Topics:

- Reinforcement Learning Part 1
  - Markov Decision Processes
  - Value Iteration
  - (Deep) Q Learning
Administrative

• HW4 is due EOD 11/11. Grace period ends 11/13
• HW3 grades will be released by the end of this week
• Milestone Report grades and feedback will be released by Sunday, 11/13
Reinforcement Learning
Introduction
Reinforcement Learning

- Evaluative feedback in the form of reward
- No supervision on the right action

Types of Machine Learning

Supervised Learning
- Train Input: \( \{X, Y\} \)
- Learning output: \( f : X \rightarrow Y, P(y|x) \)
- e.g. classification

Unsupervised Learning
- Input: \( \{X\} \)
- Learning output: \( P(x) \)
- Example: Clustering, density estimation, generative modeling

Unsupervised Learning
- Example: Clustering, density estimation, generative modeling
RL: Sequential decision making in an environment with evaluative feedback.

- **Environment** may be unknown, non-linear, stochastic and complex.
- **Agent** learns a **policy** to map states of the environments to actions.
  - Seeking to maximize cumulative reward in the long run.

Figure Credit: Rich Sutton
Example: Robot Locomotion

- **Objective**: Make the robot move forward without falling
- **State**: Angle and position of the joints
- **Action**: Torques applied on joints
- **Reward**: +1 at each time step upright and moving forward

Figures copyright John Schulman et al., 2016. Reproduced with permission.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Example: Robot Manipulation

- **Objective**: Pick up object and place to sorting bin
- **State**: Pose of the object and the bin, joint state and velocity of robots
- **Action**: End effector motion
- **Reward**: Inverse distance between the object and the bin
Example: Atari Games

- **Objective**: Complete the game with the highest score
- **State**: Raw pixel inputs of the game state
- **Action**: Game controls e.g. Left, Right, Up, Down
- **Reward**: Score increase/decrease at each time step

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.
Example: Go

- **Objective**: Defeat opponent
- **State**: Board pieces
- **Action**: Where to put next piece down
- **Reward**: +1 if win at the end of game, 0 otherwise

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Deep Learning for Decision Making

state input → Deep Neural Nets → action output
Deep Learning for Decision Making

Problem: we don’t know the correct action label to supervise the output!
Deep Learning for Decision Making

Problem: we don’t know the correct action label to supervise the output!
All we know is the step-wise task reward
Problem: we don’t know the correct action label to supervise the output!
All we know is the step-wise task reward

(Deep) Reinforcement Learning!
Markov Decision Processes
MDPs: Theoretical framework underlying RL
**MDPs**: Theoretical framework underlying RL

An MDP is defined as a tuple \((S, A, R, T, \gamma)\)

- \(S\) : Set of possible states
- \(A\) : Set of possible actions
- \(R(s, a, s')\) : Distribution of reward
- \(T(s, a, s')\) : Transition probability distribution, also written as \(p(s'|s, a)\)
- \(\gamma\) : Discount factor
- **MDPs**: Theoretical framework underlying RL

- An MDP is defined as a tuple \((S, A, R, T, \gamma)\)
  - \(S\): Set of possible states
  - \(A\): Set of possible actions
  - \(R(s, a, s')\): Distribution of reward
  - \(T(s, a, s')\): Transition probability distribution, also written as \(p(s'|s, a)\)
  - \(\gamma\): Discount factor

- **Experience**: \(\ldots s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}, \ldots\)
**MDPs**: Theoretical framework underlying RL

An MDP is defined as a tuple \((S, A, R, T, \gamma)\)

- \(S\) : Set of possible states
- \(A\) : Set of possible actions
- \(R(s,a,s')\) : Distribution of reward
- \(T(s,a,s')\) : Transition probability distribution, also written as \(p(s'|s,a)\)
- \(\gamma\) : Discount factor

**Experience**: \(\ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \ldots\)

**Markov property**: Current state completely characterizes state of the environment

**Assumption**: Most recent observation is a sufficient statistic of history

\[
p(S_{t+1} = s'|S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \ldots S_0 = s_0) = p(S_{t+1} = s'|S_t = s_t, A_t = a_t)
\]
### Fully observed MDP
- Agent receives the true state $s_t$ at time $t$
- Example: Chess, Go

### Partially observed MDP
- Agent perceives its own partial observation $o_t$ of the state $s_t$ at time $t$, using past states e.g. with an RNN
- Example: Poker, First-person games (e.g. Doom)

Source: https://github.com/mwydmuch/ViZDoom
We will assume **fully observed MDPs** for this lecture.
In Reinforcement Learning, we assume an underlying MDP with unknown:

- Transition probability distribution \( T \)
- Reward distribution \( R \)
In **Reinforcement Learning**, we assume an underlying **MDP** with unknown:

- Transition probability distribution $T$
- Reward distribution $R$

Put simply: without learning, the agent doesn’t know how their actions will change the environment and what reward they will receive.

Reinforcement learning is to learn the environment transition and (future) reward by actively interacting with the environment and learning from the experience.
A Grid World MDP

Agent lives in a 2D grid environment

- **State:** Agent's 2D coordinates
- **Actions:** N, E, S, W
- **Rewards:** +1/-1 at absorbing states
- **Walls** block agent's path
- Actions do not always go as planned
  - 20% chance that agent drifts one cell left or right of direction of motion (except when blocked by wall).

Figure credits: Pieter Abbeel
Agent lives in a 2D grid environment

- State: Agent's 2D coordinates
- Actions: N, E, S, W
- Rewards: +1/-1 at absorbing states
- Walls block agent's path
- Actions may not always go as planned
  - 20% chance the agent drifts one cell left or right of the direction of motion (except when blocked by a wall).
- Agent lives in a 2D grid environment
- State: Agent’s 2D coordinates
- Actions: N, E, S, W
- Rewards: +1/-1 at absorbing states

Figure credits: Pieter Abbeel
Agent lives in a 2D grid environment

- State: Agent’s 2D coordinates
- Actions: N, E, S, W
- Rewards: +1/-1 at absorbing states

- Walls block agent’s path
- Actions to not always go as planned
  - 20% chance that agent drifts one cell left or right of direction of motion (except when blocked by wall).

Figure credits: Pieter Abbeel
Solving MDPs by finding the best/optimal policy
Solving MDPs by finding the **best/optimal policy**

Formally, a **policy** is a mapping from states to actions

### Example

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>
• Solving MDPs by finding the **best/optimal policy**

• Formally, a **policy** is a mapping from states to actions
  
  • Deterministic \( \pi(s) = a \)
Solving MDPs by finding the **best/optimal policy**

- Formally, a **policy** is a mapping from states to actions
  - Deterministic $\pi(s) = a$

\[ n = |S| \]
\[ m = |A| \]
Solving MDPs by finding the **best/optimal policy**

Formally, a **policy** is a mapping from states to actions:

- **Deterministic** \( \pi(s) = a \)
- **Stochastic** \( \pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \)

\[ n = |S| \]
\[ m = |A| \]
Solving MDPs by finding the **best/optimal policy**

Formally, a **policy** is a mapping from states to actions

- **Deterministic** \( \pi(s) = a \)
- **Stochastic** \( \pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \)

\[ n = |S| \]
\[ m = |A| \]
Solving MDPs by finding the best/optimal policy

Formally, a policy is a mapping from states to actions
- Deterministic \( \pi(s) = a \)
- Stochastic \( \pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \)

What is a good policy?
- Maximize current reward? Sum of all future rewards?
Solving MDPs by finding the **best/optimal policy**

Formally, a **policy** is a mapping from states to actions

- **Deterministic** \( \pi(s) = a \)
- **Stochastic** \( \pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \)

What is a good policy?

- Maximize **current reward**? Sum of all **future rewards**?
- **Discounted sum of future rewards**!
- How much to value future rewards
  - Discount factor: \( \gamma \)
  - Typically 0.9 - 0.99
Formally, the **optimal policy** is defined as:

$$
\pi^* = \arg \max_\pi \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid \pi \right]
$$
Formally, the **optimal policy** is defined as:

$$\pi^* = \arg\max_\pi \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi \right]$$

(discounted sum of future rewards)
Formally, the **optimal policy** is defined as:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid \pi \right]$$

**discounted sum of future rewards**

Solving MDPs: Optimal policy
Formally, the **optimal policy** is defined as:

\[ \pi^* = \arg\max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid \pi \right] \]

\[ s_0 \sim p(s_0), a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim p(\cdot \mid s_t, a_t) \]

Expectation over initial state, actions from policy, next states from transition distribution.
A value function predicts the sum of discounted future reward.
A value function predicts the sum of discounted future reward

State value function / V-function / $V : S \rightarrow \mathbb{R}$

- How good is this state?
- Am I likely to win/lose the game from this state (reward-to-go)?
- A **value function** predicts the sum of discounted future reward

- **State value function / V-function /** $V : S \rightarrow \mathbb{R}$
  - How good is this state?
  - Am I likely to win/lose the game from this state (reward-to-go)?

- **State-Action value function / Q-function /** $Q : S \times A \rightarrow \mathbb{R}$
  - How good is this state-action pair?
  - In this state, what is the impact of this action on my future?
For a policy that produces a trajectory sample \((s_0, a_0, s_1, a_1, s_2 \cdots)\)
- For a policy that produces a trajectory sample \((s_0, a_0, s_1, a_1, s_2 \cdots)\)

- The **V-function** of the policy at state \(s\), is the expected cumulative reward from state \(s\):

\[
V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right]
\]
For a policy that produces a trajectory sample \( (s_0, a_0, s_1, a_1, s_2 \cdots) \)

The **V-function** of the policy at state \( s \), is the expected cumulative reward from state \( s \):

\[
V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right]
\]
For a policy that produces a trajectory sample \((s_0, a_0, s_1, a_1, s_2 \cdots)\)

The **V-function** of the policy at state \(s\), is the expected cumulative reward from state \(s\):

\[
V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right]
\]

\[
s_0 \sim p(s_0), a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim p(\cdot \mid s_t, a_t)
\]
For a policy that produces a trajectory sample \((s_0, a_0, s_1, a_1, s_2 \cdots)\)

The **Q-function** of the policy at state \(s\) and action \(a\), is the expected cumulative reward upon taking action \(a\) in state \(s\) (and following policy thereafter):
For a policy that produces a trajectory sample \((s_0, a_0, s_1, a_1, s_2 \cdots)\)

The \textbf{Q-function} of the policy at state \(s\) and action \(a\), is the expected cumulative reward upon taking action \(a\) in state \(s\) (and following policy thereafter):

\[
Q^\pi(s, a) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]
\]

\[s_0 \sim p(s_0), a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim p(\cdot \mid s_t, a_t)\]
The V and Q functions corresponding to the optimal policy $\pi^*$

$$V^*(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi^* \right]$$

$$Q^*(s, a) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi^* \right]$$

$$V^*(s) = \max_a Q^*(s, a)$$
Recursive Bellman expansion (from definition of $Q$)

$$Q^*(s, a) = \mathbb{E}_{a_t \sim \pi^*(\cdot | s_t)} \mathbb{E}_{s_{t+1} \sim p(\cdot | s_t, a_t)} \left[ \sum_{t \geq 0} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a \right]$$

(Expected) return from $t = 0$
Recursive Bellman expansion (from definition of Q)

\[ Q^*(s, a) = \mathbb{E}_{a_t \sim \pi^*(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)} \left[ \sum_{t \geq 0} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a \right] \]

\[ = \gamma^0 r(s, a) + \mathbb{E}_{s' \sim p(\cdot | s, a)} \left[ \gamma \mathbb{E}_{a_t \sim \pi^*(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)} \left[ \sum_{t \geq 1} \gamma^{t-1} r(s_t, a_t) \mid s_1 = s' \right] \right] \]

\[ = r(s, a) + \gamma \mathbb{E}_{s' \sim p(s' | s, a)} [V^*(s')] \]

\[ = \mathbb{E}_{s' \sim p(s' | s, a)} [r(s, a) + \gamma V^*(s')] \]

(Reward at t = 0) + gamma * (Return from expected state at t=1)

Bellman Optimality Equations
Equations relating optimal quantities

\[ V^*(s) = \max_{a} Q^*(s, a) \]
\[ \pi^*(s) = \arg \max_{a} Q^*(s, a) \]

Recursive Bellman optimality equation

\[
Q^*(s, a) = \mathbb{E}_{s' \sim p(s'|s,a)} \left[ r(s,a) + \gamma V^*(s) \right]
= \sum_{s'} p(s'|s,a) \left[ r(s,a) + \gamma V^*(s) \right]
= \sum_{s'} p(s'|s,a) \left[ r(s,a) + \gamma \max_{a'} Q^*(s', a') \right]
\]
Equations relating optimal quantities

\[ V^*(s) = \max_a Q^*(s, a) \]
\[ \pi^*(s) = \arg \max_a Q^*(s, a) \]
Equations relating optimal quantities

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

Recursive Bellman optimality equation

\[ Q^*(s, a) = \mathbb{E}_{s' \sim p(s'|s, a)} \left[ r(s, a) + \gamma V^*(s) \right] \]
\[ = \sum_{s'} p(s'|s, a) \left[ r(s, a) + \gamma V^*(s) \right] \]
\[ = \sum_{s'} p(s'|s, a) \left[ r(s, a) + \gamma \max_{a'} Q^*(s', a') \right] \]

\[ V^*(s) = \max_a \sum_{s'} p(s'|s, a) \left[ r(s, a) + \gamma V^*(s') \right] \]
Algorithm: Value Iteration

- Initialize values of all states to arbitrary values, e.g., all 0’s.
- While not converged:
  - For each state: 

\[ V_{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a) \left[ r(s,a) + \gamma V^i(s') \right] \]

- Repeat until convergence (no change in values)

\[ V^0 \rightarrow V^1 \rightarrow V^2 \rightarrow \cdots \rightarrow V^i \rightarrow \cdots \rightarrow V^* \]

**Time complexity per iteration** \( O(|S|^2|A|) \)
Value Iteration Update:

\[
V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a) \left[ r(s,a) + \gamma V^i(s') \right]
\]

Q-Iteration Update:

\[
Q^{i+1}(s,a) \leftarrow \sum_{s'} p(s'|s,a) \left[ r(s,a) + \gamma \max_{a'} Q^i(s',a') \right]
\]

The algorithm is same as value iteration, but it loops over actions as well as states.
Value iteration is almost never used in practice!

Time complexity per iteration \( O(|S|^2|A|) \)

\[
|S| = 11, \quad |A| = 4
\]

\[
|S| \approx 3^{361}, \quad |A| \approx 361
\]

\[
|S| \approx ?, \quad |A| = ?
\]

Can’t iterate over all \((s, a)\) pairs -> need approximation!

We also don’t know the transition function (model) -> need a model-free method!
Q-Learning

- We’d like to do Q-value updates to each Q-state:

\[ Q'(s_t, a_t) \approx \sum_{s'} T(s_{t+1}|s_t, a_t)[r_t + \gamma \max_a Q(s_{t+1}, a)] \]

- But can’t compute this update without knowing all possible next states \( s' \)

- Instead, approximate the expectation with (lots of) experience samples
  - Take an action in the environment following policy \( \arg\max_a Q(s, a) \)
  - receive a sample transition \( (s_t, a_t, r_t, s_{t+1}) \)
  - This sample suggests: \( Q(s_t, a_t) \approx r_t + \gamma \max_a Q(s_{t+1}, a) \)
  - Keep a running average to approximate the expectation:
    \[ Q'(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)] \]

Still need to represent all \((s, a)\) pairs in a Q value table!
Q-Learning

Idea: represent the Q value table as a parametric function $Q_\theta(s, a)$.

How do we learn the function?

\[
Q'(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)] \\
= Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
\]

Now, at optimum, $Q(s_t, a_t) = Q'(s_t, a_t) = Q^*(s_t, a_t)$; This gives us:

\[
0 = 0 + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
\]

Learning problem:

\[
\arg\min_\theta ||r_t + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)||
\]

Target Q value
Deep Q-Learning
\begin{itemize}
  \item **Q-Learning with linear function approximators**
    \[
    Q(s, a; w, b) = w_a^T s + b_a
    \]
    - Has some theoretical guarantees
  \item **Deep Q-Learning**: Fit a deep Q-Network
    - Works well in practice
    - Q-Network can take arbitrary input (e.g. RGB images)
    - Assume discrete action space (e.g., left, right)
\end{itemize}
Assume we have collected a dataset:

\[ \{(s, a, s', r)\}_{i=1}^{N} \]

We want a Q-function that satisfies bellman optimality (Q-value)

\[
Q^*(s, a) = \mathbb{E}_{s' \sim p(s'|s,a)} \left[ r(s, a) + \gamma \max_{a'} Q^*(s', a') \right]
\]

Loss for a single data point:

\[
\text{MSE Loss} := \left( Q_{new}(s, a) - (r + \gamma \max_{a} Q_{old}(s', a)) \right)^2
\]

- Predicted Q-Value
- Target Q-Value
Minibatch of \( \{(s, a, s', r)\}_i \)

Forward pass:
- State \( B \times D \) → Q-Network \( B \times n_{actions} \) → Q-Values per action

Compute loss:
\[
\left( Q_{new}(s, a) - (r + \gamma \max_a Q_{old}(s', a)) \right)^2
\]

\( \theta_{new} \) - \( \theta_{old} \)

Backward pass:
\[
\frac{\partial \text{Loss}}{\partial \theta_{new}}
\]
\[ \text{MSE Loss} := \left( Q_{new}(s, a) - (r + \max_a Q_{old}(s', a)) \right)^2 \]

- In practice, for stability:
  - Freeze \( Q_{old} \) and update \( Q_{new} \) parameters
  - Set \( Q_{old} \leftarrow Q_{new} \) at regular intervals or update as running average
  - \( \theta_{old} = \beta \theta_{old} + (1 - \beta) \theta_{new} \)
How to gather experience?

$$\{(s, a, s', r)_i\}_{i=1}^N$$

This is why RL is hard
How to gather experience?

Challenge 1: Exploration vs Exploitation

Challenge 2: Non iid, highly correlated data
What should \( \pi_{\text{gather}} \) be?

- Greedy? -> no exploration, always choose the most confident action

\[
\arg \max_a Q(s, a; \theta)
\]

- An exploration strategy:

- \( \epsilon \)-greedy

\[
a_t = \begin{cases} 
\arg \max_a Q(s, a) & \text{with probability } 1 - \epsilon \\
\text{random action} & \text{with probability } \epsilon 
\end{cases}
\]
• Samples are correlated => high variance gradients => **inefficient learning**

• Current Q-network parameters determines next training samples => can lead to **bad feedback loops**
  - e.g. if maximizing action is to move right, training samples will be dominated by samples going right, may fall into local minima
Correlated data: addressed by using experience replay

- A replay buffer stores transitions \((s, a, s', r)\)

- Continually update replay buffer as game (experience) episodes are played, older samples discarded

- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

- Larger the buffer, lower the correlation
Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights

for episode $= 1, M$ do
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ do
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_i, a_i, r_i, \phi_{i+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{i+1}, a'; \theta) & \text{for non-terminal } \phi_{i+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for
Atari Games

- **Objective**: Complete the game with the highest score
- **State**: Raw pixel inputs of the game state
- **Action**: Game controls e.g. Left, Right, Up, Down
- **Reward**: Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.
Atari Games

https://www.youtube.com/watch?v=V1eYniJ0Rnk
Different RL Paradigms

- **Value-based RL**
  - (Deep) Q-Learning, approximating $Q^*(s, a)$ with a deep Q-network

- **Policy-based RL**
  - Directly approximate optimal policy $\pi^*$ with a parametrized policy $\pi^*_\theta$

- **Model-based RL**
  - Approximate transition function $T(s', a, s)$ and reward function $R(s, a)$
  - Plan by looking ahead in the (approx.) future!
Today, we saw

- **MDPs**: Theoretical framework underlying RL, solving MDPs
- **Policy**: How an agent acts at states
- **Value function (Utility)**: How good is a particular state or state-action pair?

Solving an MDP with known rewards/transition

- **Value Iteration**: Bellman update to state value estimates
- **Q-Value Iteration**: Bellman update to (state, action) value estimates

**Policy Iteration**

- Policy evaluation + refinement
Next Time: RL continued --- Policy Gradient and Actor-Critic