represent policies as $|S| \times |A|$ matrices with non-negative entries and rows that sum to one. When numbers are used to define a policy, such that using different numbers results in different policies, we refer to the numbers as policy parameters. Unlike other areas of machine learning, often we will discuss different functions and/or distributions that can be parameterized in this way, and so it is important to refer to these parameters as policy parameters and not just parameters (unless it is exceedingly obvious which parameters we are referring to). We refer to the representation described in Figure 3 as a tabular policy, since the policy is stored in a table, with one entry per state-action pair. Notice that we can view this table as a vector by appending the rows or columns into a vector in $\mathbb{R}^{|S| \times |A|}$.

Although this parameterization of the policy is simple, it requires constraints on the solution vectors that standard BBO algorithms are not designed to handle—they require the rows in the tabular policy to sum to one and the entries to always be positive. Although BBO algorithms might be adapted to work with this policy representation, it is usually easier to change how we represent the policy. That is, we want to store $\pi$ as a $|S| \times |A|$ matrix, $p$, that has no constraints on its entries. Furthermore, increasing $p(s,a)$ (the entry in the $s$'th row and $a$'th column) should increase $\pi(s,a)$. Notice that, using this notation, $p(s,a)$ is a policy parameter for each $(s,a)$ pair.

One common way to achieve this is to use softmax action selection. Softmax action selection defines the policy in terms of $p(s,a)$ as:

$$\pi(s,a) := \frac{e^{\sigma p(s,a)}}{\sum_{a' \in A} e^{\sigma p(s,a')}}$$

where $\sigma > 0$ is a hyperparameter that scales how differences in values of $p(s,a)$ and $p(s,a')$ change the probabilities of the actions $a$ and $a'$. For now, assume that $\sigma = 1$. To see that this is a valid definition of $\pi$, we must show that $\pi(s, \cdot)$ is a probability distribution over $A$ for all $s \in S$. That is, $\sum_{a \in A} \pi(s,a) = 1$ for all $s$ and $\pi(s,a) \geq 0$ for all $s$ and $a$. We now show these two properties. First, for all $s \in S$:

$$\sum_{a \in A} \pi(s,a) = \sum_{a \in A} \frac{e^{\sigma \pi(s,a)}}{\sum_{a' \in A} e^{\sigma p(s,a')}}$$

$$= \sum_{a' \in A} \frac{e^{\sigma \pi(s,a)}}{\sum_{a' \in A} e^{\sigma p(s,a')}}$$

$$= 1.$$ (75)

Second, for all $s$ and $a$, $e^{\sigma p(s,a)} > 0$, and so all terms in the numerator and denominator of (72) are non-negative, and thus $\pi(s,a)$ is non-negative.

This distribution is also known as the Boltzman distribution or Gibbs distribution. A drawback of using this distribution is that it cannot exactly represent deterministic policies without letting $p(s,a) = \pm \infty$. 

40
We typically denote the parameters of a policy by \( \theta \), not \( p \), and we define a \textit{parameterized policy} to be a function \( \pi : S \times A \times \mathbb{R}^n \to [0, 1] \) such that \( \pi(s, a, \theta) = \Pr(A_t = a | S_t = s, \theta) \). Thus, changing the policy parameter vector \( \theta \) results in the parameterized policy being a different policy. So, you might see the equivalent definition of a tabular softmax policy:

\[
\pi(s, a, \theta) := \frac{e^{\sigma_{\theta,s,a}}}{\sum_{a'} e^{\sigma_{\theta,s,a'}}}, \tag{76}
\]

Tabular softmax policies are just one way that the policy might be parameterized. In general, you should pick the policy parameterization (the way that policy parameter vectors map to stochastic policies) to be one that you think will work well for the problem you want to solve. This is more of an art than a science—it is something that you will learn from experience.

Now that we have represented the policy as a vector in \( \mathbb{R}^n \) (with \( n = |S||A| \) when using a tabular softmax policy), we must redefine our objective function to be a function of policy parameter vectors rather than policies. That is, let

\[
J(\theta) := \mathbb{E}[G|\theta], \tag{77}
\]

where conditioning on \( \theta \) denotes that \( A_t \sim \pi(S_t, \cdot, \theta) \). Often a policy parameterization will be used that cannot represent all policies. In these cases, the goal is to find the best policy that can be represented, i.e., the optimal policy parameter vector:

\[
\theta^* \in \arg \max_{\theta \in \mathbb{R}^n} J(\theta). \tag{78}
\]

Examples of other parameterized policies include deep neural networks, where the input to the network is the state, \( s \), and the network has one output per action. One can then use softmax action selection over the outputs of the network. If the actions are continuous, one might assume that the action, \( A_t \), should be normally distributed given \( S_t \), where the mean is parameterized by a neural network with parameters \( \theta \), and the variance is a fixed constant (or another parameter of the policy parameter vector).

One might also choose to represent deterministic policies, where the input is a state and the output is the action that is always chosen in that state. For example, consider the problem of deciding how much insulin an insulin pump should inject prior to a person eating a meal. One common (and particularly simple) policy parameterization is:

\[
\text{injection size} = \frac{\text{current blood glucose} - \text{target blood glucose}}{\theta_1} + \frac{\text{meal size}}{\theta_2}, \tag{79}
\]

where \( \theta = [\theta_1, \theta_2] \top \in \mathbb{R}^2 \) is the policy parameter vector, the current blood glucose and meal size form the state, the target blood glucose is a constant value specified by a diabetologist, and the injection size is the action (Bastani, 2014). Notice that this representation results in a small number of policy parameters—far fewer than if we were to use a neural network. Similarly, many control problems
can use policies parameterized using few parameters (Schaal et al., 2005) (see also PID and PD controllers).

Another common policy representation (typically used when states are not discrete) is softmax action selection with linear function approximation. Later we will discuss linear function approximation in general and in more detail. Here we will consider a single specific use. Let $\phi : S \rightarrow \mathbb{R}^m$ be a function that takes a state as input and returns a vector of features as output. Notice that $\phi(s)$ can be a short vector even when $S$ is continuous. For example, if $S = \mathbb{R}$, we could define $\phi(s) = [s, s^2, s^3, ..., s^4]^T$. For now we define $\phi(s)$ vector to be of length $m$ because we use $n$ to denote the total number of policy parameters. Later we may use $n$ to denote the number of features when using linear function approximation to represent structures other than the policy (e.g., when we discuss linear temporal difference learning, we will use $n$ to denote the number of features).

To use linear function approximation for softmax action selection, we store a different vector, $\theta_a$, for each action $a \in A$. We then compute $p(s,a)$ from (72) as $\theta_a^T \phi(s)$, where $v^T$ denotes the transpose of the vector $v$. That is,

$$p(s,a) = \theta_a^T \phi(s)$$

$$= \theta_a \cdot \phi(s)$$

$$= \sum_{i=1}^{m} \theta_{a,i} \phi_i(s),$$

where $\theta_{a,i}$ is the $i$th entry in the vector $\theta_a$ and $\phi_i(s)$ is the $i$th entry in the vector $\phi(s)$. Hence, in softmax action selection using linear function approximation:

$$\pi(s,a) = \frac{e^{\theta_a^T \phi(s)}}{\sum_{a' \in A} e^{\theta_{a'}^T \phi(s)}},$$

where $\theta$ is one big vector containing $m|A|$ parameters: one vector $\theta_a \in \mathbb{R}^m$ for each of the $|A|$ actions. Although you might think of $\theta$ as being a matrix, I encourage you to think of $\theta$ as a vector (the matrix with all of the columns stacked into one big column)—this will simplify later math. Also, we refer to this as “linear” because $\theta_a^T \phi(s)$ is linear in feature space, even though it may not be a linear function of $s$ due to nonlinearities in $\phi$.

The question remains: how should we define $\phi$? Which features allow the agent to represent a good policy? You can view a neural network as learning these features (the output of the second to last layer is $\phi(s)$). However, there are also known choices for $\phi(s)$ that tend to result in strong performance (Konidaris et al., 2011). We will discuss some of these in more detail later.

Selecting policy representations with small numbers of parameters often speeds up learning. This is because the space of policy parameter vectors that an algorithm must search is lower dimensional—when the problem is phrased as in (68), $n$ is the number of policy parameters, and so fewer policy parameters results in a smaller search space. Thus, deep neural networks should usually
be a last resort when selecting a policy representation, since they often have thousands, if not millions of policy parameters.

3.3 Cross-Entropy Method

The cross-entropy (CE) method for policy search is a simple BBO algorithm that has achieved remarkable performance on domains like playing Tetris (Szita and Lörincz, 2006). We present a variant of CE based on the work of Stulp and Sigaud (2012). Intuitively, CE starts with a multivariate Gaussian distribution over policy parameter vectors. This distribution has mean $\theta$ and covariance matrix $\Sigma$. It then samples some fixed number, $K$, of policy parameter vectors from this distribution. Let $\theta_1, \ldots, \theta_K$ denote these samples. It evaluates these $K$ sampled policies by running each one for $N$ episodes and averaging the resulting returns. It then picks the $K_e$ best performing policy parameter vectors (for some constant $K_e$) and fits a multivariate Gaussian to these parameter vectors. The mean and covariance matrix for this fit are stored in $\theta$ and $\Sigma$ and this process is repeated. We present pseudocode for CE in Algorithm 3, which uses the `evaluate` function defined in Algorithm 4.

**Algorithm 3: Cross-Entropy (CE) for Policy Search.**

**Input:**
1) Initial mean policy parameter vector, $\theta \in \mathbb{R}^n$
2) Initial $n \times n$ covariance matrix, $\Sigma$
3) Population, $K \in \mathbb{N}_{>1}$ [for example, $K = 20$]
4) Elite population, $K_e \in \mathbb{N}_{>0}$, where $K_e < K$ [for example, $K_e = 10$]
5) Number of episodes to sample per policy, $N \in \mathbb{N}_{>0}$ [for example, $N = 10$]
6) Small numerical stability parameter $\epsilon \in \mathbb{R}$ [for example, $\epsilon = 0.0001$]

1 while true do
2 for $k = 1$ to $K$ do
3 $\theta_k \sim N(\theta, \Sigma)$;
4 $\hat{J}_k = \text{evaluate}(\theta_k, N)$;
5 sort((($\theta_1, \hat{J}_1$), ($\theta_2, \hat{J}_2$), ..., ($\theta_K, \hat{J}_K$)), descending);
6 $\theta = \frac{1}{K - K_e} \sum_{k=1}^{K_e} \theta_k$;
7 $\Sigma = \frac{1}{\epsilon + K_e} \left( \epsilon I + \sum_{k=1}^{K_e} (\theta_k - \theta)(\theta_k - \theta)^\top \right)$;

**Algorithm 4: `evaluate`**

**Input:**
1) Policy parameter vector, $\theta \in \mathbb{R}^n$
2) Number of episodes to sample, $N \in \mathbb{N}_{>0}$ [for example, $N = 10$]

1 Run the parameterized policy using policy parameters $\theta$ for $N$ episodes;
2 Compute the resulting $N$ returns, $G^1, G^2, \ldots, G^N$, where $G^i = \sum_{t=0}^{\infty} R^i_t$;
3 Return $\frac{1}{N} \sum_{i=1}^{N} G^i$;
3.4 First-Choice Hill-Climbing

Another simple BBO algorithm is First-Choice Hill-Climbing (FCHC) (Russell et al., 2003). CE, presented above, has several hyperparameters that must be set properly for the algorithm to work well. FCHC has remarkably few hyperparameters, which makes it easy to apply. Pseudocode for a variant of FCHC, created for this class, is presented in Algorithm 5.

Algorithm 5: First-Choice Hill-Climbing (FCHC) for Policy Search.

Input:
1) Initial mean policy parameter vector, $\theta \in \mathbb{R}^n$
2) Exploration parameter, $\sigma \in \mathbb{R}$
3) Number of episodes to sample per policy, $N \in \mathbb{N}_{>0}$ [for example, $N = 10$]

1. $\hat{J} = \text{evaluate}(\theta, N);$
2. while true do
3. $\theta' \sim N(\theta, \sigma I);$ 
4. $\hat{J}' = \text{evaluate}(\theta', N);$ 
5. if $\hat{J}' > \hat{J}$ then
6. $\theta = \theta';$
7. $\hat{J} = \hat{J}';$
8. end
9. end

3.5 Evaluating RL Algorithms

There are many different ways that researchers report the performance of their algorithms. The most common method is for researchers to optimize all of the hyperparameters for all of the algorithms, and then plot learning curves for each algorithm, which show how quickly the agent learns when using each RL algorithm. The details of this process are not standardized across the community. An example of a learning curve is depicted in Figure 8.

There is no agreed upon standard for how the optimization of hyperparameters should be performed. Some researchers use grid searches and others random searches. Research on hyperparameter optimization suggests that you should at least use a random search (Bergstra and Bengio, 2012) rather than a grid search. You might consider using a BBO algorithm to optimize the hyperparameters of the algorithm. There is also no agreed upon objective for the hyperparameter optimization: some authors choose the hyperparameters that maximize the area under the learning curve (the sum of returns over a fixed number of episodes, averaged over several trials), while others optimize for the performance of the final policy after a fixed number of episodes.

Notice that this approach for reporting the performance of RL algorithms does not capture how difficult it is to find good hyperparameters for each algorithm. This problem is not new—there has been a push for many years now to report the sensitivity of algorithms to their hyperparameters, and increasingly many papers provide plots showing hyperparameter sensitivity. Notice also that some
Figure 8: Example learning curve. The horizontal axis shows the number of episodes that the agent has been learning for, and the vertical axis shows the mean return. Here the “expected return” is averaged over many trials—many lifetimes of the agent. This plot was averaged over 100,000 trials. That is, 100,000 of each type of agent (Sarsa, Q-learning, and Actor-Critic) were created and trained from scratch for 200 episodes each. The discounted returns of these 100,000 agents on the 20th episode were averaged to create the points at the 20th episode mark on the horizontal axis. Standard error error-bars are included, but are too small to be seen. It is not uncommon for researchers to provide standard deviation error bars. Researchers also sometimes plot total time-steps on the horizontal axis (particularly when episode lengths vary significantly with policy performance), and cumulative reward (total reward since the first step of the first episode) on the vertical axis.
papers present similar looking plots, but report statistics other than expected return (Johns, 2010, Page 80).
**Implement Cart-Pole**

In this assignment you will implement the cart-pole domain using the (frictionless) dynamics described by Florian (2007, Equations 23 and 24), we provide the expressions below. You must use a forward Euler approximation of the dynamics. If the cart hits the boundary of the track, the pole falls below a fail angle, or the time limit is exceeded, then terminate the episode. **You may not use existing RL code for this problem—you must implement the agent and environment entirely on your own and using the class template.**

The Cart-pole environments consists of two interacting bodies: a cart with position $x$ and velocity $\dot{x}$, and a pole with angle $\theta$ and angular velocity $\dot{\theta}$. An expression for the pole’s angular acceleration is

$$\ddot{\theta} = \frac{g \sin \theta + \cos \theta \left( \frac{-F - m_p \dot{\theta}^2 \sin \theta}{m_c + m_p} \right)}{l \left( \frac{4}{3} - \frac{m_p \cos^2 \theta}{m_c + m_p} \right)}, \quad (84)$$

where $g$ is the acceleration due to gravity, $F$ is the force applied to the cart, $m_p$ is the mass of the pole, $m_c$ is the mass of the cart, and $l$ is half the length of the pole. An expression for acceleration of the cart is

$$\ddot{x} = \frac{F + m_p l (\dot{\theta}^2 \sin \theta - \dot{\theta} \cos \theta)}{m_c + m_p} \quad (85)$$

The system state is described by the state vector $x := [x, v, \theta, \omega]$, where $v = \dot{x}$ and $\omega = \dot{\theta}$. 
The state space equations of motion are:

\[
\begin{align*}
\dot{x}_t &= v_t \quad (86) \\
\dot{v}_t &= \frac{F + m_p l (\dot{\theta}_t^2 \sin \theta_t - \ddot{\omega}_t \cos \theta_t)}{m_c + m_p} \quad (87) \\
\dot{\theta}_t &= \omega_t \quad (88) \\
\dot{\omega}_t &= \frac{g \sin \theta_t + \cos \theta_t \left( -\frac{F - m_p \omega_t^2 \sin \theta_t}{m_c + m_p} \right)}{l \left( \frac{1}{4} - \frac{m_p \cos^2 \theta_t}{m_c + m_p} \right)} \quad (89)
\end{align*}
\]

Using the Euler approximation the system evolves as:

\[
x_{t+1} = x_t + \Delta t \dot{x}_t, \quad (90)
\]

where \( x_t = [\dot{x}_t, \dot{v}_t, \dot{\theta}_t, \dot{\omega}_t] \) and \( \Delta t \) is the time in seconds between updates. In this environment, an agent’s action specifies a force \( F \in [-F_{\text{mag}}, F_{\text{mag}}] \) on the cart. Traditionally, in RL, the cart pole environment is specified with two discrete actions, where \( a_0 = -F_{\text{mag}} \) and \( a_1 = F_{\text{mag}} \). You must use this action space in creating the environment.

Use the following values for the cart-pole constants:

- Fail angle = \( \pi/12 \). (If it exceeds this value or its negative, the episode ends in failure.)
- Cart boundaries are at \( x = -3m \) and \( x = 3m \).
- Max motor force magnitude \( F_{\text{mag}} = 10.0 \) (force on cart in Newtons).
- Gravitational constant \( g \) is 9.8.
- Cart mass \( m_c = 1.0 \) kg.
- Pole mass \( m_p = 0.1 \) kg.
- Pole half-length \( l = 0.5m \).
- \( \Delta t = 0.02 \) seconds (time step).
- Max time before end of episode = 20 seconds.

**Genetic Algorithms**

Genetic algorithms (GAs) are a family of biologically inspired BBO algorithms that are empirically competitive with deep reinforcement learning methods (Such et al., 2017). Intuitively, the GA starts with a population of candidate solutions (in our case, parameterized policies \( \theta \)) and iteratively modifies, or evolves, the population. We call the number of iterations over which a population is evolved the number of generations.
At every generation, the GA uses three main methods, called genetic operators, to create the next generation. These genetic operators are known as parent selection, mutation operators, and crossover operators. **Parent selection** chooses which candidate solutions in the current generation will be used as parents for creating children in the next generation. **Mutation operators** apply random changes to parents to create children for the next generation. **Crossover operators** combine two or more parents to create children for the next generation.

Example pseudocode for a simple GA is below. Pseudocode for the functions `get_parents` and `get_children` are purposefully not provided. At each generation, every candidate solution, \( \theta_i \) \( i \in \mathbb{N} \), in the population, \( P = \{ \theta_i \}_{i=0}^K \), is evaluated and a fitness score for \( \theta_i \) is produced. The top \( K_p \) candidate solutions are then chosen to become parents of the next generation (this parent selection method is known as truncation selection). Children are created by selecting a parent uniformly at random to mutate by applying additive Gaussian noise to the parent: \( \theta_{\text{child}} = \theta_{\text{parent}} + \alpha \epsilon \), where \( \epsilon \sim \mathcal{N}(0,1) \) and \( \alpha \) is a hyperparameter. \( K_e \) candidate solutions in the new generation are unmodified copies of the top \( K_e \) candidate solutions from the previous generation. This is known as elitism.

### Algorithm 6: Genetic Algorithm (GA) for Policy Search

**Input:**
1) Number of generations, \( G \in \mathbb{N}_{>1} \) [for example, \( G = 300 \)]
2) Initial population, \( P = \{ \theta_i \}_{i=0}^K \)
3) Population size, \( K \in \mathbb{N}_{>1} \) [for example, \( K = 20 \)]
4) Truncation index, \( K_p \in \mathbb{N}_{>0} \), where \( K_p < K \) [for example, \( K_p = 10 \)]
5) Elite population, \( K_e \in \mathbb{N}_{>0} \), where \( K_e < K \) [for example, \( K_e = 10 \)]
6) Number of episodes to sample per policy, \( N \in \mathbb{N}_{>0} \) [for example, \( N = 10 \)]
7) Learning parameter \( \alpha \in \mathbb{R} \) [for example, \( \alpha = 2.5 \)]

```plaintext
1 for g = 1 to G do
  2 for k = 1 to K do
    3 \( \hat{J}_k = \text{evaluate}(\theta_k, N) \);  
    4 sorted = sort((\( \theta_1, \hat{J}_1 \), (\( \theta_2, \hat{J}_2 \)),..., (\( \theta_K, \hat{J}_K \)), descending);  
    5 parents = get_parents(\( K_p, \text{sorted} \));  
    6 next_gen = sorted[1 : K_e].append(get_children(\( \alpha, \text{parents} \ ));
```

### Algorithm 7: evaluate

**Input:**
1) Policy parameter vector, \( \theta \in \mathbb{R}^n \)
2) Number of episodes to sample, \( N \in \mathbb{N}_{>0} \) [for example, \( N = 10 \)]

```plaintext
1 Run the parameterized policy using policy parameters \( \theta \) for \( N \) episodes;  
2 Compute the resulting \( N \) returns, \( G^1, G^2, ..., G^N \), where \( G^i = \sum_{t=0}^{\infty} R^i_t \);  
3 Return \( \frac{1}{N} \sum_{i=1}^{N} G^i \);
```

**Note** that we do not provide pseudocode for the `get_parents` and `get_children` methods. It will be up to you to define these methods.
Problems

Six questions ask for a plot. You must report two plots: one for cart-pole and one for 687-Gridworld, where each of these two plots has a curve corresponding to the cross-entropy method, a curve corresponding to the GA method, and a curve corresponding to first-choice hill-climbing. In the coding problems your solutions will be graded via an autograder on Gradescope. To submit assignments you must zip the rl687 folder and its structure must be the same as the template. You must use the same files provided in the template and fill in the missing function bodies. Correct solutions can be fit within the provided files, but you can create additional files as necessary for the assignment. However, if any errors occur in using them the questions will be marked as zero. The functions will need to match our outputs exactly so pay attention to types (we use np.float64 and np.int for arrays, int, float, and bool else where). The autograder will give results before the assignment is due so you can check your implementations for correctness.

Coding Problems:

1. (15 Points) Implement the cart-pole environment as specified above.

2. (5 Points) Implement the tabular softmax policy.

3. (10 Points) Implement the cross-entropy method as described in the class notes.

4. (10 Points) Implement the First-choice hill-climbing algorithm as described in the class notes.

5. (10 Points) Implement the GA algorithm described earlier in this assignment.

Written Response:

1. (7.5 Points) Apply the CEM algorithm to the More-Watery 687-Gridworld. Use a tabular softmax policy. Search the space of hyperparameters for hyperparameters that work well. Report how you searched the hyperparameters, what hyperparameters you found worked best, and present a learning curve plot using these hyperparameters, as described in class. This plot may be over any number of episodes, but should show convergence to a nearly optimal policy. The plot should average over at least 500 trials and should include standard deviation error bars.

2. (7.5 Points) Repeat the previous question, but using first-choice hill-climbing on the More-Watery 687-Gridworld domain. Report the same quantities.

3. (7.5 Points) Repeat the previous question, but using the GA (as described earlier in this assignment) on the More-Watery 687-Gridworld domain. Report the same quantities.
4. (7.5 Points) Repeat the previous question, but using the cross-entropy method on the cart-pole domain. Notice that the state is not discrete, and so you cannot directly apply a tabular softmax policy. It is up to you to create a representation for the policy for this problem. Consider using the softmax action selection using linear function approximation as described in the notes. For descriptions of common choices of \( \phi \), see the work of Konidaris et al. (2011). Report the same quantities, as well as how you parameterized the policy.

5. (7.5 Points) Repeat the previous question, but using first-choice hill-climbing (as described in class) on the cart-pole domain. Report the same quantities and how the policy was parameterized.

6. (7.5 Points) Repeat the previous question, but using the GA (as described earlier in this homework) on the cart-pole domain. Report the same quantities and how the policy was parameterized.

7. (5 Points) Reflect on this problem. Was it easier or harder than you expected to get these methods working? In the previous assignment you hypothesized how long it would take an agent to solve the More-Watery 687-Gridworld problem. Did it take more or fewer episodes than you expected? Why do you think this happened?
4 Value Functions

So far we have described the problem we want to solve mathematically, and have described how BBO methods can be applied. These BBO algorithms do not leverage the assumed MDP structure of the environment. We will now present value functions, which are a tool that we will use to leverage the MDP structure of the environment. Notice that value functions are not a complete agent on their own.

4.1 State-Value Function

The state-value function, \( v^\pi : S \rightarrow \mathbb{R} \), is defined as follows, for all \( s \):

\[
v^\pi(s) := \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s, \pi \right].
\]

Using the \( G_t \) notation we have the equivalent definition (you may provide this as the definition when asked):

\[
v^\pi(s) := \mathbb{E} [G_t \mid S_t = s, \pi].
\]

In English, \( v^\pi(s) \) is the expected discounted return if the agent follows policy \( \pi \) from state \( s \). Notice that this quantity depends on the policy, \( \pi \). More informally, \( v^\pi(s) \), is a measure of how “good” it is for the agent to be in state \( s \) when using policy \( \pi \). We call \( v^\pi(s) \) the value of state \( s \).

Notice that we use \( t \) on the right side of (91), even though it does not appear on the left side. This is because the right side takes the same value for all \( t \). We
can show this as follows:

\[ v^\pi(s) := \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \bigg| S_t = s, \pi \right] \quad (93) \]

\[ = \sum_{k=0}^{\infty} \gamma^k \mathbb{E} \left[ R_{t+k} \bigg| S_t = s, \pi \right] \quad (94) \]

\[ = \mathbb{E}[R_t|S_t = s, \pi] + \gamma \mathbb{E}[R_{t+1}|S_t = s, \pi] + \gamma^2 \mathbb{E}[R_{t+2}|S_t = s, \pi] + \cdots \quad (95) \]

\[ = \sum_{a \in A} \Pr(A_t = a|S_t = s, \pi) \mathbb{E}[R_t|S_t = s, A_t = a, \pi] \quad (96) \]

\[ + \gamma \sum_{a \in A} \Pr(A_t = a|S_t = s, \pi) \sum_{s' \in S} \Pr(S_{t+1} = s'|S_t = s, A_t = a, \pi) \quad (97) \]

\[ \times \sum_{a' \in A} \Pr(A_{t+1} = a'|S_{t+1} = s', S_t = s, A_t = a, \pi) \quad (98) \]

\[ \times \mathbb{E}[R_{t+1}|S_{t+1} = s', A_{t+1} = a', S_t = s, A_t = a, \pi] \quad (99) \]

\[ \ldots \quad (100) \]

\[ = \sum_{a \in A} \pi(s, a) R(s, a) \quad (101) \]

\[ + \gamma \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P(s, a, s') \sum_{a' \in A} \pi(s', a') R(s', a') \quad (102) \]

\[ + \gamma^2 \sum_{a \in A} \pi(s, a) \sum_{s' \in S} \sum_{a' \in A} \pi(s', a') \sum_{s'' \in S} P(s', a', s'') \sum_{a'' \in A} \pi(s'', a'') R(s'', a'') \quad (103) \]

\[ \ldots, \quad (104) \]

where \( \times \) denotes scalar multiplication split across two lines. Notice that \( t \) does not show up in any terms in the last expression, and so regardless of the value of \( t \), \( v^\pi(s) \) takes the same value. Hence, the following definitions are equivalent (do not provide these when asked for the definition of the state value function):

\[ v^\pi(s) := \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \bigg| S_t = s, \pi \right] \quad (105) \]

\[ v^\pi(s) := \mathbb{E}[G|S_0 = s, \pi] \quad (106) \]

\[ v^\pi(s) := \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_t \bigg| S_0 = s, \pi \right] \quad (107) \]
As an example, consider the MDP depicted in Figure 9. For this MDP:

\[ v^{\pi_1}(s_1) = 0 \]  
\[ v^{\pi_1}(s_2) = 12\gamma^0 = 12 \]  
\[ v^{\pi_1}(s_3) = 0\gamma^0 + 12\gamma^1 = 6 \]  
\[ v^{\pi_1}(s_4) = 0\gamma^0 + \gamma^1 + 12\gamma^2 = 3 \]  
\[ v^{\pi_1}(s_5) = 0\gamma^0 + \gamma^1 + 0\gamma^2 + 12\gamma^3 = 1.5 \]  
\[ v^{\pi_1}(s_6) = 0 \]  

\[ v^{\pi_2}(s_1) = 0 \]  
\[ v^{\pi_2}(s_2) = 0\gamma^0 + \gamma^1 + 0\gamma^2 + 2\gamma^3 = 1/4 \]  
\[ v^{\pi_2}(s_3) = 0\gamma^0 + \gamma^1 + 2\gamma^2 = 1/2 \]  
\[ v^{\pi_2}(s_4) = 0\gamma^0 + 2\gamma^1 = 1 \]  
\[ v^{\pi_2}(s_5) = 2\gamma^0 = 2 \]  
\[ v^{\pi_2}(s_6) = 0. \]

Figure 9: A simple MDP that we will call the “chain” MDP. There are many “chain” MDPs used in the RL literature—this is not a standard one. In each state the agent can choose to move left (L) or right (R), and the transition function is deterministic in implementing these transitions. In states \( s_1 \) and \( s_6 \), both actions cause a transition to \( s_\infty \). The rewards are always zero, except for when the agent transitions from \( s_2 \) to \( s_1 \), in which case the reward is +12, or when the agent transitions from \( s_5 \) to \( s_6 \), in which case the reward is +2. The initial state distribution is not specified. For simplicity, let \( \gamma = 0.5 \). We will consider two policies for this MDP. The first, \( \pi_1 \), always selects the left action, while the second, \( \pi_2 \), always selects the right action.
4.2 Action-Value Function

The action-value function, also called the state-action value function or \(Q\)-function, is defined as:

\[
q^\pi : S \times A \to \mathbb{R}
\]

\[
q^\pi(s, a) := \mathbb{E}[G_t | S_t = s, A_t = a, \pi],
\]

for all \(s\) and \(a\). Recall that conditioning on \(\pi\) denotes that \(A_t \sim \pi(S_t, \cdot)\) for all times, \(t\), where \(A_t\) has not otherwise been specified. Here \(A_t\) has been specified, and so conditioning on \(\pi\) only applies to time steps other than \(t\). That is, \(q^\pi(s, a)\) is the expected discounted return if the agent takes action \(a\) in state \(s\) and follows the policy \(\pi\) thereafter. Equivalent definitions of \(q^\pi\) are:

\[
q^\pi(s, a) := \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \bigg| S_t = s, A_t = a, \pi \right]
\]

\(122\)

\[
q^\pi(s, a) := \mathbb{E} [G | S_0 = s, A_0 = a, \pi]
\]

\(123\)

\[
q^\pi(s, a) := \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_t \bigg| S_0 = s, A_0 = a, \pi \right].
\]

\(124\)

For the chain MDP depicted in Figure 9:

\[
q^{\pi_1}(s_1, L) = 0
\]

\(125\)

\[
q^{\pi_1}(s_1, R) = 0
\]

\(126\)

\[
q^{\pi_1}(s_2, L) = 12\gamma^0 = 12
\]

\(127\)

\[
q^{\pi_1}(s_2, R) = 0\gamma^0 + 0\gamma^1 + 12\gamma^2 = 3
\]

\(128\)

\[
q^{\pi_1}(s_3, L) = 0\gamma^0 + 12\gamma^1 = 6
\]

\(129\)

\[
q^{\pi_1}(s_3, R) = 0\gamma^0 + 0\gamma^1 + 0\gamma^2 + 12\gamma^3 = 1.5
\]

\(130\)

Notice that \(q^\pi(s, a)\) and \(v^\pi(s)\) are both always zero if \(s\) is a terminal state. Also, take particular note of \((128)\)—it shows a nuance of \(q\)-values that is often missed. That is, the agent begins in \(s_2\) and takes the action to go right. It then takes the action to go left, bringing it back to \(s_2\). Now when it is again in \(s_2\), it takes the action to go left. In this sense, the \(q\)-function considers the behavior of an agent that is not following a fixed policy. For more on this topic, see the work of Bellemare et al. (2016) for further discussion of this “inconsistency”.

4.3 The Bellman Equation for \(v^\pi\)

The Bellman equation is a recursive expression for the value function—a sort of consistency condition that the value function satisfies. Specifically, the Bellman equation for the state-value function can be derived from the definition of the
state-value function:

\[ v^\pi(s) := \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \middle| S_t = s, \pi \right] \]  

(131)

\[ = \mathbb{E} \left[ R_t + \sum_{k=1}^{\infty} \gamma^k R_{t+k} \middle| S_t = s, \pi \right] \]  

(132)

\[ = \mathbb{E} \left[ R_t + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} \middle| S_t = s, \pi \right] \]  

(133)

(a) \[ \sum_{a \in A} \pi(s,a)R(s,a) + \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, \pi \right] \]  

(134)

(b) \[ \sum_{a \in A} \pi(s,a)R(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') \]  

(135)

\[ \times \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a, S_{t+1} = s', \pi \right] \]  

(136)

(c) \[ \sum_{a \in A} \pi(s,a)R(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') \]  

(137)

\[ \times \gamma \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_{t+1} = s', \pi \right] \]  

(138)

(d) \[ \sum_{a \in A} \pi(s,a)R(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') \gamma v^\pi(s') \]  

(139)

\[ = \sum_{s' \in S} P(s,a,s') (R(s,a) + \gamma v^\pi(s')) . \]  

(140)

where × denotes scalar multiplication split across two lines, (a) comes from modifying the indexing of the sum to start at zero instead of one, but changes all uses of \( k \) within the sum to \( k + 1 \), (b) comes from marginalizing over \( A_t \) and \( S_{t+1} \), (c) follows from the Markov property, and (d) comes from the definition of the state-value function (see (91)).

This final expression gives the Bellman equation for \( v^\pi \):

\[ v^\pi(s) = \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') (R(s,a) + \gamma v^\pi(s')) . \]  

(141)

Another way to understand the Bellman equation for \( v^\pi \) is to consider the
Consider the “Cookie MDP” depicted in Figure 10, which was created for this course (it is not a standard domain). Clearly \( v^\pi(s_3) = 0 \). We can then compute \( v^\pi(s_2) \) in two different ways. The first is to use the definition of the value function (and the property that state transitions are deterministic in this case):

\[
v^\pi(s_2) = R_2 + \gamma R_3 = 10 + \gamma 0 = 10.
\]  
(146)

The second approach is to use the Bellman equation (and the property that state transitions are deterministic in this case):

\[
v^\pi(s_2) = R_2 + \gamma v^\pi(s_3) = 10 + \gamma 0 = 10.
\]  
(147)

We can also compute \( v^\pi(s_0) \) both ways:

\[
v^\pi(s_0) = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 = 0 + \gamma 0 + \gamma^2 10 + \gamma^3 0 = \gamma^2 10.
\]  
(150)

\[
v^\pi(s_0) = R_0 + \gamma v^\pi(s_1) = 0 + \gamma 10 = \gamma^2 10.
\]  
(151)
Notice that already it is becoming easier to compute the value of states using the Bellman equation than it is to compute the values of states from the definition of the value function. This is because the Bellman equation only needs to look forward one time step into the future, while the definition of the value function must consider the entire sequence of states that will occur until the end of the episode. When considering larger problems, and particularly problems with stochastic policies, transition functions, and rewards, the Bellman equation will be increasingly useful, since computing state-values from the definition of the value function would require reasoning about every possible sequence of events that could occur from the occurrence of that state until the end of the episode.

For more intuition about the Bellman equation, imagine that the current state is $s$. We can view the Bellman equation as breaking the expected return that will occur into two parts: the reward that we will obtain during the next time step, and the value of the next state that we end up in. That is,

$$v^\pi(s) = \mathbb{E} \left[ R(s, A_t) + \gamma v^\pi(S_{t+1}) \mid S_t = s, \pi \right].$$  \hspace{1cm} (152)

This should make intuitive sense, because the value of the next state is the expected discounted sum of rewards that we will obtain from the next state, and so summing the expected immediate reward and the expected discounted sum of rewards thereafter gives the expected discounted sum of rewards from the current state.

### 4.4 The Bellman Equation for $q^\pi$

While the Bellman equation for $v^\pi$ is a recurrent expression for $v^\pi$, the Bellman equation for $q^\pi$ is a recurrent expression for $q^\pi$. Specifically:

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s, a, s') \sum_{a' \in A} \pi(s', a') q^\pi(s', a').$$  \hspace{1cm} (153)

**Question 13.** Can you derive the Bellman equation for $q^\pi(s, a)$ from the definition of $q^\pi$?
4.5 Optimal Value Functions

The optimal value function, \( v^* \), is a function \( v^* : \mathcal{S} \to \mathbb{R} \) defined by:

\[
    v^*(s) := \max_{\pi \in \Pi} v^\pi(s). \tag{162}
\]

Notice that for each state \( v^* \) “uses” the policy, \( \pi \), that maximizes \( v^\pi(s) \). Thus, \( v^* \) is not necessarily associated with a particular policy—\( v^*(s) \) can be the value function associated with different policies depending on which state, \( s \), it is evaluated on.

Consider the relation \( \geq \) for policies defined as:

\[
    \pi \geq \pi' \text{ if and only if } \forall s \in \mathcal{S}, v^\pi(s) \geq v^\pi'(s). \tag{163}
\]

Notice that this relation produces a partial ordering on the set of policies. This
is not a total order on the set of policies because there can exist two policies, \( \pi \) and \( \pi' \), such that both \( \pi \not\geq \pi' \) and \( \pi' \not\geq \pi \).

**Question 14.** Give an example MDP for which there exist policies \( \pi \) and \( \pi' \) such that \( \pi \not\geq \pi' \) and \( \pi' \not\geq \pi \).

We now present a definition of an optimal policy that differs from our earlier definition. Specifically, an optimal policy, \( \pi^* \) is any policy that is at least as good as all other policies. That is, \( \pi^* \) is an optimal policy if and only if

\[
\forall \pi \in \Pi, \ \pi^* \geq \pi.
\]  

(164)

Particularly given that \( \geq \) only produces a partial ordering, at this point it may not be clear than such an optimal policy exists.

Later we will prove that for all MDPs where \(|S| < \infty, |A| < \infty, R_{\text{max}} < \infty, \) and \( \gamma < 1 \), there exists at least one optimal policy, \( \pi^* \) under this definition of an optimal policy. That is, Property 1 holds for this definition of an optimal policy as well as the definition of an optimal policy in (17). From the definition of \( v^* \) in (162), it follows that \( v^* = v^{\pi^*} \) for all optimal policies, \( \pi^* \).

We now have two different definitions of an optimal policy. Both definitions are standard in RL research. The definition presented in (17) is common in papers that focus on policy optimization, like BBO algorithms and algorithms that compute the gradient of \( J(\theta) \) (we will talk more about these policy gradient algorithms later). The definition presented in (164) is common in theoretical reinforcement learning papers and in papers that emphasize the use of value functions.

**Question 15.** Which definition of an optimal policy is stricter, the definition in (17) or the definition in (164)?

Answer 14. The definition in (164) is stricter. That is, every optimal policy according to (164) is an optimal policy according to (17), but not every optimal policy according to (17) is an optimal policy according to (164). The difference between these two definitions stems from their requirements on the behavior of an optimal policy for states that are not reachable from any state in the support of the initial state distribution.

Also, notice that even when \( \pi^* \) is not unique, the optimal value function, \( v^* \) is unique—all optimal policies share the same state-value function.
Instructions: You have 5 minutes to complete this quiz. This quiz is closed notes—do not use your notes or a laptop. Do not discuss problems with your neighbors until after everyone has handed in their quiz.

Use the notation from class. Don’t forget to capitalize your random variables. Presenting equalities that are true is not enough—you must provide the math definitions of the symbols on the left.

Fill in the definitions for the following terms:

1. \( R(s, a) = \mathbb{E}[R_t|S_t = s, A_t = a] \)

2. \( G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \)

3. \( v^\pi(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k}|S_t = s, \pi\right] \)

4. \( q^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k}|S_t = s, A_t = a, \pi\right] \)

5. The Bellman equation says that for all states \( s \in \mathcal{S} \),

\[
v^\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R(s, a) + \gamma v^\pi(s') \right]
\]
Just as we defined an optimal state-value function, we can define the optimal action-value function, \( q^* : S \times A \rightarrow \mathbb{R} \), where
\[
q^*(s, a) := \max_{\pi \in \Pi} q^\pi(s, a).
\] (165)

Also like the optimal state-value function, \( q^* = q^\pi^* \) for all optimal policies, \( \pi^* \).

**Question 16.** Given \( v^* \), can you compute \( \pi^* \) if you do not know \( P \) and \( R \)?

**Answer 15.** No. Any action in \( \arg \max_a \sum_{s'} P(s, a, s') \left[ R(s, a) + \gamma v^*(s') \right] \) is an optimal action in state \( s \). Computing these actions requires knowledge of \( P \) and \( R \).

**Question 17.** Given \( q^* \), can you compute \( \pi^* \) if you do not know \( P \) and \( R \)?

**Answer 16.** Yes. Any action in \( \arg \max_a q^*(s, a) \) is an optimal action in state \( s \).

### 4.6 Bellman Optimality Equation for \( v^* \)

The **Bellman optimality equation for \( v^* \)** is a recurrent expression for \( v^* \). We will show later that it holds for optimal policies, and only for optimal policies.

To try to get a recurrent expression for \( v^* \), we can imagine what would happen if there was an optimal policy \( \pi^* \), with value function \( v^* \). We can begin with the Bellman equation for this policy \( \pi^* \):
\[
v^*(s) = \sum_{a \in A} \pi^*(s, a) \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma v^*(s') \right].
\] (168)

Notice that in state \( s \), \( \pi^* \) will pick the action that maximizes \( q^*(s, a) \). So, we do not need to consider all possible actions, \( a \)—we only need to consider those that
cause the $q^*(s,a)$ term in (168) to be maximized. Thus,

$$v^*(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma v^*(s') \right].$$

(169)

The above equation is the Bellman optimality equation for $v^*$. We say that a policy, $\pi$, satisfies the Bellman optimality equation if for all states $s \in S$:

$$v^\pi(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma v^\pi(s') \right].$$

(170)

A possible misconception is that we have “derived” the Bellman optimality equation formally. We have not—we have not established that there actually exists a policy $\pi^*$ such that $v^{\pi^*} = v^*$, a property that we used when introducing the Bellman optimality equation. Rather, one should view the Bellman optimality equation at this point as an equation that policies may, or may not satisfy. The Bellman optimality equation will be useful to us because we will establish that 1) if a policy $\pi$ satisfies the Bellman optimality equation, then it is an optimal policy, and 2) if the state and action sets are finite, rewards are bounded, and $\gamma < 1$, then there exists a policy $\pi$ that satisfies the Bellman optimality equation. With these two results, we will have established the existence of an optimal policy, $\pi^*$.

Once we have established these results, we will have that all optimal policies satisfy the Bellman optimality equation, there exists at least one optimal policy, and all optimal policies $\pi$ have value functions $v^\pi$ that are equal to $v^*$.

Like the Bellman optimality equation for $v^*$, we can define Bellman optimality equation for $q^*$ as:

$$q^*(s, a) = \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma \max_{a' \in A} q^*(s', a') \right].$$

(171)

**Question 18.** Derive the Bellman optimality equation for $q^*$ starting with the Bellman equation for $q^{\pi^*}$.

We now establish the first link in our line of reasoning that will allow us to establish the existence of an optimal policy:

**Theorem 1.** If a policy $\pi$ satisfies the Bellman optimality equation, then $\pi$ is an optimal policy.

**Proof.** By the assumption that $\pi$ satisfies the Bellman optimality equation, we have that for all states $s$:

$$v^\pi(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') [R(s, a) + \gamma v^\pi(s')].$$

(172)
Applying the Bellman optimality equation again, this time for \( v^\pi(s') \), we can expand the above equation:

\[
v^\pi(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma \left( \max_{a' \in A} \sum_{s''} P(s', a', s'')(R(s', a') + \gamma v^\pi(s'')) \right) \right]
\]

(173)

We could continue this process, constantly replacing the final \( v^\pi \) term using the Bellman optimality equation. If we could do this infinitely often, we would eliminate \( \pi \) from the expression entirely. Consider what this expression would become if \( s \) corresponds to the state at time \( t \). \( R(s, a) \) is the first reward, and \( R(s', a') \) is the second reward (which is discounted by \( \gamma \)). We would eventually obtain an \( R(s'', a'') \) term, which would be discounted by \( \gamma^2 \). The \( P \) terms are capturing state transition dynamics. So, ignoring for a moment how actions are selected, this expression is the expected discounted sum of rewards. Now, when computing this expected discounted sum of rewards, which actions are being chosen? At each time, \( t \), the action is chosen that maximizes the expected discounted sum of future rewards (given that in the future the actions are chosen to also maximize the discounted sum of future rewards).

Consider now any policy \( \pi' \). What would happen if we replaced each \( \max_{a \in A} \) with \( \sum_{a \in A} \pi'(s, a) \)? Would the expression become bigger or smaller? We argue that the expression could not become bigger. That is, for any policy \( \pi' \):

\[
v^\pi(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma \left( \max_{a' \in A} \sum_{s''} P(s', a', s'')(R(s', a') + \gamma \ldots) \right) \right]
\]

(174)

\[
\geq \sum_{a \in A} \pi'(s, a) \sum_{s' \in S} P(s, a, s') \left[ R(s, a) + \gamma \left( \sum_{a' \in A} \pi'(s', a') \sum_{s''} P(s', a', s'')(R(s', a') + \gamma \ldots) \right) \right].
\]

(175)

If you do not see why this is true, consider any finite set \( \mathcal{X} \), any distribution \( \mu \) over the set \( \mathcal{X} \), and any function \( f : \mathcal{X} \to \mathbb{R} \). Convince yourself that the following property holds:

\[
\max_{x \in \mathcal{X}} f(x) \geq \sum_{x \in \mathcal{X}} \mu(x) f(x).
\]

(176)

We have simply applied this property repeatedly, where \( \mathcal{X} \) is \( A \), \( \mu \) is \( \pi(s, \cdot) \), \( x \) is \( a \), and \( f \) is the remainder of the expression as a function of \( a \).

Continuing the proof, given that the above holds for all policies \( \pi' \), we have
that for all states $s \in \mathcal{S}$ and all policies $\pi' \in \Pi$:

$$v^\pi(s) = \max_{a \in A} \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R(s, a) + \gamma \left( \max_{a' \in A} \sum_{s''} P(s', a', s'') (R(s', a') + \gamma \ldots) \right) \right]$$

$$= \mathbb{E}[G_t | S_t = s, \pi']$$

$$= v^{\pi'}(s).$$

Hence, for all states $s \in \mathcal{S}$, and all policies $\pi' \in \Pi$, $v^\pi(s) \geq v^{\pi'}(s)$, i.e., for all policies $\pi' \in \Pi$, we have that $\pi \geq \pi'$, and hence $\pi$ is an optimal policy.

Notice that this theorem has not established the existence of an optimal policy because we did not show that there exists a policy that satisfies the Bellman optimality equation.

End of Lecture 9, October 1, 2019

5 Reward Design [Guest Lecture by Yash Chandak]

So far in this course, we have seen how a reward function is essential to specify the objective a researcher wants an agent to optimize for. However, in many cases, it is not simple to design a reward function that incentivizes an agent to find a policy that aligns with the researcher’s goal for the agent. This section first discusses how some heuristically designed reward functions can distract the agent and result in sub-optimal policies. Subsequently, two techniques: (a) Optimal reward functions, and (b) Reward shaping, are discussed which can help in mitigating this problem.

This section also has an associated set of powerpoint slides, which are available here. While not a part of the main syllabus, these slides also contain information related to designing reward functions from languages that can be encoded as a regular expression. Additionally, some methods for teaching robots through active human interaction or demonstrations are also briefly reviewed in the presentation.

5.1 Examples of Bad Reward Functions

There are several noteworthy instances where researchers have unknowingly designed reward functions that did not align with what they wanted. For example, consider the example discussed in this blog post, where the researchers wanted to train a robotic arm simulator to grasp and move a cube. To make the agent do so, they designed a reward function that rewarded the agent based on how much it ultimately moves the cube. Instantly, the agent exploited this reward function and learned to smack the cube hard such that it rolls away the
maximum distance. On further training, it learned to grasp the cube and toss it in the air such that it goes even farther!

In another blogpost, the author recalls an incident where an agent was trained to make a simulated half-cheetah run. Similar to the previous example, a positive reward was given when the half-cheetah moved a certain distance. Surprisingly, the agent discovered a unique gait to move by flipping on its back and kicking in the air! While this did help it cover the distance and optimize for the total reward, it certainly did not align with what the researchers had aimed to make the agent learn.

5.2 Optimal Reward Function

The foundations of the reinforcement learning framework are motivated by how animals learn and interact with their surroundings. A popular computational abstraction of this interaction process is provided in Figure 11 (left), where the agent observes a state and executes an action that results in a reward and the next state. From the point of view of this abstraction, these rewards, which drive the agent’s behavior, are generated completely by the environment. However, Singh et al. (2009a) argue that such an abstraction fails to capture an important aspect of internal motivation in animal learning.

Figure 11 (right) presents an alternate abstraction, where the reward functions that drive the animal behavior is generated by the animal itself. Here, Singh et al. (2009b) argue that the environment only provides the evolutionary basis for the animal (eg., survival, happiness, etc.), and the animal sets its own goals and objectives that help it maximize the evolutionary fitness. This viewpoint has a key advantage. It allows different animals to have different goals even though the final evolutionary objective is the same. The internal reward can depend on the way an animal interacts with the environment and can thus possibly help the animal overcome its limitations to better optimize for the evolutionary objective.

We now elaborate on this idea of internal rewards from a more computational point of view. In the following paragraphs, we provide an example that demon-
strates how internal rewards can be useful. Then we define what an optimal reward function is for an agent and provide a simple algorithm to find it.

The following example succinctly highlights how an internal reward function can be beneficial for the agent. Consider the four-state MDP in the Figure 12(a). The agent starts in the grey state and has two available actions. Action $a_1$ takes the agent to the red state and rewards $+1$. Action $a_2$ takes the agent to purple state with probability $0.6$ and the blue state with probability $0.4$, yielding rewards of $+2$ and $-10$, respectively. The optimal action is $a_1$.

Now consider an agent that enumerates several random trajectories, estimates a model of the environment using those trajectories, and then acts according to the estimated model. Additionally, due to certain constraints, it can only represent deterministic transitions. Therefore, the agent estimates a model where action $a_2$ transitions deterministically to the purple state (Figure 12(b)). According to this model, unfortunately, the agent will consider action $a_2$ to be optimal when using the original reward functions. However, if there is an alternate reward function that assigns the red state a reward of $+3$ then the agent can learn the optimal policy despite its model error.

While this particular example might seem contrived, it captures the essence of the problem. The original reward function does not depend on the limitations (e.g., restricted expressibility of function approximators, computational and memory constraints, modeling errors, etc.) an agent might have while interacting with the environment. But an alternate, agent dependent, reward function can be designed that can help mitigate the negative consequences of these limitations.

Let $J : A \times M \times R \rightarrow \mathbb{R}$ be an objective function that maps the performance of any agent $A \in A$, in an MDP $M \in M$, trained using a reward function $r \in R$. Then the optimal reward function for an agent $A$ in MDP $M$ is,

$$r^* \in \arg \max_{r \in \mathbb{R}} J(A, M, r).$$

An outline of a black-box algorithm for searching for the optimal reward function is given below,
5.3 Reward Shaping

One of the major challenges for a reinforcement learning agent is the availability of only sparse rewards. This makes finding the optimal trajectory extremely hard as there is no intermediate help during exploration. For example, if an agent has four actions available at any state and the horizon length is 20, then the number of possible trajectories is in order of $4^{20}$. Among them, if there exists only one trajectory that yields the goal reward then random exploration is unlikely to stumble across that trajectory even once.

Consider the problem in Figure 13, where the task for an agent is to ride a bicycle and reach the finish line. The episode ends when the agent reaches the goal line, otherwise, there is no time limit for solving the problem. On reaching the goal, the agent receives a reward of +100 and 0 otherwise. This domain also suffers from sparse rewards problem. One natural heuristic to help the agent reach the goal is by awarding a +1 helping reward whenever the agent moves towards the finish line. However, this heuristic has a major drawback: As the episode never ends until the agent reaches the finish line, the agent can keep...
moving to and fro to accumulate the helping reward whose net worth is more than the goal reward.

While there are other possible heuristics that will not have this problem (e.g., additionally penalizing the agent with $-1$ when it moves away from the finish line), it shows that heuristics can often fail. As the domains become increasingly more complex, fixing this heuristic can become difficult. In the following paragraphs, we discuss the underlying problem with such heuristics and present a principled way to design helping rewards.

The core reason behind the problem in the bicycle domain can be reduced to the fact that the agent can find a loop that has a net positive return from the helping reward (Note: A net positive return from the actual reward function is fine). This permanently distracts the agent from the main task as the agent can exploit this loop to obtain a return higher than what it would have by actually solving the desired task. To mitigate this problem, the helping reward should be designed such that it neither positively nor negatively distracts the agent permanently. That is, while it would be beneficial to have a helping reward that can help the agent quickly find the desired trajectory, its net impact should be zero.

Let $R(s, a, s')$ be the original reward function and $F(s, a, s')$ be the helping reward function, then the new reward function $R'(s, a, s') := R(s, a, s') + F(s, a, s')$ should be such that for any trajectory,

$$\sum_{t=0}^{\infty} \gamma^t R'(s_t, a_t, s_{t+1}) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}).$$

This would imply that $\sum_{t=0}^{\infty} \gamma^t F(s_t, a_t, s_{t+1}) = 0$. Ng et al. (1999) showed that using any function $\Phi(s) : S \rightarrow \mathbb{R}$, if $F := \gamma \Phi(s') - \Phi(s)$ then (181) can be satisfied. To see this intuitively, consider unrolling (181) using the $F$ as defined above,

$$\sum_{t=0}^{\infty} \gamma^t R'(s_t, a_t, s_{t+1}) = R(s_0, a_0, s_1) + \gamma \Phi(s_1) - \Phi(s_0)$$

$$+ \gamma (R(s_1, a_1, s_2) + \gamma \Phi(s_2) - \Phi(s_1))$$

$$...$$

$$+ \gamma^t (R(s_t, a_t, s_{t+1}) + \gamma \Phi(s_{t+1}) - \Phi(s_t))$$

$$...$$

$$= -\Phi(s_0) + \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$$

It can be noticed that in (186), all the $\Phi(s_t)$ terms cancel each other out, except $\Phi(s_0)$ and $\gamma^{t+1} \Phi(s_{t+1})$. For continuing tasks, as $t \rightarrow \infty$, $\gamma^{t+1} \rightarrow 0$, therefore $\gamma^{t+1} \Phi(s_{t+1}) \rightarrow 0$. For episodic tasks, $s_{t+1}$ corresponds to the absorbing state and $\Phi(s_{t+1})$ can be easily defined to be 0. Regarding the starting state, to ensure (181) holds with exact equality, $\Phi(s_0)$ is also be defined to be 0. Ng et al. (1999)
refer to such a $F$ as the potential-based shaping reward.\(^4\) The following theorem formally establishes this intuition.

**Theorem 2** (Ng et al. (1999)). For any $\Phi : \mathcal{S} \rightarrow \mathbb{R}$, if $F(s, a, s') := \gamma \Phi(s') - \Phi(s)$, then any optimal policy for the MDP $M := (\mathcal{S}, \mathcal{A}, P, \gamma, d_0, R)$ is also an optimal policy for the MDP $M' := (\mathcal{S}, \mathcal{A}, P, \gamma, d_0, R + F)$, and vice-versa.

**Proof.** Consider the Bellman optimality equation for $M$,

$$q^*_M(s, a) = \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} q^*_M(s', a') \right]. \quad (188)$$

Note that (188) can also be expressed as,

$$q^*_M(s, a) - \Phi(s) = \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R(s, a, s') + \gamma \Phi(s') - \Phi(s) + \gamma \max_{a' \in \mathcal{A}} (q^*_M(s', a') - \Phi(s')) \right]. \quad (189)$$

Let, $\hat{q}_{M'}(s, a) := q^*_M(s, a) - \Phi(s)$, then (189) is,

$$\hat{q}_{M'}(s, a) = \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R(s, a, s') + F(s, a, s') + \gamma \max_{a' \in \mathcal{A}} \hat{q}_{M'}(s', a') \right]. \quad (190)$$

$$= \sum_{s' \in \mathcal{S}} P(s, a, s') \left[ R'(s, a, s') + \gamma \max_{a' \in \mathcal{A}} \hat{q}_{M'}(s', a') \right]. \quad (191)$$

Notice that (191) is the Bellman optimality equation for $M'$. Therefore,

$$\pi^*_M(s) \in \arg \max_{a \in \mathcal{A}} \hat{q}_{M'}(s, a) \quad (192)$$

$$= \arg \max_{a \in \mathcal{A}} q^*_M(s, a) - \Phi(s) \quad (193)$$

$$= \arg \max_{a \in \mathcal{A}} q^*_M(s, a) \quad (194)$$

$$= \pi^*_M(s). \quad (195)$$

\(\square\)

Theorem 2 shows that $F(s, a, s') := \gamma \Phi(s') - \Phi(s)$ is sufficient to preserve the optimal policy. Ng et al. (1999) in their work also prove that this is not only sufficient but also the necessary condition. In other words, there is no other way of defining $F(s, a, s')$ that ensures this property, unless some other assumptions are made. The proof of this is beyond the scope of this lecture and interested readers can refer appendix in the work by Ng et al. (1999).

\(^4\)My guess is that this terminology has its root in physics, where traditionally potential functions are used when either the net work done in moving a particle around a loop is 0, or else it only depends on the start and the final position of the particle.
Question 19. How is the optimal policy affected if \( \Phi(s_0) \neq 0 \)?

Answer 18. It will not impact on the optimal policy as reward in the starting state does not impact any decision making steps.

Question 20. What is the optimal policy when only a potential-based shaping reward is provided to the agent?

Question 21. What is the impact of reward shaping on policies which are not optimal? Are the partial orderings preserved?

End of Lecture 10, October 3, 2019

6 Policy Iteration and Value Iteration

So far we have defined the problem that we would like to solve, discussed BBO algorithms, which ignore the MDP structure of the problem, and defined value functions that more sophisticated algorithms will use to leverage the MDP structure of the environment. In this section we will show how value functions can be used to efficiently solve for the optimal policies of finite MDPs when the transition function and reward function are known. That is, in this section we will present standard planning algorithms. We present these algorithms because the later RL algorithms (which do not require \( P \) and \( R \) to be known) are closely related to these algorithms (they can be viewed as stochastic approximations to these planning algorithms). We begin with the question of how the value function for a policy can be computed.

6.1 Policy Evaluation

Here we consider the question: given a policy, \( \pi \), how can we efficiently compute \( v^\pi \)? Notice that the Bellman equation provides us with \(|S|\) equations and \(|S|\) unknown variables, \( v^\pi(s_1), v^\pi(s_2), \ldots, v^\pi(s_{|S|}) \). These equations are:

\[
v^\pi(s_1) = \sum_{a \in A} \pi(s_1, a) \sum_{s' \in S} P(s_1, a, s') \left( R(s_1, a) + \gamma v^\pi(s') \right)
\]

\[
v^\pi(s_2) = \sum_{a \in A} \pi(s_2, a) \sum_{s' \in S} P(s_2, a, s') \left( R(s_2, a) + \gamma v^\pi(s') \right)
\]

\[
\vdots
\]

\[
v^\pi(s_{|S|}) = \sum_{a \in A} \pi(s_{|S|}, a) \sum_{s' \in S} P(s_{|S|}, a, s') \left( R(s_{|S|}, a) + \gamma v^\pi(s') \right).
\]
Notice that this is a system of linear equations for which we know there is
a unique solution (the value function—we know this is unique because these
equations were derived from the definition of the value function in (91), which is
clearly unique). This system can be solved in \( O(|S|^3) \) operations (in general this
problem requires \( \Omega(|S|^2) \) operations, and in Fall 2018 the algorithm with the
best asymptotic runtime is that of Coppersmith and Winograd (1987), which
requires \( O(n^{2.736}) \) operations.

An alternative approach is to use dynamic programming. Although not nec-
essarily more efficient, this dynamic programming approach will later allow us to
efficiently interleave steps of evaluating the current policy and then improving the
current policy. Here, when we talk about evaluating a policy or policy evaluation,
we refer to estimating the state-value or action-value function associated with
the policy. Later we will discuss a different form of policy evaluation wherein
the goal is to estimate \( J(\pi) \), not the entire value function, \( v_\pi \).

Let \( v_0, v_1, v_2, \ldots \) denote a sequence of functions where each \( v_i : S \to \mathbb{R} \). Note
that some people like to include hat symbols on estimators. In the notes we will
continue by writing \( \hat{v}_i \) to denote the \( i \)th estimator of \( v_\pi \), but you are free in your
assignments to write \( \hat{v}_i \) (as we are doing in lecture) in place of \( \hat{v}_i \). Intuitively,
this sequence of functions represents the sequence of estimates of \( v_\pi \) produced
by an algorithm estimating \( v_\pi \) in an incremental manner. This has been a point
of confusion for students in the past, who did not recognize that \( v_i \) and \( v_\pi \) are
different. So, to say it again, notice that \( v_i \) is our \( i \)th approximation of \( v_\pi \). It is
not necessarily \( v_\pi \). Also, although we typically “know” \( v_i \) (we have an analytic
form for it, or code to evaluate it), we often do not precisely know \( v_\pi \) (if we did, we
would have no need to approximate it!).

Consider the setting where \( v_0 \) is chosen arbitrarily. One way to improve our
estimate is to try to make the two sides of the Bellman equation equal. Recall
that the Bellman equation for \( v_\pi \) is:

\[
v_\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_\pi(s') \right).
\] (199)

We can therefore try to make the two sides of the Bellman equation equal with
the following update:

\[
v_{i+1}(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_i(s') \right).
\] (200)

Applying this update to compute \( v_{i+1}(s) \) for ever state \( s \) given \( v_i \), is called a full
backup. Applying the update to a compute \( v_{i+1}(s) \) for a single state, \( s \), is called a backup.

To see why this is a dynamic programming approach, consider what this
algorithm does if we stored \( v_0, v_1, \ldots \) as a matrix with \( v_0 \) as the first column, \( v_1 \)
and the second column, etc. This update rule fills this matrix from the left to
the right, where the values for entries depend on previously computed values for
the previous column.

From our derivation of the Bellman equation, it should be clear that \( v_\pi \) is
a fixed point of this iterative procedure (that is, if \( v_i = v_\pi \), then \( v_{i+1} = v_\pi \) as

72
well). Less obviously, as \( i \to \infty, v_i \to v^\pi \). We will not prove this property (this update is a stepping stone to the value iteration update that we present later, and we will focus on proving the convergence of the value iteration update).

**Question 22.** Consider a \( 4 \times 4 \) gridworld where the agent starts in the top left, the bottom right state is terminal, rewards are always \(-1\), \( \gamma = 1 \), and state transitions are deterministic. Consider the policy that always chooses the action to move down, except when it is on the bottom row, at which point it chooses the action to move right. Starting with \( v_0(s) = 0 \) for all \( s \), compute \( v_1, v_2, \ldots, v_7 \).

**Answer 19.**

\[
\begin{align*}
v_0 &= 0 \\
v_1 &= -1, -1, -1, -1, -1, -1, -1 \\
v_2 &= -2, -2, -2, -2, -2, -2, -2 \\
v_3 &= -3, -3, -3, -3, -3, -3, -3 \\
v_4 &= -4, -4, -4, -4, -4, -4, -4 \\
v_5 &= -5, -5, -5, -5, -5, -5, -5 \\
v_6 &= -6, -6, -6, -6, -6, -6, -6 \\
v_7 &= -7, -7, -7, -7, -7, -7, -7 \\
\end{align*}
\]

Notice that in Question 22 information appears to flow backwards across state transitions. Also notice that in this example the process has reached its fixed point after only seven iterations. In general, this policy evaluation algorithm is only guaranteed to converge in the limit, and so practical implementations might halt the process when all changes to the current state-value approximation are smaller than some predefined constant value.

The dynamic programming algorithm for policy evaluation that we have described thus far can be implemented by storing \( 2|S| \) values—by storing \( v_i \) only until \( v_{i+1} \) has been computed. When a new estimate of \( v^\pi(s) \) has been computed, it is placed in \( v_{i+1}(s) \) in order to not overwrite the value stored in \( v_i(s) \), which might be used when computing the next values for other states. An alternative in-place implementation keeps only a single table, performs individual state backups (rather than full backups) and stores updated state-value approximations directly in the same table from which they were computed. This variant has also been shown to converge, even if states are updated in any order or if some states are
updated more frequently than others (as long as every state is updated infinitely often). Notice that in these in-place variants, the order that states are updated matters. In Question 22, updating states from the bottom left to the top right can result in a single sweep of the state space being sufficient for the algorithm to reach its fixed point, while updating states from the top left to bottom right will take many sweeps before convergence.

6.2 Policy Improvement

Once we have estimated \( v^\pi \) or \( q^\pi \) for some initial policy, \( \pi \), how can we find a new policy that is at least as good as \( \pi \)? Notice that if we have \( v^\pi \), we can easily compute \( q^\pi(s,a) \) for any \((s,a)\) by the equation

\[
q^\pi(s,a) = \sum_{s' \in S} P(s,a,s')(R(s,a) + \gamma v^\pi(s')).
\]

So, for now consider how we could find a policy \( \pi' \) that is always at least as good as \( \pi \) if we have already computed \( q^\pi \).

Consider a greedy approach, where we define \( \pi' \) to be a deterministic policy that selects the action that maximizes \( q^\pi(s,\cdot) \) when in state \( s \). That is, \( \pi' : S \rightarrow A \) is a deterministic policy defined such that

\[
\pi'(s) \in \arg \max_{a \in A} q^\pi(s,a).
\] (201)

This policy is the \textit{greedy policy with respect to} \( q^\pi \). It is greedy because it optimizes for the immediate future without considering long-term ramifications. Recall that \( q^\pi(s,a) \) is the expected discounted return if the agent takes action \( a \) in state \( s \) and follows the policy \( \pi \) thereafter. So, when \( \pi' \) chooses actions, \( a \), that maximize \( q^\pi(s,a) \), it not necessarily choosing actions that cause \( q^\pi'(s,a) \) or \( v^\pi'(s) \) to be maximized. It is choosing the actions that maximize the expected discounted return if the action is chosen at the current step, and then afterwards the policy \( \pi \) (not \( \pi' \)! ) is used. Can this greedy update to the policy, which only considers using \( \pi' \) to take a single action, cause \( \pi' \) to be worse than \( \pi \)?

Perhaps surprisingly, the greedy policy with respect to \( q^\pi \) is always at least as good as \( \pi \), i.e., \( \pi' \geq \pi \) (recall the definition of \( \geq \) for policies, presented in (163)). This result is described by the \textit{policy improvement theorem}, Theorem 3.

**Theorem 3** (Policy Improvement Theorem). For any policy \( \pi \), if \( \pi' \) is a deterministic policy such that \( \forall s \in S \),

\[
q^\pi(s,\pi'(s)) \geq v^\pi(s),
\] (202)

then \( \pi' \geq \pi \).
Proof.

\begin{align}
\pi(v(s)) & \leq q(s', \pi'(s)) \\
= & E[R_t + \gamma u(S_{t+1})] \mid S_t = s, \pi'] \\
\leq & E[R_t + \gamma q(S_{t+1}, \pi'(S_{t+1}))] \mid S_t = s, \pi'] \\
= & E[R_t + \gamma E[R_{t+1} + \gamma v(S_{t+2})] \mid S_t = s, \pi'] \\
= & E[R_t + \gamma R_{t+1} + \gamma^2 v(S_{t+2})] \mid S_t = s, \pi'] \\
\leq & E[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 v(S_{t+3})] \mid S_t = s, \pi'] \\
& \vdots \\
\leq & E[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^4 R_{t+3} + \cdots] \mid S_t = s, \pi'] \\
= & v'(s),
\end{align}

where each step follows from the previous using mixtures of the definitions of value functions and the assumption within the theorem statement. Notice that (204) conditions on \( \pi' \), not \( \pi \). This is because the only action that this conditioning impacts is \( A_t \)--all subsequent actions are captured by the \( v(S_{t+1}) \) term, which uses the policy \( \pi \). Notice also that the inner expectation in (206) does not condition on \( S_{t+1} \) taking a particular value, which one might expect given that it is an expansion of \( q(S_{t+1}, \pi'(S_{t+1})) \). This is because the state, \( S_{t+1} \), that \( q(S_{t+1}, \pi'(S_{t+1})) \) takes as its first argument is a random variable. So, the condition that one might expect in (206) when expanding \( q(S_{t+1}, \pi'(S_{t+1})) \) is that \( S_{t+1} = S_{t+1} \). This condition is a tautology, and so it can be ignored. \( \square \)

The policy improvement theorem also holds for stochastic greedy policies, as described in Theorem 4, for which we do not provide a proof.

**Theorem 4** (Policy Improvement Theorem for Stochastic Policies). For any policy \( \pi \), if \( \pi' \) satisfies

\begin{equation}
\sum_{a \in A} \pi'(s, a) q(s, a) \geq v(s),
\end{equation}

for all \( s \in S \), then \( \pi' \geq \pi \).
Figure 14: Diagram of the policy iteration algorithm. It begins with an arbitrary policy, \( \pi_0 \), and evaluates it using the dynamic programming approach described previously to produce \( v^{\pi_0} \). It then performs greedy policy improvement to select a new deterministic policy, \( \pi_1 \), that is at least as good as \( \pi_0 \). It then repeats this process, evaluating \( \pi_1 \), and using the resulting value function to obtain a new policy, \( \pi_2 \), etc.

<table>
<thead>
<tr>
<th>Algorithm 8: Policy iteration. This pseudocode assumes that policies are deterministic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize ( \pi_0 ) arbitrarily;</td>
</tr>
<tr>
<td>2. for ( i = 0 ) to ( \infty ) do</td>
</tr>
<tr>
<td>\hspace{1cm} /* Policy Evaluation */</td>
</tr>
<tr>
<td>\hspace{1cm} Initialize ( v_0 ) arbitrarily;</td>
</tr>
<tr>
<td>\hspace{1cm} for ( k = 0 ) to ( \infty ) do</td>
</tr>
<tr>
<td>\hspace{2cm} For all ( s \in S ):</td>
</tr>
<tr>
<td>\hspace{3cm} ( v_{k+1}(s) = \sum_{s' \in S} P(s, \pi_i(s), s') (R(s, \pi_i(s)) + \gamma v_k(s')) ) (213)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>\hspace{2cm} with ties broken by selecting actions according to some strict total order on ( A );</td>
</tr>
</tbody>
</table>

Notice that the number of deterministic policies for a finite MDP is finite. So, either policy iteration terminates after a finite number of iterations (if rewards are bounded and \( \gamma < 1 \)) or some policy must occur at least twice (there is a cycle in the sequence of policies). We now show that there cannot be a cycle of policies, so we can conclude that policy iteration terminates after a finite number
Theorem 5. It cannot occur that $\pi_j = \pi_k$ for $j \neq k$ when using the policy iteration algorithm.

Proof. This was not covered in lecture, and you are not responsible for it. We provide it for completeness.

We assume without loss of generality that $j < k$. We have from the policy improvement theorem that $\pi_j \leq \pi_{j+1} \leq \cdots \leq \pi_k$. Since $\pi_j = \pi_k$, and thus $v^{\pi_j} = v^{\pi_k}$, we therefore have that $v^{\pi_j} = v^{\pi_{j+1}} = v^{\pi_k}$. So (recall that the policies are deterministic policies):

\[
v^{\pi_j}(s) = v^{\pi_{j+1}}(s) = R(s, \pi_{j+1}(s)) + \sum_{s' \in S} P(s, \pi_{j+1}(s), s') \gamma v^{\pi_{j+1}}(s') \tag{215}\]

(a) comes from the Bellman equation, (b) holds because $v^{\pi_j} = v^{\pi_{j+1}}$, and (c) holds by the definition of $\pi_{j+1}$. Furthermore, by the Bellman equation for $v^{\pi_j}$ we have that:

\[
v^{\pi_j}(s) = R(s, \pi_j(s)) + \sum_{s' \in S} P(s, \pi_j(s), s') \gamma v^{\pi_j}(s'). \tag{219}\]

For (219) and (218) to hold simultaneously, we have that

\[
\max_{a \in A} R(s, a) + \sum_{s' \in S} P(s, a, s') \gamma v^{\pi_j}(s') = R(s, \pi_j(s)) + \sum_{s' \in S} P(s, \pi_j(s), s') \gamma v^{\pi_j}(s'), \tag{220}\]

and hence that

\[
\pi_j(s) \in \arg \max_{a \in A} R(s, a) + \sum_{s' \in S} P(s, a, s') \gamma v^{\pi_j}(s'). \tag{221}\]

However, this means that the termination condition for policy iteration would have been satisfied with $\pi_j$. \hfill \Box

Now that we know policy iteration terminates, consider what we know about the policy that it converges to. When this it terminates, we have that for all
where (a) is the Bellman equation (and where we view \( \pi_{i+1} \) as a distribution rather than a mapping from states to actions), (b) both follow from the starting assumption that the process has terminated, and so \( \pi_{i+1} = \pi_i \), and (c) comes from the fact that \( \pi_{i+1} \) is greedy with respect to \( v^{\pi_i} \). Since (225) is the Bellman optimality equation, \( \pi_{i+1} \) is an optimal policy—when policy iteration stops, the policy is optimal.

Notice also that the policy evaluation algorithm could terminate when \( v^{\pi_{i+1}} = v^{\pi_i} \). Using this termination condition would not require \( \pi_{i+1} \) to break ties in any particular order and is equivalent, but makes the analysis of the final policy less straightforward.

### 6.3 Value Iteration

Notice that the policy iteration algorithm is not efficient. Even though policy evaluation using dynamic programming is guaranteed to converge to \( v^{\pi} \), it is not guaranteed to reach \( v^{\pi} \), except in the limit as the number of iterations of policy evaluation goes to infinity. Thus, each iteration of the outer loop in policy iteration (the loop over \( i \)), may require an infinite amount of computation. An obvious question is whether or not the policy evaluation algorithm can be stopped early—when \( v_{k+1} \neq v_k \), but perhaps after some fixed number of iterations (i.e., the loop over \( k \) goes from 0 to \( K \), for some constant \( K \)).

If policy evaluation is stopped early, then the estimate of \( v^{\pi_i} \) will have error, and so the policy that is greedy with respect to the estimate of \( v^{\pi_i} \) may not be an improvement over the current policy. However, perhaps surprisingly, this process does still converge to an optimal policy. A particularly popular variant of this algorithm is value iteration, which uses \( K = 1 \)—it performs a single iteration of policy evaluation between policy improvement steps. Importantly, each iteration of policy evaluation begins with the value function estimate used in the previous step (rather than a random initial value function). Pseudocode
for value iteration is presented in Algorithm 9.

**Algorithm 9: Value Iteration.** This pseudocode is an inefficient implementation that we use as a stepping-stone to the common pseudocode.

1. Initialize $\pi_0$ and $v_0$ arbitrarily;
2. for $i = 0$ to $\infty$ do
   /* Policy Evaluation */
   3. For all $s \in \mathcal{S}$:
      $$v_{i+1}(s) = \sum_{s' \in \mathcal{S}} P(s, \pi_i(s), s') (R(s, \pi_i(s)) + \gamma v_i(s'))$$ (226)
   /* Check for Termination */
   4. if $v_{i+1} = v_i$ then terminate;
   /* Policy Improvement */
   5. Compute $\pi_{i+1}$ such that for all $s$,
      $$\pi_{i+1}(s) \in \arg \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s, a, s') (R(s, a) + \gamma v_{i+1}(s')).$$ (227)

If we were to following the pseudocode in Algorithm 9, we would obtain the following sequence of policies and value functions:

- $v_0: \text{arbitrary}$
- $\pi_0: \text{arbitrary}$
- $v_1 : \forall s, v_1(s) = \sum_{s' \in \mathcal{S}} P(s, \pi_0(s), s') (R(s, \pi_0(s)) + \gamma v_0(s'))$ (230)
- $\pi_1 : \forall s, \pi_1(s) \in \arg \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s, a, s') (R(s, a) + \gamma v_1(s'))$ (231)
- $v_2 : \forall s, v_2(s) = \sum_{s' \in \mathcal{S}} P(s, \pi_1(s), s') (R(s, \pi_1(s)) + \gamma v_1(s'))$ (232)
- $\pi_2 : \forall s, \pi_2(s) \in \arg \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s, a, s') (R(s, a) + \gamma v_2(s'))$ (233)
- $v_3 : \forall s, v_3(s) = \sum_{s' \in \mathcal{S}} P(s, \pi_2(s), s') (R(s, \pi_2(s)) + \gamma v_2(s'))$ (234)
- $\pi_3 : \forall s, \pi_3(s) \in \arg \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s, a, s') (R(s, a) + \gamma v_3(s'))$ (235)

... (236)

Notice the similarity between the updates to the policy and the value function estimate. When computing $v_2(s)$ we use $\pi_1(s)$, which is the action, $a$, that maximizes the expression on the right side of (231). This expression is the same
expression as that in the right side of (232). Thus, the expression for $v_2(s)$ can be written as:

$$v_2(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_1(s') \right). \quad (237)$$

This same trend holds for $v_3$ and $v_2$. In general, we can compute $v_{i+1}$ directly from $v_i$ without explicitly computing $\pi_i$. This results in the more efficient form for the value iteration algorithm:

$$v_{i+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_i(s') \right). \quad (238)$$

Notice that, while the policy evaluation algorithm is an iterative form for the Bellman equation, the value iteration update in (238) is an iterative form for the Bellman optimality equation. Pseudocode for the value iteration algorithm using this more efficient update is presented in Algorithm 10.

<table>
<thead>
<tr>
<th>Algorithm 10: Value Iteration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Initialize $v_0$ arbitrarily;</td>
</tr>
<tr>
<td>2 for $i = 0$ to $\infty$ do</td>
</tr>
<tr>
<td>3 /* Policy Evaluation */</td>
</tr>
<tr>
<td>4 For all $s \in S$:</td>
</tr>
<tr>
<td>5 $v_{i+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_i(s') \right). \quad (239)$</td>
</tr>
<tr>
<td>6 /* Check for Termination */</td>
</tr>
<tr>
<td>7 if $v_{i+1} = v_i$ then</td>
</tr>
<tr>
<td>8 terminate;</td>
</tr>
</tbody>
</table>

6.4 The Bellman Operator and Convergence of Value Iteration

In this subsection we prove that value iteration converges to a single unique value function. We then argue that this result implies all of the claims that we previously stated we would prove later: a deterministic optimal policy exists for all finite MDPs with bounded rewards and $\gamma < 1$, and the Bellman optimality equation only holds for $v^\pi$, where $\pi^*$ is an optimal policy.

Before we present the main theorem of this subsection, we will establish additional notation. First notice that, for finite MDPs, we can view value function estimates as vectors in $\mathbb{R}^{|S|}$, where each element in the vector corresponds to the value of one state. Also, recall that an operator is a function that takes elements of a space as input and produces elements of the same space as output. Let $T : \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ be an operator that we call the Bellman operator, which takes value function estimates as input and produces as output new value function

80
estimates, and such that
\[ T(v_i) := v_{i+1}, \]  
(240)
where the sequence of value function approximations, \( v_0, v_1, \ldots \), is as defined by (238). That is,
\[ T(v_i) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') \left( R(s, a) + \gamma v_i(s') \right). \]  
(241)
That is, the Bellman operator is the operator that encodes a single iteration of value iteration. We will abuse notation and omit parenthesis, writing \( T v_i = v_{i+1} \), and further assume that the order of operations prioritizes evaluation of the Bellman operator over evaluation of the value function approximation, so that \( T v(s) \) denotes \( T(v) \) evaluated at \( s \).

An operator is a contraction mapping if there exists a \( \lambda \in [0, 1) \) such that
\[ \forall x, y \in X, \forall f(x), f(y) \leq \lambda d(x, y), \]  
where \( d \) is a distance function. Figure 15 presents a diagram that may assist in understanding the definition of a contraction mapping.

Figure 15: Diagram to assist with interpreting the definition of a contraction mapping. Here \( x \) and \( y \) denote two points in the space \( X \). The function, \( f \), maps \( x \) to \( f(x) \), and \( y \) to \( f(y) \). If \( f \) is a contraction mapping, then for every possible \( x \) and \( y \), the distance between \( x \) and \( y \) (the length of the green dotted line) must be greater than the distance between \( f(x) \) and \( f(y) \) (the length of the dotted blue line). Moreover, the ratio of these distances must be at most \( \lambda \). For example, if \( \lambda = 0.5 \), then every application of \( f \) to two points, \( x \) and \( y \) must at least halve the distance between \( x \) and \( y \).

**Question 23.** If \( f \) is a contraction mapping, then is the sequence \( x_{i+1} = f(x_i) \) guaranteed to converge? Is it guaranteed to converge to a unique point within \( X \)?
As described in the answer to the above question, if $f$ is a contraction mapping then it is guaranteed to converge to a unique fixed point. This intuition is formalized by the Banach fixed-point theorem:

**Theorem 6 (Banach Fixed-Point Theorem).** If $f$ is a contraction mapping on a non-empty complete normed vector space, then $f$ has a unique fixed point, $x^*$, and the sequence defined by $x_{k+1} = f(x_k)$, with $x_0$ chosen arbitrarily, converges to $x^*$.

**Proof.** We do not provide a proof in this course. A proof can be found on Wikipedia. \hfill $\square$

We will apply the Banach fixed-point theorem where $f \leftarrow T$, $x \in \mathbb{R}^{|S|}$, and $d(v, v') := \max_{s \in S} |v(s) - v'(s)|$. That is, we will consider the max norm, $\|v - v'\|_{\infty} = \max_{s \in S} |v(s) - v'(s)|$. Recall that the max norm is the $p$-norm, with $p = \infty$. In order to apply the Banach fixed-point theorem, first notice that $\mathbb{R}^{|S|}$ is complete under the max-norm.\footnote{This follows from the Riesz-Fisher theorem, which implies that $L^p$ space is complete for $1 \leq p \leq \infty$.} We must also show that the Bellman operator is a contraction mapping—we show this in Theorem 7.

---

End of Lecture 12, October 10, 2019
Instructions: You have 5 minutes to complete this quiz. This quiz is closed notes—do not use your notes or a laptop. Do not discuss problems with your neighbors until after everyone has handed in their quiz.

Use the notation from class. Don’t forget to capitalize your random variables.

1. The definition of the state-value function is:

   \[ v^\pi(s) := \mathbb{E}[G_t | S_t = s, \pi] \]

2. The definition of the action-value function is:

   \[ q^\pi(s, a) := \mathbb{E}[G_t | S_t = s, A_t = a, \pi] \]

3. The Bellman equation states that for all \( s \):

   \[ v^\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P(s, a, s') (R(s, a) + \gamma v^\pi(s')) \]

4. The Bellman optimality equation states that for all \( s \):

   \[ v^*(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') (R(s, a) + \gamma v^*(s')) \]

5. (True or False) Although there may be multiple optimal policies, there is only one optimal value function for a finite MDP with bounded rewards.
The Bellman operator is a contraction mapping. The Bellman operator is a contraction mapping on $\mathbb{R}^{|S|}$ with $d(v, v') := \max_{s \in S} |v(s) - v'(s)|$.

Proof.

\[ ||Tv - Tv'|| = \max_{s \in S} |Tv(s) - Tv'(s)| \tag{242} \]

\[ = \max_{s \in S} \left| \max_{a \in A} \sum_{s' \in S} P(s, a, s')(R(s, a) + \gamma v(s')) - \max_{a \in A} \sum_{s' \in S} P(s, a, s')(R(s, a) + \gamma v'(s')) \right|, \tag{243} \]

by the definition of the Bellman operator. To continue, we derive a relevant property of arbitrary functions, $f : \mathcal{X} \to \mathbb{R}$ and $g : \mathcal{S} \to \mathbb{R}$, for arbitrary sets, $\mathcal{X}$. We begin with a simple expression and then list inequalities implied by the preceding inequalities to obtain the desired expression:

\[ \forall x, f(x) - g(x) \leq |f(x) - g(x)| \tag{244} \]

\[ \forall x, f(x) \leq |f(x) - g(x)| + g(x) \tag{245} \]

\[ \max_{x \in \mathcal{X}} f(x) \leq \max_{x \in \mathcal{X}} |f(x) - g(x)| + g(x) \tag{246} \]

\[ \max_{x \in \mathcal{X}} f(x) \leq \max_{x \in \mathcal{X}} |f(x) - g(x)| + \max_{x \in \mathcal{X}} g(x) \tag{247} \]

\[ \max_{x \in \mathcal{X}} f(x) - \max_{x \in \mathcal{X}} g(x) \leq \max_{x \in \mathcal{X}} |f(x) - g(x)|. \tag{248} \]

If $\max_{x \in \mathcal{X}} f(x) - \max_{x \in \mathcal{X}} g(x) \geq 0$, then it follows from (248) that

\[ \left| \max_{x \in \mathcal{X}} f(x) - \max_{x \in \mathcal{X}} g(x) \right| \leq \max_{x \in \mathcal{X}} |f(x) - g(x)|. \tag{249} \]

If $\max_{x \in \mathcal{X}} f(x) - \max_{x \in \mathcal{X}} g(x) < 0$, then we have from (248) that:

\[ \max_{x \in \mathcal{X}} g(x) - \max_{x \in \mathcal{X}} f(x) \leq \max_{x \in \mathcal{X}} |g(x) - f(x)|, \tag{250} \]

which also implies (249), since $\max_{x \in \mathcal{X}} f(x) - \max_{x \in \mathcal{X}} g(x) \geq 0$ and $|f(x) - g(x)| = |g(x) - f(x)|$. Applying (249) to (243), we obtain:

\[ ||Tv - Tv'|| \leq \max_{s \in S} \max_{a \in A} \sum_{s' \in S} P(s, a, s')(R(s, a) + \gamma v(s')) - \sum_{s' \in S} P(s, a, s')(R(s, a) + \gamma v'(s')) \tag{251} \]

\[ = \gamma \max_{s \in S} \max_{a \in A} \sum_{s' \in S} P(s, a, s')(v(s') - v'(s')) \tag{252} \]

\[ = \gamma \max_{s \in S} \max_{a \in A} \sum_{s' \in S} P(s, a, s') |v(s') - v'(s')| \tag{253} \]

\[ \leq \gamma \max_{s \in S} \max_{a \in A} \max_{s' \in S} |v(s') - v'(s')| \tag{254} \]

\[ = \gamma \max_{s' \in S} |v(s') - v'(s')| \tag{255} \]

\[ = \gamma \|v - v'\|. \tag{256} \]
Thus, we have that the Bellman operator is a contraction mapping, and so by the Banach fixed point theorem it follows that the value iteration algorithm converges to a unique fixed point, which we denote here by $v^\infty$.

**Theorem 8.** Value iteration converges to a unique fixed point $v^\infty$ for all MDPs with finite state and action sets, bounded rewards, and $\gamma < 1$.

**Proof.** This follows from the Banach fixed point theorem (Theorem 6) and the fact that the Bellman operator (which encodes the value iteration update) is a contraction (Theorem 7). □

Although we do not provide a proof, policy iteration and value iteration both converge in a number of iterations that is polynomial in $|S|$ and $|A|$. Notice also that the Bellman operator is a contract with parameter $\gamma$—as $\gamma$ approaches one the speed of convergence slows, while small values for $\gamma$ speed up convergence. This is intuitive because small $\gamma$ mean that events that occur in the distant future are of little importance, and so the value function will become accurate after fewer backups.

We can now establish that the existence of deterministic optimal policies:

**Theorem 9.** All MDPs with finite state and action sets, bounded rewards, and $\gamma < 1$ have at least one optimal policy.

**Proof.** By Theorem 8 we have that value iteration converges to a unique fixed point $v^\infty$. Consider any deterministic policy $\pi^\infty$ satisfying:

$$\pi^\infty(s) \in \arg\max_{a \in A} \sum_{s' \in S} P(s, a, s') (R(s, a) + \gamma v^\infty(s)).$$

At least one such policy exists, since $A$ is a finite set. Recall that value iteration corresponds to one iteration of policy iteration, but where policy evaluation only conducts a single full backup. This $\pi^\infty$ is the greedy policy with respect to $v^\infty$. Since $v^\infty$ is a fixed point of value iteration, performing one full backup of policy evaluation for $\pi^\infty$ results in $v^\infty$ again. This means that $v^\infty$ is a fixed-point of policy evaluation for $\pi^\infty$. That is:

$$v^\infty(s) = \sum_{s' \in S} P(s, \pi^\infty(s), s')(R(s, \pi^\infty(s)) + \gamma v^\infty(s')).$$

As this is the Bellman equation, we have that $v^\infty$ is the state-value function for $\pi^\infty$. Next, since $v^\infty$ is a fixed point of the value iteration algorithm, we have that for all $s \in S$:

$$v^\infty(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s')(R(s, a) + \gamma v^\infty(s)),$$

which is the Bellman optimality equation. Since $v^\infty$ is the value function for $\pi^\infty$, we therefore have that $\pi^\infty$ satisfies the Bellman optimality equation. We showed in Theorem 1 that any policy satisfying the Bellman optimality equation is an optimal policy, and so we have that $\pi^\infty$ is an optimal policy. □
7 Monte Carlo Methods

Monte Carlo algorithms, which have a history worth reading about, use randomness to solve problems that are deterministic in principle. A classical example is the estimation of $\pi$. Consider the unit circle, drawn with its center at the bottom left corner of a unit square. The percent of the area inside the square that is also inside the circle is $\pi/4$. Hence, once can estimate $\pi$ by throwing darts at the unit square (such that the darts land with a uniform distribution over the square). An estimate of $\pi$ is then given by:

$$\pi \approx 4 \frac{\text{number of darts inside the unit square and the circle}}{\text{number of darts inside the unit square}}. \quad (260)$$

In this example, the random dart throws are used to provide an (unbiased) estimate of the deterministic value, $\pi$.

7.1 Monte Carlo Policy Evaluation

Consider using a Monte Carlo approach to estimate the state-value function for a policy $\pi$. In this case, a “dart throw” corresponds to sampling history (running an episode) using the policy, $\pi$. More specifically, consider the task of estimating the value of a state, $s \in S$. If we generate a history, $H = (S_0, A_0, R_0, S_1, A_1, R_1, \ldots)$, starting from $S_0 = s$, how can we construct an unbiased estimator of $v^\pi(s)$?

Question 24. Which of the following three estimators of $v^\pi(s)$ is an unbiased estimator?

1. $\sum_{k=0}^{\infty} \gamma^k R_k$.
2. $\sum_{k=0}^{\infty} \gamma^k R_{t_{\text{last}} + k}$, where $t_{\text{last}}$ is the last time step where the state was $s$.
3. Other estimators that average the returns from each occurrence of $s$ within the history.
Part One: Written (50 Points Total)

1. (3 Points) One day while working in the engineering department of the Starship Enterprise, your friend Geordi comes to you with an idea. He points out that the warp core (engine) uses a reinforcement learning algorithm to regulate its temperature. He hypothesizes that the value function that it uses would be easier to represent and/or faster to approximate in two distinct parts: one that estimates the value of a state given that the next state is safe (within desirable thresholds), and another that estimates the value of a state given that the next state is not safe. Working with Geordi, who of course uses the notation from this class, you decide to define $X$ to be the set of safe states, and $X^\complement$ to be the set of unsafe states, i.e., $X^\complement = S \setminus X$. In order to continue, you and Geordi decide to establish some notation. Specifically, you want to define $v^\pi_Y(s)$ to be the expected discounted return given that the agent begins in state $s$, follows policy $\pi$, and the next state (but not necessarily the states after the next state) happens to be in $Y$. Give a mathematical definition for $v^\pi_Y(s)$ like our definition for $v^\pi$:

$$v^\pi_Y(s) :=$$

(261)

2. (5 Points) Having defined $v^\pi_Y$, you decide to relate your new value functions, $v^\pi_X$ and $v^\pi_{X^\complement}$, to the standard value function, $v^\pi$. Derive an expression for $v^\pi(s)$ that only uses the following terms: $\pi, P, A, S, X, v^\pi_X$ and $v^\pi_{X^\complement}$. Note: You may introduce variables when summing over sets, e.g., $x$ in $\sum_{x \in X}$. Your final answer should not include expectations or any random variables like $S_t$ or $R_t$. You should begin with the definition of $v^\pi(s)$ and end with an expression that only contains the allowed terms. Show your work (show the steps, don’t just jump to your final answer). You may want to derive some properties before proceeding with the derivation for $v^\pi(s)$—that is allowed.

$$v^\pi(s) =$$

(262)

---

6In latex, here we are using the symbols \complement and \setminus for $\mathbb{C}$ and \ respectively.
3. (13 Points) Having related your new value functions to the standard value function, you now talk to Geordi about what to do next to design a reinforcement learning algorithm using your new value functions. Another friend named Data loads the course notes from CMPSCI 687 in Fall 2019. He finds that the next step towards developing an algorithm with this value function may be to write out a new Bellman equation for $v^\pi_X$. Derive a Bellman-like equation for this new value function. You should begin with the definition of $v^\pi_X$ according to your answer to the first question, and should end with a recursive expression for $v^\pi_X$ that is written only in terms of $S, A, P, R, d_0, \gamma, \pi, X$, and $X^\mathcal{C}$. For this problem, use an alternate definition of $R$: $R(s, a, s') := \mathbb{E}[R_t | S_t = s, A_t = a, S_{t+1} = s']$. (Hint: Using font size “tiny”, our answer spans two lines—do not expect a short answer).

$$v^\pi_X(s) = \text{expression}(263)$$

4. (5 Points) Consider the following definition of an optimal policy:

For any finite MDP with $\gamma < 1$ and precisely two actions, $a_1$ and $a_2$, for any two policies $\pi$ and $\pi'$, $\pi \geq \pi'$ iff $\forall s \in S$, $q^\pi(s, a_1) \geq q^{\pi'}(s, a_1)$. A policy $\pi$ is optimal iff $\pi \geq \pi'$ for all policies $\pi'$.

Is this definition equivalent to the definition from Section 4.5 in the course notes? Prove that your answer is correct.

5. (5 Points) Consider a different definition of $\geq$ for policies: $\pi \geq \pi'$ iff $\sum_{s \in S} d_0(s) v^\pi(s) \geq \sum_{s \in S} d_0(s) v^{\pi'}(s)$. Using this modified version of $\geq$, we can still define an optimal policy to be any policy $\pi$ such that $\pi \geq \pi'$ for all $\pi'$. Prove that using this definition of an optimal policy is equivalent to using our first definition:

$$\pi^* \in \arg \max_{\pi \in \Pi} J(\pi). \quad (264)$$

6. (20 Points) In class we proved that the Bellman operator is a contraction, and used this to show that value iteration converges to a unique fixed point. In this problem you will prove that the dynamic programming policy evaluation operator is a contraction, and so the policy evaluation algorithm converges to a unique fixed-point. (From the Bellman equation, it should then be clear that this fixed point is $v^\pi$, establishing that our dynamic programming policy evaluation algorithm converges to $v^\pi$.) Let $f$ denote the dynamic programming policy evaluation operator (this is (200), viewed as an operator on value function approximations):

$$fv(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P(s, a, s') (R(s, a) + \gamma v(s')). \quad (265)$$

Notice that the definition of $f$ relies on a specific policy $\pi$—this is the policy being evaluated by the policy evaluation algorithm. Prove that $f$ is
a contraction under the $L^\infty$ norm (the same max norm used in our proof that the Bellman operator is a contraction).

Part Two: Programming (25 Points Total)

For this part of the assignment, you will implement value iteration (modified to terminate when the value function estimate has not changed significantly between two iterations). Your program will read an MDP from a file, run value iteration on the MDP, and output the final estimate of the optimal value function and the policies that are greedy with respect to this value function. As a soft introduction to C++, we are providing you with most of the code here: your job is to fill in the missing lines in the function `valueIteration`, marked with a comment saying “TODO”. Do not change the code logic outside of the `valueIteration` function (you may add new functions if you like, but do not modify any of the other functions in your final submission or it may fail to run as expected in our auto-grader).

You are free to use any IDE or toolchain you would like to program in C++. If you are not familiar with C++, we have provided two different systems for opening and working with this C++ code. If you are using Windows, you should download Microsoft Visual Studio. The community version is perfectly sufficient, and is free online (in my opinion, this is the best C++ experience out there). Clicking on the .sln file in HW3/build/VisualStudio will open the project. On the left you should see main.cpp—open this file to see all of the code for this assignment. If you are using Mac or Linux, we have provided a CLion project. CLion is free for students. To open this project, select “Open” when launching CLion. Select the file HW3/build/CLion/CMakeLists.txt. When prompted, select “Open as Project”. If main.cpp does not immediately open, on the left click on HW3/main.cpp.

This assignment is your chance to begin to familiarize yourself with C++. Please look over all of the provided code, and feel free to ask if you have questions about what some portion of the code is doing. Also, take this opportunity to familiarize yourself with the debugger in your IDE—developing simple programs in C++ is a breeze when you are familiar with how to use the different capabilities of your debugger.

We have provided you (within the provided code) with 687Gridworld.txt, a text file containing the MDP we have been using in class. We will evaluate your program on other MDPs that we are not providing to you. You are welcome to create your own test MDPs, but do not share these with others.

You must submit your main.cpp file. A correct implementation is worth 20 points. Any incorrect output (beyond numerical issues) will result in 0/20 points. In the .pdf that you submit, answer the following questions.

1. (2 Points) Did your final code compile on your machine? (Yes or no).
2. (3 Points) Comment on your experience with this problem. Did your first implementation work, or did you introduce a bug at first? Was there anything we could do to smooth your introduction to C++? Did you implement any additional test MDPs (you do not have to in order to get full credit). Did the number of iterations required by value iteration surprise you? Do you have any other comments on this problem?
References


