Reinforcement Learning

Piyush Rai

Introduction to Machine Learning (CS771A)

November 6, 2018
Supervised Learning: Uses **explicit supervision** (input-output pairs)

In many learning problems that need supervision, it is hard to provide explicit supervision.
Reinforcement Learning

- Supervised Learning: Uses explicit supervision (input-output pairs)
- In many learning problems that need supervision, it is hard to provide explicit supervision
- Example: Learning a policy/strategy of an agent acting in an environment
Reinforcement Learning

- Supervised Learning: Uses **explicit supervision** (input-output pairs)
- In many learning problems that need supervision, it is hard to provide explicit supervision
- Example: Learning a policy/strategy of an **agent** acting in an **environment**

- Agent lives in **states**, takes **actions**, gets **rewards**, and goal is to maximize its **total reward**
Reinforcement Learning

- Supervised Learning: Uses explicit supervision (input-output pairs)
- In many learning problems that need supervision, it is hard to provide explicit supervision
- Example: Learning a policy/strategy of an agent acting in an environment

Agent lives in states, takes actions, gets rewards, and goal is to maximize its total reward

Reinforcement Learning (RL) is a way to solve such problems
Supervised Learning: Uses explicit supervision (input-output pairs)

In many learning problems that need supervision, it is hard to provide explicit supervision

Example: Learning a policy/strategy of an agent acting in an environment

Agent lives in states, takes actions, gets rewards, and goal is to maximize its total reward

Reinforcement Learning (RL) is a way to solve such problems

Many applications: Robotics, autonomous driving, computer game playing, online advertising, etc.
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions

States = [Standing, Moving, Fallen]
Actions = [Slow, Fast]

\[
T(Standing, Slow, Moving) = 1 \\
T(Standing, Fast, Moving) = 0.6 \\
T(Standing, Fast, Fallen) = 0.4 \\
... \\
T(Falling, Slow, Moving) = 0.6 \\
T(Falling, Fast, Fallen) = 0.4 \\
... \\
\]
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states

States = [Standing, Moving, Fallen]
Actions = [Slow, Fast]

\[T(Standing, Slow, Moving) = 1\]
\[T(Standing, Fast, Moving) = 0.6\]
\[T(Standing, Fast, Fallen) = 0.4\]
...
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)

\[\begin{align*}
\text{States} &= \text{[Standing, Moving, Fallen]} \\
\text{Actions} &= \text{[Slow, Fast]} \\
T(\text{Standing, Slow, Moving}) &= 1 \\
T(\text{Standing, Fast, Moving}) &= 0.6 \\
T(\text{Standing, Fast, Fallen}) &= 0.4 \\
\text{...}
\end{align*}\]
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)
  - For discrete states, \(P_{sa}\) is an \(|S|\) length probability vector

States = [Standing, Moving, Fallen]
Actions = [Slow, Fast]

\[
\begin{align*}
T(Standing, Slow, Moving) &= 1 \\
T(Standing, Fast, Moving) &= 0.6 \\
T(Standing, Fast, Fallen) &= 0.4 \\
&\quad \ldots
\end{align*}
\]
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)
  - For discrete states, \(P_{sa}\) is an \(|S|\) length probability vector
  - Another notation for \(P_{sa}(s')\): \(T(s, a, s')\)

States = [Standing, Moving, Fallen]
Actions = [Slow, Fast]

\[
\begin{align*}
T(\text{Standing, Slow, Moving}) &= 1 \\
T(\text{Standing, Fast, Moving}) &= 0.6 \\
T(\text{Standing, Fast, Fallen}) &= 0.4 \\
&\ldots
\end{align*}
\]
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)
  - For discrete states, \(P_{sa}\) is an \(|S|\) length probability vector
  - Another notation for \(P_{sa}(s')\): \(T(s, a, s')\)
- \(R : S \times A \rightarrow \mathbb{R}\) is the reward function
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)
  - For discrete states, \(P_{sa}\) is an \(|S|\) length probability vector
  - Another notation for \(P_{sa}(s')\): \(T(s, a, s')\)
- \(R : S \times A \rightarrow \mathbb{R}\) is the reward function
  - Reward for reaching state \(s\): \(R(s, a)\)

Intro to Machine Learning (CS771A)
Reinforcement Learning
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple \((S, A, \{P_{sa}\}, \gamma, R)\)
- \(S\) is a set of states (discrete or continuous valued)
- \(A\) is a set of actions
- \(P_{sa}\) is a probability distribution over possible states
  - \(P_{sa}(s')\): Prob. of switching to \(s'\) if we took action \(a\) in \(s\)
  - For discrete states, \(P_{sa}\) is an \(|S|\) length probability vector
  - Another notation for \(P_{sa}(s')\): \(T(s, a, s')\)
- \(R : S \times A \rightarrow \mathbb{R}\) is the reward function
  - Reward for reaching state \(s\): \(R(s, a)\)
- \(\gamma \in [0, 1)\) is called discount factor for future rewards

Intro to Machine Learning (CS771A)
Markov Decision Processes (MDP)

- MDP gives a formal way to define RL problems
- An MDP consists of a tuple $(S, A, \{P_{sa}\}, \gamma, R)$
- $S$ is a set of states (discrete or continuous valued)
- $A$ is a set of actions
- $P_{sa}$ is a probability distribution over possible states
  - $P_{sa}(s')$: Prob. of switching to $s'$ if we took action $a$ in $s$
  - For discrete states, $P_{sa}$ is an $|S|$ length probability vector
  - Another notation for $P_{sa}(s')$: $T(s, a, s')$
- $R: S \times A \mapsto \mathbb{R}$ is the reward function
  - Reward for reaching state $s$: $R(s, a)$
- $\gamma \in [0, 1)$ is called discount factor for future rewards
- $P_{sa}$ and $R$ may be unknown (may need to be learned)

States = [Standing, Moving, Fallen]
Actions = [Slow, Fast]
$T($Standing, Slow, Moving$) = 1$
$T($Standing, Fast, Moving$) = 0.6$
$T($Standing, Fast, Fallen$) = 0.4$
...
Payoff and Expected Payoff

- Payoff defines the cumulative reward

Payoff upon visiting states $s_0, s_1, ...$ with actions $a_0, a_1, ...$ is:

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

Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)

We care more about immediate rewards, rather than the future rewards.

If rewards defined in terms of states only, then the payoff:

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

We want to choose actions over time (by learning a “policy”) to maximize the expected payoff:

$$E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...]$$

The expectation $E$ is w.r.t. all possibilities of the initial state $s_0$. 
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:
  \[ R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \]
- Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:
  \[ R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \]
- Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)
  - We care more about immediate rewards, rather than the future rewards
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:

\[ R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \]

- Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)
  - We care more about immediate rewards, rather than the future rewards
- If rewards defined in terms of states only, then the payoff:

\[ R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \]
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:
  \[ R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \]
- Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)
  - We care more about immediate rewards, rather than the future rewards
- If rewards defined in terms of states only, then the payoff:
  \[ R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \]
- We want to choose actions over time (by learning a “policy”) to maximize the expected payoff
  \[ \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots] \]
Payoff and Expected Payoff

- Payoff defines the cumulative reward
- Upon visiting states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, the payoff:
  \[ R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \]
- Reward at time $t$ is discounted by $\gamma^t$ (note: $\gamma < 1$)
  - We care more about immediate rewards, rather than the future rewards
- If rewards defined in terms of states only, then the payoff:
  \[ R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \]
- We want to choose actions over time (by learning a “policy”) to maximize the expected payoff
  \[ \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots] \]
- The expectation $\mathbb{E}$ is w.r.t. all possibilities of the initial state $s_0$
Policy Function and Value Function

- Policy Function or Policy is a function $\pi : S \mapsto A$, mapping from states to actions.
Policy Function and Value Function

- Policy Function or **Policy** is a function $\pi : S \mapsto A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$.
Policy Function and Value Function

- Policy Function or **Policy** is a function $\pi : S \mapsto A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$. Want to learn the best $\pi$. 

*Bellman's Equation:* 
$$V_\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi}(s) \cdot V_\pi(s')$$ 
It's the immediate reward + expected sum of future discounted rewards.
Policy Function and Value Function

- Policy Function or **Policy** is a function \( \pi : S \mapsto A \), mapping from states to actions.
- For an agent with policy \( \pi \), the action in state \( s \): \( a = \pi(s) \). Want to learn the best \( \pi \).
- For any policy \( \pi \), we can define the **Value Function**

\[
V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi]
\]
Policy Function and Value Function

- Policy Function or Policy is a function $\pi : S \rightarrow A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$. Want to learn the best $\pi$.
- For any policy $\pi$, we can define the Value Function

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi]$$

- $V^\pi(s)$ is the expected payoff starting in state $s$ and following policy $\pi$. 
Policy Function and Value Function

- Policy Function or Policy is a function $\pi : S \mapsto A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$. Want to learn the best $\pi$.
- For any policy $\pi$, we can define the Value Function $V^\pi(s)$:
  \[ V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi] \]
  - $V^\pi(s)$ is the expected payoff starting in state $s$ and following policy $\pi$.
- For finite state spaces, $V^\pi(s)$ will be a vector of size $|S|$. 

Bellman's Equation:

Gives a recursive definition of the above Value Function:

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

It's the immediate reward + expected sum of future discounted rewards.
Policy Function and Value Function

- Policy Function or **Policy** is a function \( \pi : S \mapsto A \), mapping from states to actions.
- For an agent with policy \( \pi \), the action in state \( s \): \( a = \pi(s) \). Want to learn the best \( \pi \).
- For any policy \( \pi \), we can define the **Value Function**

\[
V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi]
\]

- \( V^\pi(s) \) is the expected payoff starting in state \( s \) and following policy \( \pi \).
- For finite state spaces, \( V^\pi(s) \) will be a vector of size \( |S| \).
- **Bellman’s Equation**: Gives a recursive definition of the above Value Function:

\[
V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi}(s')(s') \times V^\pi(s')
\]
Policy Function and Value Function

- Policy Function or Policy is a function $\pi : S \mapsto A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$. Want to learn the best $\pi$.
- For any policy $\pi$, we can define the Value Function

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi]$$

- $V^\pi(s)$ is the expected payoff starting in state $s$ and following policy $\pi$.
- For finite state spaces, $V^\pi(s)$ will be a vector of size $|S|$.
- Bellman’s Equation: Gives a recursive definition of the above Value Function:

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

$$= R(s) + \gamma \mathbb{E}_{s' \sim P_{s\pi}(s)} [V^\pi(s')]$$

---

Intro to Machine Learning (CS771A)
Reinforcement Learning
Policy Function and Value Function

- Policy Function or **Policy** is a function $\pi : S \mapsto A$, mapping from states to actions.
- For an agent with policy $\pi$, the action in state $s$: $a = \pi(s)$. Want to learn the best $\pi$.
- For any policy $\pi$, we can define the **Value Function**

$$V^\pi(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots | s_0 = s, \pi]$$

- $V^\pi(s)$ is the expected payoff starting in state $s$ and following policy $\pi$.
- For finite state spaces, $V^\pi(s)$ will be a vector of size $|S|$.
- **Bellman’s Equation**: Gives a recursive definition of the above Value Function:

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

$$= R(s) + \gamma \mathbb{E}_{s' \sim P_{s\pi}(s')} [V^\pi(s')]$$

- It’s the immediate reward + expected sum of future discounted rewards.
Computing the Value Function

- Given $\pi$, Bellman’s equation can be used to compute the value function $V^\pi(s)$

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

For finite-state MDP, it gives us $|S|$ equations with $|S|$ unknowns $\Rightarrow$ Efficiently solvable

The Optimal Value Function is defined as

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

It’s the best possible expected payoff that any policy $\pi$ can give.

The Optimal Value Function can also be defined as:

$$V^*(s) = R(s) + \max_a \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$$
Computing the Value Function

Given $\pi$, Bellman’s equation can be used to compute the value function $V^\pi(s)$

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

For finite-state MDP, it gives us $|S|$ equations with $|S|$ unknowns $\Rightarrow$ Efficiently solvable

The Optimal Value Function is defined as $V^*(s) = \max_{\pi} V^\pi(s)$

It's the best possible expected payoff that any policy $\pi$ can give
Computing the Value Function

- Given $\pi$, Bellman’s equation can be used to compute the value function $V^\pi(s)$

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

- For finite-state MDP, it gives us $|S|$ equations with $|S|$ unknowns $\Rightarrow$ Efficiently solvable

- The **Optimal Value Function** is defined as

$$V^*(s) = \max_{\pi} V^\pi(s)$$
Computing the Value Function

- Given $\pi$, Bellman’s equation can be used to compute the value function $V^\pi(s)$

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

- For finite-state MDP, it gives us $|S|$ equations with $|S|$ unknowns $\Rightarrow$ Efficiently solvable

- The **Optimal Value Function** is defined as

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

- It’s the **best possible expected payoff** that any policy $\pi$ can give
Computing the Value Function

- Given $\pi$, Bellman’s equation can be used to compute the value function $V^\pi(s)$

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

- For finite-state MDP, it gives us $|S|$ equations with $|S|$ unknowns $\Rightarrow$ Efficiently solvable

- The **Optimal Value Function** is defined as

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

- It’s the **best possible expected payoff** that any policy $\pi$ can give

- The Optimal Value Function can also be defined as:

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

- The **Optimal Value Function**:

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

Given the optimal value function \( V^* \), the **Optimal Policy** \( \pi^* \):

\[
\pi^*(s) = \text{arg max}_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
\]

Three popular methods to find the optimal policy:

- **Value Iteration**: Estimate \( V^* \) and then use Eq 1
- **Policy Iteration**: Iterate between learning the optimal policy \( \pi^* \) and learning \( V^* \)
- **Q Learning** (a variant of value iteration)
Optimal Policy

- The **Optimal Value Function**:

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

- Given the optimal value function \( V^* \), the **Optimal Policy** \( \pi^* : S \mapsto A \):

  \[
  \pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
  \]

  (1)
Optimal Policy

- The **Optimal Value Function**:

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

- Given the optimal value function \( V^* \), the **Optimal Policy** \( \pi^* : S \mapsto A \):

\[
\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
\]

- \( \pi^*(s) \) gives the action \( a \) that maximizes the optimal value function for that state.
Optimal Policy

- The **Optimal Value Function**: 

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

- Given the optimal value function \( V^* \), the **Optimal Policy** \( \pi^* : S \mapsto A \):

\[
\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
\]

- \( \pi^*(s) \) gives the action \( a \) that maximizes the optimal value function for that state

- Three popular methods to find the optimal policy:
  - **Value Iteration**: Estimate \( V^* \) and then use Eq 1
  - **Policy Iteration**: Iterate between learning the optimal policy \( \pi^* \) and learning \( V^* \)
  - **Q Learning** (a variant of value iteration)
The **Optimal Value Function**:

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

Given the optimal value function \(V^*\), the **Optimal Policy** \(\pi^* : S \mapsto A\):

\[
\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
\]  

\(\pi^*(s)\) gives the action \(a\) that maximizes the optimal value function for that state.

- Three popular methods to find the optimal policy
  - **Value Iteration**: Estimate \(V^*\) and then use Eq 1

Optimal Policy

- **The Optimal Value Function:**

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

- Given the optimal value function \( V^* \), the **Optimal Policy** \( \pi^* : S \mapsto A \):

\[
\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')
\] (1)

- \( \pi^*(s) \) gives the action \( a \) that maximizes the optimal value function for that state

- Three popular methods to find the optimal policy
  - **Value Iteration:** Estimate \( V^* \) and then use Eq 1
  - **Policy Iteration:** Iterate between learning the optimal policy \( \pi^* \) and learning \( V^* \)
  - **Q Learning** (a variant of value iteration)
Finding the Optimal Policy: Value Iteration

- Iteratively compute the optimal value function $V^*$ as follows:
  
  For each state $s$, initialize $V(s) = 0$
  
  Repeat until convergence:
  - For each state $s$, update $V(s)$ as:
    $$V(s) = R(s) + \max_{a \in A} \sum_{s'} P_{s,a}(s') V(s')$$
  
  Value Iteration property: $V$ converges to $V^*$

  Upon convergence, use:
  $$\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{s,a}(s') V^*(s')$$

  Note: The inner loop can update $V(s)$ for all states simultaneously, or in some order.
Finding the Optimal Policy: Value Iteration

- Iteratively compute the optimal value function $V^*$ as follows
  - For each state $s$, initialize $V(s) = 0$
Finding the Optimal Policy: Value Iteration

- Iteratively compute the optimal value function $V^*$ as follows
  - For each state $s$, initialize $V(s) = 0$
  - Repeat until convergence
Finding the Optimal Policy: Value Iteration

Iteratively compute the optimal value function $V^*$ as follows

- For each state $s$, initialize $V(s) = 0$
- Repeat until convergence
  - For each state $s$, update $V(s)$ as
    $$V(s) = R(s) + \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

Value Iteration property: $V$ converges to $V^*$

Upon convergence, use
$$\pi^*(s) = \arg \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

Note: The inner loop can update $V(s)$ for all states simultaneously, or in some order.
Finding the Optimal Policy: Value Iteration

- Iteratively compute the optimal value function $V^*$ as follows
  - For each state $s$, initialize $V(s) = 0$
  - Repeat until convergence
    - For each state $s$, update $V(s)$ as
      $$V(s) = R(s) + \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

- Value Iteration property: $V$ converges to $V^*$
Finding the Optimal Policy: Value Iteration

- Iteratively compute the **optimal** value function $V^*$ as follows
  - For each state $s$, initialize $V(s) = 0$
  - Repeat until convergence
    - For each state $s$, update $V(s)$ as
      $$V(s) = R(s) + \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

- **Value Iteration property:** $V$ converges to $V^*$

- Upon convergence, use $\pi^*(s) = \arg \max_{a \in A} \sum_{s'} P_{sa}(s') V^*(s')$
Finding the Optimal Policy: Value Iteration

- Iteratively compute the optimal value function $V^*$ as follows
  - For each state $s$, initialize $V(s) = 0$
  - Repeat until convergence
    - For each state $s$, update $V(s)$ as
      $$V(s) = R(s) + \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$$

- Value Iteration property: $V$ converges to $V^*$

- Upon convergence, use $\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')$

- **Note:** The inner loop can update $V(s)$ for all states simultaneously, or in some order
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy $\pi$ until convergence
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy \( \pi \) until convergence
  - Initialize \( \pi \) randomly
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy $\pi$ until convergence
  - Initialize $\pi$ randomly
  - Repeat until convergence

Step (1) computes the value function for the current policy $\pi$. Can be done using Bellman's equations (solving $|S|$ equations in $|S|$ unknowns). Step (2) gives the policy that is greedy w.r.t. $V$.
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy $\pi$ until convergence
  - Initialize $\pi$ randomly
  - Repeat until convergence
    - Let $V = V^\pi$
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy $\pi$ until convergence
  - Initialize $\pi$ randomly
  - Repeat until convergence
    1. Let $V = V^\pi$
    2. For each state $s$, set $\pi(s) = \arg \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$
Finding the Optimal Policy: Policy Iteration

• Iteratively compute the policy $\pi$ until convergence
  • Initialize $\pi$ randomly
  • Repeat until convergence
    1. Let $V = V^\pi$
    2. For each state $s$, set $\pi(s) = \arg\max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$

• Step (1) the computes the value function for the current policy $\pi$
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy \( \pi \) until convergence
  - Initialize \( \pi \) randomly
  - Repeat until convergence
    1. Let \( V = V^\pi \)
    2. For each state \( s \), set \( \pi(s) = \arg \max_{a \in A} \sum_{s'} P_{sa}(s') V(s') \)

- Step (1) computes the value function for the current policy \( \pi \)
  - Can be done using Bellman’s equations (solving \(|S|\) equations in \(|S|\) unknowns)
Finding the Optimal Policy: Policy Iteration

- Iteratively compute the policy $\pi$ until convergence
  - Initialize $\pi$ randomly
  - Repeat until convergence
    1. Let $V = V^\pi$
    2. For each state $s$, set $\pi(s) = \arg \max_{a \in A} \sum_{s'} P_{sa}(s') V(s')$

Step (1) computes the value function for the current policy $\pi$
- Can be done using Bellman’s equations (solving $|S|$ equations in $|S|$ unknowns)

Step (2) gives the policy that is greedy w.r.t. $V$
Finding the Optimal Policy: Q Learning

- This is a variant of value iteration
- However, instead of iterating over $V$, we iterate over a “Q function”

Recall the optimal value function $V^*(s) = R(s) + \max_a \gamma \sum_{s'} P_{sa}(s') V^*(s')$

Define $Q(s, a) = R(s) + \gamma \sum_{s'} P_{sa}(s') \max_{a'} Q(s', a')$ then

$V^*(s) = \max_a Q(s, a)$

We can iteratively learn $Q(s, a)$ until convergence $Q_{t+1}(s, a) = R(s) + \gamma \sum_{s'} P_{sa}(s') \max_{a'} Q_t(s', a')$

Then set $V^*(s) = \arg \max_a Q(s, a)$
Finding the Optimal Policy: Q Learning

- This is a variant of value iteration
- However, instead of iterating over $V$, we iterate over a “Q function”
- Recall the optimal value function

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$$
Finding the Optimal Policy: Q Learning

- This is a variant of value iteration
- However, instead of iterating over $V$, we iterate over a “Q function”
- Recall the optimal value function
  \[
  V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')
  \]
- Define $Q(s, a) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$ then
  \[
  V^*(s) = \max_{a \in A} Q(s, a)
  \]
Finding the Optimal Policy: Q Learning

- This is a variant of value iteration
- However, instead of iterating over $V$, we iterate over a “Q function”
- Recall the optimal value function

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

- Define $Q(s, a) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$ then

$$V^*(s) = \max_{a \in A} Q(s, a)$$

- We can iteratively learn $Q(s, a)$ until convergence

$$Q_{t+1}(s, a) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') \max_{a' \in A} Q_t(s, a')$$
Finding the Optimal Policy: Q Learning

- This is a variant of value iteration
- However, instead of iterating over $V$, we iterate over a “Q function”
- Recall the optimal value function

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

- Define $Q(s, a) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$ then

$$V^*(s) = \max_{a \in A} Q(s, a)$$

- We can iteratively learn $Q(s, a)$ until convergence

$$Q_{t+1}(s, a) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') \max_{a' \in A} Q_t(s, a')$$

- Then set $V^*(s) = \arg \max_{a \in A} Q(s, a)$
Learning an MDP Model

- So far we assumed:
  - State transition probabilities $\{P_{sa}\}$ are given
  - Rewards $R(s)$ at each state are known

Often we don't know these and want to learn these. These are learned using experience (i.e., a set of previous trials).

$s(j)_i$ is the state at time $i$ of trial $j$

$a(j)_i$ is the corresponding action at that state
Learning an MDP Model

- So far we assumed:
  - State transition probabilities $\{P_{sa}\}$ are given
  - Rewards $R(s)$ at each state are known

- Often we don’t know these and want to learn these
Learning an MDP Model

- So far we assumed:
  - State transition probabilities $\{P_{sa}\}$ are given
  - Rewards $R(s)$ at each state are known

- Often we don’t know these and want to learn these

- These are learned using experience (i.e., a set of previous trials)
Learning an MDP Model

- So far we assumed:
  - State transition probabilities \( \{P_{sa}\} \) are given
  - Rewards \( R(s) \) at each state are known

- Often we don’t know these and want to learn these

- These are learned using experience (i.e., a set of previous trials)

\[
\begin{align*}
S_0^{(1)} &\rightarrow S_1^{(1)} \rightarrow S_2^{(1)} \rightarrow S_3^{(1)} \rightarrow \ldots \\
S_0^{(2)} &\rightarrow S_1^{(2)} \rightarrow S_2^{(2)} \rightarrow S_3^{(2)} \rightarrow \ldots \\
& \vdots \\
S_i^{(j)} & \text{is the state at time } i \text{ of trial } j
\end{align*}
\]
Learning an MDP Model

- So far we assumed:
  - State transition probabilities \( \{P_{sa}\} \) are given
  - Rewards \( R(s) \) at each state are known

- Often we don’t know these and want to learn these

- These are learned using experience (i.e., a set of previous trials)

\[
\begin{align*}
S_0^{(1)} & \xrightarrow{a_0^{(1)}} S_1^{(1)} & \xrightarrow{a_1^{(1)}} S_2^{(1)} & \xrightarrow{a_2^{(1)}} S_3^{(1)} & \xrightarrow{a_3^{(1)}} \ldots \\
S_0^{(2)} & \xrightarrow{a_0^{(2)}} S_1^{(2)} & \xrightarrow{a_1^{(2)}} S_2^{(2)} & \xrightarrow{a_2^{(2)}} S_3^{(2)} & \xrightarrow{a_3^{(2)}} \ldots \\
& \ldots
\end{align*}
\]

- \( s_i^{(j)} \) is the state at time \( i \) of trial \( j \)
- \( a_i^{(j)} \) is the corresponding action at that state
Learning an MDP Model

- Maximum likelihood estimate of state transition probabilities:

\[ P_{sa}(s') = \frac{\text{# of times we took action } a \text{ in state } s \text{ and got to } s'}{\text{# of times we took action } a \text{ in state } s} \]

Note: if action \( a \) is never taken in state \( s \), the above ratio is \( 0/0 \). In that case:

\[ P_{sa}(s') = \frac{1}{|S|} \text{ (uniform distribution over all states)} \]

\( P_{sa} \) is easy to update if we gather more experience (i.e., do more trials) just add counts in the numerator and denominator.

Likewise, the expected reward \( R(s) \) in state \( s \) can be computed:

\[ R(s) = \text{average reward in state } s \text{ across all the trials} \]
Learning an MDP Model

- Maximum likelihood estimate of state transition probabilities:

\[ P_{sa}(s') = \frac{\text{# of times we took action } a \text{ in state } s \text{ and got to } s'}{\text{# of times we took action } a \text{ in state } s} \]

- Note: if action \( a \) is never taken in state \( s \), the above ratio is \( 0/0 \)

  - In that case: \( P_{sa}(s') = 1/|S| \) (uniform distribution over all states)
Learning an MDP Model

- Maximum likelihood estimate of state transition probabilities:
  \[ P_{sa}(s') = \frac{\text{# of times we took action } a \text{ in state } s \text{ and got to } s'}{\text{# of times we took action } a \text{ in state } s} \]

- Note: if action \( a \) is never taken in state \( s \), the above ratio is 0/0
  
  - In that case: \( P_{sa}(s') = 1/|S| \) (uniform distribution over all states)

- \( P_{sa} \) is easy to update if we gather more experience (i.e., do more trials)
  
  - .. just add counts in the numerator and denominator
Learning an MDP Model

- Maximum likelihood estimate of state transition probabilities:
  \[ P_{sa}(s') = \frac{\text{# of times we took action } a \text{ in state } s \text{ and got to } s'}{\text{# of times we took action } a \text{ in state } s} \]

- Note: if action \( a \) is never taken in state \( s \), the above ratio is 0/0
  - In that case: \( P_{sa}(s') = 1/|S| \) (uniform distribution over all states)

- \( P_{sa} \) is easy to update if we gather more experience (i.e., do more trials)
  - Just add counts in the numerator and denominator

- Likewise, the expected reward \( R(s) \) in state \( s \) can be computed
  - \( R(s) = \text{average reward in state } s \text{ across all the trials} \)
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

The Algorithm (assuming value iteration)

1. Randomly initialize policy $\pi$
2. Repeat until convergence
   1. Execute policy $\pi$ in the MDP to generate a set of trials
   2. Use this "experience" to estimate $P_{sa}$ and $R$
   3. Apply value iteration with the estimated $P_{sa}$ and $R$ ⇒ Gives a new estimate of the value function $V$
   4. Update policy $\pi$ as the greedy policy w.r.t. $V$

Note: Step 3 can be made more efficient by initializing $V$ with values from the previous iteration.
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy.

Policy learning step can be done using value iteration or policy iteration.
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

Policy learning step can be done using value iteration or policy iteration

**The Algorithm (assuming value iteration)**

1. Randomly initialize policy $\pi$
2. Repeat until convergence
   1. Execute policy $\pi$ in the MDP to generate a set of trials
   2. Use this “experience” to estimate $P_{sa}$ and $R$
   3. Apply value iteration with the estimated $P_{sa}$ and $R$ $\Rightarrow$ Gives a new estimate of the value function $V$
   4. Update policy $\pi$ as the greedy policy w.r.t. $V$

Note: Step 3 can be made more efficient by initializing $V$ with values from the previous iteration
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy. Policy learning step can be done using value iteration or policy iteration.

**The Algorithm (assuming value iteration)**

- Randomly initialize policy $\pi$
Alternate between learning the MDP \((P_{sa} \text{ and } R)\), and learning the policy

Policy learning step can be done using value iteration or policy iteration

**The Algorithm (assuming value iteration)**

- Randomly initialize policy \(\pi\)
- Repeat until convergence
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy.

Policy learning step can be done using value iteration or policy iteration.

**The Algorithm (assuming value iteration)**

1. Randomly initialize policy $\pi$
2. Repeat until convergence
   1. Execute policy $\pi$ in the MDP to generate a set of trials
   2. Use this "experience" to estimate $P_{sa}$ and $R$
   3. Apply value iteration with the estimated $P_{sa}$ and $R$
      $\Rightarrow$ Gives a new estimate of the value function $V$
   4. Update policy $\pi$ as the greedy policy w.r.t. $V$

Note: Step 3 can be made more efficient by initializing $V$ with values from the previous iteration.
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

Policy learning step can be done using value iteration or policy iteration

The Algorithm (assuming value iteration)

- Randomly initialize policy $\pi$
- Repeat until convergence
  - Execute policy $\pi$ in the MDP to generate a set of trials
  - Use this “experience” to estimate $P_{sa}$ and $R$
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

Policy learning step can be done using value iteration or policy iteration

The Algorithm (assuming value iteration)

- Randomly initialize policy $\pi$
- Repeat until convergence
  - Execute policy $\pi$ in the MDP to generate a set of trials
  - Use this “experience” to estimate $P_{sa}$ and $R$
  - Apply value iteration with the estimated $P_{sa}$ and $R$
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

Policy learning step can be done using value iteration or policy iteration

**The Algorithm (assuming value iteration)**

- Randomly initialize policy $\pi$
- Repeat until convergence
  1. Execute policy $\pi$ in the MDP to generate a set of trials
  2. Use this “experience” to estimate $P_{sa}$ and $R$
  3. Apply value iteration with the estimated $P_{sa}$ and $R$
     ⇒ Gives a new estimate of the value function $V$
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy
Policy learning step can be done using value iteration or policy iteration

The Algorithm (assuming value iteration)

- Randomly initialize policy $\pi$
- Repeat until convergence
  - Execute policy $\pi$ in the MDP to generate a set of trials
  - Use this “experience” to estimate $P_{sa}$ and $R$
  - Apply value iteration with the estimated $P_{sa}$ and $R$
    - Gives a new estimate of the value function $V$
  - Update policy $\pi$ as the greedy policy w.r.t. $V$
Alternate between learning the MDP ($P_{sa}$ and $R$), and learning the policy

Policy learning step can be done using value iteration or policy iteration

**The Algorithm (assuming value iteration)**

- Randomly initialize policy $\pi$

- Repeat until convergence
  1. Execute policy $\pi$ in the MDP to generate a set of trials
  2. Use this “experience” to estimate $P_{sa}$ and $R$
  3. Apply value iteration with the estimated $P_{sa}$ and $R$
     - Gives a new estimate of the value function $V$
  4. Update policy $\pi$ as the greedy policy w.r.t. $V$

**Note:** Step 3 can be made more efficient by initializing $V$ with values from the previous iteration
Value Iteration vs Policy Iteration

- **Small state spaces**: Policy Iteration typically very fast and converges quickly
- Large state spaces: Policy Iteration may be slow. Reason: Policy Iteration needs to solve a large system of linear equations. Value iteration is preferred in such cases.
- Very large state spaces: Value function can be approximated using some regression algorithm. Optimality guarantee is lost however.
Value Iteration vs Policy Iteration

- **Small state spaces**: Policy Iteration typically very fast and converges quickly
- **Large state spaces**: Policy Iteration may be slow
Value Iteration vs Policy Iteration

- **Small state spaces:** Policy Iteration typically very fast and converges quickly

- **Large state spaces:** Policy Iteration may be slow
  - Reason: Policy Iteration needs to solve a large system of linear equations

Very large state spaces: Value function can be approximated using some regression algorithm. Optimality guarantee is lost however.
Value Iteration vs Policy Iteration

- **Small state spaces**: Policy Iteration typically very fast and converges quickly
- **Large state spaces**: Policy Iteration may be slow
  - Reason: Policy Iteration needs to solve a large system of linear equations
  - Value iteration is preferred in such cases

[Very large state spaces: Value function can be approximated using some regression algorithm. Optimality guarantee is lost however]
Value Iteration vs Policy Iteration

- **Small state spaces:** Policy Iteration typically very fast and converges quickly

- **Large state spaces:** Policy Iteration may be slow
  - Reason: Policy Iteration needs to solve a **large system of linear equations**
  - Value iteration is preferred in such cases

- **Very large state spaces:** Value function can be *approximated* using some regression algorithm
Value Iteration vs Policy Iteration

- **Small state spaces**: Policy Iteration typically very fast and converges quickly
- **Large state spaces**: Policy Iteration may be slow
  - Reason: Policy Iteration needs to solve a large system of linear equations
  - Value iteration is preferred in such cases
- **Very large state spaces**: Value function can be approximated using some regression algorithm
  - Optimality guarantee is lost however
A car moving in 2D: $s = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$. Thus $S = \mathbb{R}^6$
MDP with Continuous State Spaces

- A car moving in 2D: $s = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$. Thus $S = \mathbb{R}^6$

- Helicopter flying in 3D: $s = (x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$. Thus $S = \mathbb{R}^{12}$
A car moving in 2D: $s = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$. Thus $S = \mathbb{R}^6$

Helicopter flying in 3D: $s = (x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$. Thus $S = \mathbb{R}^{12}$

In general, the state space could be infinite $S = \mathbb{R}^n$
A car moving in 2D: \( s = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}) \). Thus \( S = \mathbb{R}^6 \)

Helicopter flying in 3D: \( s = (x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}) \). Thus \( S = \mathbb{R}^{12} \)

In general, the state space could be infinite \( S = \mathbb{R}^n \)

How to handle these continuous state spaces?
Suppose the state space is 2D: $s = (s_1, s_2)$. Can discretize it.
Suppose the state space is 2D: \( s = (s_1, s_2) \). Can discretize it

Call each discrete state \( \bar{s} \), discretized state space \( \bar{S} \), and define the MDP as

\[
(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)
\]
Suppose the state space is 2D: \( s = (s_1, s_2) \). Can discretize it.

Call each discrete state \( \bar{s} \), discretized state space \( \bar{S} \), and define the MDP as

\[
(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)
\]

Can now use value iteration or policy iteration on this discrete state space.

Limitations: Piecewise constant \( V^* \) and \( \pi^* \) (isn’t realistic). Doesn’t work well in high-dim state spaces (resulting discrete space too huge).

Discretization usually done only for 1D or 2D state-spaces.
Discretization

- Suppose the state space is 2D: \( s = (s_1, s_2) \). Can discretize it

- Call each discrete state \( \bar{s} \), discretized state space \( \bar{S} \), and define the MDP as

\[
(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)
\]

- Can now use value iteration or policy iteration on this discrete state space

- Limitations? Piecewise constant \( V^* \) and \( \pi^* \) (isn’t realistic). Doesn’t work well in high-dim state spaces (resulting discrete space space too huge)
Discretization

- Suppose the state space is 2D: \( s = (s_1, s_2) \). Can discretize it

- Call each discrete state \( \bar{s} \), discretized state space \( \bar{S} \), and define the MDP as

\[
(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)
\]

- Can now use value iteration or policy iteration on this discrete state space

- Limitations? Piecewise constant \( V^* \) and \( \pi^* \) (isn’t realistic). Doesn’t work well in high-dim state spaces (resulting discrete space space too huge)

- Discretization usually done only for 1D or 2D state-spaces
Policy Learning in Continuous State Spaces

- Policy learning requires learning the value function $V^*$.
Policy Learning in Continuous State Spaces

- Policy learning requires learning the value function $V^*$
- Can we do away with discretization and approximate $V^*$ directly?
Policy Learning in Continuous State Spaces

- Policy learning requires learning the value function $V^*$
- Can we do away with discretization and approximate $V^*$ directly?
- To do so, we will need (an approximate) model of the underlying MDP
Approximating the MDP Model

- Execute a set of trials

\[
\begin{align*}
S_0^{(1)} & \rightarrow S_1^{(1)} \rightarrow S_2^{(1)} \rightarrow \ldots \rightarrow S_T^{(1)} \\
S_0^{(2)} & \rightarrow S_1^{(2)} \rightarrow S_2^{(2)} \rightarrow \ldots \rightarrow S_T^{(2)} \\
\vdots & \\
S_0^{(m)} & \rightarrow S_1^{(m)} \rightarrow S_2^{(m)} \rightarrow \ldots \rightarrow S_T^{(m)}
\end{align*}
\]
Approximating the MDP Model

- Execute a set of trials

\[
\begin{align*}
S_0^{(1)} &\xrightarrow{a_0^{(1)}} S_1^{(1)} &\xrightarrow{a_1^{(1)}} S_2^{(1)} &\cdots &\xrightarrow{a_{T-1}^{(1)}} S_T^{(1)} \\
S_0^{(2)} &\xrightarrow{a_0^{(2)}} S_1^{(2)} &\xrightarrow{a_1^{(2)}} S_2^{(2)} &\cdots &\xrightarrow{a_{T-1}^{(2)}} S_T^{(2)} \\
&\cdots \\
S_0^{(m)} &\xrightarrow{a_0^{(m)}} S_1^{(m)} &\xrightarrow{a_1^{(m)}} S_2^{(m)} &\cdots &\xrightarrow{a_{T-1}^{(m)}} S_T^{(m)}
\end{align*}
\]

- Use this data to learn a function that predicts \( s_{t+1} \) given \( s_t \) and \( a \), e.g.,

\[
s_{t+1} = As_t + Ba_t
\]

\[
\arg\min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left( As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2
\]
Approximating the MDP Model

- Execute a set of trials

\[
\begin{align*}
S_0^{(1)} & \rightarrow S_1^{(1)} & a_0^{(1)} & \rightarrow S_2^{(1)} & a_1^{(1)} & \rightarrow \cdots & a_{T-1}^{(1)} & \rightarrow S_T^{(1)} \\
S_0^{(2)} & \rightarrow S_1^{(2)} & a_0^{(2)} & \rightarrow S_2^{(2)} & a_1^{(2)} & \rightarrow \cdots & a_{T-1}^{(2)} & \rightarrow S_T^{(2)} \\
\vdots & & & & & & & \\
S_0^{(m)} & \rightarrow S_1^{(m)} & a_0^{(m)} & \rightarrow S_2^{(m)} & a_1^{(m)} & \rightarrow \cdots & a_{T-1}^{(m)} & \rightarrow S_T^{(m)}
\end{align*}
\]

- Use this data to learn a function that predicts \( s_{t+1} \) given \( s_t \) and \( a \), e.g.,

\[
s_{t+1} = As_t + Ba_t
\]

- \( A \) and \( B \) can be estimated from the trial data
Approximating the MDP Model

- Execute a set of trials

\[
\begin{align*}
S_0^{(1)} & \rightarrow S_1^{(1)} & \rightarrow S_2^{(1)} & \rightarrow \ldots & \rightarrow S_T^{(1)} \\
S_0^{(2)} & \rightarrow S_1^{(2)} & \rightarrow S_2^{(2)} & \rightarrow \ldots & \rightarrow S_T^{(2)} \\
\vdots \\
S_0^{(m)} & \rightarrow S_1^{(m)} & \rightarrow S_2^{(m)} & \rightarrow \ldots & \rightarrow S_T^{(m)}
\end{align*}
\]

- Use this data to learn a function that predicts \(s_{t+1}\) given \(s_t\) and \(a\), e.g.,

\[
s_{t+1} = As_t + Ba_t
\]

\[
\arg\min_{A,B} \sum_i \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - (As_t^{(i)} + Ba_t^{(i)}) \right\|^2
\]

- \(A\) and \(B\) can be estimate from the trial data

- Can also make the function stochastic/noisy: \(s_{t+1} = As_t + Ba_t + \epsilon_t\) where \(\epsilon_t \sim \mathcal{N}(0, \Sigma)\) is the random noise (\(\Sigma\) can also be learned)
Approximating the MDP Model

- Can also learn nonlinear functions $s_{t+1} = f(s_t)$

$$s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$$

- Any nonlinear regression algorithm can be used here
We will use “Fitted Value Iteration” methods.
Approximating the Value Function

- We will use “Fitted Value Iteration” methods
- Recall the value iteration

\[
V(s) := R(s) + \gamma \max_a \int_{s'} P_{sa}(s') V(s') ds'
\]

\[
= R(s) + \gamma \max_a \mathbb{E}_{s' \sim P_{sa}} [V(s')]
\]

- Note: sum replaced by integral (since the state space \( S \) is continuous)
Approximating the Value Function

- We will use “Fitted Value Iteration” methods
- Recall the value iteration

\[
V(s) := R(s) + \gamma \max_a \int_s P_{sa}(s') V(s') ds'
\]

\[
= R(s) + \gamma \max_a E_{s' \sim P_{sa}} [V(s')]
\]

- Note: sum replaced by integral (since the state space \( S \) is continuous)
- Want a model for \( V(s) \). Let’s assume \( V(s) = \theta^T \phi(s) \)
Approximating the Value Function

- We will use “Fitted Value Iteration” methods
- Recall the value iteration

\[ V(s) := R(s) + \gamma \max_a \int s' P_{sa}(s') V(s') ds' \]

\[ = R(s) + \gamma \max_a E_{s' \sim P_{sa}} [V(s')] \]

- Note: sum replaced by integral (since the state space \( S \) is continuous)
- Want a model for \( V(s) \). Let’s assume \( V(s) = \theta^\top \phi(s) \)
- We would need some training data in order to learn \( \theta \)

\[ \{V(s^i), \phi(s^i)\}_{i=1}^m \]
Approximating the Value Function

- We will use “Fitted Value Iteration” methods
- Recall the value iteration

\[ V(s) := R(s) + \gamma \max_a \int_{s'} P_{sa}(s')V(s')ds' \]

\[ = R(s) + \gamma \max_a E_{s' \sim P_{sa}}[V(s')] \]

- Note: sum replaced by integral (since the state space \( S \) is continuous)
- Want a model for \( V(s) \). Let’s assume \( V(s) = \theta^\top \phi(s) \)
- We would need some training data in order to learn \( \theta \)

\[ \{ V(s^i), \phi(s^i) \}_{i=1}^m \]
- We will generate such training data and learn \( \theta \) in an alternating fashion
Fitted Value Iteration: The Full Algorithm

1. Randomly sample \( m \) states \( s^{(1)}, s^{(2)}, \ldots, s^{(m)} \in S \).
2. Initialize \( \theta := 0 \).
3. Repeat {
   
   For \( i = 1, \ldots, m \) {
      
      For each action \( a \in A \) {
         
         Sample \( s'_1, \ldots, s'_k \sim P_{s^{(i)}_a} \) (using a model of the MDP).
         
         Set \( q(a) = \frac{1}{k} \sum_{j=1}^{k} R(s^{(i)}_j) + \gamma V(s'_j) \)
         
         // Hence, \( q(a) \) is an estimate of \( R(s^{(i)}) + \gamma \mathbb{E}_{s' \sim P_{s^{(i)}_a}} [V(s')] \).
      }
      
      Set \( y^{(i)} = \max_a q(a) \).
      
      // Hence, \( y^{(i)} \) is an estimate of \( R(s^{(i)}) + \gamma \max_a \mathbb{E}_{s' \sim P_{s^{(i)}_a}} [V(s')] \).
   }

   // In the original value iteration algorithm (over discrete states)
   // we updated the value function according to \( V(s^{(i)}) := y^{(i)} \).
   // In this algorithm, we want \( V(s^{(i)}) \approx y^{(i)} \), which we’ll achieve
   // using supervised learning (linear regression).
   
   Set \( \theta := \arg \min_{\theta} \frac{1}{2} \sum_{i=1}^{m} (\theta^T \phi(s^{(i)}) - y^{(i)})^2 \)
Fitted Value Iteration

- Other nonlinear regression algorithms can also be used

\[ V(s) = f(\phi(s)) \]

where \( f \) is a nonlinear function (e.g., modeled by a Gaussian Process)

- Note: Fitted value iteration is not guaranteed to converge (though, in practice, mostly it does)

- The final output is \( V \) (an approximation to \( V^* \))

- \( V \) implicitly represents our policy \( \pi \). The optimal action

\[ \arg \max_a \mathbb{E}_{s' \sim P_{sa}}[V(s')] \]
Other Topics related to RL

- Inverse Reinforcement Learning (IRL)
  - Doesn’t assume the reward function to be known. Learns it
Other Topics related to RL

- Inverse Reinforcement Learning (IRL)
  - Doesn’t assume the reward function to be known. Learns it

- Imitation Learning: Imitate an demonstrator/demonstrations
Other Topics related to RL

- Inverse Reinforcement Learning (IRL)
  - Doesn’t assume the reward function to be known. Learns it

- Imitation Learning: Imitate an demonstrator/demonstrations

- Deep Reinforcement Learning
Summary

- Basic introduction to Reinforcement Learning
- Looked at the definition of a Markov Decision Process (MDP)
- Looked at methods for learning the MDP parameters from data
  - Easily and exactly for the discrete state-space case
  - Using function approximation methods in the continuous case
- Looked at methods for Policy Learning
  - MDP Learning and Policy Learning usually done jointly