CS 4803 / 7643: Deep Learning

Topic:

- Reinforcement Learning (RL)
  - Overview
  - Markov Decision Processes

Zsolt Kira

Georgia Tech

Administrative

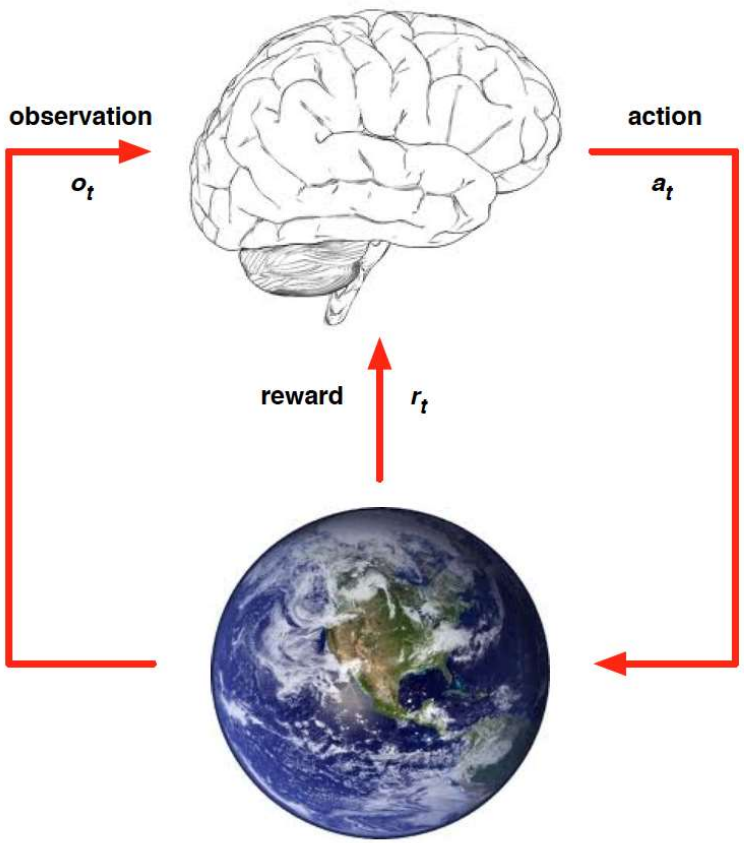
- PS3/HW3 due March 15th!
- Projects
  - 2 new FB projects up (https://www.cc.gatech.edu/classes/AY2020/cs7643_spring/fb_projects.html)
    - Project 1: Confident Machine Translation
    - Project 2: Habitat Embodied Navigation Challenge @ CVPR20
    - Project 3: MRI analysis
    - Project 4: Transfer learning for machine translation quality estimation
  - Tentative FB plan:
    - March 20th: Phone call with FB
    - April 5th: Written Q&A
    - April 15th: Phone call with FB
  - Fill out spreadsheet: https://gtvault-my.sharepoint.com/:x:/g/personal/sdharur3_gatech_edu/EVXbNc4oxelMmj1T5WsEIRQBE4Hn532GeLQVcmOnWdG2Jg?e=dIGNfX

From Last Time

- Overview of RL
  - RL vs other forms of learning
  - RL "API"
  - Applications
- Framework: Markov Decision Processes (MDP's)
  - Definitions and notations
  - Policies and Value Functions
  - Solving MDP's
    - Value Iteration
    - Policy Iteration
- Reinforcement learning
  - Value-based RL (Q-learning, Deep-Q Learning)
  - Policy-based RL (Policy gradients)

Last lecture:
– Focus on MDP's
– No learning (deep or otherwise)

RL API



- At each step t the agent:
  - Executes action a_t
  - Receives observation o_t
  - Receives scalar reward r_t
- The environment:
  - Receives action a_t
  - Emits observation o_{t+1}
  - Emits scalar reward r_{t+1}

Markov Decision Process (MDP)

- RL operates within a framework called a Markov Decision Process
- MDP's: General formulation for decision making under uncertainty

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{T}, \gamma)$

$\mathcal{S}$: set of possible states [start state = s_0, optional terminal / absorbing state]
$\mathcal{A}$: set of possible actions
$\mathcal{R}(s, a, s')$: distribution of reward given (state, action, next state) tuple
$\mathbb{T}(s, a, s')$: transition probability distribution, also written as $p(s'|s, a)$
γ : discount factor

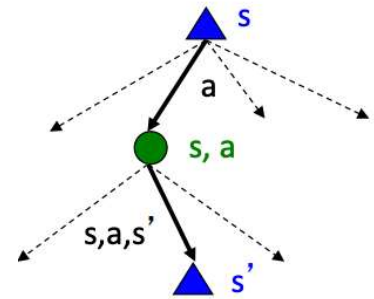
- Life is trajectory: $\ldots \underline{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}}, r_{t+2}, s_{t+2}, \ldots$

- **Markov property**: Current state completely characterizes state of the world
- **Assumption**: Most recent observation is sufficient statistic of history

$$p\left(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \ldots S_0 = s_0\right) = p\left(S_{t+1} = s' | S_t = s_t, A_t = a_t\right)$$

Markov Decision Process (MDP)

- MDP state projects a search tree



- **Observability**:
  - **Full:** In a fully observable MDP, $o_t = s_t$
    - Example: Chess
  - **Partial:** In a partially observable MDP, agent *constructs* its own state, using history, of beliefs of world state, or an RNN, …
    - Example: Poker

Markov Decision Process (MDP)

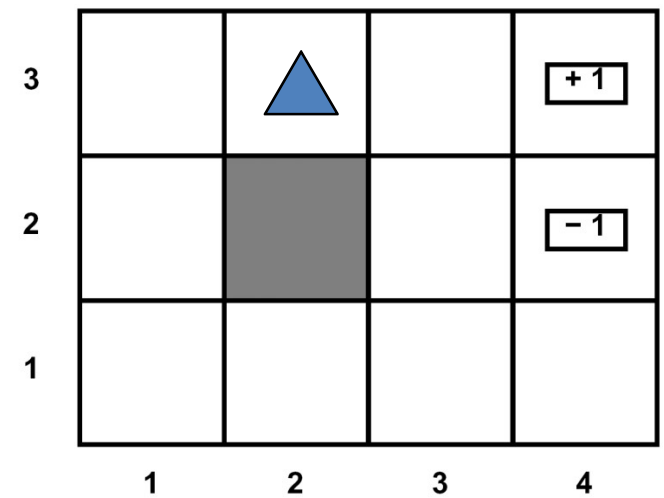
- In RL, we don't have access to $\mathbb{T}$ or $\mathcal{R}$ (i.e. the environment)
  - Need to *actually try* actions and states out to learn
  - Sometimes, need to model the environment

- Last time, assumed we *do* have access to how the world works

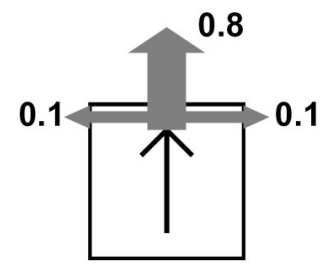
- And that our goal is to find an optimal behavior strategy for an agent

Canonical Example: Grid World

- Agent lives in a grid
- Walls block the agent's path
- Actions do not always go as planned
  - 80% of the time, action North takes the agent North (if there is no wall)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall, the agent stays put



- State: Agent's location
- Actions: N, E, S, W
- Rewards: +1 / -1 at absorbing states
  - Also small "living" reward each step (negative)



Policy

- A policy is how the agent acts

e.g.

State	Action
A $\longrightarrow$	2
B $\longrightarrow$	1

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

The optimal policy π*

What's a good policy?

Maximizes current reward? Sum of all future reward?

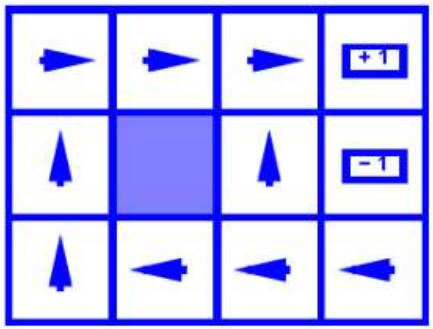
Discounted future rewards!

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

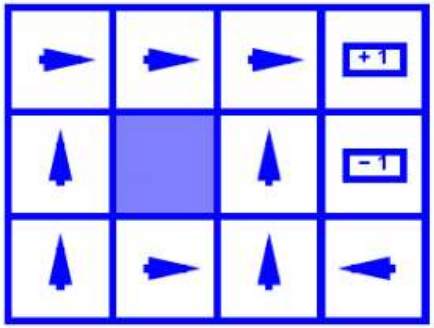
(Typically for a fixed horizon T)

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

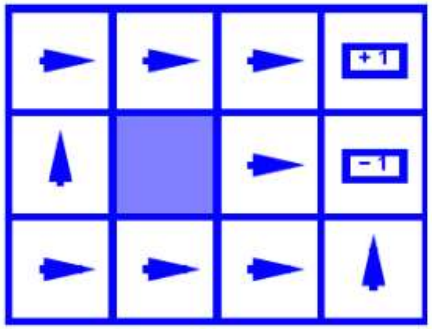
The optimal policy π^*



R(s) = -0.03



R(s) = -0.4



R(s) = -2.0

Reward

at every non-terminal state (living reward/ penalty)

Value Function

- A value function is a prediction of future reward

- State Value Function or simply **Value Function**
  - How good is a state?
  - Am I screwed? Am I winning this game?

- Action-Value Function or **Q-function**
  - How good is a state action-pair?
  - Should I do this now?

Value Function

Following policy π that produces sample trajectories s_0, a_0, r_0, s_1, a_1, …

How good is a state?

The **value function** at state s, is the expected cumulative reward from state s (and following the policy thereafter):

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

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s (and following the policy thereafter):

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

Optimal Quantities

Given *optimal* policy π^* that produces sample trajectories s_0, a_0, r_0, s_1, a_1, …

How good is a state?

The **optimal value function** at state s, and acting optimally thereafter

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

How good is a state-action pair?

The **optimal Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and acting optimally thereafter

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

Recursive definition of value

- Extracting optimal value / policy from Q-values:

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

Recursive definition of value

- Extracting optimal value / policy from Q-values:

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

- Bellman Equations:

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

Recursive definition of value

- Extracting optimal value / policy from Q-values:

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

- Bellman Equations:

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

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

Recursive definition of value

- Extracting optimal value / policy from Q-values:

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

- Bellman Equations:

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

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

- Characterize optimal values in a way we'll use over and over

Value Iteration (VI)

- Bellman equations characterize optimal values, VI is a fixed-point DP solution method to *compute* it

Value Iteration (VI)

- Bellman equations characterize optimal values, VI is a fixed-point DP solution method to *compute* it

- Algorithm
  - Initialize values of all states $V_0(s) = 0$
  - Update: $V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]$
  - Repeat until convergence (to V^*)

Value Iteration (VI)

- Bellman equations characterize optimal values, VI is a fixed-point DP solution method to *compute* it

- Algorithm
  - Initialize values of all states $V_0(s) = 0$
  - Update: $V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]$
  - Repeat until convergence (to V^*)

- Complexity per iteration (DP): $O(|S|^2|A|)$

Value Iteration (VI)

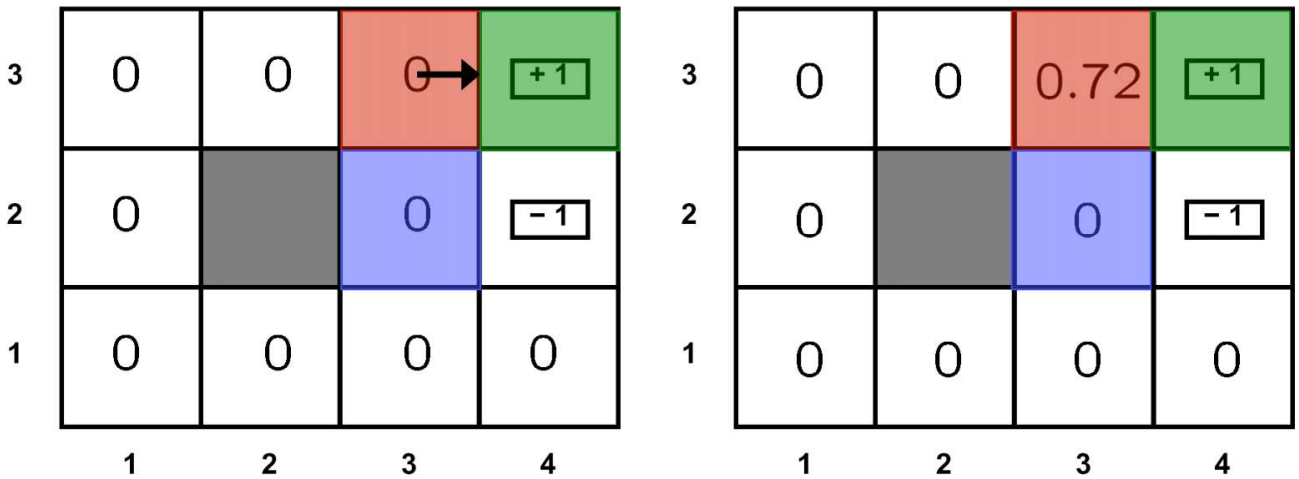
- Bellman equations characterize optimal values, VI is a fixed-point DP solution method to *compute* it

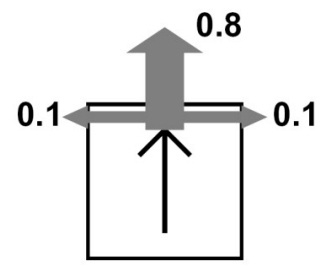
- Algorithm
  - Initialize values of all states $V_0(s) = 0$
  - Update: $V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]$
  - Repeat until convergence (to V^*)

- Complexity per iteration (DP): $O(|S|^2|A|)$

- Convergence
  - Guaranteed for $\gamma < 1$
  - Sketch: Approximations get refined towards optimal values
  - In practice, policy may converge before values do

Value Iteration (VI)





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

$$V^2(\langle 3,3\rangle) = \sum_{s'} P\left(s' | \text{right}, \langle 3,3\rangle\right)\left[r(\langle 3,3\rangle) + \gamma V^1\left(s'\right)\right]$$

$$= 0.9[0.8 \cdot 1 + 0.1 \cdot 0 + 0.1 \cdot 0]$$

[NOTE: Here we are showing calculations for the action we know is argmax (go right), but in general we have to calculate this for *each* actions and return max]

Slide credit: Pieter Abbeel

Q-Value Iteration

- Value Iteration Update:

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

- Remember: $Q^*(s,a) = \sum_{s'} p\left(s'|s,a\right)\left[r(s,a) + \gamma V^*\left(s'\right)\right]$
- Q-Value Iteration Update:

$$Q^{i+1}(s,a) \leftarrow$$

Q-Value Iteration

- Value Iteration Update:

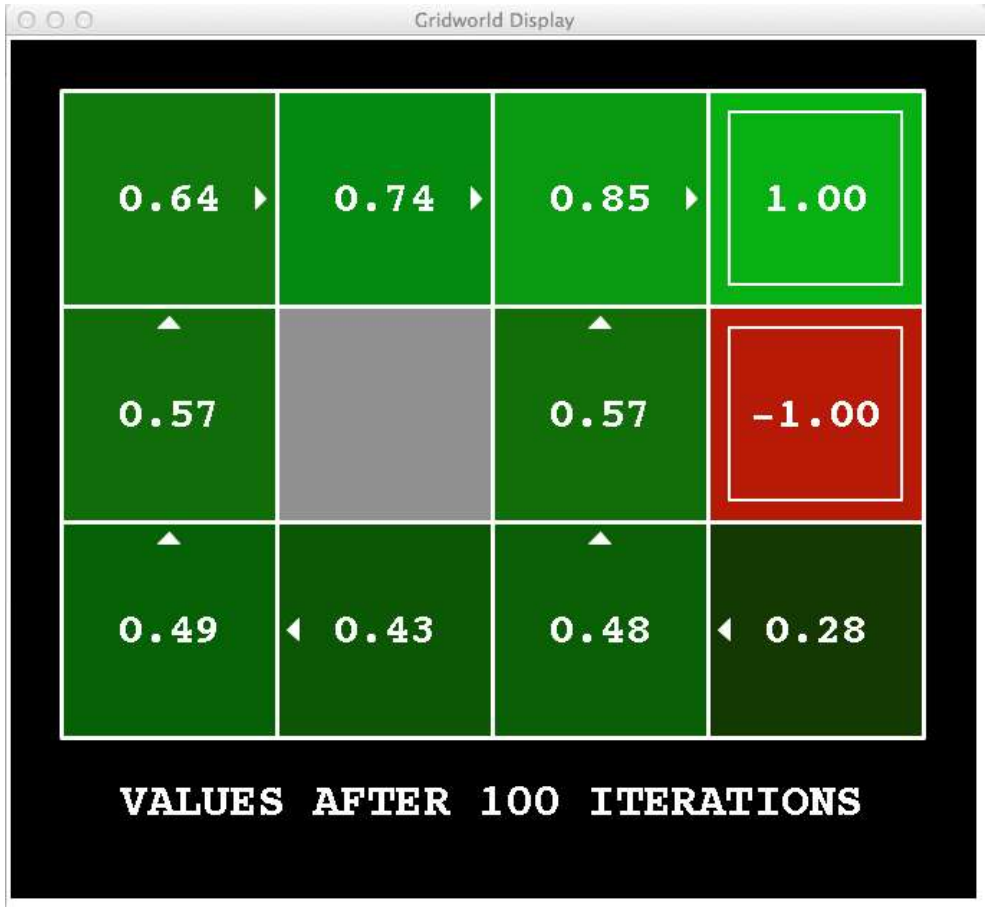
$$V^{i+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V^i(s')\right]$$

- Remember: $Q^*(s,a) = \sum_{s'} p\left(s'|s,a\right)\left[r(s,a) + \gamma V^*\left(s'\right)\right]$
- Q-Value Iteration Update:

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

The algorithm is same as value iteration,
but it loops over actions as well as states

Snapshot of Demo – Gridworld V Values



Noise = 0.2

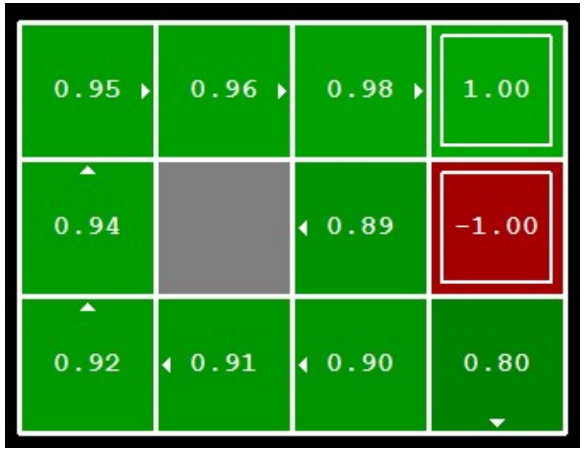
Discount = 0.9

Living reward = 0

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

Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!
- We need to do a one step calculation

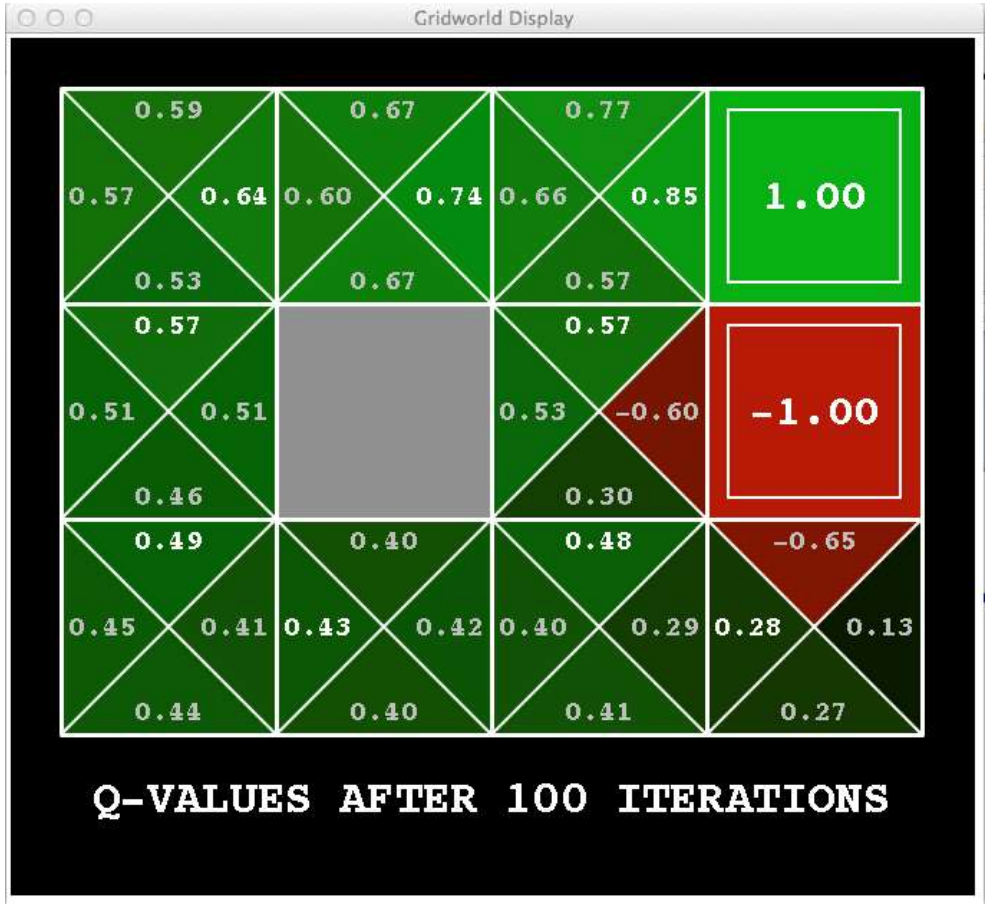


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- This is called policy extraction, since it gets the policy implied by the values

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

Snapshot of Demo – Gridworld Q Values



Noise = 0.2

Discount = 0.9

Living reward = 0

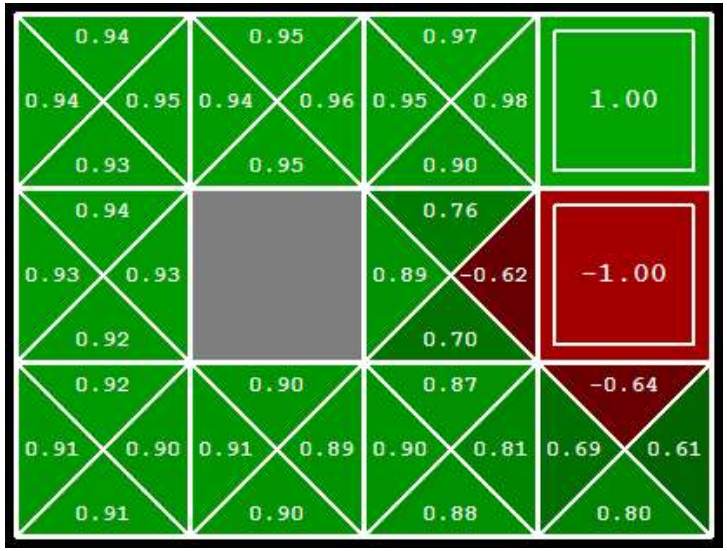
Slide Credit: http://ai.berkeley.edu

Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

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



- Important lesson: actions are easier to select from q-values than values!

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

Demo

- https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html

Next class

- Solving MDP's
  - Policy Iteration
- Reinforcement learning
  - Value-based RL
    - Q-learning
    - Deep Q Learning

Policy Iteration

Policy Iteration

- Policy iteration: Start with arbitrary π_0 and refine it.

$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \quad \ldots \quad \rightarrow \pi^*$$

Policy Iteration

- Policy iteration: Start with arbitrary π_0 and refine it.

$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \quad \ldots \quad \rightarrow \pi^*$$

- Involves repeating two steps:
  - Policy Evaluation: Compute V^π (similar to VI)
  - Policy Refinement: Greedily change actions as per V^π

$$\pi_0 \longrightarrow V^{\pi_0} \longrightarrow \pi_1 \longrightarrow V^{\pi_1} \longrightarrow \quad \ldots \quad \longrightarrow \pi^* \longrightarrow V^{\pi^*}$$

Policy Iteration

- Policy iteration: Start with arbitrary π_0 and refine it.

$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \quad \ldots \quad \rightarrow \pi^*$$

- Involves repeating two steps:
  - Policy Evaluation: Compute V^π (similar to VI)
  - Policy Refinement: Greedily change actions as per V^π

$$\pi_0 \longrightarrow V^{\pi_0} \longrightarrow \pi_1 \longrightarrow V^{\pi_1} \longrightarrow \quad \ldots \quad \longrightarrow \pi^* \longrightarrow V^{\pi^*}$$

- Why do policy iteration?
  - π_i often converges to π^* much sooner than V^{π_i}

Summary

- Value Iteration
  - Bellman update to state value estimates
- Q-Value Iteration
  - Bellman update to (state, action) value estimates
- Policy Iteration
  - Policy evaluation + refinement

Learning Based Methods

Learning Based Methods

- Typically, we don't know the environment
  - $\mathbb{T}(s, a, s')$ unknown, how actions affect the environment.
  - $\mathcal{R}(s, a, s')$ unknown, what/when are the good actions?

Learning Based Methods

- Typically, we don't know the environment
  - $\mathbb{T}(s, a, s')$ 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.

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

  - Approximate unknown quantities from data.

$\boxed{\text{Reinforcement Learning}}$

Learning Based Methods

- 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

Reinforcement Learning

Sample-Based Policy Evaluation?

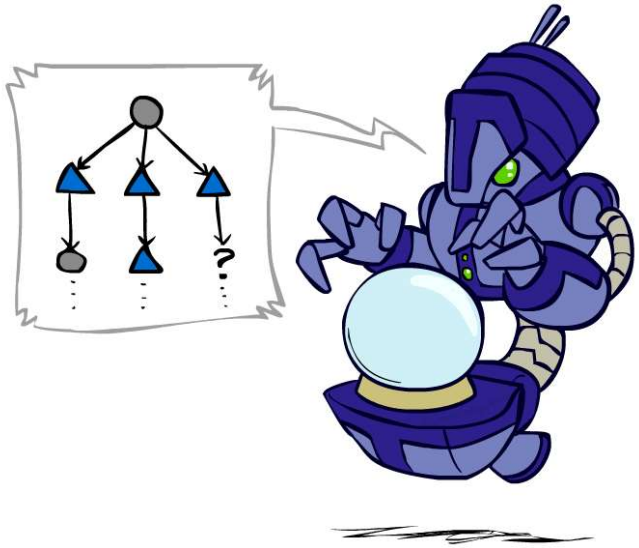
- We want to improve our estimate of V by computing these averages:

$$V^{\pi}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{k}(s')]$$

- Idea: Take samples of outcomes s' (by doing the action!) and average

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V^{\pi}_{k}(s'_1)$$
$$sample_2 = R(s, \pi(s), s'_2) + \gamma V^{\pi}_{k}(s'_2)$$
$$\ldots$$
$$sample_n = R(s, \pi(s), s'_n) + \gamma V^{\pi}_{k}(s'_n)$$

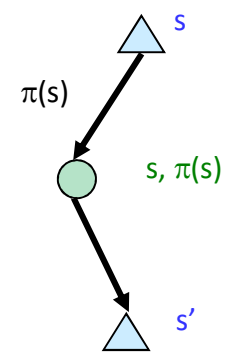
$$V^{\pi}_{k+1}(s) \leftarrow \frac{1}{n} \sum_{i} sample_i$$



What's the difficulty of this algorithm?

Temporal Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of V(s): $$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

Update to V(s): $$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$$

Same update: $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

Exponential Moving Average

- Exponential moving average
  - The running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
  - Makes recent samples more important:

$$\bar{x}_n = \frac{x_n + (1-\alpha) \cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \ldots}{1 + (1-\alpha) + (1-\alpha)^2 + \ldots}$$

  - Forgets about the past
- Decreasing learning rate (alpha) can give converging averages

Why do we want to forget about the past?

(distant past values were wrong anyway)

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

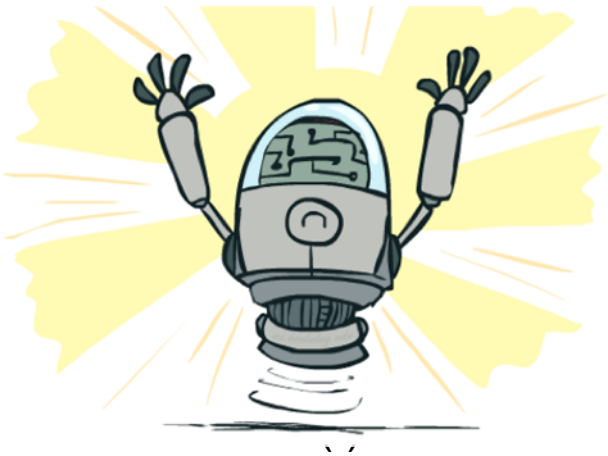
$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

  - 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



(Deep) Learning Based Methods

(Deep) Learning Based Methods

- In addition to not knowing the environment, sometimes the state space is too large.

(Deep) Learning Based Methods

- In addition to not knowing the environment, sometimes the state space is too large.

- A value iteration updates takes $O(|\mathcal{S}|^2|\mathcal{A}|)$
  - Not scalable to high dimensional states e.g.: RGB images.

(Deep) Learning Based Methods

- In addition to not knowing the environment, sometimes the state space is too large.

- A value iteration updates takes $O(|\mathcal{S}|^2|\mathcal{A}|)$
  - Not scalable to high dimensional states e.g.: RGB images.

- Solution: Deep Learning!
  - Use deep neural networks to learn low-dimensional representations.

Deep Reinforcement Learning

Reinforcement Learning

Reinforcement Learning

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

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^*

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!

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!

Value-based Reinforcement Learning

Deep Q-Learning

Deep Q-Learning

- Q-Learning with linear function approximators

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

  – Has some theoretical guarantees

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

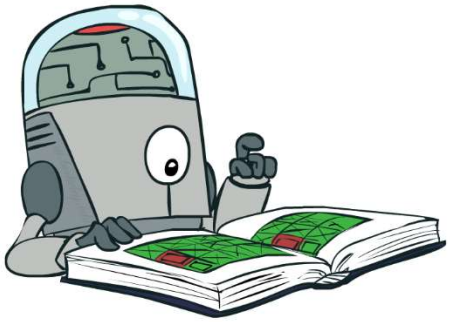
$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

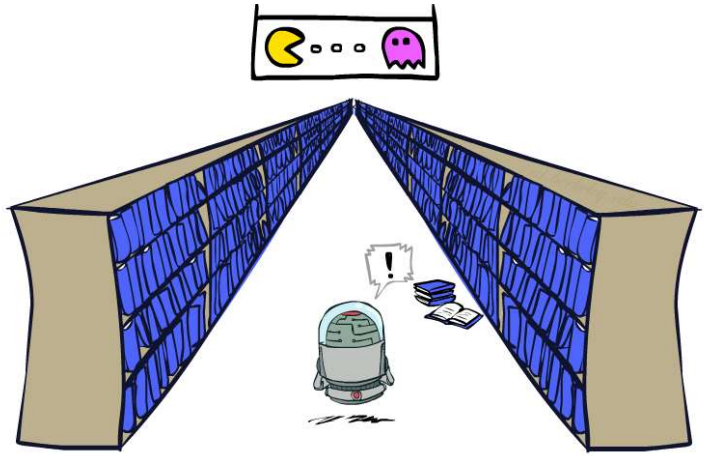
  - But we want to average over results from (s,a)
  - So keep a running average

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

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!

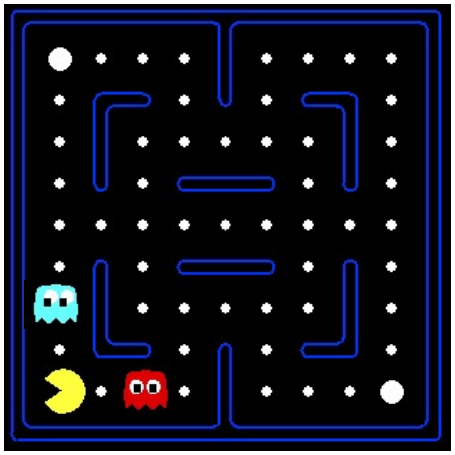




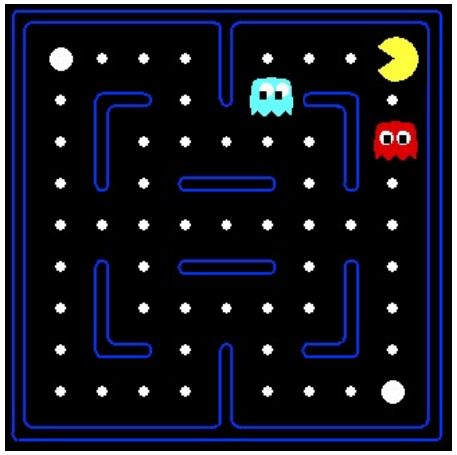
[demo – RL pacman]

Example: Pacman

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



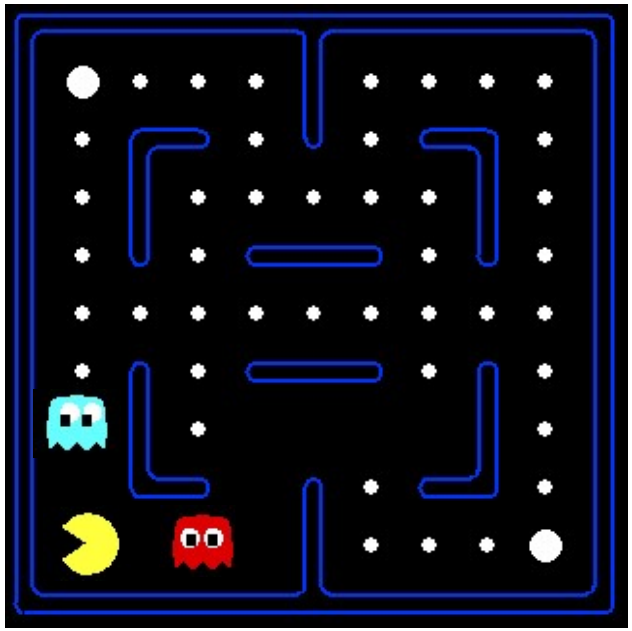
In naïve q-learning, we know nothing about this state:



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 / (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 food)



Linear Value Functions

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

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

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Advantage: our experience is summed up in a few powerful numbers

- Disadvantage: states may share features but actually be very different in value!
- Want to optimize weights. What should our loss be?

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s',a')\right] - Q(s,a)$$

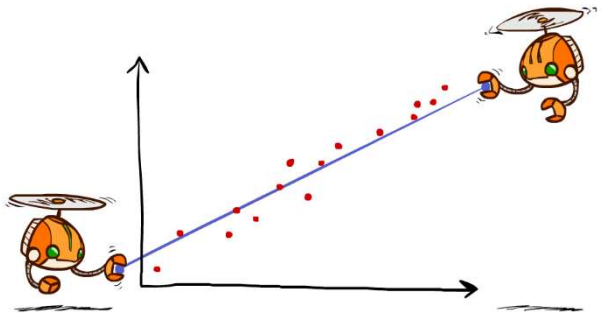
Minimizing Error*

Imagine we had only one point x, with features f(x), target value y, and weights w:

$$\text{error}(w) = \frac{1}{2}\left(y - \sum_k w_k f_k(x)\right)^2$$

$$\frac{\partial\ \text{error}(w)}{\partial w_m} = -\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

$$w_m \leftarrow w_m + \alpha\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$



Approximate q update explained:

$$w_m \leftarrow w_m + \alpha\left[Q(s,a) - r + \gamma \max_a Q(s',a')\right] f_m(s,a)$$

"prediction" "target"

Deep Q-Learning

- 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

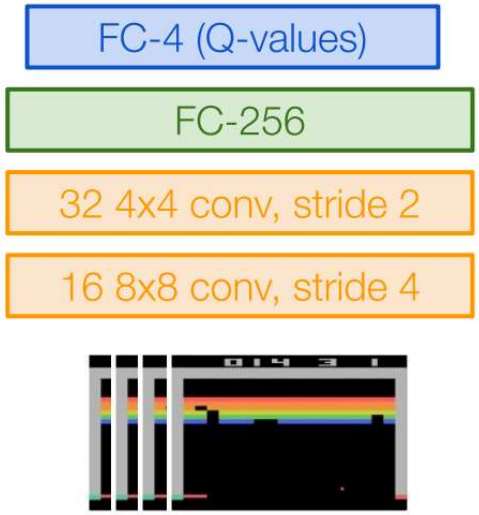


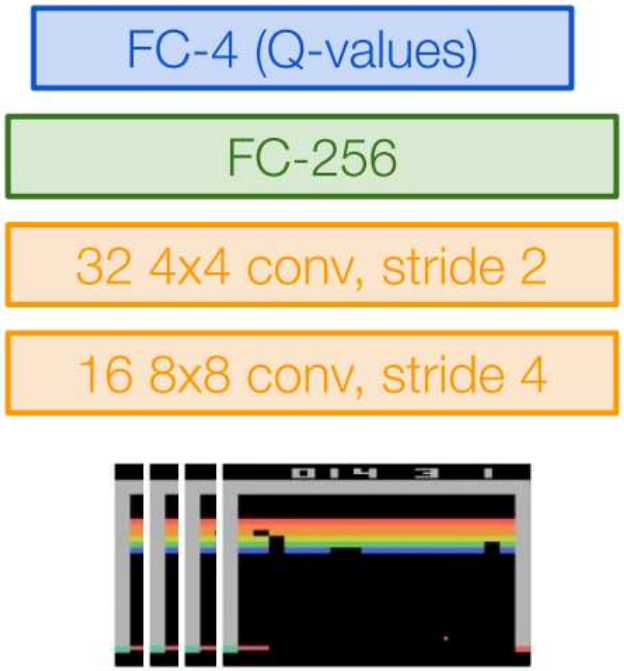
Image Credits: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Playing Atari Games

- Q-Network architecture

- State:
  - Stack of 4 image frames, grayscale conversion, down-sampling and cropping to (84 x 84 x 4)

- Last FC layer has #(actions) dimensions (predicts Q-values)



Deep Q-Learning

Deep Q-Learning

- Assume we have collected a dataset

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

- We want a Q-function that satisfies:

Q-Value Bellman Optimality

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

- Loss for a single data point:

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

Deep Q-Learning

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

- Forward pass: State → Q-Network → Q-Values per action

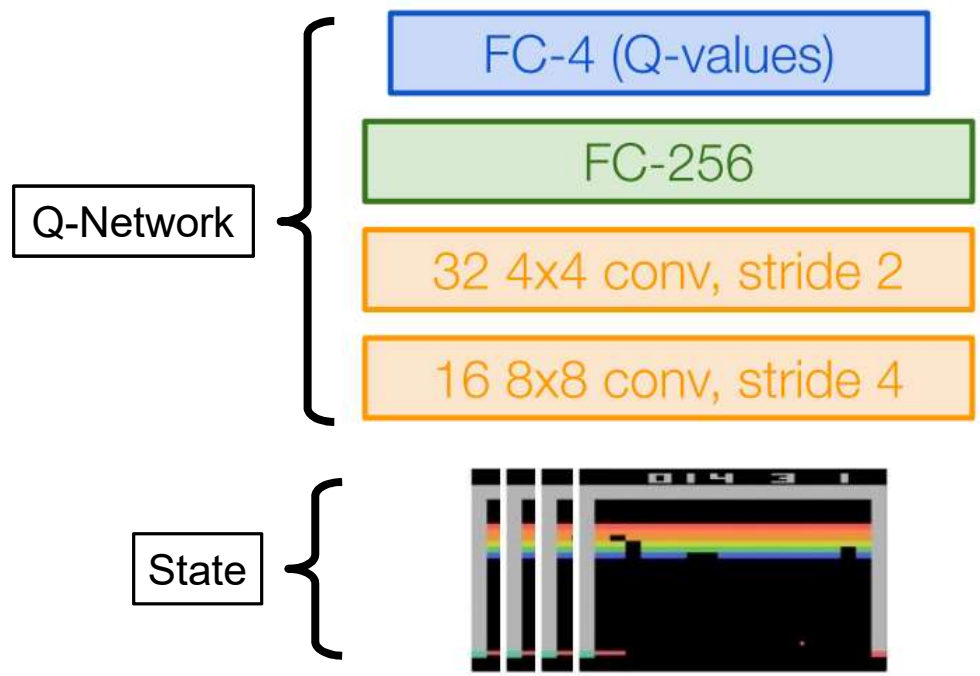
  $B \times D$ $\qquad$ $B \times n_{actions}$

Deep Q-Learning

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

- Forward pass: State → Q-Network → Q-Values per action

  $B \times D$ $\qquad$ $B \times n_{actions}$



Deep Q-Learning

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

- Forward pass: State → Q-Network → Q-Values per action

  $B \times D$ $\qquad$ $B \times n_{actions}$

- Compute loss: $$\Big(\underbrace{Q_{new}(s,a)}_{\theta_{new}} - (r + \gamma \max_a \underbrace{Q_{old}(s',a)}_{\theta_{old}})\Big)^2$$

Deep Q-Learning

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

- Forward pass: State → Q-Network → Q-Values per action

  $B \times D$ $\qquad$ $B \times n_{actions}$

- Compute loss: $$\Big(\underbrace{Q_{new}(s,a)}_{\theta_{new}} - (r + \gamma \max_a \underbrace{Q_{old}(s',a)}_{\theta_{old}})\Big)^2$$

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

Deep Q-Learning

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

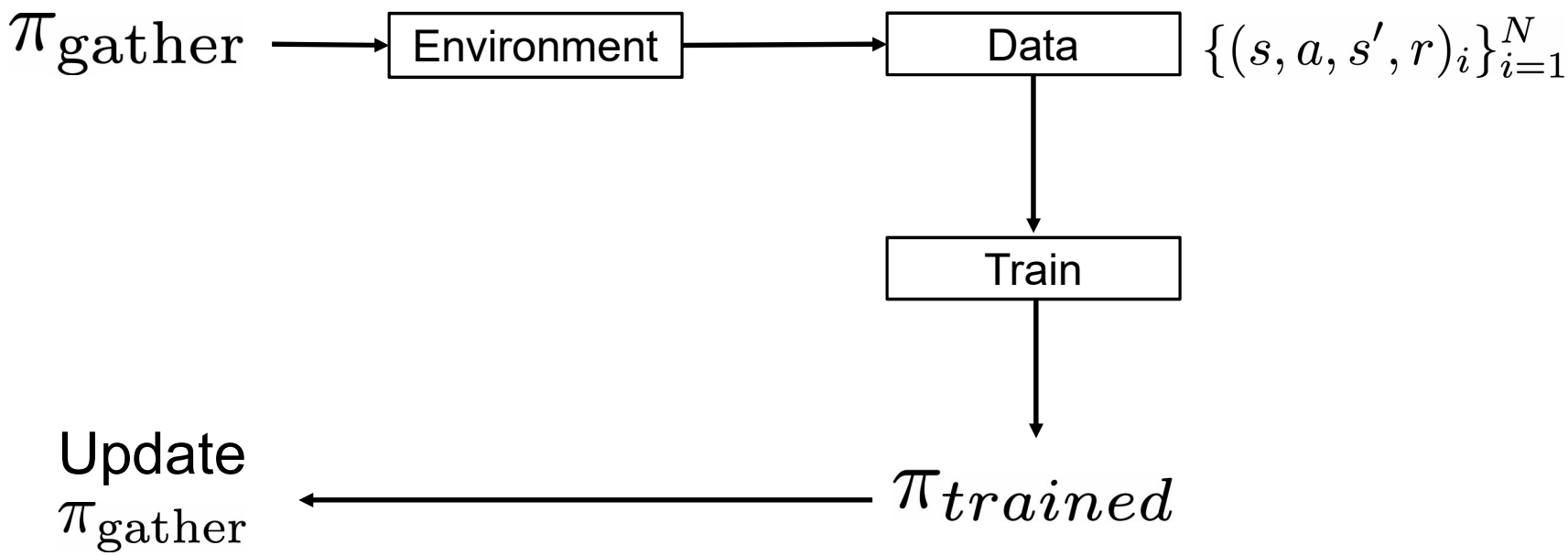
- In practice, for stability:
  - Freeze Q_{old} and update Q_{new} parameters
  - Set $Q_{old} \leftarrow Q_{new}$ at regular intervals

How to gather experience?

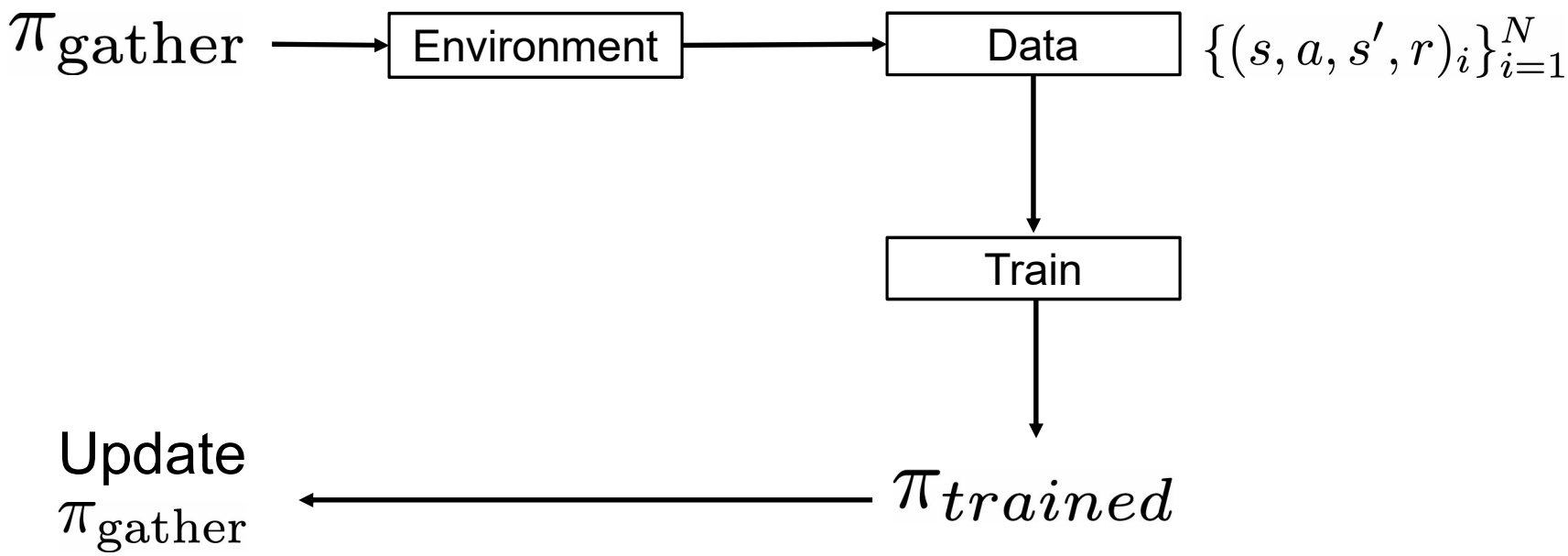
$$\{(s, a, s', r)_i\}_{i=1}^{N}$$

This is why RL is hard

How To Gather Experience?



How To Gather Experience?



Challenge 1: Exploration vs Exploitation

Challenge 2: Non iid, highly correlated data

Exploration Problem

- What should π_{gather} be?
  - Greedy? -> Local minimas, no exploration

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

Exploration Problem

- What should π_{gather} be?
  - Greedy? -> Local minimas, no exploration

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

- An exploration strategy:
  - ϵ-greedy

$$a_t = \begin{cases} \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

Correlated Data Problem

- 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 left, training samples will be dominated by samples from left-hand size.

Experience Replay

- Address this problem using experience replay
  - A replay buffer stores transitions (s, a, s', r)

Experience Replay

- Address this problem 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

Experience Replay

- Address this problem 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

Q-Learning Algorithm

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

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

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

  end for

end for

Experience Replay

Epsilon-greedy

Q Update

Case study: Playing Atari Games

- Objective: Complete the game with the highest score

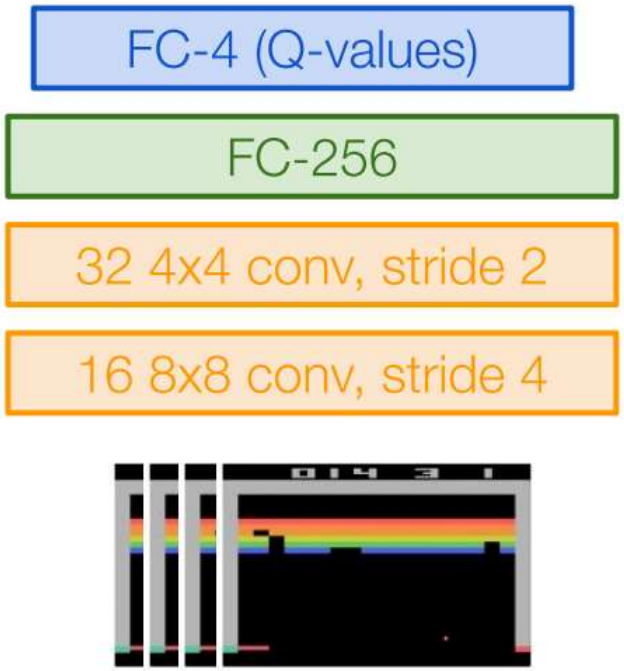
- State: Raw pixel inputs from the game state
- Action: Game controls e.g.: Left, Right, Up, Down
- Reward: Score increase/decrease at each time step

Playing Atari Games

- Q-Network architecture

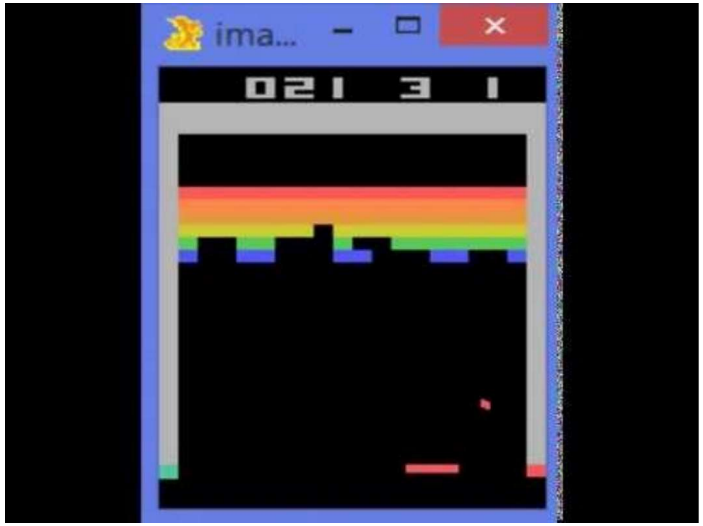
- State:
  - Stack of 4 image frames, grayscale conversion, down-sampling and cropping to (84 x 84 x 4)

- Last FC layer has #(actions) dimensions (predicts Q-values)



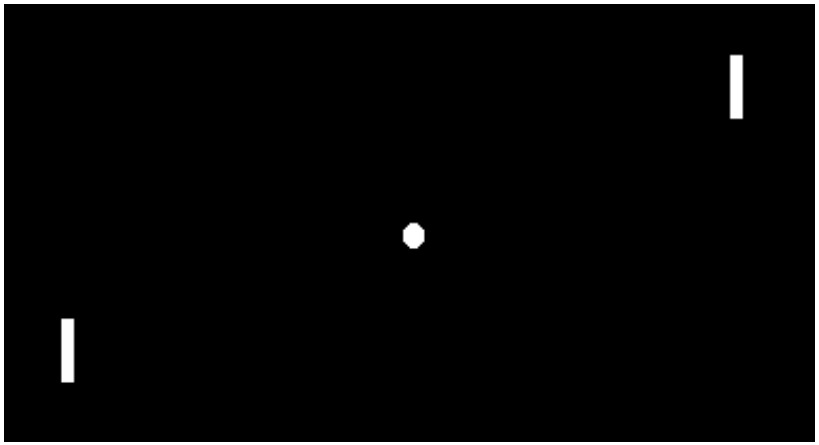
Atari Games

Breakout



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

Pong



Summary

So far, we looked at

- Dynamic Programming
  - Q-Value Iteration
  - Policy Iteration
- Reinforcement Learning (RL)
  - The challenges of (deep) learning based methods
  - Value-based RL algorithms
    - Deep Q-Learning

Next :

  - Policy-based RL algorithms