

Topics:

- Reinforcement Learning Part 1
 - Markov Decision Processes
 - Value Iteration

CS 4803-DL / 7643-A
ZSOLT KIRA

- **Assignment 4 out**
 - Due date **extended** to **April 8th 11:59pm EST.**
- **Projects**
 - Will try to get **feedback** back to you before project period starts
- **Outline of rest of course:**
 - Reinforcement Learning
 - Guest lectures/other topics (e.g. self-supervised learning, audio)
 - **April 7th:** Wav2Vec !!
 - **April 9th:** Ishan Misra (FB) on Self-Supervised Learning
 - **April 14th:** Automatic Speech Recognition Systems
 - Generative models (VAEs / GANs)



Nirbhay Modhe

Nirbhay Modhe is a PhD Student in the School of Interactive Computing at Georgia Tech advised by Prof. Dhruv Batra. His research interests within Reinforcement Learning (RL) include model based RL, generalization guarantees in RL and unsupervised or reward-free RL for exploration. Prior to starting his PhD program in 2017, he received his Bachelor's degree in Computer Science at the Indian Institute of Technology (IIT), Kanpur where he worked with Prof. Piyush Rai on Bayesian ML applied to multi-label learning.

Slides Brought to You By...



Previous Lecture

- ◆ **RL:** Definitions, interaction API, tasks/challenges
- ◆ **MDPs:** Theoretical framework underlying RL, solving MDPs

Today

- ◆ **Policy** (continued): How an agents acts at states
- ◆ **Value function (Utility):** How good is a particular state or state-action pair?
- ◆ **Algorithms** for solving MDPs (Value Iteration)
- ◆ Departure from known rewards and transitions: **Reinforcement Learning (RL), Deep RL**

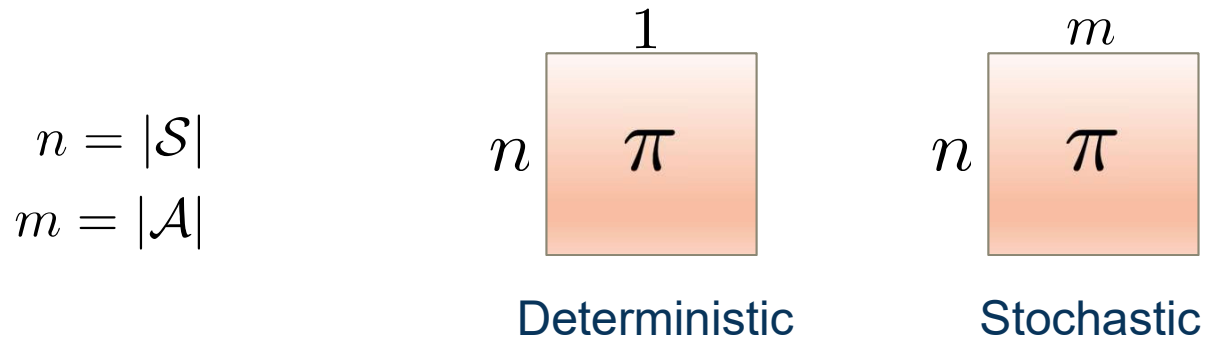
Recap & Overview

- Markov Decision Processes (MDPs)
 - States, Actions, Reward dist., Transition dist., Discount factor (gamma)

MDP
 $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$

- Policy:
 - Mapping from states to actions (deterministic)
 - Distribution of actions given states (stochastic)

State	Action
A	→ 2
B	→ 1



Recap: MDPs, Policy

- Markov Decision Processes (MDPs)
 - States, Actions, Reward dist., Transition dist., Discount factor (gamma)
- Policy:
 - Mapping from states to actions (deterministic)
 - Distribution of actions given states (stochastic)
- What is a good policy?
 - Maximize **discounted sum of future rewards**
 - Discount factor: γ

MDP
 $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$

State	Action
A	→ 2
B	→ 1



1

Worth Now



γ

Worth Next Step



γ^2

Worth In Two Steps

Recap: MDPs, Policy

- Formally, the **optimal policy** is defined as:

discounted sum of future rewards

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

?

- Formally, the **optimal policy** is defined as:

discounted sum of future rewards

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

$$\mathbf{s}_0 \sim p(\mathbf{s}_0), a_t \sim \pi(\cdot | \mathbf{s}_t), \mathbf{s}_{t+1} \sim p(\cdot | \mathbf{s}_t, a_t)$$

Expectation over initial state, actions from policy,
next states from transition distribution

- A **value function** is a prediction of discounted sum of future reward
- **State** value function / **V**-function / $V : \mathcal{S} \rightarrow \mathbb{R}$
 - How good is this state?
 - Am I likely to win/lose the game from this state?
- **State-Action** value function / **Q**-function / $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
 - 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 $(s_0, a_0, s_1, a_1, s_2 \dots)$
- The **V-function** of the policy at state s , is the expected cumulative reward from state s :

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

$$s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$$

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

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

$$\mathbf{s}_0 \sim p(\mathbf{s}_0), a_t \sim \pi(\cdot \mid \mathbf{s}_t), \mathbf{s}_{t+1} \sim p(\cdot \mid \mathbf{s}_t, a_t)$$

- The V and Q functions corresponding to the optimal policy π^*

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

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

● Recursive Bellman expansion (from definition of Q)

$$\begin{aligned}
 Q^*(s, a) &= \mathbb{E}_{\substack{a_t \sim \pi^*(\cdot | s_t) \\ s_{t+1} \sim p(\cdot | s_t, a_t)}} \left[\sum_{t \geq 0} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a \right] \\
 &\quad \text{(Expected) return from } t = 0 \\
 &= \gamma^0 r(s, a) + \mathbb{E}_{s' \sim p(\cdot | s, a)} \left[\gamma \mathbb{E}_{\substack{a_t \sim \pi^*(\cdot | s_t) \\ s_{t+1} \sim p(\cdot | s_t, a_t)}} \left[\sum_{t \geq 1} \gamma^{t-1} r(s_t, a_t) \mid s_1 = s' \right] \right] \\
 &\quad \text{(Reward at } t = 0) + \text{(Return for expected state at } t=1) \\
 &= r(s, a) + \gamma \mathbb{E}_{s' \sim p(s' | s, a)} [V^*(s')] \\
 &= \mathbb{E}_{s' \sim p(s' | s, a)} [r(s, a) + \gamma V^*(s')]
 \end{aligned}$$

- Equations relating optimal quantities

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

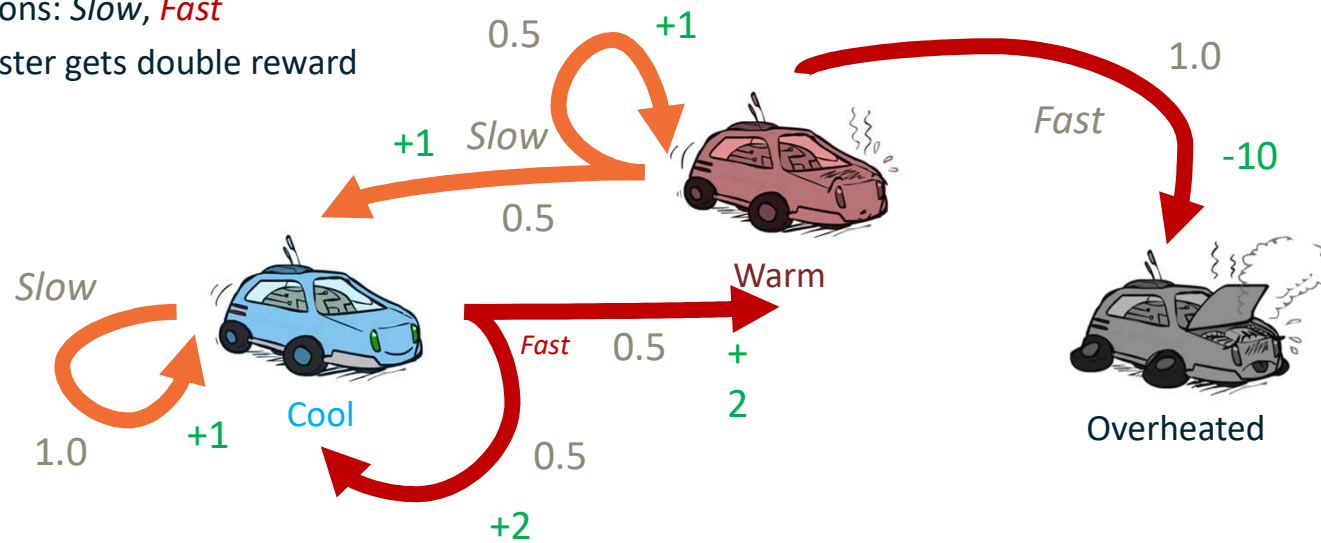
$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

- Recursive Bellman optimality equation

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

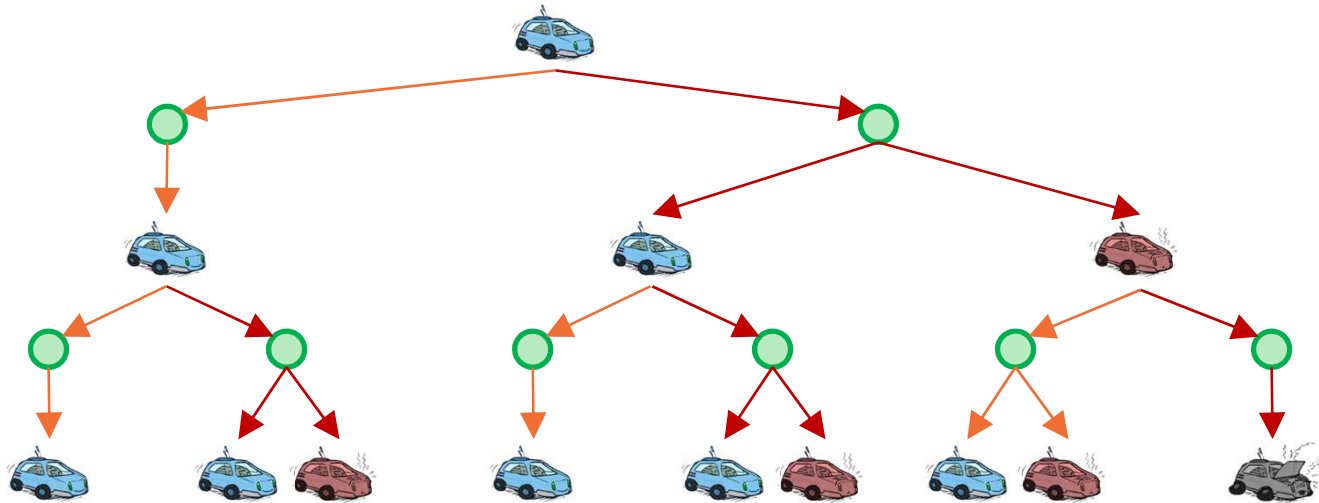
$$V^*(s) = \max_a \sum_{s'} p(s' | s, a) [r(s, a) + \gamma V^*(s')]$$

- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward



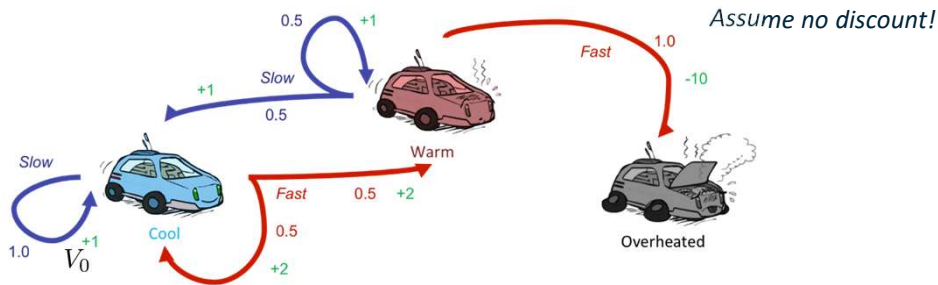
Slide Credit: <http://ai.berkeley.edu>

Example: Racing






Slide Credit: <http://ai.berkeley.edu>

Racing Search Tree

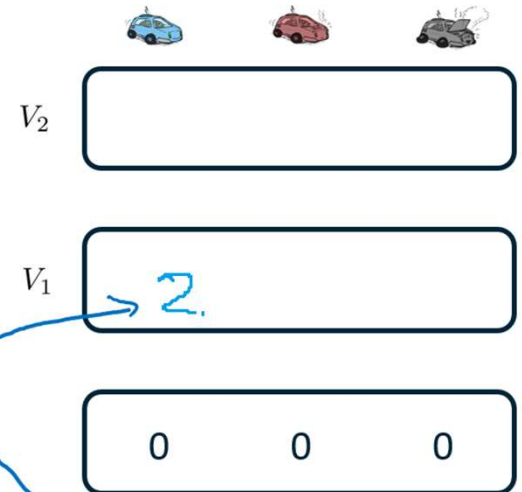
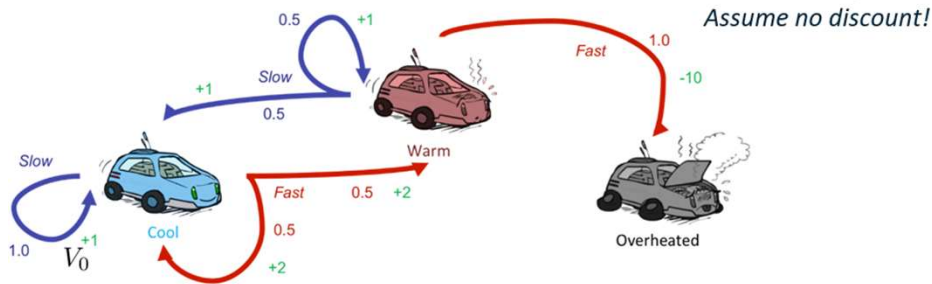


$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

			
V_2	3.5	2.5	0
V_1	2	1	0
	0	0	0

Slide Credit: <http://ai.berkeley.edu>

Racing Search Tree



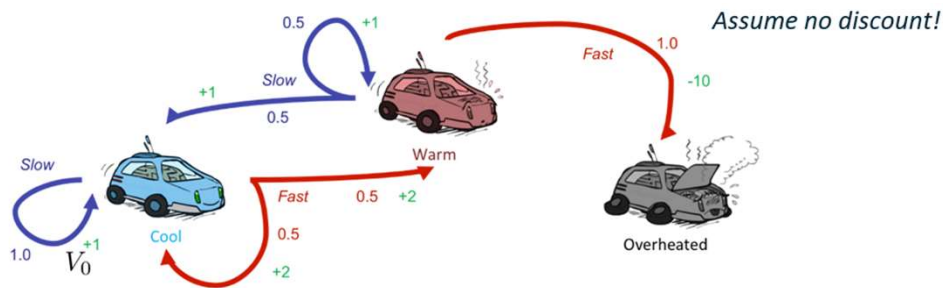
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

$$V_1(s_0) = \max \left(\begin{array}{l} \overset{\text{slow}}{1.0 \cdot (+1 + V(s_0))} \\ \overset{\text{Fast}}{0.5 [+2 + V(s_1)] + 0.5 [+2 \cdot V(s_0)]} \end{array} \right) = 2$$

Slide Credit: <http://ai.berkeley.edu>

Racing Search Tree





$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

$$V_2(s_0) = \max \left[\begin{array}{l} \text{slow} \\ 1.0 (+1 + V_1(s_0)) \quad \Rightarrow 3 \\ \text{fast} \\ 0.5 (+2 + V_1(s_0)) + 0.5 (+2 + V(s_1)) \quad \Rightarrow 3.5 \end{array} \right]$$

$\Rightarrow 3.5$

V_2	3.5	2.5	0
V_1	2	1	0
	0	0	0

Slide Credit: <http://ai.berkeley.edu>

Racing Search Tree

Value Iteration Update:

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

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

Policy iteration: Start with arbitrary π_0 and refine it.

$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \dots \rightarrow \pi^*$$

Involves repeating two steps:

- **Policy Evaluation:** Compute V^π (similar to Value Iteration)
- **Policy Refinement:** Greedily change actions as per V^π at next states

$$\pi_i(s) \leftarrow \operatorname{argmax}_a \sum_{s'} p(s' | s, a) [r(s, a) + \gamma V^{\pi_i}(s')]$$

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \dots \rightarrow \pi^* \rightarrow V^{\pi^*}$$

Why do policy iteration?

- π_i often converges to π^* much sooner than V^{π_i} to V^{π^*}

For Value Iteration:

Theorem: will converge to unique optimal values

Basic idea: approximations get refined towards optimal values

Policy may converge long before values do

Time complexity per iteration $O(|\mathcal{S}|^2|\mathcal{A}|)$

Feasible for:

- ◆ 3x4 Grid world?
- ◆ Chess/Go?
- ◆ Atari Games with integer image pixel values [0, 255] of size 16x16 as state?

Summary: MDP Algorithms

Value Iteration

- ◆ Bellman update to state value estimates

Q-Value Iteration

- ◆ Bellman update to (state, action) value estimates

Policy Iteration

- ◆ Policy evaluation + refinement



Reinforcement Learning, Deep RL

- Recall RL assumptions:
 - $\mathbb{T}(s, a, s')$ unknown, how actions affect the environment.
 - $\mathcal{R}(s, a, s')$ unknown, what/when are the good actions?
- But, we can learn by trial and error.
 - Gather experience (data) by performing actions.
 - Approximate unknown quantities from data.

Reinforcement Learning

- Old Dynamic Programming Demo
 - https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html
- RL Demo
 - https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_td.html

Slide credit: Dhruv Batra

Q-Learning

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

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right]$$

- But can't compute this update without knowing T, R

- Instead, compute average as we go

- Receive a sample transition (s,a,r,s')
- This sample suggests

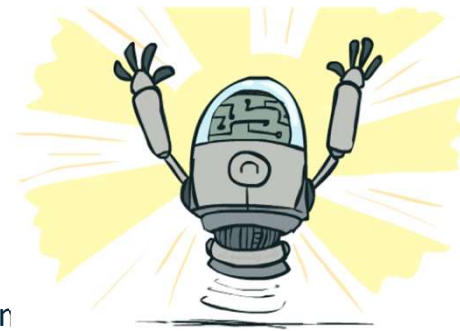
$$Q(s, a) \approx r + \gamma \max_{a'} Q(s', a')$$

- But we want to average over results from (s,a)
- So keep a running average

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha) \left[r + \gamma \max_{a'} Q(s', a') \right]$$

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called **off-policy learning**
- Caveats:
 - You have to explore enough
 - You have to eventually make the learning rate small enough
 - ... but not decrease it too quickly
 - Basically, in the limit, it doesn't matter how you select action

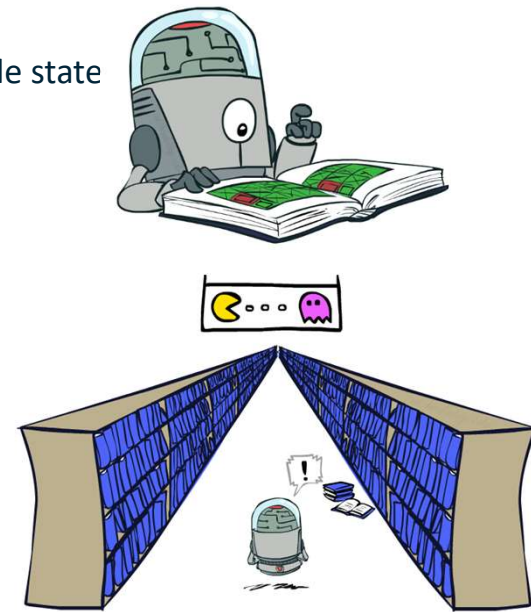


[Demo: Q-lear

Deep Q-Learning

Generalizing Across States

- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state
 - Too many states to visit them all in training
 - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
 - Learn about some small number of training states from experience
 - Generalize that experience to new, similar situations
 - This is the fundamental idea in machine learning!



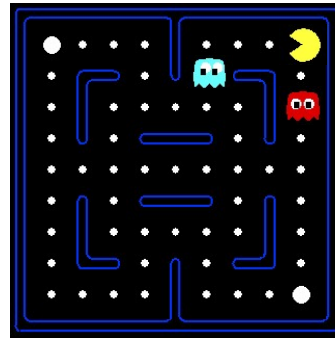
[demo – RL pacman]

Example: Pacman

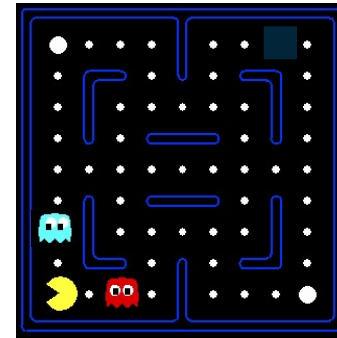
Let's say we discover through experience that this state is bad:



In naïve q-learning, we know nothing about this state:

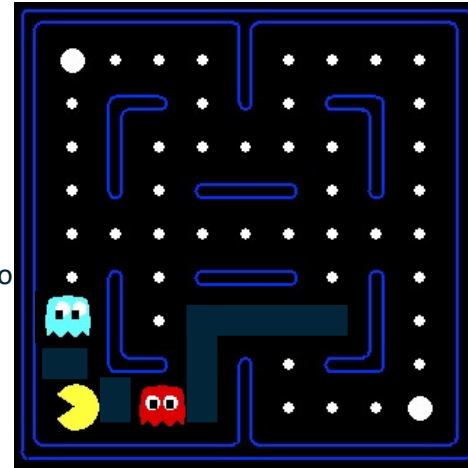


Or even this one!



Feature-Based Representations

- Solution: describe a state using a vector of features (properties)
 - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
 - Example features:
 - Distance to closest ghost
 - Distance to closest dot
 - Number of ghosts
 - $1 / (\text{dist to dot})^2$
 - Is Pacman in a tunnel? (0/1)
 - etc.
 - Is it the exact state on this slide?
 - Can also describe a q-state (s, a) with features (e.g. action moves closer to



Linear Value Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

- Advantage: our expected value function is $Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$
- Disadvantage: states may share features but actually be very different in value!

- State space is too large and complicated for feature engineering though!
- Recall: Value iteration not scalable (chess, RGB images as state space, etc)
- Solution: Deep Learning! ... more precisely, function approximation.
 - Use deep neural networks to learn state representations
 - Useful for continuous action spaces as well

Deep Reinforcement Learning

- **Value-based RL**

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

- **Policy-based RL**

- Directly approximate optimal policy π^* with a parametrized policy π_θ^*

- **Model-based RL**

- Approximate transition function $T(s', a, s)$ and reward function $\mathcal{R}(s, a)$
- Plan by looking ahead in the (approx.) future!

Homework

- **Q-Learning with linear function approximators**

$$Q(s, a; w, b) = w_a^\top s + b_a$$

- Has some theoretical guarantees
- **Deep Q-Learning: Fit a deep Q-Network** $Q(s, a; \theta)$
 - Works well in practice
 - Q-Network can take RGB images

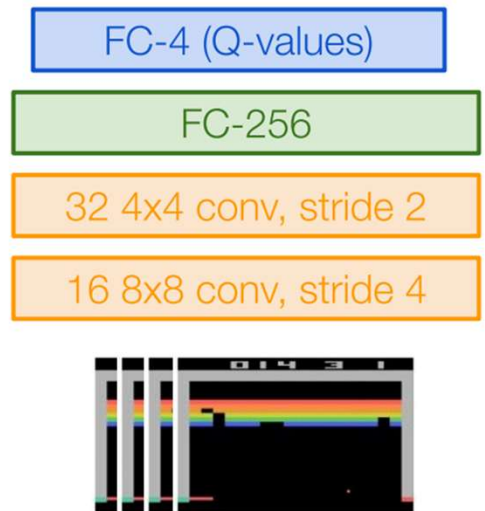


Image Credits: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

- Assume we have collected a dataset:

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

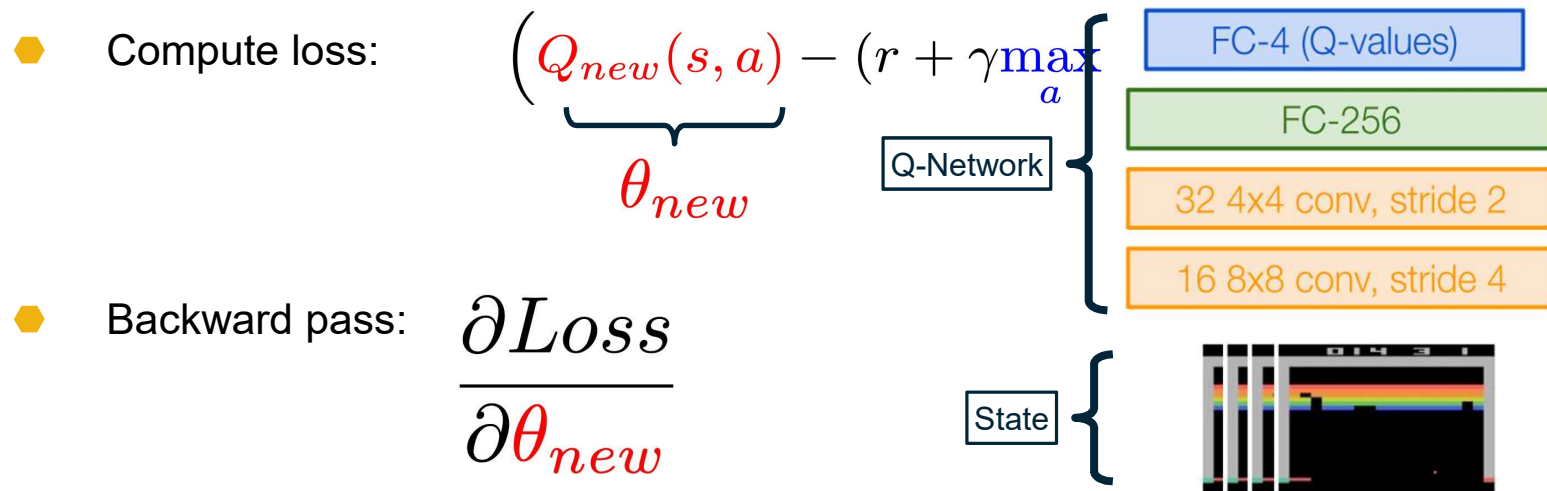
- We want a Q-function that satisfies bellman optimality (Q-value)

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

- Loss for a single data point:

$$\text{MSE Loss} := \left(\underbrace{Q_{\text{new}}(s, a)}_{\text{Predicted Q-Value}} - \underbrace{\left(r + \gamma \max_a Q_{\text{old}}(s', a) \right)}_{\text{Target Q-Value}} \right)^2$$

- Minibatch of $\{(s, a, s', r)_i\}_{i=1}^B$



- Backward pass: $\frac{\partial Loss}{\partial \theta_{new}}$

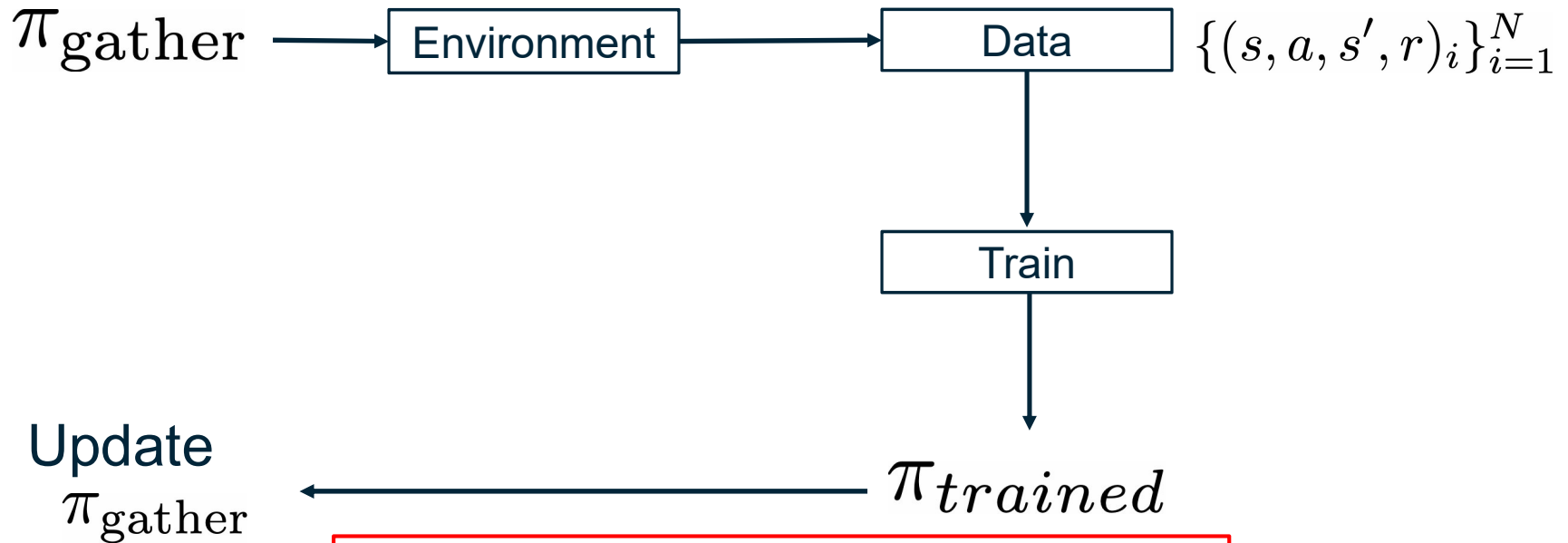
$$\text{MSE Loss} := \left(Q_{new}(s, a) - \left(r + \max_a Q_{old}(s', a) \right) \right)^2$$

- ◆ In practice, for stability:
 - ◆ Freeze Q_{old} and update Q_{new} parameters
 - ◆ Set $Q_{old} \leftarrow Q_{new}$ at regular intervals

How to gather experience?

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

This is why RL is hard



Challenge 1: Exploration vs Exploitation

Challenge 2: Non iid, highly correlated data

How to gather experience?

◆ What should π_{gather} be?

◆ Greedy? -> Local minimas, no exploration

$$\arg \max_a Q(s, a; \theta)$$

◆ An exploration strategy:

◆ ϵ -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

Experience Replay

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

Epsilon-greedy

 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 $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

Q Update

 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

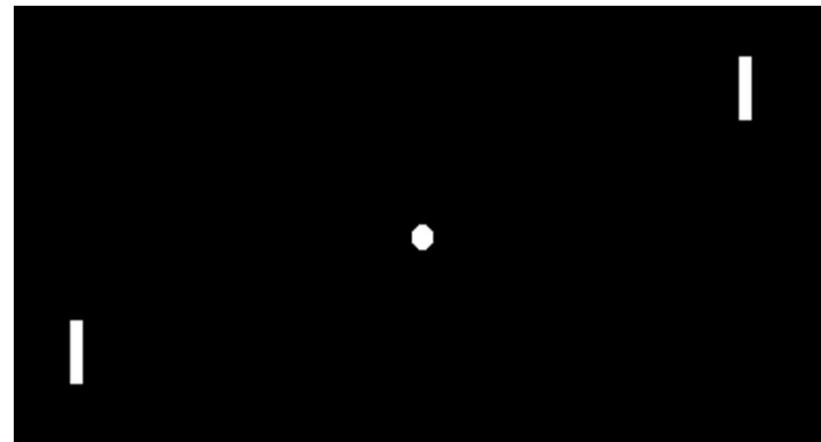
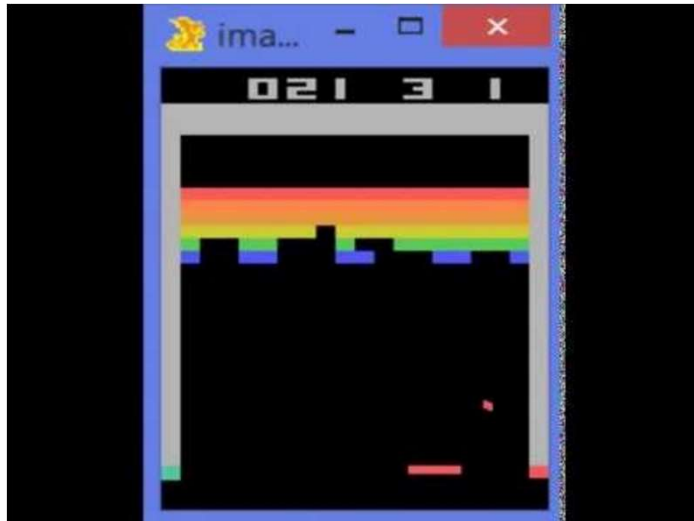
Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

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

Case study: Playing Atari Games



Atari Games



<https://www.youtube.com/watch?v=V1eYniJORnk>

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

Case study: Playing Atari Games



In today's class, we looked at

- ◆ **Dynamic Programming**
 - ◆ Value, Q-Value Iteration
 - ◆ Policy Iteration

- ◆ **Reinforcement Learning (RL)**
 - ◆ The challenges of (deep) learning based methods
 - ◆ Value-based RL algorithms
 - ◆ Deep Q-Learning

Next class:

- ◆ **Policy-based RL algorithms** (policy gradients)

Summary