Today

- Learn to play games
- Reinforcement Learning
Playing Games: Atari

https://www.youtube.com/watch?v=V1eYniJ0Rnk
Playing Games: Super Mario

https://www.youtube.com/watch?v=wfL4L_14U9A
Making Pancakes!

https://www.youtube.com/watch?v=W_gxLKSsSIE
Reinforcement Learning Resources

- RL tutorial – on course website
Learning algorithms differ in the information available to learner.
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
- **Reinforcement learning**
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
- **Reinforcement learning**

More realistic learning scenario:

- Continuous stream of input information, and actions
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
- **Reinforcement learning**

More realistic learning scenario:

- Continuous stream of input information, and actions
- Effects of action depend on state of the world
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
- **Reinforcement learning**

More realistic learning scenario:

- Continuous stream of input information, and actions
- Effects of action depend on state of the world
- Obtain reward that depends on world state and actions
Learning algorithms differ in the information available to learner

- **Supervised**: correct outputs
- **Unsupervised**: no feedback, must construct measure of good output
- **Reinforcement learning**

More realistic learning scenario:

- Continuous stream of input information, and actions
- Effects of action depend on state of the world
- Obtain reward that depends on world state and actions
  - not correct response, just some feedback
Reinforcement Learning

State: $s$
Reward: $r$

Agent

Environment

Actions: $a$

[pic from: Peter Abbeel]
Example: Tic Tac Toe, Notation

environment
Example: Tic Tac Toe, Notation

\begin{tikzpicture}
\draw[step=1cm,gray,very thin] (0,0) grid (3,3);
\filldraw[fill=red] (1,1) circle (1cm);
\filldraw[fill=blue] (2,2) circle (1cm);
\filldraw[fill=red] (0,0) circle (1cm);
\filldraw[fill=blue] (1,2) circle (1cm);
\filldraw[fill=red] (2,0) circle (1cm);
\filldraw[fill=blue] (1,0) circle (1cm);
\end{tikzpicture}

(current) state
Example: Tic Tac Toe, Notation

\[ \begin{array}{ccc}
O & X & O \\
X & O & X \\
O & X & O \\
\end{array} \]

action
Example: Tic Tac Toe, Notation

reward
(Here: -1)
World described by a discrete, finite set of states and actions
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:

- Take an action $a_t$ (possibly null action)
Formulating Reinforcement Learning

- World described by a discrete, finite set of states and actions
- At every time step $t$, we are in a state $s_t$, and we:
  - Take an action $a_t$ (possibly null action)
  - Receive some reward $r_{t+1}$
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:

- Take an action $a_t$ (possibly null action)
- Receive some reward $r_{t+1}$
- Move into a new state $s_{t+1}$
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:

- Take an action $a_t$ (possibly null action)
- Receive some reward $r_{t+1}$
- Move into a new state $s_{t+1}$

An RL agent may include one or more of these components:

- **Policy** $\pi$: agent’s behaviour function
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:

- Take an action $a_t$ (possibly null action)
- Receive some reward $r_{t+1}$
- Move into a new state $s_{t+1}$

An RL agent may include one or more of these components:

- Policy $\pi$: agent’s behaviour function
- Value function: how good is each state and/or action
World described by a discrete, finite set of states and actions

At every time step $t$, we are in a state $s_t$, and we:

- Take an action $a_t$ (possibly null action)
- Receive some reward $r_{t+1}$
- Move into a new state $s_{t+1}$

An RL agent may include one or more of these components:

- **Policy** $\pi$: agent’s behaviour function
- **Value function**: how good is each state and/or action
- **Model**: agent’s representation of the environment
A policy is the agent’s behaviour.

It’s a selection of which action to take, based on the current state

Deterministic policy:  \( a = \pi(s) \)

Stochastic policy: \( \pi(a|s) = P[a_t = a|s_t = s] \)

[Slide credit: D. Silver]
Value Function

- **Value function** is a prediction of future reward
- Used to evaluate the goodness/badness of states
Value Function

- **Value function** is a prediction of future reward
- Used to evaluate the goodness/badness of states
- Our aim will be to maximize the value function (the total reward we receive over time): find the policy with the highest expected reward

\[
V_\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots
\]

$\gamma$ is called a discount rate, and it is always $0 \leq \gamma \leq 1$.

If $\gamma$ close to 1, rewards further in the future count more, and we say that the agent is "farsighted".

$\gamma$ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later).

[Slide credit: D. Silver]
Value Function

- **Value function** is a prediction of future reward.
- Used to evaluate the goodness/badness of states.
- Our aim will be to maximize the value function (the total reward we receive over time): find the policy with the highest expected reward.
- By following a policy $\pi$, the value function is defined as:
  \[
  V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots
  \]
- $\gamma$ is called a **discount rate**, and it is always $0 \leq \gamma \leq 1$. 

[Slide credit: D. Silver]
**Value Function**

- **Value function** is a prediction of future reward
- Used to evaluate the goodness/badness of states
- Our aim will be to maximize the value function (the total reward we receive over time): find the policy with the highest expected reward
- By following a policy $\pi$, the value function is defined as:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

- $\gamma$ is called a **discount rate**, and it is always $0 \leq \gamma \leq 1$
- If $\gamma$ close to 1, rewards further in the future count more, and we say that the agent is “farsighted”
- $\gamma$ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later)

[Slide credit: D. Silver]
The model describes the environment by a distribution over rewards and state transitions:

\[ P(s_{t+1} = s', r_{t+1} = r'|s_t = s, a_t = a) \]

We assume the Markov property: the future depends on the past only through the current state.
Maze Example

- Rewards:

Actions:
- N, E, S, W

States:
- Agent's location

[Slide credit: D. Silver]

Zemel, Urtasun, Fidler (UofT)
CSC 411: 19-Reinforcement Learning
November 29, 2016 17 / 38
Maze Example

- Rewards: $-1$ per time-step
- Actions:

Start

Goal
Maze Example

- Rewards: $-1$ per time-step
- Actions: N, E, S, W
- States:
Maze Example

- **Rewards:** $-1$ per time-step
- **Actions:** N, E, S, W
- **States:** Agent’s location

[Slide credit: D. Silver]
Arrows represent policy $\pi(s)$ for each state $s$
Maze Example

- Numbers represent value $V^\pi(s)$ of each state $s$

[Slide credit: D. Silver]
Example: Tic-Tac-Toe

Consider the game tic-tac-toe:
Consider the game tic-tac-toe:

- **reward:**
  - win/lose/tie the game (+1/−1/0) [only at final move in given game]

- **state:** positions of X's and O's on the board

- **policy:** mapping from states to actions
  - based on rules of game: choice of one open position

- **value function:** prediction of reward in future, based on current state

In tic-tac-toe, since state space is tractable, can use a table to represent value function
Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/ −1/0) [only at final move in given game]
Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/ −1/0) [only at final move in given game]
- **state**: positions of X's and O's on the board
- **policy**: mapping from states to actions
  - based on rules of game: choice of one open position
- **value function**: prediction of reward in future, based on current state

In tic-tac-toe, since state space is tractable, can use a table to represent value function.
Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/−1/0) [only at final move in given game]
- **state**: positions of X’s and O’s on the board
Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/−1/0) [only at final move in given game]
- **state**: positions of X’s and O's on the board
- **policy**: mapping from states to actions
Example: Tic-Tac-Toe

Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/ − 1/0) [only at final move in given game]
- **state**: positions of X’s and O’s on the board
- **policy**: mapping from states to actions
  - based on rules of game: choice of one open position
Example: Tic-Tac-Toe

Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/ − 1/0) [only at final move in given game]
- **state**: positions of X’s and O's on the board
- **policy**: mapping from states to actions
  - based on rules of game: choice of one open position
- **value function**: prediction of reward in future, based on current state
Consider the game tic-tac-toe:

- **reward**: win/lose/tie the game (+1/ − 1/0) [only at final move in given game]
- **state**: positions of X’s and O's on the board
- **policy**: mapping from states to actions
  - based on rules of game: choice of one open position
- **value function**: prediction of reward in future, based on current state

In tic-tac-toe, since state space is tractable, can use a table to represent value function
Each board position (taking into account symmetry) has some probability

<table>
<thead>
<tr>
<th>State</th>
<th>Probability of a win (Computer plays “o”)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>etc</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Each board position (taking into account symmetry) has some probability

Simple learning process:

- start with all values = 0.5
- policy: choose move with highest probability of winning given current legal moves from current state
- update entries in table based on outcome of each game
- After many games value function will represent true probability of winning from each state

Can try alternative policy: sometimes select moves randomly (exploration)
Each board position (taking into account symmetry) has some probability

**Simple learning process:**

- start with all values $= 0.5$

<table>
<thead>
<tr>
<th>State</th>
<th>Probability of a win (Computer plays “o”)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(empty)</td>
<td>0.5</td>
</tr>
<tr>
<td>(one x)</td>
<td>0.5</td>
</tr>
<tr>
<td>(two x)</td>
<td>1.0</td>
</tr>
<tr>
<td>(three x)</td>
<td>0.0</td>
</tr>
<tr>
<td>(three o)</td>
<td>0.0</td>
</tr>
<tr>
<td>(two o)</td>
<td>0.5</td>
</tr>
<tr>
<td>etc</td>
<td></td>
</tr>
</tbody>
</table>
Each board position (taking into account symmetry) has some probability.

- **Simple learning process:**
  - start with all values $= 0.5$
  - **policy**: choose move with highest probability of winning given current legal moves from current state

<table>
<thead>
<tr>
<th>State</th>
<th>Probability of a win (Computer plays “o”)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
</tbody>
</table>

etc
Each board position (taking into account symmetry) has some probability

Simple learning process:

- start with all values = 0.5
- policy: choose move with highest probability of winning given current legal moves from current state
- update entries in table based on outcome of each game

<table>
<thead>
<tr>
<th>State</th>
<th>Probability of a win (Computer plays “o”)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>etc</td>
<td>0.5</td>
</tr>
</tbody>
</table>
RL & Tic-Tac-Toe

Each board position (taking into account symmetry) has some probability

Simple learning process:

- start with all values $= 0.5$
- **policy**: choose move with highest probability of winning given current legal moves from current state
- update entries in table based on outcome of each game
- After many games value function will represent true probability of winning from each state

<table>
<thead>
<tr>
<th>State</th>
<th>Probability of a win (Computer plays “o”)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>etc</td>
<td></td>
</tr>
</tbody>
</table>
Each board position (taking into account symmetry) has some probability

Simple learning process:

- start with all values = 0.5
- policy: choose move with highest probability of winning given current legal moves from current state
- update entries in table based on outcome of each game
- After many games value function will represent true probability of winning from each state

Can try alternative policy: sometimes select moves randomly (exploration)
Markov Decision Problem (MDP): tuple \((S, A, P, \gamma)\) where \(P\) is

\[
P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)
\]
Basic Problems

- Markov Decision Problem (MDP): tuple $(S, A, P, \gamma)$ where $P$ is

  \[ P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a) \]

- Standard MDP problems:
Markov Decision Problem (MDP): tuple \((S, A, P, \gamma)\) where \(P\) is

\[ P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a) \]

Standard MDP problems:

1. **Planning**: given complete Markov decision problem as input, compute policy with optimal expected return

[Pic: P. Abbeel]
Markov Decision Problem (MDP): tuple \((S, A, P, \gamma)\) where \(P\) is

\[
P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)
\]

Standard MDP problems:

1. **Planning**: given complete Markov decision problem as input, compute policy with optimal expected return
2. **Learning**: We don’t know which states are good or what the actions do. We must try out the actions and states to learn what to do

[P. Abbeel]
Example of Standard MDP Problem

1. **Planning**: given complete Markov decision problem as input, compute policy with optimal expected return

2. **Learning**: Only have access to experience in the MDP, learn a near-optimal strategy

\[ r(s, a) \text{ (immediate reward)} \]
Example of Standard MDP Problem

1. **Planning**: given complete Markov decision problem as input, compute policy with optimal expected return

2. **Learning**: Only have access to experience in the MDP, learn a near-optimal strategy

We will focus on learning, but discuss planning along the way
Exploration vs. Exploitation

- If we knew how the world works (embodied in \( P \)), then the policy should be deterministic.
Exploration vs. Exploitation

- If we knew how the world works (embodied in $P$), then the policy should be deterministic
  - just select optimal action in each state
Exploration vs. Exploitation

- If we knew how the world works (embodied in $P$), then the policy should be deterministic
  - just select optimal action in each state
- Reinforcement learning is like trial-and-error learning
Exploration vs. Exploitation

- If we knew how the world works (embodied in $P$), then the policy should be deterministic
  - just select optimal action in each state
- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment
- Without losing too much reward along the way
If we knew how the world works (embodied in $P$), then the policy should be deterministic

- just select optimal action in each state

Reinforcement learning is like trial-and-error learning

The agent should discover a good policy from its experiences of the environment

Without losing too much reward along the way

Since we do not have complete knowledge of the world, taking what appears to be the optimal action may prevent us from finding better states/actions
Exploration vs. Exploitation

- If we knew how the world works (embodied in $P$), then the policy should be deterministic
  - just select optimal action in each state
- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment
- Without losing too much reward along the way
- Since we do not have complete knowledge of the world, taking what appears to be the optimal action may prevent us from finding better states/actions
- Interesting trade-off:
  - immediate reward (exploitation) vs. gaining knowledge that might enable higher future reward (exploration)
Examples

○ Restaurant Selection
  ▶ **Exploitation**: Go to your favourite restaurant
  ▶ **Exploration**: Try a new restaurant

○ Online Banner Advertisements
  ▶ **Exploitation**: Show the most successful advert
  ▶ **Exploration**: Show a different advert

○ Oil Drilling
  ▶ **Exploitation**: Drill at the best known location
  ▶ **Exploration**: Drill at a new location

○ Game Playing
  ▶ **Exploitation**: Play the move you believe is best
  ▶ **Exploration**: Play an experimental move

[Slide credit: D. Silver]
Goal: find policy $\pi$ that maximizes expected accumulated future rewards $V^\pi(s_t)$, obtained by following $\pi$ from state $s_t$:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$
Goal: find policy $\pi$ that maximizes expected accumulated future rewards $V^\pi(s_t)$, obtained by following $\pi$ from state $s_t$:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Game show example:
Goal: find policy $\pi$ that maximizes expected accumulated future rewards $V^\pi(s_t)$, obtained by following $\pi$ from state $s_t$:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Game show example:

- assume series of questions, increasingly difficult, but increasing payoff
**Goal:** find policy $\pi$ that maximizes expected accumulated future rewards $V^\pi(s_t)$, obtained by following $\pi$ from state $s_t$:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

**Game show example:**

- assume series of questions, increasingly difficult, but increasing payoff
- choice: accept accumulated earnings and quit; or continue and risk losing everything

**Notice that:**

$$V^\pi(s_t) = r_t + \gamma V^\pi(s_{t+1})$$
What to Learn

- We might try to learn the function $V$ (which we write as $V^*$)

$$V^*(s) = \max_a \left[ r(s, a) + \gamma V^*(\delta(s, a)) \right]$$

- Here $\delta(s, a)$ gives the next state, if we perform action $a$ in current state $s$
We might try to learn the function $V$ (which we write as $V^*$)

$$V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

Here $\delta(s, a)$ gives the next state, if we perform action $a$ in current state $s$

We could then do a lookahead search to choose best action from any state $s$:

$$\pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$
What to Learn

- We might try to learn the function $V$ (which we write as $V^*$)

$$V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

- Here $\delta(s, a)$ gives the next state, if we perform action $a$ in current state $s$

- We could then do a lookahead search to choose best action from any state $s$:

$$\pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

- But there’s a problem:
What to Learn

- We might try to learn the function $V$ (which we write as $V^*$)
  \[
  V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))]
  \]
- Here $\delta(s, a)$ gives the next state, if we perform action $a$ in current state $s$
- We could then do a lookahead search to choose best action from any state $s$:
  \[
  \pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]
  \]
- But there's a problem:
  - This works well if we know $\delta()$ and $r()$
What to Learn

- We might try to learn the function $V$ (which we write as $V^*$)

$$V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

- Here $\delta(s, a)$ gives the next state, if we perform action $a$ in current state $s$
- We could then do a lookahead search to choose best action from any state $s$:

$$\pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

- But there’s a problem:
  - This works well if we know $\delta()$ and $r()$
  - But when we don’t, we cannot choose actions this way
Q Learning

- Define a new function very similar to $V^*$

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$
Q Learning

- Define a new function very similar to $V^*$

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$

- If we learn $Q$, we can choose the optimal action even without knowing $\delta$!

$$\pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$
Q Learning

- Define a new function very similar to $V^*$

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$

- If we learn $Q$, we can choose the optimal action even without knowing $\delta$!

$$\pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

$$= \arg\max_a Q(s, a)$$
Q Learning

- Define a new function very similar to $V^*$
  \[
  Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))
  \]

- If we learn $Q$, we can choose the optimal action even without knowing $\delta$!
  \[
  \pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]
  = \arg\max_a Q(s, a)
  \]

- $Q$ is then the evaluation function we will learn
\( \gamma = 0.9 \)

\( r(s, a) \) (immediate reward) values

\( Q(s, a) \) values

\( V^*(s) \) values

\[ V^*(s_5) = 0 + \gamma 100 + \gamma^2 0 + \ldots = 90 \]

One optimal policy
Q and $V^*$ are closely related:

$$V^*(s) = \max_a Q(s, a)$$
Training Rule to Learn Q

- $Q$ and $V^*$ are closely related:

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

- So we can write $Q$ recursively:

\[
Q(s_t, a_t) = r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))
\]
Training Rule to Learn $Q$

- $Q$ and $V^*$ are closely related:

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

- So we can write $Q$ recursively:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))$$

$$= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a')$$
Training Rule to Learn $Q$

- $Q$ and $V^*$ are closely related:
  \[ V^*(s) = \max_a Q(s, a) \]

- So we can write $Q$ recursively:
  \[
  Q(s_t, a_t) = r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t)) \\
  = r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a')
  \]

- Let $\hat{Q}$ denote the learner’s current approximation to $Q$
Training Rule to Learn $Q$

- $Q$ and $V^*$ are closely related:
  \[ V^*(s) = \max_a Q(s, a) \]

- So we can write $Q$ recursively:
  \[
  Q(s_t, a_t) = r(s_t, a_t) + \gamma V^*(s_t+1) = r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a')
  \]

- Let $\hat{Q}$ denote the learner’s current approximation to $Q$
- Consider training rule
  \[
  \hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')
  \]
  where $s'$ is state resulting from applying action $a$ in state $s$
Q Learning for Deterministic World

- For each \( s, a \) initialize table entry \( \hat{Q}(s, a) \leftarrow 0 \)
Q Learning for Deterministic World

- For each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$
- Start in some initial state $s$
Q Learning for Deterministic World

- For each state \( s \), action \( a \) initialize table entry \( \hat{Q}(s, a) \leftarrow 0 \)
- Start in some initial state \( s \)
- Do forever:
  - Select an action \( a \) and execute it
  - Receive immediate reward \( r \)
  - Observe the new state \( s' \)
  - Update the table entry for \( \hat{Q}(s, a) \) using Q-learning rule:
    \[
    \hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_a \hat{Q}(s', a)
    \]
Q Learning for Deterministic World

- For each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$
- Start in some initial state $s$
- Do forever:
  - Select an action $a$ and execute it
Q Learning for Deterministic World

- For each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$
- Start in some initial state $s$
- Do forever:
  - Select an action $a$ and execute it
  - Receive immediate reward $r$

If we get to absorbing state, restart to initial state, and run thru "Do forever" loop until reach absorbing state
Q Learning for Deterministic World

- For each \( s, a \) initialize table entry \( \hat{Q}(s, a) \leftarrow 0 \)
- Start in some initial state \( s \)
- Do forever:
  - Select an action \( a \) and execute it
  - Receive immediate reward \( r \)
  - Observe the new state \( s' \)
Q Learning for Deterministic World

- For each \( s, a \) initialize table entry \( \hat{Q}(s, a) \leftarrow 0 \)
- Start in some initial state \( s \)
- Do forever:
  - Select an action \( a \) and execute it
  - Receive immediate reward \( r \)
  - Observe the new state \( s' \)
  - Update the table entry for \( \hat{Q}(s, a) \) using \( Q \) learning rule:
    \[
    \hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')
    \]
Q Learning for Deterministic World

- For each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$
- Start in some initial state $s$
- Do forever:
  - Select an action $a$ and execute it
  - Receive immediate reward $r$
  - Observe the new state $s'$
  - Update the table entry for $\hat{Q}(s, a)$ using $Q$ learning rule:
    \[ \hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a') \]
  - $s \leftarrow s'$

If we get to absorbing state, restart to initial state, and run thru "Do forever" loop until reach absorbing state
Q Learning for Deterministic World

- For each \( s, a \) initialize table entry \( \hat{Q}(s, a) \leftarrow 0 \)
- Start in some initial state \( s \)
- Do forever:
  - Select an action \( a \) and execute it
  - Receive immediate reward \( r \)
  - Observe the new state \( s' \)
  - Update the table entry for \( \hat{Q}(s, a) \) using Q learning rule:
    \[
    \hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')
    \]
  - \( s \leftarrow s' \)
- If we get to absorbing state, restart to initial state, and run thru "Do forever" loop until reach absorbing state
Updating Estimated $Q$

- Assume the robot is in state $s_1$; some of its current estimates of $Q$ are as shown; executes rightward move

\[
\hat{Q}(s_1, a_{\text{right}}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')
\]

Important observation: at each time step (making an action $a$ in state $s$ only one entry of $\hat{Q}$ will change (the entry $\hat{Q}(s, a)$)).

Notice that if rewards are non-negative, then $\hat{Q}$ values only increase from 0, approach true $Q$. 

\[81, 100 \rightarrow 90, 100\]
Assume the robot is in state $s_1$; some of its current estimates of $Q$ are as shown; executes rightward move

\[
\hat{Q}(s_1, a_{\text{right}}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')
\]
Updating Estimated $Q$

- Assume the robot is in state $s_1$; some of its current estimates of $Q$ are as shown; executes rightward move

\[ \hat{Q}(s_1, a_{\text{right}}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \]

\[ \leftarrow r + 0.9 \max_{a} \{63, 81, 100\} \leftarrow 90 \]
Updating Estimated $Q$

- Assume the robot is in state $s_1$; some of its current estimates of $Q$ are as shown; executes rightward move

\[
\hat{Q}(s_1, a_{\text{right}}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')
\]

\[
\leftarrow r + 0.9 \max_a \{63, 81, 100\} \leftarrow 90
\]

- Important observation: at each time step (making an action $a$ in state $s$ only one entry of $\hat{Q}$ will change (the entry $\hat{Q}(s, a)$)
Updating Estimated $Q$

Assume the robot is in state $s_1$; some of its current estimates of $Q$ are as shown; executes rightward move

$$\hat{Q}(s_1, a_{\text{right}}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow r + 0.9 \max_a \{63, 81, 100\} \leftarrow 90$$

Important observation: at each time step (making an action $a$ in state $s$ only one entry of $\hat{Q}$ will change (the entry $\hat{Q}(s, a)$)

Notice that if rewards are non-negative, then $\hat{Q}$ values only increase from 0, approach true $Q$
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action \( a \) results in transition from state \( s_i \) to \( s_j \); algorithm updates \( \hat{Q}(s_i, a) \) using the learning rule

Intuition for simple grid world, reward only upon entering goal state

1. All \( \hat{Q}(s, a) \) start at 0
2. First episode – only update \( \hat{Q}(s, a) \) for transition leading to goal state
3. Next episode – if go thru this next-to-last transition, will update \( \hat{Q}(s, a) \) another step back
4. Eventually propagate information from transitions with non-zero reward throughout state-action space
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action \( a \) results in transition from state \( s_i \) to \( s_j \); algorithm updates \( \hat{Q}(s_i, a) \) using the learning rule
- Intuition for simple grid world, reward only upon entering goal state \( \rightarrow Q \) estimates improve from goal state back
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action $a$ results in transition from state $s_i$ to $s_j$; algorithm updates $\hat{Q}(s_i, a)$ using the learning rule
- Intuition for simple grid world, reward only upon entering goal state $\rightarrow Q$ estimates improve from goal state back
  1. All $\hat{Q}(s, a)$ start at 0
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action \( a \) results in transition from state \( s_i \) to \( s_j \); algorithm updates \( \hat{Q}(s_i, a) \) using the learning rule
- Intuition for simple grid world, reward only upon entering goal state \( \rightarrow Q \) estimates improve from goal state back
  1. All \( \hat{Q}(s, a) \) start at 0
  2. First episode – only update \( \hat{Q}(s, a) \) for transition leading to goal state
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action $a$ results in transition from state $s_i$ to $s_j$; algorithm updates $\hat{Q}(s_i, a)$ using the learning rule
- Intuition for simple grid world, reward only upon entering goal state $\rightarrow Q$ estimates improve from goal state back
  1. All $\hat{Q}(s, a)$ start at 0
  2. First episode – only update $\hat{Q}(s, a)$ for transition leading to goal state
  3. Next episode – if go thru this next-to-last transition, will update $\hat{Q}(s, a)$ another step back
Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state.
- Each executed action $a$ results in transition from state $s_i$ to $s_j$; algorithm updates $\hat{Q}(s_i, a)$ using the learning rule.
- Intuition for simple grid world, reward only upon entering goal state $\rightarrow Q$ estimates improve from goal state back.
  1. All $\hat{Q}(s, a)$ start at 0.
  2. First episode – only update $\hat{Q}(s, a)$ for transition leading to goal state.
  3. Next episode – if go thru this next-to-last transition, will update $\hat{Q}(s, a)$ another step back.
  4. Eventually propagate information from transitions with non-zero reward throughout state-action space.
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)

\[ \text{Can choose actions to maximize } \hat{Q}(s, a) \]

Good idea?

Can instead employ stochastic action selection (policy):

\[ P(a_i|s) = \exp(\kappa \hat{Q}(s, a_i)) / \sum_j \exp(\kappa \hat{Q}(s, a_j)) \]

Can vary \( \kappa \) during learning:

- more exploration early on, shift towards exploitation.
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
- Good idea?
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
- Good idea?
- Can instead employ stochastic action selection (policy):

$$P(a_i|s) = \frac{\exp(k\hat{Q}(s, a_i))}{\sum_j \exp(k\hat{Q}(s, a_j))}$$
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
- Good idea?
- Can instead employ stochastic action selection (policy):

$$P(a_i|s) = \frac{\exp(k\hat{Q}(s, a_i))}{\sum_j \exp(k\hat{Q}(s, a_j))}$$

- Can vary $k$ during learning
Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
- Good idea?
- Can instead employ stochastic action selection (policy):

$$P(a_i|s) = \frac{\exp(k\hat{Q}(s, a_i))}{\sum_j \exp(k\hat{Q}(s, a_j))}$$

- Can vary $k$ during learning
  - more exploration early on, shift towards exploitation
Non-deterministic Case

- What if reward and next state are non-deterministic?

\[
\begin{align*}
V_\pi(s) &= E_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots] \\
Q(s, a) &= E [r(s, a) + \gamma V^*(\delta(s, a))] \\
&= E [r(s, a) + \gamma \sum_{s'} p(s' | s, a) \max_{a'} Q(s', a')]
\end{align*}
\]
What if reward and next state are non-deterministic?

We redefine $V, Q$ based on probabilistic estimates, expected values of them:

$$
V^\pi(s) = E_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots]
$$

$$
= E_\pi[\sum_{i=0}^{\infty} \gamma^i r_{t+i}]
$$
Non-deterministic Case

- What if reward and next state are non-deterministic?
- We redefine $V, Q$ based on probabilistic estimates, expected values of them:

\[
V^\pi(s) = E_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots ] = E_\pi \left[ \sum_{i=0}^{\infty} \gamma^i r_{t+i} \right]
\]

and

\[
Q(s, a) = E[r(s, a) + \gamma V^*(\delta(s, a))] = E[r(s, a) + \gamma \sum_{s'} p(s'|s, a) \max_{a'} Q(s', a')]
\]
Non-deterministic Case: Learning $Q$

- Training rule does not converge (can keep changing $\hat{Q}$ even if initialized to true $Q$ values)

\[
\hat{Q}(s, a) \leftarrow (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n \left[ r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') \right]
\]

where $s'$ is the state land in after $s$, and $a'$ indexes the actions that can be taken in state $s'$.

$\alpha_n = \frac{1}{1 + \text{visits}(n, a)}$
Non-deterministic Case: Learning $Q$

- Training rule does not converge (can keep changing $\hat{Q}$ even if initialized to true $Q$ values)

- So modify training rule to change more slowly

$$\hat{Q}(s, a) \leftarrow (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')]$$

where $s'$ is the state land in after $s$, and $a'$ indexes the actions that can be taken in state $s'$

$$\alpha_n = \frac{1}{1 + \text{visits}_n(s, a)}$$

where visits is the number of times action $a$ is taken in state $s$