CS 457/557: Machine Learning

Lecture 08-*:
Markov Decision Processes

University of Wisconsin
LA CROSSE

Prof. Jason Sauppe
jsauppe@uwlax.edu
https://cs.uwlax.edu/~jsauppe/

Fall 2024

Lecture 08-1a:
Reasoning in a Complex Environment

Limitations of Supervised Learning

Consider the task of teaching a machine to play chess.

- We could try to assemble a **training set** of

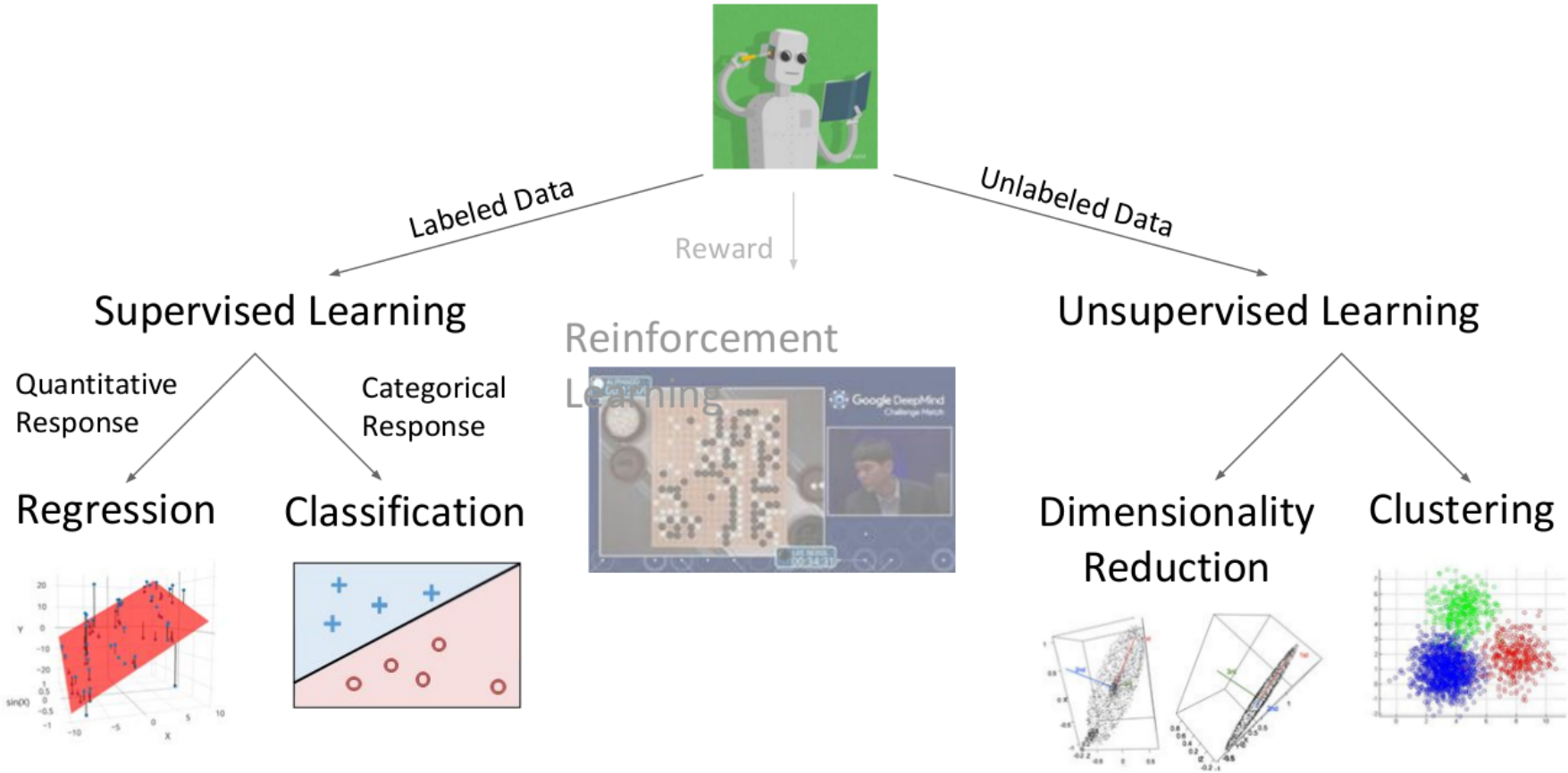
$$((\text{board configuration}, \text{move}), \text{value})$$

  records and **learn a hypothesis function** that can estimate the value of any move from any board configuration
- But **lots** of board positions are possible (about 10^{40}), **far more** than we could possibly include in a training set
- Sooner rather than later, the machine will encounter a configuration that it has no ideas about, and start doing **very poorly**

"The AI revolution will not be supervised."

— Yann LeCun and Alyosha Efros

Machine Learning Taxonomy



(Image source: DS 100 lecture notes)

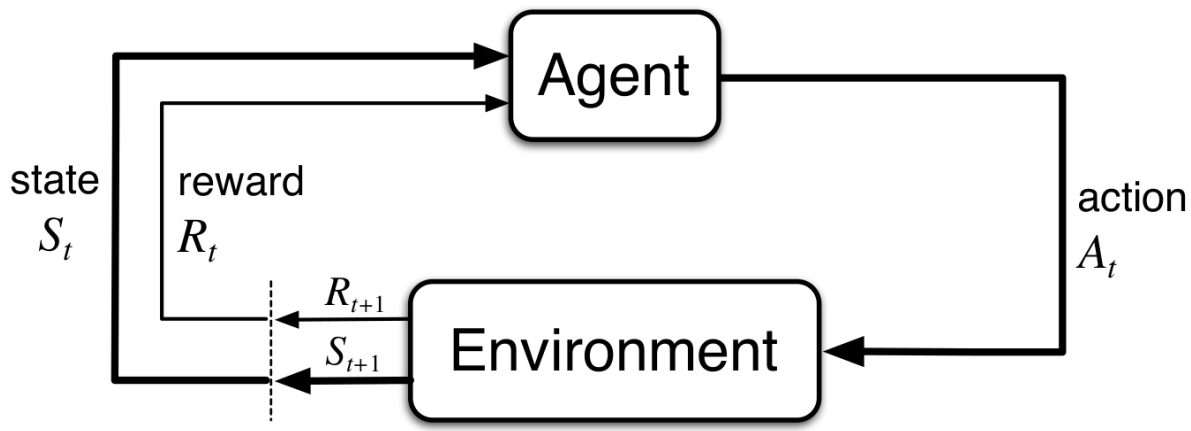
Alternative Learning Methods

Definition

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.

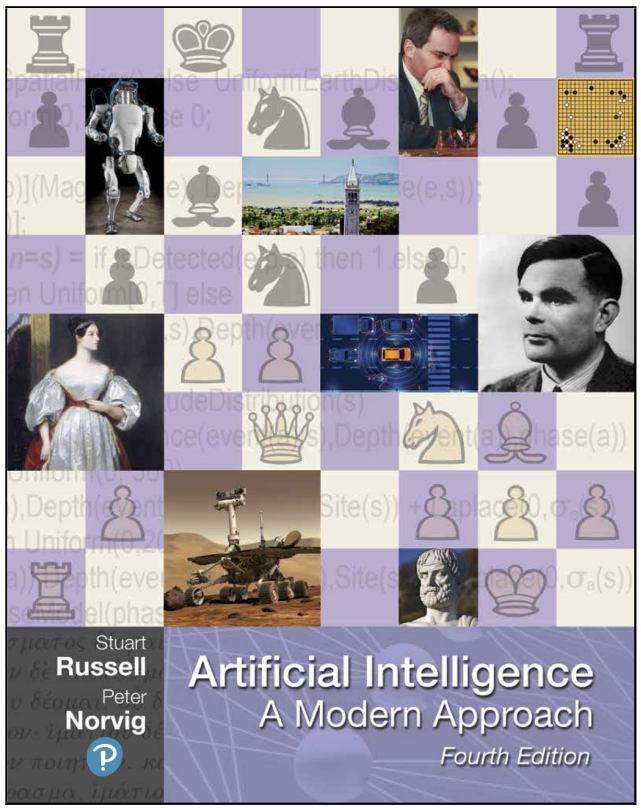
(*Reinforcement Learning* by Sutton and Barto)

Useful for learning how to play games (e.g., Tic-tac-toe, Go)



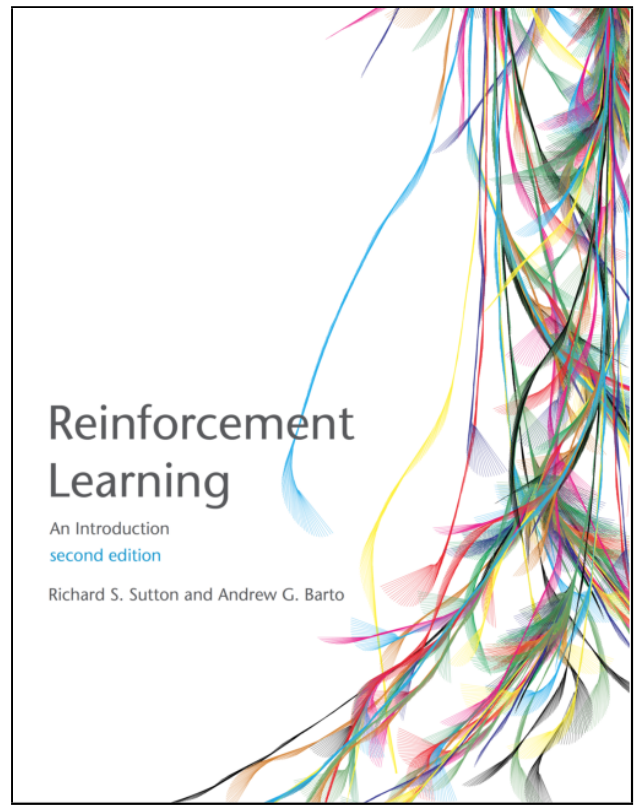
Textbook Reminder

Primary Textbook:



(Assigned readings are taken from here)

Secondary Textbook:



Available for free online, and great for extra reading and context!

Reinforcement Learning

Instead of providing a machine with **labeled training data**, **reinforcement learning** lets the machine **explore** its **environment** on its own by trying various **actions**.

Periodically during exploration, the machine will receive a **reward** (**reinforcement**). The **goal** is for the machine to figure out which actions should be used to **maximize** its total reward over time.

Human example: You play a new game without knowing the rules. After a little while you are told that you lost. Then you restart and try again. And again. And again...

- Providing a **reward signal** is usually much **easier** than providing labeled examples of how to behave
- Also, we don't need to know the "correct" answer for every example (as would be necessary in supervised learning)

Components of Reinforcement Learning

Reinforcement learning considers a machine (agent) operating in a **complex and uncertain** environment **over time** with periodic **rewards**.

- **Uncertainty**: We need **probability theory**
- **Rewards**: We need **value-based planning**, as in decision theory
- **Time**: We need a **temporal model** of how the environment changes

To get started, we will use a model known as a **Markov decision process** (MDP). In an MDP:

- The current configuration of the environment is called a **state**
- The **state** can change based on the **actions** of the agent
- These state changes may be **stochastic** (i.e., random)
- **Rewards** (or punishments) may be received as part of a state change

Lecture 08-1b:
Markov Decision Processes

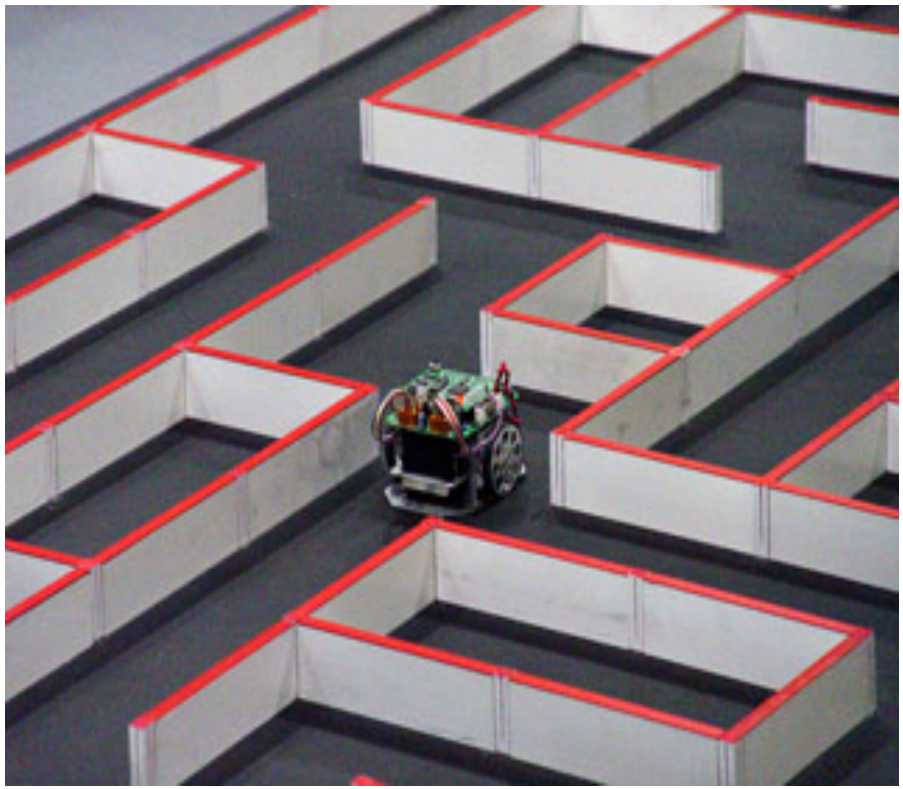
Markov Decision Processes

Definition

A **Markov decision process** (MDP) $M = (S, A, P, R, T)$ is a 5-tuple containing the following components:

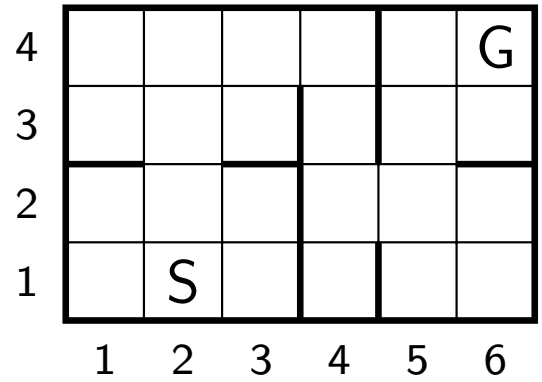
- S is a set of **states** in the world;
- A is a set of **actions** that the agent can take;
- P is a **state-transition function**: $P(s, a, s')$ denotes the probability of ending up in state s' if action a is taken in state s;
- R is a **reward function**: $R(s, a, s')$ is the one-step reward obtained if the agent enters state s' after taking action a in state s;
- T is a **time horizon** (i.e., a limit on the number of steps that can be taken).

An Example: Maze Navigation



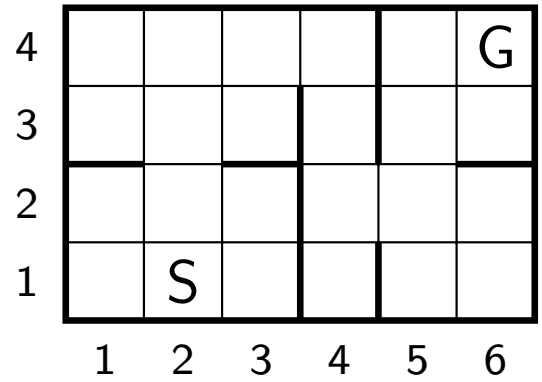
- Suppose we have a robot in a maze, looking for exit
- The robot can see where it is currently, and where surrounding walls are, but doesn't know anything else
- We would like it to be able to learn the shortest route out of the maze, no matter where it starts
- How can we formulate this problem as an MDP?

Modeling the Maze as an MDP



- States: $S = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid 1 \leq x \leq 6 \wedge 1 \leq y \leq 4\}$
- Actions: $A = \{\text{Left}, \text{Right}, \text{Up}, \text{Down}\}$
- State-transition function:
  - Can include **deterministic** movement, e.g.,
    - $P((2, 1), \text{Up}, (2, 2)) = 1$ and
    - $P((2, 1), \text{Up}, (x, y)) = 0$ for all other $(x, y) \in S$
  - Can be used to enforce walls, e.g., $P((2, 1), \text{Down}, (2, 1)) = 1$

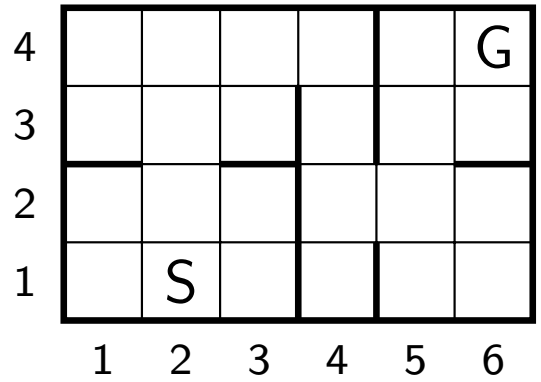
Adding Uncertainty



- The state-transition function can also include **non-deterministic movement**, e.g.:
  - $P((2,1), \text{Up}, (2,2)) = 0.8$
  - $P((2,1), \text{Up}, (1,1)) = 0.1$
  - $P((2,1), \text{Up}, (3,1)) = 0.1$

  (could represent an agent that staggers occasionally)

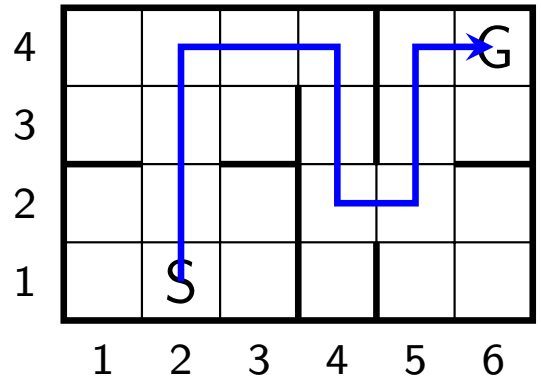
Rewards



- Reward function:
  - Can be used to **encourage** the agent to find the goal state G, e.g.:
    - $R((5,4), \text{Right}, (6,4)) = +100$
    - $R((6,3), \text{Up}, (6,4)) = +100$
  - Can also **discourage** spending too much time in the maze, e.g.,:
    - $R((x,y), a, (x',y')) = -1$ for all other states (x,y) and (x',y') in S and actions a in A

Lecture 08-1c:
Policies for Markov Decision Processes

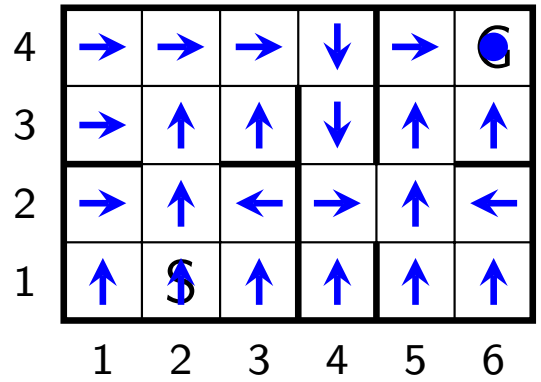
Solving an MDP: The Wrong Way



Providing a **single path** through the maze doesn't constitute a solution.

- What happens if the agent deviates from the path by accident?
- How would it know what to do next?

Solving an MDP: The Correct Way



Definition

For an MDP $M = (S, A, P, R, T)$, a **policy** π is a function from S to A (i.e., $\pi : S \to A$) that specifies the action to take in each state.

- Having a policy takes care of all possible contingencies
- The goal state is **terminal**, so no further action gets taken

Stationary and Non-stationary Policies

Definition

A **stationary policy** $\pi : S \to A$ is a policy in which the action to take depends only on the current state.

However, if an MDP's time horizon T is **finite**, then the action to take in a state s **might depend on** how much time remains!

- Example: If the maze agent has the option of standing still with a reward of 0 (e.g., conserving battery life), then it might be better to stand still instead of moving once the agent realizes it cannot reach the goal in time

Definition

A **non-stationary policy** $\pi : S \times \{0, 1, \ldots, T-1\} \to A$ is a policy in which the action to take may depend on the current state and time.

This makes it harder to formulate a policy, let alone find a good one!

Infinite Time Horizon and Stationary Policies

For an MDP M with an **infinite** time horizon $T = \infty$, we only need to consider **stationary policies**:

- The current action should **only depend** on the current state
- How long we took to get to the current state is **irrelevant**
- How long we have to go is **irrelevant** (because we always have an infinite amount of time left!)

So to simplify our discussion going forward, we'll focus on MDPs with an **infinite** time horizon.

Lecture 08-1d:
Measuring Policy Quality

Towards Finding a Good Policy

How can an agent **find** a **good policy**?

If the agent **knows the entire problem** in advance, then it can **plan**:

- For an MDP $M = (S, A, P, R, T)$, the agent has everything it needs to evaluate the impact and reward obtained from any action in any state, even if these impacts and rewards are **stochastic** (random)
- Thus, the agent can "solve" the MDP **ahead of time** to determine what it should do, without having to do any trial-and-error

We'll see how the agent can do this in order to show some **useful ideas**!

If the agent **doesn't know everything** in advance, then it must **learn**:

- The agent is forced to do trial-and-error to explore its environment and determine (estimate) the impact and reward of each action in each state
- This is the subject of **reinforcement learning** proper, which we'll get to a bit later on!

Policy Quality and the Environment History

To **find** a **good policy**, we first need a notion of the **quality** of a policy.

Suppose we start in initial state s_0 and follow a policy π. What happens?

We observe some **sequence** of states and actions:

$$[s_0, a_0, s_1, a_1, s_2, a_2, s_3, \ldots, s_{n-1}, a_{n-1}, s_n]$$

where

- $a_t = \pi(s_t)$ for all $t \in \{0, 1, 2, \ldots, n-1\}$,
- s_n is a terminal state (assuming we reach one).

This sequence is called an **environment history**.

How **good** was the policy π? We can measure it via the sum of **one-step** rewards obtained during each transition!

$$\text{Quality}(\pi) = R_{\text{total}} = \sum_{t=0}^{n-1} R(s_t, a_t, s_{t+1})$$

Computing Policy Quality

The **quality** of a policy π can be measured as the **total reward** obtained by following that policy:

$$R_{\text{total}} = \sum_{t=0}^{n-1} R(s_t, a_t, s_{t+1})$$

Issue: The time horizon T of the MDP may impact our policy.

- If T is **finite**, then:
  - We can always calculate R_{total} because $n \leq T$,
  - **but** the optimal policy may be **non-stationary**
- If T is **infinite**, then:
  - The optimal policy is **stationary** (which is **good**),
  - **but** R_{total} may be an **infinite sum**! (e.g., the agent might never reach a terminal state)

Lecture 08-2a:
Discounted Rewards

Rewards with an Infinite Time Horizon

For an MDP $M = (S, A, P, R, T)$ with $T = \infty$, we saw that an environment history $[s_0, a_0, s_1, a_1, s_2, a_2, s_3, \ldots]$ of infinite length can have sum of rewards

$$R_{\text{total}} = \sum_{t=0}^{\infty} R(s_t, a_t, s_{t+1})$$

that is **also infinite**!

This makes policy comparison **problematic**:

- Suppose $\pi^{(1)}$, $\pi^{(2)}$, and $\pi^{(3)}$ are three policies giving total rewards $+\infty$, $+\infty$, and $-\infty$, respectively
- Obviously we prefer $\pi^{(1)}$ and $\pi^{(2)}$ over $\pi^{(3)}$
- But **how should we compare** $\pi^{(1)}$ and $\pi^{(2)}$?

Discounted Rewards

To solve the problem of **infinite total reward** in an MDP, we use a **discount rate** $\gamma \in [0, 1]$. Then the total reward is

$$\begin{aligned} R_{\text{total}} &= R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots \\ &= \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}). \end{aligned}$$

- If time horizon T is finite, then $\gamma = 1$ gives the standard sum of rewards.
- If time horizon T is infinite, then $\gamma < 1$ ensures that

$$R_{\text{total}} = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = \frac{R_{\max}}{1-\gamma},$$

  where $R_{\max}$ is the maximum value of any one-step reward.
- If $\gamma = 0$, then we have a **greedy algorithm**!

Lecture 08-2b:
Incorporating Uncertainty

Dealing with Uncertainty

With **discounted rewards**, we can compute the **total reward** obtained for any **particular** environment history $[s_0, a_0, s_1, a_1, s_2, \ldots]$.

Issue: The environment history that we get is **random**!

Solution: Introduce **random variables**!

- Let S_t be the **random variable** representing the state that the agent is in at time t
- Let A_t be the **random variable** representing the action that the agent takes at time t

Then the **total reward** we get is

$$R_{\text{total}} = \sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1}).$$

(Note: R_{total} is **also** a **random variable**!)

Probability Background: Random Variables

Definition (informal)

A **random variable** (RV) is a variable whose value depends on some process whose outcome is unknown in advance.

By convention, random variables are denoted with capital letters (e.g., X, Y, Z).

Examples:

- Let RV X represent the value obtained from flipping a coin, H or T
- Let RV X represent the value obtained from rolling a six-sided die

In both of these examples, we **don't know** the **value** of X **before** we do the experiment (i.e., flip the coin or roll the die).

However, we **can** talk about **possible outcomes** for X and the **likelihoods** of those outcomes!

Probability Background: Range and Probabilities for a Random Variable

Definition

The **range** of a (discrete) random variable is the set of values that the random variable can have (after the associated experiment is performed). Each value in the range has an associated likelihood, or **probability**, of occurring.

Example: Let X represent the value obtained from rolling a six-sided die. Then range$(X) = \{1, 2, 3, 4, 5, 6\}$.

Additionally, if the die is **fair**, then each value is **equally likely**:

$$P(X = 1) = \tfrac{1}{6},\ P(X = 2) = \tfrac{1}{6},\ \ldots,\ P(X = 6) = \tfrac{1}{6},$$

where $P(X = x)$ denotes the **probability** of X having value x.

Probability Background: Expected Value of a Random Variable

Definition

For a (discrete) random variable X, the **expected value** of X, denoted $E[X]$, is

$$E[X] = \sum_{x \in \text{range}(X)} x \cdot P(X = x).$$

So $E[X]$ is a **weighted average** of the possible values for X, where each value is weighted by its likelihood.

For the die-rolling example from the previous slide, we have:

$$E[X] = \sum_{i=1}^{6} i \cdot P(X = i) = \sum_{i=1}^{6} i \cdot \tfrac{1}{6} = \frac{21}{6} = 3.5.$$

(So the expected value of a random variable is just a **number**, and the number might not even be a value that the random variable itself could have.)

Probability Background: Conditional Probability

Definition

For (discrete) random variables X and Y, the **conditional probability** of X having value x **given that** Y has value y is

$$P(X = x \mid Y = y) = \frac{P(X = x \wedge Y = y)}{P(Y = y)},$$

provided that $P(Y = y) \neq 0$.

Conditional probability allows us to **revise** an **unconditional probability** based on information about other related outcomes. Example:

- Let X count the number of heads in two flips of a fair coin. Then: $P(X = 0) = 0.25$, $P(X = 1) = 0.5$, $P(X = 2) = 0.25$.
- Let Y be an RV that is 1 if the first flip is heads, 0 otherwise. Then:
  - $P(X = 0 \mid Y = 1) = 0.0$, $P(X = 1 \mid Y = 1) = 0.5$, $P(X = 2 \mid Y = 1) = 0.5$
  - $P(X = 0 \mid Y = 0) = 0.5$, $P(X = 1 \mid Y = 0) = 0.5$, $P(X = 2 \mid Y = 0) = 0.0$

Probability Background: Conditional Expectation

Definition

For (discrete) random variables X and Y, the **conditional expected value** of X **given that** Y has value y, denoted $E[X \mid Y = y]$, is

$$E[X \mid Y = y] = \sum_{x \in \text{range}(X)} x \cdot P(X = x \mid Y = y),$$

where $P(X = x \mid Y = y)$ is the **conditional probability** of X having value x **given that** Y has value y.

For the random variables X and Y used in the example on the previous slide, we have:

$$\begin{aligned} E[X \mid Y = 1] &= \sum_{x \in \text{range}(X)} x \cdot P(X = x \mid Y = 1) \\ &= 0 \cdot P(X = 0 \mid Y = 1) + 1 \cdot P(X = 1 \mid Y = 1) + 2 \cdot P(X = 2 \mid Y = 1) \\ &= 0 \cdot 0 + 1 \cdot 0.5 + 2 \cdot 0.5 = 1.5. \end{aligned}$$

Brief Aside: Law of Total Expectation

Definition

Given (discrete) random variables X and Y, the **law of total expectation** (iterated expectation) says $E[X] = E[E[X \mid Y]]$.

Proof for discrete random variables:

$$
\begin{aligned}
E[E[X \mid Y]] &= \sum_{y \in \text{range}(Y)} y \cdot E[X \mid Y = y] \\
&= \sum_{y \in \text{range}(Y)} y \cdot \left(\sum_{x \in \text{range}(X)} x \cdot P(X = x \mid Y = y) \right) \\
&= \sum_{x \in \text{range}(X)} x \cdot \sum_{y \in \text{range}(Y)} y \cdot P(X = x \mid Y = y) \\
&= \sum_{x \in \text{range}(X)} x \cdot P(X = x) \\
&= E[X].
\end{aligned}
$$

Lecture 08-3a:
The State-Value Function

Quick Recap: Discounted Total Reward of a Policy

For an MDP M with an infinite time horizon and a policy π, the discounted total reward obtained by following π is

$$R_{\text{total}} = \sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1})$$

where S_t and A_t are **random variables** representing the agent's state and the action it takes at time t (with $A_t = \pi(S_t)$).

- Caveat: The initial state S_0 is generally known, **not random**

Because R_{total} is a sum of functions of random variables, R_{total} is also a **random variable**!

We'd like to compute $E[R_{\text{total}} \mid S_0 = s]$, the **expected value** of R_{total} given that the agent starts in state s. **Intuitively**, this is the value of R_{total} averaged across all possible environment histories starting from s.

The Value of a State

Definition

For an MDP M and policy π, the **state-value function** $V^\pi : S \to \mathbb{R}$ computes the expected total reward obtained by following policy π starting from an initial state s. I.e.,

$$V^\pi(s) = E\left[R_{\text{total}} \mid S_0 = s\right] = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s\right],$$

where the expectation is taken over the probability distribution of all environment histories generated by π starting from s.

Intuitively, this is the average total reward across **all possible histories**.

- $V^\pi(s)$ tells us how **desirable** it is to be in state s when following policy π
- We can use this information to **evaluate** (and also **improve**!) a policy

Rewriting the State-Value Function

From the definition of the **state-value function**, we have:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s\right].$$

We can use the **law of total expectation** to rewrite the above expectation by **conditioning** on **another random variable**, say S_1:

$$\begin{aligned}
&E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s\right] \\
&\quad = E\left\{E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1\right]\right\} \quad [\textit{outer expectation over } S_1] \\
&\quad = \sum_{s' \in S} P(S_1 = s' \mid S_0 = s) \cdot E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right].
\end{aligned}$$

Rewriting the State-Value Function (continued)

Putting this together, we have:

$$V^{\pi}(s) = \sum_{s' \in S} P(S_1 = s' \mid S_0 = s) \cdot E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right]$$
$$= \sum_{s' \in S} P(s, \pi(s), s') \cdot E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right].$$

Note that there is some slight **notational overlap** here:

- In $P(S_1 = s' \mid S_0 = s)$, the P denotes a (conditional) **probability**
- In $P(s, \pi(s), s')$, the P denotes the **state-transition function** from the MDP

Rewriting the State-Value Function (continued)

In the expected value portion, we can rewrite the summation inside by pulling out the first term:

$$
\begin{aligned}
& E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right] \\
& \quad = E\left[R(S_0, \pi(S_0), S_1) + \sum_{t=1}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right] \\
& \quad = E\left[R(s, \pi(s), s') + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right] \\
& \quad = R(s, \pi(s), s') + \gamma E\left[\sum_{t=1}^{\infty} \gamma^{t-1} R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s'\right] \\
& \quad = R(s, \pi(s), s') + \gamma E\left[\sum_{t=0}^{\infty} \gamma^{t} R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s'\right] \quad [\textit{shift indices down by } 1] \\
& \quad = R(s, \pi(s), s') + \gamma V^{\pi}(s') \qquad [\textit{replace expected value by } V^{\pi}(s')]
\end{aligned}
$$

The Bellman Equation

Combining this all together gives the **Bellman equation**:

$$V^{\pi}(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right].$$

- $V^{\pi}(s)$: expected value of following policy π starting in state s
- $\sum_{s' \in S}$: Sum over all possible next states
- $P(s, \pi(s), s')$: transition probability of going from s to s' following action $\pi(s)$
- $R(s, \pi(s), s')$: One-step reward for going from s to s' following action $\pi(s)$
- $\gamma V^{\pi}(s')$: discounted expected value of continuing to follow policy π starting from state s'

This is a **recursive** definition for the **state-value function** V^{π} which says that the **value** of starting in state s can be calculated from the **values** of **all possible next state(s)** that can be reached by taking the action dictated by the policy π.

Lecture 08-3b:
Policy Evaluation

Solving the Bellman Equation

Suppose we have $S = \{s_1, s_2, \ldots, s_n\}$ for our MDP M. If we write the **Bellman equation** out for each state, we get:

$$V^\pi(s_1) = \sum_{i=1}^{n} P(s_1, \pi(s_1), s_i) \cdot [R(s_1, \pi(s_1), s_i) + \gamma V^\pi(s_i)]$$

$$V^\pi(s_2) = \sum_{i=1}^{n} P(s_2, \pi(s_2), s_i) \cdot [R(s_2, \pi(s_2), s_i) + \gamma V^\pi(s_i)]$$

$$\vdots$$

$$V^\pi(s_n) = \sum_{i=1}^{n} P(s_n, \pi(s_n), s_i) \cdot [R(s_n, \pi(s_n), s_i) + \gamma V^\pi(s_i)]$$

This is a system of $|S|$ equations in $|S|$ unknowns (the $V^\pi(s)$ values).

- We can solve this system of equations in $O(|S|^3)$ time.
- For large S, we'll use an **iterative approach** to find the $V^\pi(s)$ values, similar to how we used gradient descent to find the optimal weights in linear regression instead of solving the normal equations directly.

Evaluating a Policy Iteratively

Evaluating a Policy for an MDP

procedure PolicyEvaluation(MDP M; Policy π; Discount factor γ; Threshold Θ)

  for each state $s \in S$ **do**

    $v_s^{(0)} \leftarrow 0$ ▷ $v_s^{(t)}$ is estimate of $V^\pi(s)$ at iteration t

  $t \leftarrow 0$

  do

    $\Delta \leftarrow 0$

    for each state $s \in S$ **do**

      $v_s^{(t+1)} \leftarrow \sum_{s' \in S} P(s, \pi(s), s') \left[R(s, \pi(s), s') + \gamma v_s^{(t)} \right]$ ▷ Update v_s estimate

      $\Delta \leftarrow \max \left\{ \Delta, \left| v_s^{(t+1)} - v_s^{(t)} \right| \right\}$ ▷ Update max change in estimates

    $t \leftarrow t + 1$

  while $\Delta \geq \Theta$ ▷ If $\Theta = \frac{\epsilon(1-\gamma)}{\gamma}$, then error is at most ϵ

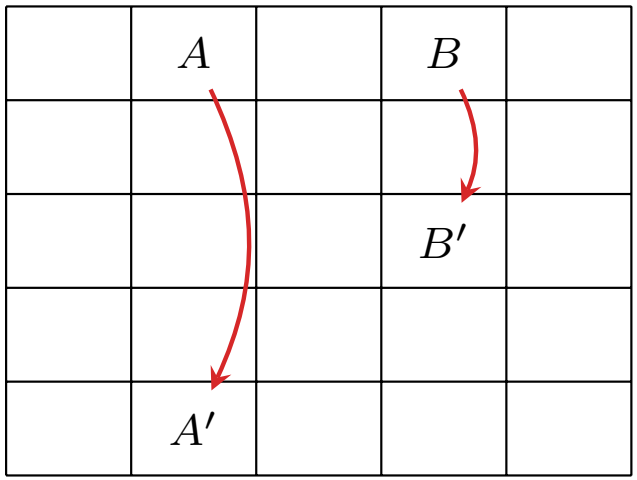
  for each state $s \in S$ **do**

    $V^\pi(s) \leftarrow v_s^{(t)}$ ▷ For ϵ sufficiently small, $v_s^{(t)} \approx V^\pi(s)$

  return V^π

The above is an **iterative process** for finding a function that satisfies the **Bellman equation**, which is exactly the **state-value function** V^π.

Using the State-Value Function



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Simple grid world: Agent moves around with the following notable behavior:

- Any action in states A or B causes move to state A' and B', respectively
- Rewards:
  - Hitting a wall incurs reward of -1 (and no movement)
  - Moving from A to A' has reward of $+10$; from B to B' is $+5$
  - All other movement has reward of 0
- **Policy**: Agent moves **randomly** (Up, Down, Left, Right)

State-value function V^π values shown on right with discount factor $\gamma = 0.9$.
Even with a **random policy**, we can see what states **look promising**!

Lecture 08-3c:
Towards an Optimal Policy

Policy Improvement: Finding Better Actions

To create a **new policy** π' from π, we pick, for each state, the action that yields the **most perceived benefit** using one-step look-ahead:

$$\pi'(s) = \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{\pi}(s') \right] \right\}.$$

The **perceived benefit** of an action a is a weighted sum over all possible next states s' of the **immediate one-step reward** $R(s, a, s')$ and the **discounted (long-term) value** of being in state s', $\gamma V^{\pi}(s')$.

- This is a **greedy** choice based on the **current** V^{π} state values
- If $\pi' \neq \pi$, then $V^{\pi'}(s) \neq V^{\pi}(s)$, and so we need to update the state-value function for new policy π'

Some Additional Notation: The Action-Value Function

The **Bellman equation** specifies the **state-value function** V^π as:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^\pi(s') \right].$$

Definition

For an MDP M and policy π, the **action-value function** $Q^\pi : S \times A \to \mathbb{R}$ is defined as follows for all $(s, a) \in S \times A$:

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^\pi(s') \right].$$

In words, $Q^\pi(s, a)$ is the **expected total reward** of

- **first** taking action a in state s, **regardless of policy**, and
- **then** following policy π in all subsequent steps.

Using the Action-Value Function

The **action-value function** Q^π provides **compact notation** for specifying the **best-looking** action for each state in a new policy π':

$$\begin{aligned}\pi'(s) &= \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s,a,s') \cdot \left[R(s,a,s') + \gamma V^\pi(s') \right] \right\} \\ &= \arg\max_{a \in A} \left\{ Q^\pi(s,a) \right\}.\end{aligned}$$

Additionally, from the **Bellman equation**, it can be shown that $V^\pi(s) = Q^\pi(s, \pi(s))$. This allows us to define the **action-value function** recursively as:

$$\begin{aligned}Q^\pi(s,a) &= \sum_{s' \in S} P(s,a,s') \cdot \left[R(s,a,s') + \gamma V^\pi(s') \right] \\ &= \sum_{s' \in S} P(s,a,s') \cdot \left[R(s,a,s') + \gamma Q^\pi(s', \pi(s')) \right].\end{aligned}$$

We'll come back to this idea later on!

Lecture 08-4a:
Policy Iteration

Policy Iteration Algorithm

Policy Iteration for Solving an MDP

procedure PolicyIteration(MDP M; Initial Policy π; Discount factor γ; Threshold Θ)
  $\pi^{(0)} \leftarrow \pi$
  $t \leftarrow 0$
  do
    $V^{(t)} \leftarrow \text{PolicyEvaluation}(M, \pi^{(t)}, \gamma, \Theta)$ ▷ $V^{(t)}$ is state-value func. for $\pi^{(t)}$
    $policyChanged \leftarrow false$
    for each state $s \in S$ **do**

$$\pi^{(t+1)}(s) \leftarrow \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{(t)}(s') \right] \right\}.$$

      if $\pi^{(t+1)}(s) \neq \pi^{(t)}(s)$ **then**
        $policyChanged \leftarrow true$
    $t \leftarrow t + 1$
  while $(policyChanged)$
  return $\pi^{(t)}$

This is an **iterative process** for finding an **optimal policy** π^* for the given MDP.

- The PolicyEvaluation procedure computes the state-value function V^π for the current policy as before

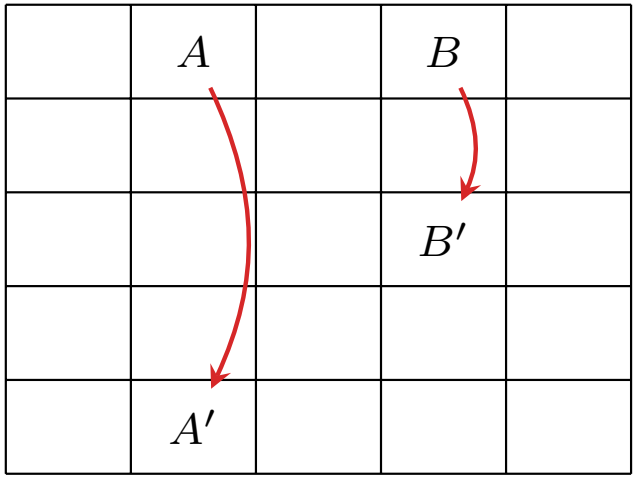
Policy Iteration Convergence

For any MDP M, the PolicyIteration procedure is **guaranteed to converge** to an **optimal** policy π^*:

- If the current policy is not optimal, then the policy update process will find a better policy
- For an MDP with finite state and action spaces, the number of possible policies is finite, $|A|^{|S|}$
- Eventually, we'll arrive at the best policy!

There are a few caveats for numerical issues and tolerances if we use the iterative procedure for estimating the state-value functions $V^{(t)}$, but these can be removed if we compute $V^{(t)}$ exactly by solving the system of Bellman equations.

Example with Optimal Policy

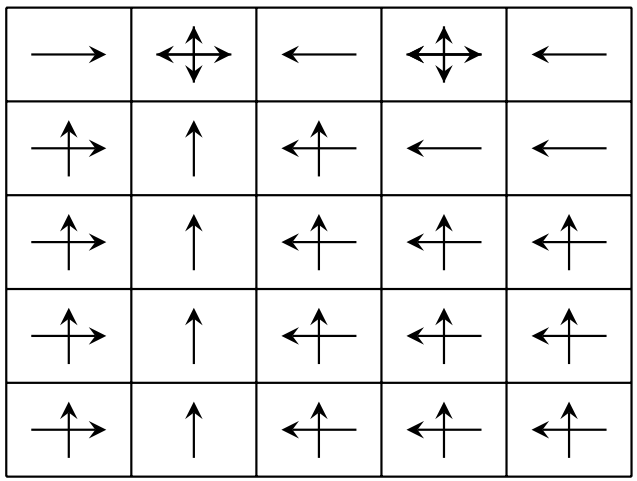


22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Simple grid world example with policy iteration.

The top right shows the state values for the optimal policy.

The bottom right shows the permissible actions in an optimal policy.



Lecture 08-4b: Bellman Optimality Equations and Value Iteration

Optimal State Values

Recall that $V^\pi(s)$ is the **expected total reward** that can be obtained starting from state s and **following policy** π, specified via the **Bellman equation** as:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^\pi(s') \right],$$

What we'd **ultimately** like to know is the **expected total reward** that can be obtained starting from state s, assuming that we **always do the right thing**!

Definition

The **optimal state-value function** $V^* : S \to \mathbb{R}$ computes the best expected total reward when starting in state s, across all possible policies:

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

The Bellman Optimality Equation

For the **optimal state-value function** V^*, we have:

$$
\begin{aligned}
V^*(s) &= \max_{\pi} V^{\pi}(s) \\
&= \max_{\pi} \left[\sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right] \right] \\
&= \max_{a \in A} \left[\sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma \max_{\pi} V^{\pi}(s') \right] \right] \\
&= \max_{a \in A} \left[\sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^*(s') \right] \right]
\end{aligned}
$$

This is the **Bellman optimality equation**.

- Also a **recursive** formula, just like the **Bellman equation**!
- The $\max_{a \in A}$ component complicates evaluation, though.

Comparing Bellman Equations

The **Bellman equation** for a fixed policy π is

$$V^{\pi}(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right].$$

We saw earlier how to use V^{π} for the current policy to find a **better** policy.

The **Bellman optimality equation** is

$$V^{*}(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{*}(s') \right] \right\}.$$

This **does not require** policies at all! Once we have V^{*}, we can find an **optimal policy** using

$$\pi^{*}(s) = \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{*}(s') \right] \right\}.$$

Solving the Bellman Optimality Equations: Value Iteration

Value Iteration for Solving an MDP

procedure ValueIteration(MDP M; Discount factor γ; Threshold Θ)

  for each state $s \in S$ **do**

    $v_s^{(0)} \leftarrow 0$ ▷ $v_s^{(t)}$ is estimate of $V^*(s)$ at iteration t

  $t \leftarrow 0$

  do

    $\Delta \leftarrow 0$

    for each state $s \in S$ **do**

      $v_s^{(t+1)} \leftarrow \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \left[R(s, a, s') + \gamma v_s^{(t)} \right] \right\}$ ▷ Update v_s estimate

      $\Delta \leftarrow \max \left\{ \Delta, \left| v_s^{(t+1)} - v_s^{(t)} \right| \right\}$ ▷ Update max change in estimates

    $t \leftarrow t + 1$

  while $(\Delta > \Theta)$ ▷ If $\Theta = \frac{\epsilon(1-\gamma)}{\gamma}$, then error is at most ϵ

  for each state $s \in S$ **do**

    $V^*(s) \leftarrow v_s^{(t)}$ ▷ For ϵ sufficiently small, $v_s^{(t)} \approx V^*(s)$

  return V^*

Two Methods for Solving MDPs: Policy Iteration and Value Iteration

Policy iteration:

1. Maintains current policy $\pi^{(t)}$
2. Solves the (regular) **Bellman equations** for policy $\pi^{(t)}$ to obtain state-value function $V^{\pi^{(t)}}$ (or $V^{(t)}$)
   - Solving is done iteratively (e.g., using PolicyEvaluation) or exactly via linear algebra
3. Generates a new policy $\pi^{(t+1)}$ by using the state values of the current policy
4. Repeats until no changes are made

Value iteration:

1. Does not maintain a current policy
2. Solves the **Bellman optimality equations** to obtain the optimal state-value function V^*
   - Solving is done iteratively (as on previous slide) or via linear programming
3. Uses V^* to generate an **optimal** policy by picking the best available action in each state

Each method has pros and cons; **policy iteration** tends to work well for small state spaces, but **value iteration** is better for larger ones.

Lecture 08-4c:
The Optimal Action-Value Function

The Optimal Action-Value Function

With the **Bellman equation**, we have the **state-value function** V^π and **action-value function** Q^π:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')] .$$

With the **Bellman optimality equations**, we have the **optimal state-value function** V^* and **optimal action-value function** Q^*:

$$V^*(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\}$$

$$Q^*(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] .$$

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

Additional Notes on the Optimal Action-Value Function

1. We can derive an **optimal policy** π^* from Q^* using:

$$\begin{aligned}\pi^*(s) &= \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s,a,s') \cdot [R(s,a,s') + \gamma V^*(s')] \right\} \\ &= \arg\max_{a \in A} \{Q^*(s,a)\}\end{aligned}$$

2. The observation that $V^*(s) = \max_{a \in A} Q^*(s,a)$ allows us to write the **optimal action-value function recursively**:

$$\begin{aligned}Q^*(s,a) &= \sum_{s' \in S} P(s,a,s') \cdot [R(s,a,s') + \gamma V^*(s')] \\ &= \sum_{s' \in S} P(s,a,s') \cdot \left[R(s,a,s') + \gamma \max_{a' \in A} \{Q^*(s',a')\} \right].\end{aligned}$$

Lecture 08-4d:
(Optional) More Complex Models

Partially Observable MDPs (POMDPs)

- The basic MDP model assumes that the agent fully observes their current state, and can therefore follow any policy with certainty
- Not all AI planning problems are accurately modeled in this way, since we may have, among other things:
  1. Incomplete information
  2. Incorrect information
  3. Noisy sensors
- The POMDP is a model for some of these situations

Formal Definition of a POMDP

- A POMDP extends the MDP by adding two new components:

$$M = < S, A, P, \Omega, O, R, T >$$

1. S = a set of states of the world
2. A = a set of actions an agent can take
3. P = a state-transition function: $P(s, a, s')$ is the probability of state s', starting in state s and taking action a: $\mathrm{P}(s' \mid s, a)$
4. Ω = a set of observations an agent can make
5. O = an observation function: $O(s, e)$ is the probability of observing evidence e when in state s: $\mathrm{P}(e \mid s)$
6. R = a reward function: $R(s, a, s')$ is the one-step reward you get if you go from state s to state s' after taking action a
7. T = a time horizon: we assume that every state-transition, following a single action, takes a single unit of time

Tractability of the Models

- Basic complexity analysis has shown that there are tractable algorithms for finite/infinite horizon MDPs
  - Linear programming techniques are among the most efficient
  - Very complex MDPs for real-world problems (including control of commercial aircraft) have been solved optimally
- POMDPs are considerably more complex
  - Conversion to a "belief-state" MDP produces models that are generally intractable to solve
  - It has been shown that finding optimal policies for infinite-horizon POMDPs is non-computable
  - Many advances have been made in optimal algorithms for finite-horizon problems, and in the use of approximation methods

Even More Complex Models

- Both MDPs and POMDPs are single-agent models
  - In each case, a single decision-maker is assumed to be acting
- Things get much more complicated once multiple agents are involved, and must interact with one another
  1. Competing agents: when agents have distinct reward functions, and do not always share the same interests, then game-theoretic techniques come into play
  2. Cooperating agents: when agents share a common reward function, and want to work as a team, it can be shown that the complexity of finding optimal/approximate plans increases greatly, making such planning infeasible almost always

CS 457/557: Machine Learning

Lecture 09-*:
Reinforcement Learning

University of Wisconsin
LA CROSSE

Prof. Jason Sauppe
jsauppe@uwlax.edu
https://cs.uwlax.edu/~jsauppe/

Fall 2024

Lecture 09-1a:
Passive Reinforcement Learning

Moving Into the Unknown...

Markov decision processes are used to model **known environments**:

- S: A set of states
- A: A set of actions
- P: Probabilities of all state transitions for all actions
- R: Rewards for all state transitions for all actions
- T: Time horizon

We saw how to **solve** an MDP via the **Bellman optimality equation**:

$$V^*(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^*(s') \right] \right\}.$$

In **reinforcement learning** proper though, things **aren't so easy!**

Basic Ideas in RL

As a starting point, we can think of a **reinforcement learning** problem as an MDP where **the agent doesn't know** the **state-transition probabilities** P and the **one-step rewards** R.

Despite this lack of information, we would like **the agent** to be able to:

1. Evaluate the quality of any given policy $\pi : S \to A$ (**passive reinforcement learning**)
2. Find an optimal (or at least good) policy (**active reinforcement learning**)

To make this happen, **the agent** needs to **learn**!

> *We say that a machine learns with respect to a particular task T, performance metric P, and type of experience E, if the system reliably improves its performance P at task T, following experience E.*
>
> — *Mitchell, The Discipline of Machine Learning*

Learning from Experience

The only winning move is not to play.

You can't win if you don't play.

The agent needs to play in order to learn!

General Idea: Perform many **learning episodes** (trials)

- For each episode:
  - **The agent** begins in a start state
  - **The agent** executes an action (possibly dictated by a policy) and moves to a next state, collecting a one-step reward along the way
  - This process repeats until the episode ends
- Based on **the agent's** experiences, it can update estimates of various quantities of interest, either after or during each episode

Lecture 09-1b:
Monte Carlo Methods

Monte Carlo Methods

Definition

A **Monte Carlo** method is a computational algorithm that uses repeated random sampling to obtain numerical results.

The basic idea behind **Monte Carlo** methods is to have **the agent** play many episodes and record various **observations** during each episode.

- Assuming that the agent makes a **sufficient number** of observations regarding an **unknown quantity**, it can construct a high-quality estimate for this quantity!

Monte Carlo Methods: Approach 1

Approach 1: In each episode, record, for each $(s, a, s') \in S \times A \times S$:

- The number of times **the agent** visited s and did a
- The number of times **the agent** ended up in state s' after this
- The reward **the agent** obtained by entering s' from s by a

After all episodes have completed, estimate $P(s, a, s')$ and $R(s, a, s')$ from this data, and then apply an MDP method to **solve** for an optimal policy (based on the estimates of P and R).

This can be made to work, but it has some **limitations**:

- Depending on the sizes of S and A, it may take **many trials** to get good estimates of P and R
- Learning only occurs **after** all episodes are done

Monte Carlo Methods: Approach 2

Approach 2: In each episode, record the **states** that **the agent** visited and the **one-step rewards** received for each transition. This generates an **environment history** (**trajectory**):

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, \ldots, r_n, s_n$$

At the end of each episode, for each visited state s:

- Let i be the time at which s is **first visited**, and compute $r = \sum_{t=i}^{n} \gamma^{t-i} r_t$ (discounted sum of rewards earned after first visit)
- Append r to a list (initially empty) associated with state s

After all episodes are finished, **the agent** estimates the value of each state s, $V(s)$, as the average of the total reward values recorded for s.

Compared to **Approach 1**, there is **less to learn** here, but learning still happens **only at the end**.

Lecture 09-2a:
Temporal Difference Learning

Learning the State-Value Function: Temporal Difference Updates

Passive Reinforcement Learning: Evaluating a Policy

procedure TDPolicyEvaluation(MDP M; Policy π; Discount factor γ; Step size α)
  for each state $s \in S$ **do**
    $V(s) \leftarrow 0$ ▷ $V(s)$ is current estimate of $V^\pi(s)$
  for each episode **do**
    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M
    while episode has not ended **do**
      Execute action $\pi(s)$ ▷ π may be a "random action" policy
      Observe next state s' and one-step reward $r = R(s, \pi(s), s')$
      $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$ ▷ Update estimated value of state s
      $s \leftarrow s'$ ▷ Move to next state
  return V ▷ $V(s) \approx V^\pi(s)$ for all $s \in S$

This is called the **temporal difference** (TD) update method:

- The estimated value of the state s' at the **next time step** is used to update the estimated value of the state s at **current time step**

The TD(0) Update

The TD(0) update:

$$V(s) = V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]$$

Every time the agent leaves a state s and moves to another state s', **the agent** updates its **estimated value** of s based on:

- its current estimate of the value of s, $V(s)$
- the one-step reward it gets, $r = R(s, \pi(s), s')$
- its (discounted) current estimate of the value of s', $\gamma V(s')$

Intuition:

- If $V(s') > V(s)$, then the value of s **tends to go up**
- If $V(s') < V(s)$, then the value of s **tends to go down**

The **value** of the **current state** should be **comparable** to the **value** of the **next state**, after factoring in the one-step reward and discounting.

The TD(0) Update and Step Size

In the TD(0) update, the **step-size parameter** α controls **how quickly the agent** updates its estimated state values:

$$\begin{aligned} V(s) &= V(s) + \alpha \left[r + \gamma V(s') - V(s) \right] \\ &= (1 - \alpha) V(s) + \alpha \left[r + \gamma V(s') \right] \end{aligned}$$

- If $\alpha = 0$, then **the agent** makes **no change** to $V(s)$
- If $\alpha = 1$, then **the agent replaces** the current estimated value of $V(s)$ with the reward and discounted value of the next state
- For $\alpha \in (0, 1)$, **the agent** tries to **compromise** between keeping the current value and replacing it based on the most recent information

The TD(0) Update and the Bellman Equation

In the TD(0) update with $\alpha = 1$, we have:

$$\begin{aligned} V(s) &= V(s) + \alpha \left[r + \gamma V(s') - V(s) \right] \\ &= (1 - \alpha) V(s) + \alpha \left[r + \gamma V(s') \right] \\ &= \left[r + \gamma V(s') \right]. \end{aligned}$$

Recall the **Bellman equation**:

$$\begin{aligned} V^{\pi}(s) &= \sum_{s'' \in S} P(s, a, s'') \cdot \left[R(s, a, s'') + \gamma V^{\pi}(s'') \right] \\ &\approx R(s, a, s') + \gamma V^{\pi}(s') \quad \text{if } P(s, a, s') = 1 \end{aligned}$$

So the TD(0) update **looks like** a **Bellman update** where **the agent** assumes $P(s, a, s') = 1$ for the **one state** s' that it actually observes!

The TD(0) Update: TD Target and TD Error

The TD(0) update

$$V(s) = V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]$$

can be viewed as an attempt to construct $V(s)$ values that **satisfy** the **Bellman equation** under the assumption of **deterministic** transitions.

In the TD(0) update:

- The expression $r + \gamma V(s')$ is the **TD target**
  (i.e., the value that $V(s)$ "should" have based on above)
- The expression $r + \gamma V(s') - V(s)$ is the **TD error**
  (i.e., the difference between the value that $V(s)$ currently has and what it "should" have)

The TD(0) Update and Convergence

With time indexing and some rearranging, we can write the TD(0) update as

$$V^{(t+1)}(s) = (1-\alpha)V^{(t)}(s) + \alpha\left[r + \gamma V^{(t)}(s')\right].$$

- With $\alpha = 1$, $V(s)$ is replaced with the reward and next state value from the **most recent** transition.
- With $\alpha < 1$, the updates to $V(s)$ will **retain** some information from past transitions, not just the most recent one.

If α shrinks slowly over time, e.g., $\alpha^{(t)} = 0.9/(1 + t/100)$, then **the agent** can ensure that its $V(s)$ values **converge** to the actual $V^{\pi}(s)$ values.

Brief Aside: Some Calculations

Example: Suppose we visit state s at times t_1, t_2, and t_3, collecting rewards r_1, r_1, and r_3 and reaching states s'_1, s'_2, and s'_3. Because $V(s)$ only updates after we leave s, we have

$$V^{(t)}(s) = \begin{cases} V^{(0)}(s) = 0 & \text{if } t \in \{1, 2, \ldots, t_1\} \\ V^{(t_1+1)}(s) & \text{if } t \in \{t_1 + 1, \ldots, t_2\} \\ V^{(t_2+1)}(s) & \text{if } t \in \{t_2 + 1, \ldots, t_3\} \\ \ldots \end{cases}$$

After each TD(0) update, we have:

$$\begin{aligned} V^{(t_1+1)}(s) &= (1-\alpha)V^{(0)}(s) + \alpha\left[r_1 + V^{(t_1)}(s'_1)\right] \\ &= \alpha\left[r_1 + V^{(t_1)}(s'_1)\right] \\ V^{(t_2+1)}(s) &= (1-\alpha)V^{(t_1+1)}(s) + \alpha\left[r_2 + V^{(t_2)}(s'_2)\right] \\ &= (1-\alpha)\alpha\left[r_1 + V^{(t_1)}(s'_1)\right] + \alpha\left[r_2 + V^{(t_2)}(s'_2)\right] \\ V^{(t_3+1)}(s) &= (1-\alpha)V^{(t_2+1)}(s) + \alpha\left[r_3 + V^{(t_3)}(s'_3)\right] \\ &= (1-\alpha)\left\{(1-\alpha)\alpha\left[r_1 + V^{(t_1)}(s'_1)\right] + \alpha\left[r_2 + V^{(t_2)}(s'_2)\right]\right\} + \alpha\left[r_3 + V^{(t_3)}(s'_3)\right] \end{aligned}$$

So α influences how **the agent** combines past transitions with the most recent one.

Pros and Cons of TD Updates

With TD updates, **the agent**:

- Can estimate the **values** of states **without** knowing the actual state-transition probabilities $P(s, a, s')$
- Only updates the **value** of **one state** at a time, but these updates are applied immediately (i.e., during the episode)
- **Never learns values** for states that it doesn't reach

This last point **may be advantageous** if these states aren't interesting and/or if the current policy would always avoid them.

However, if these unknown states are **actually useful**, then this **can be problematic** for doing **policy improvement**. We'll revisit this idea in more detail later on.

Lecture 09-2b:
Towards Active Reinforcement Learning

Finding Better Policies in RL

The preceding discussion focused on **passive reinforcement learning**: figuring out how **the agent** can evaluate a **fixed policy** π.

In **active reinforcement learning**, we would like **the agent** to find an **optimal** (or at least good) policy.

- **The agent** will still perform episodes to acquire experiences
- But now **the agent** should also **improve** its policy as it goes!

How do we make this happen?

Policy Improvement Idea

Recall in **policy improvement** for MDPs, we can generate a **new policy** π' from the current policy π using

$$\pi'(s) = \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{\pi}(s') \right] \right\}.$$

With TD(0) updates, **the agent learns** V^{π} for its current policy π.

Problem: Without knowing P and R, **the agent** cannot figure out how to pick better actions!

One Solution: Estimate P and R from the data

Another Solution: Learn the **action-value function** Q^{π} instead of V^{π}

Policy Improvement via the Action-Value Function

The **policy improvement** process can be reformulated in terms of the **action-value function** Q^π instead of the **state-value function** V^π:

$$\begin{aligned}\pi'(s) &= \arg\max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^\pi(s') \right] \right\} \\ &= \arg\max_{a \in A} \left\{ Q^\pi(s, a) \right\}.\end{aligned}$$

So if **the agent** knows Q^π, it can figure out how to create a potentially better policy π' from it, **without** needing to know P or R!

The agent can use the same **temporal difference update** that we saw earlier for V^π to help it learn Q^π.

Lecture 09-2c:
Learning Action-Values

Revisiting the Bellman Equations

Recall the **Bellman equations** for the **state-value** and **action-value** functions:

$$V^{\pi}(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right]$$

$$Q^{\pi}(s, a) = \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma V^{\pi}(s') \right]$$

We have $V^{\pi}(s) = Q^{\pi}(s, \pi(s))$, so we can write $Q^{\pi}(s, a)$ as:

$$Q^{\pi}(s, a) = \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma Q^{\pi}(s', \pi(s')) \right].$$

During **learning**, when **the agent** takes action a in state s and ends up in state s' with one-step reward $r = R(s, a, s')$, it can assume that $P(s, a, s') = 1$ and approximate the **Bellman equation** as:

$$Q^{\pi}(s, a) \approx r + \gamma Q^{\pi}(s', \pi(s')).$$

TD Updates for Action-Values

The **TD(0) update** for **action-values** is given by:

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right].$$

TD Updates for Action-Values: Evaluating a Policy

---

procedure TDPolicyEvaluation(MDP M; Policy π; Discount factor γ; Step size α)

  for each $(s,a) \in S \times A$ **do**

    $Q(s,a) \leftarrow 0$ ▷ $Q(s,a)$ is current estimate of $Q^\pi(s,a)$

  for each episode **do**

    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M

    $a \leftarrow \pi(s)$ ▷ a is next action to take

    while episode has not ended **do**

      Execute action a

      Observe next state s' and one-step reward $r = R(s,a,s')$

      $a' \leftarrow \pi(s')$ ▷ a' is what the agent will do next

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

      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action

  return Q ▷ $Q(s,a) \approx Q^\pi(s,a)$ for all $(s,a) \in S \times A$

Lecture 09-2d:
Greedy Policy Improvement

Towards an Optimal Policy

The agent can use the following process to find an **optimal** policy:

1. Start with an initial policy π (possibly random)
2. Learn Q^π via TD(0) updates (**policy evaluation**)
3. Construct π' from Q^π values (**policy improvement**)
4. Update policy $\pi \leftarrow \pi'$ and repeat until no changes

This is potentially **slow**, though: **the agent** has to wait until it has learned Q^π before it can improve its policy.

Generalized policy iteration refers to the general idea of combining **policy evaluation** and **policy improvement** within a single process.

In this particular case, we'd like **the agent** to try to learn an **optimal** policy from the get-go, by potentially **updating** its policy after each step.

Greedy Policy Improvement

A **greedy policy** always picks the action that looks best at each step:

$$\pi^{\text{greedy}}(s) \leftarrow \arg\max_{a \in A} Q(s,a) \qquad \text{(ties broken at random)}$$

Greedy Policy Improvement

---

procedure GreedyPolicyImprovement(MDP M; Discount factor γ; Step size α)

  for each $(s,a) \in S \times A$ **do**

    $Q(s,a) \leftarrow 0$

  for each episode **do**

    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M

    $a \leftarrow \pi^{\text{greedy}}(s)$ ▷ a is next action to take, chosen greedily

    while episode has not ended **do**

      Execute action a

      Observe next state s' and one-step reward $r = R(s,a,s')$

      $a' \leftarrow \pi^{\text{greedy}}(s')$ ▷ a' is what the agent will do next

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

      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action

  return learned policy π, with actions chosen greedily for each state

---

Issues with Greedy Policy Improvement

Problem: Greedy isn't always good.

Example: Consider an agent exploring a maze.

- Once **the agent** learns a sequence of actions that arrive at the goal, it will be **strongly incentivized** to use those same actions on subsequent episodes
- This may cause **the agent** to overlook or miss other action sequences that would get it to the goal faster

A **greedy improvement process** may end up with a **suboptimal** policy.

Exploration and Exploitation

As **the agent** learns, it needs to balance two things:

- **Exploitation**: Taking the best-looking actions based on what it currently knows
- **Exploration**: Trying out new actions even if they don't look as good as other options, based on what it currently knows

Textbook quote:

> *In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.*

Lecture 09-2e:
ϵ-Greedy Policies and SARSA

Almost-Greedy Policies

The **greedy policy** focuses **almost exclusively** on **exploitation** (aside from breaking ties at random).

A relatively simple way to incorporate **exploration** is to adjust the policy to be **mostly greedy**, but **not always**.

An **epsilon-greedy** (ϵ-greedy) policy $\pi^{\epsilon\text{-greedy}}$, where $\epsilon \in [0, 1]$, chooses the next action in any state s using the following process:

1. Generate a random number $R \in [0, 1]$
2. If $R \leq \epsilon$, choose an action **at random**
3. If $R > \epsilon$, choose an action **greedily** via $\arg\max_{a \in A} Q(s, a)$

By using an ϵ-**greedy policy** in the policy improvement process, the **learned action-values** $Q(s, a)$ can be made to converge to the **optimal action-values** $Q^*(s, a)$.

SARSA Learning

ϵ-Greedy Policy Improvement: SARSA

procedure SARSALEARNING(MDP M; γ; α; ϵ)
  for each $(s, a) \in S \times A$ **do**
    $Q(s, a) \leftarrow 0$ ▷ $Q(s, a)$ is current estimate of $Q^*(s, a)$
  for each episode **do**
    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M
    $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$ ▷ a is next action to take, chosen ϵ-greedily
    while episode has not ended **do**
      Execute action a
      Observe next state s' and one-step reward $r = R(s, a, s')$
      $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$ ▷ a' is what the agent will do next
      $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma Q(s', a') - Q(s, a)\right]$
      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action
  return learned policy π, with actions chosen greedily for each state

- This algorithm is called **SARSA** for s, a, r, s', a'
- By decreasing ϵ over time, **the agent** can stabilize the policy and ensure convergence

Lecture 09-3a:
On-Policy vs. Off-Policy Updates

SARSA Learning: On-Policy Updates

ϵ-Greedy Policy Improvement: SARSA

procedure SARSALEARNING(MDP M; γ; α; ϵ)
  for each $(s, a) \in S \times A$ **do**
    $Q(s, a) \leftarrow 0$ ▷ $Q(s, a)$ is current estimate of $Q^*(s, a)$
  for each episode **do**
    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state
    $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$ ▷ a is next action to take, chosen ϵ-greedily
    while episode has not ended **do**
      Execute action a
      Observe next state s' and one-step reward $r = R(s, a, s')$
      Determine $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$ ▷ a' is what we're going to do next
      $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma Q(s', a') - Q(s, a)\right]$
      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action
  return learned policy π, with actions chosen greedily for each state

SARSA is an **on-policy** update method: The **value** of the next state s' is based on the next action a', which is picked **according to the policy**.

The Impact of On-Policy Updates

An **on-policy** update of $Q(s, a)$ uses the action value $Q(s', a')$ where action a' is chosen according to the ϵ-**greedy policy**:

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

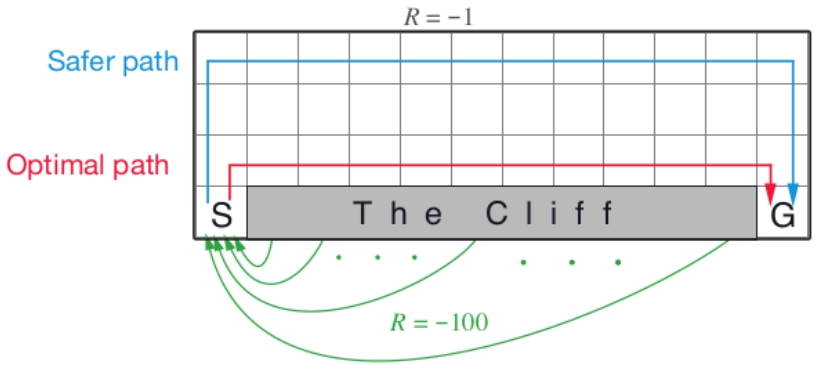
- With probability $(1 - \epsilon)$, a' **is the best possible action** in state s'
- With probability ϵ, a' is a **random** action

This means that the TD(0) update of $Q(s, a)$ is based on a **potentially suboptimal** action a' in the next state:

$$Q(s', a') \leq \max_{a'' \in A} Q(s', a'').$$

An Issue with On-Policy Updates: The Cliff Problem

Example: Consider an agent navigating to a goal, with some states adjacent to a **cliff**.



Moving into a cliff cell results in a reward of -100 and causes the agent to return to the start

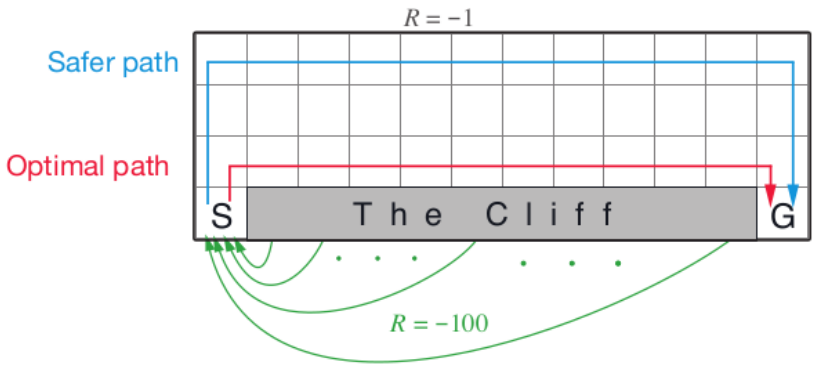
The ϵ-**greedy policy** will **sometimes** decide to jump off the cliff:

- How does **the agent** know **not** to jump off the cliff unless it tries it and sees what happens?
- The choice of random actions ensures that the agent **explores** its environment, even if it does the wrong thing occasionally

This is generally fine, except when it comes to the TD(0) update....

The Cliff Problem: More Details

Let's look at what can happen with the TD(0) update here.



Let:

- $s_0 = (1, 1)$ be the start state
- $s = (1, 2)$ be the state above the start
- $s' = (2, 2)$ be the state to the right
- U, D, L, R be the actions

The first time the agent is in s' and chooses to move down, it learns that jumping off the cliff (while in s') is bad:

$$\begin{aligned} Q(s', D) &= Q(s', D) + \alpha \left[r + \gamma Q(s_0, U) - Q(s', D) \right] \\ &= 0 + 0.9 \left[-100 + 0.9 \cdot 0.0 - 0.0 \right] = -90 \end{aligned}$$

Later, when the agent is in s and chooses to move right, it updates $Q(s, R)$ based on what it **will do next** in s':

$$Q(s, R) = Q(s, R) + \alpha \left[r + \gamma Q(s', a') - Q(s, R) \right]$$

If the next action in s' is $a' = D$, which is possible in an ϵ-greedy policy, then $Q(s, R)$ will **look worse** than it actually is!

Consequences of On-Policy Updates: Summary and the Cliff Problem

In an **on-policy** TD(0) update, the change to the value of the current action depends on the value of the next action:

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right].$$

In **SARSA**, the next action a' may be chosen at random, and hence it may be **suboptimal**:

$$Q(s',a') \leq \max_{a'' \in A} Q(s',a'')$$

In the cliff problem, this causes the **learned values** of actions leading to states along the cliff to be **lower** than they otherwise would be if the agent **acted optimally** in states along the cliff (i.e., by never jumping).

Ensuring Optimal TD Updates

If we want $Q(s, a)$ to converge to the **optimal action-value** $Q^*(s, a)$, then the TD update should be based on the **optimal action** in state s'.

Two ways to correct this:

1. Ensure that ϵ goes to 0 over time, so that eventually **the agent** always acts optimally
2. Revise the TD update to use the **optimal action** in state s', instead of the action that the agent **will take**, which may be **suboptimal**

The revised TD update is based on the **Bellman optimality equation**:

$$Q^*(s, a) = \sum_{s'' \in S} P(s, a, s'') \cdot \left[R(s, a, s'') + \gamma \max_{a'' \in A} \left\{ Q^*(s'', a'') \right\} \right]$$

$$\approx r + \gamma \max_{a'' \in A} \left\{ Q^*(s', a'') \right\} \quad \text{for observed state } s'$$

Q-Learning: Off-Policy Updates

Q-Learning with ϵ-Greedy and Off-Policy Updates

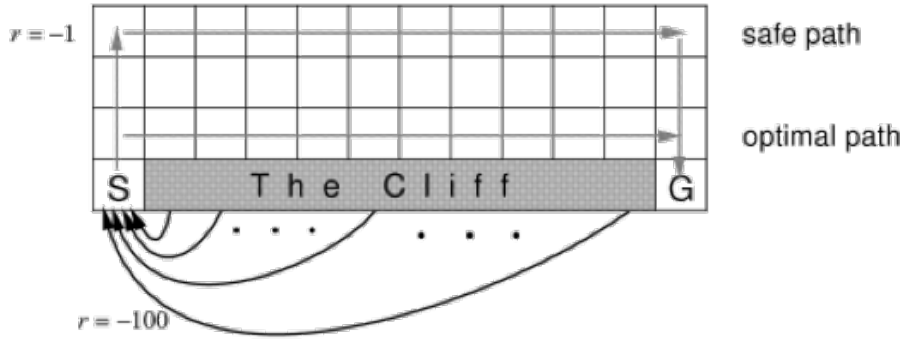
procedure QLEARNING(MDP M; γ; α; ϵ)
  for each $(s, a) \in S \times A$ **do**
    $Q(s, a) \leftarrow 0$ ▷ $Q(s, a)$ is current estimate of $Q^*(s, a)$
  for each episode **do**
    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M
    $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$ ▷ a is next action to take, chosen ϵ-greedily
    while episode has not ended **do**
      Execute action a
      Observe next state s' and one-step reward $r = R(s, a, s')$
      $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$ ▷ a' is what the agent will do next
      $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'' \in A} \{Q(s', a'')\} - Q(s, a) \right]$
      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action
  return learned policy π, with actions chosen greedily for each state

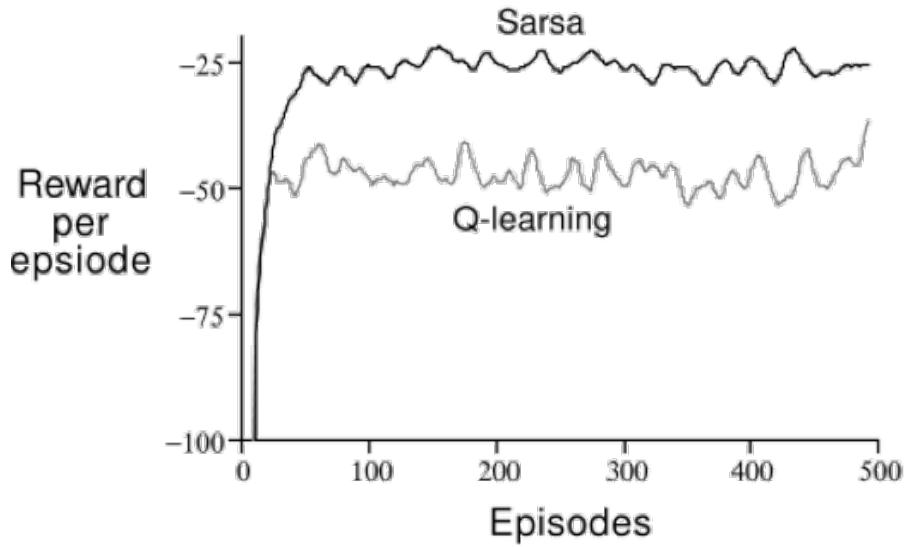
Q-Learning is an **off-policy** update method.

- **The agent** still picks actions a and a' according to policy (e.g., ϵ-greedy)
- **The agent** updates $Q(s, a)$ using the value of the **optimal action** in state s', instead of the action a' that it will take (which **may be suboptimal**)

Comparing the Methods: Cliff Problem

- Shortest path to the goal goes along edge of a cliff
  - SARSA learns safer path, since edge-states get lower values due to random falls
  - Q-Learning learns best path, since it ***ignores*** random jumps off edge
  - Why does QL do worse in the end? How can we fix this over a period of time?





Example from: Sutton & Barto, 1998

Comparing SARSA and Q-Learning

In **Q-Learning**, the **learned** action-values $Q(s, a)$ **converge** to the **optimal** action-values $Q^*(s, a)$.

SARSA **can also** achieve this provided that ϵ **decreases** over time.

The primary difference is that **Q-Learning** often converges to Q^* **faster** than **SARSA**.

- **Q-Learning** doesn't have to spend time "fixing" values of states where it has **overestimated** the risk
- This allows for reducing ϵ more rapidly so that **the agent** can learn an optimal policy sooner
  - But reducing ϵ **too rapidly** can prevent necessary exploration

Lecture 09-4a: Eligibility Traces

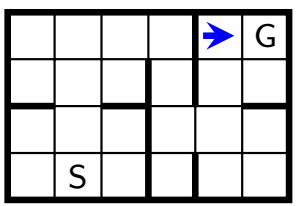
Limitations of TD(0) Updates

One **limitation** of the basic TD(0) update is that it just updates the value of **one** action at a time:

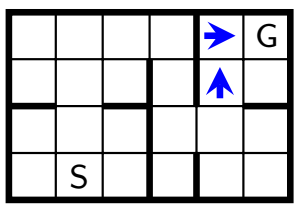
$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right].$$

In a maze problem with **sparse** rewards (e.g., only at the end), it can take a **long time** for reward values to trickle back:

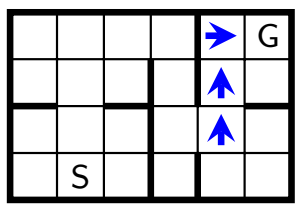
Learned action values after 1 episode:



Learned action values after 2 episodes:



Learned action values after 3 episodes:



Sometimes, we can **speed up learning** by trying to "spread out" each update now instead of letting this happen eventually over time.

Eligibility Traces

In order to spread out TD updates, **the agent** needs to track **how recently** it visited states and/or did certain actions.

For each state-action pair $(s, a) \in S \times A$, let $e(s, a)$ denote the **eligibility** of (s, a) for participating in the current TD update.

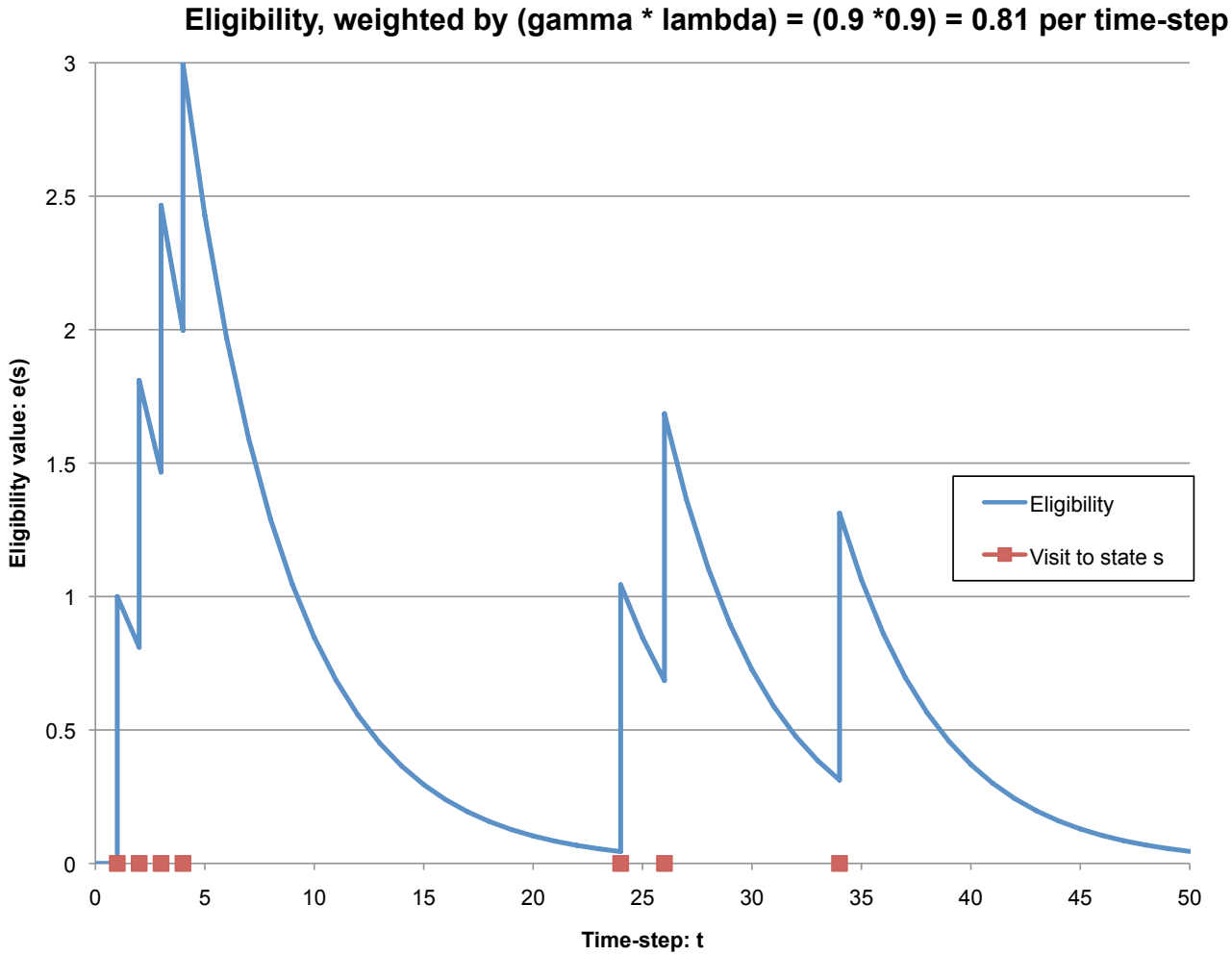
- Initially, $e(s, a) = 0$ for all (s, a) pairs
- Every time the agent visits s and does a, $e(s, a)$ goes up by 1
- As time elapses, these eligibility values **decay** back towards 0, with the rate of decay controlled by a parameter $\lambda \in [0, 1]$.

When a TD update is applied, **the agent**

- computes the **TD error** as before: $\delta = r + \gamma Q(s', a') - Q(s, a)$
- **updates all** action values in proportion to the eligibility values:

$$Q(s'', a'') = Q(s'', a'') + \alpha \delta e(s'', a'') \quad \forall (s'', a'') \in S \times A$$

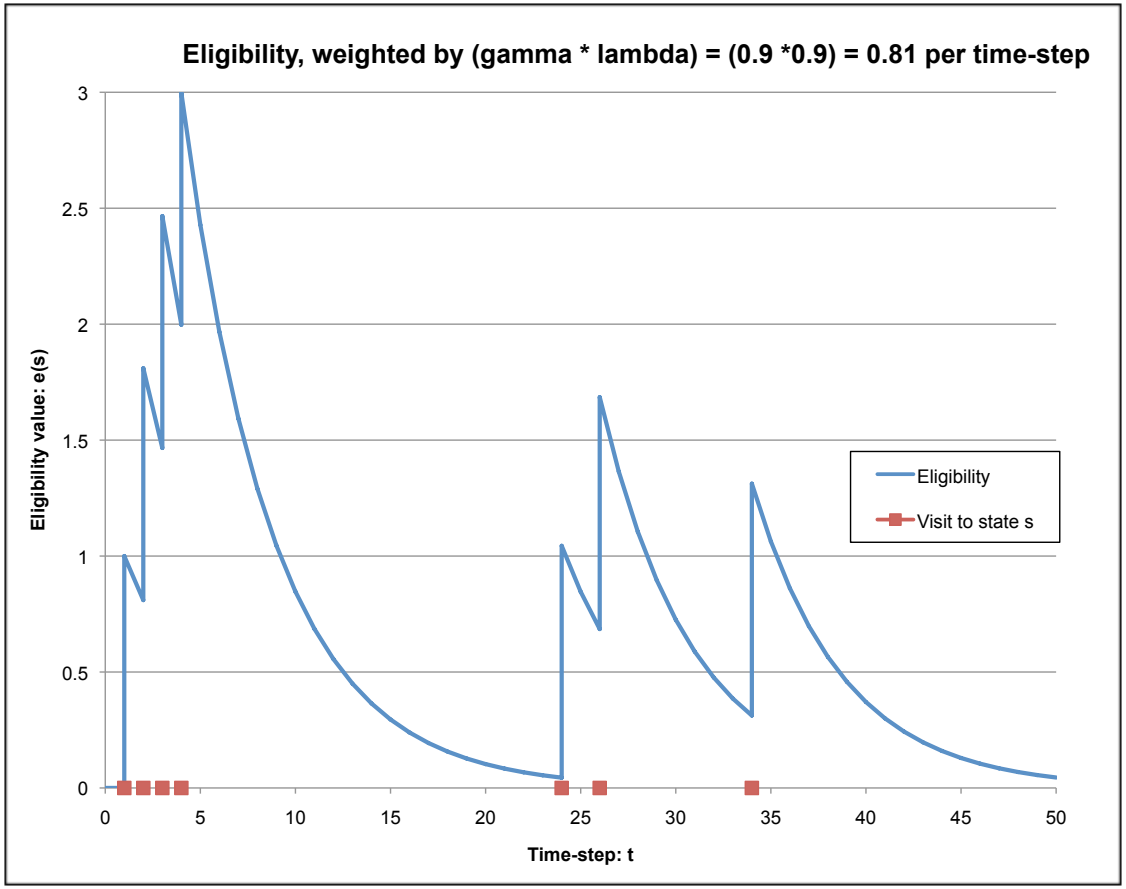
Eligibility Traces



- The eligibility trace grows each time a state is visited, and then "decays" towards 0 over time, until that state is visited again

Eligibility Traces

- We then take the TD error—the difference between values of the current state and the last state, plus current reward—and ***spread this out*** over all the past states, ***in proportion*** to their eligibility values:



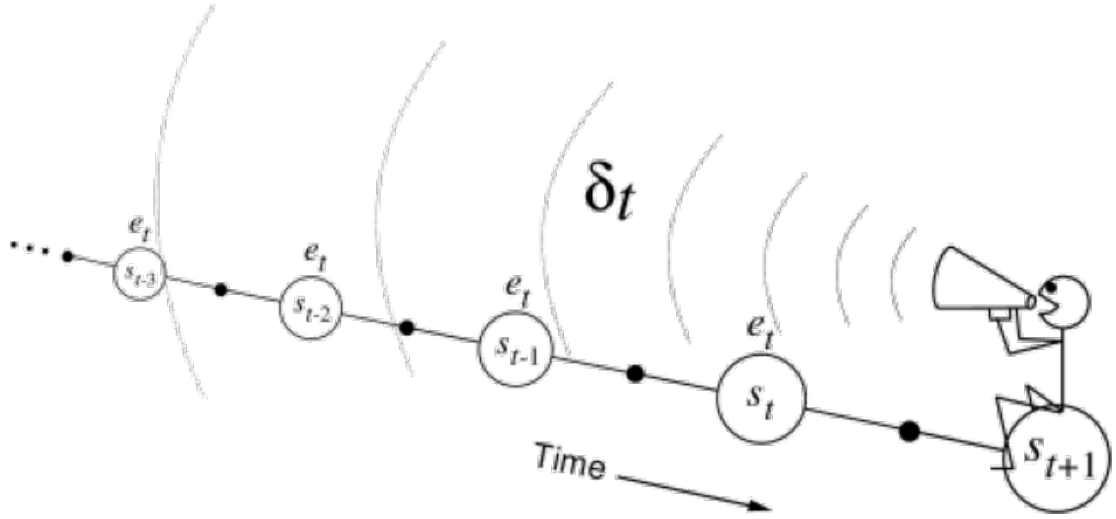
$$\delta_t = r_{t+1} + \gamma\, Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

SARSA(λ): Learning with Eligibility Traces

SARSA(λ): Learning Action Values with Eligibility Traces

procedure SARSA-λ(MDP M; γ; α; ϵ; λ)
- **for each** $(s, a) \in S \times A$: $Q(s, a) \leftarrow 0$
- **for each** episode **do**
  - **for each** $(s, a) \in S \times A$: $e(s, a) \leftarrow 0$ ▷ Reset eligibility values at start of each episode
  - $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M
  - $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$ ▷ a is next action to take, chosen ϵ-greedily
  - **while** episode has not ended **do**
    - Execute action a
    - Observe next state s' and one-step reward $r = R(s, a, s')$
    - $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$ ▷ a' is what the agent will do next
    - $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ ▷ Compute TD error
    - $e(s, a) \leftarrow e(s, a) + 1$ ▷ Update eligibility of current state-action pair
    - **for each** $(s'', a'') \in S \times A$ **do**
      - $Q(s'', a'') \leftarrow Q(s'', a'') + \alpha\delta e(s'', a'')$
      - $e(s'', a'') \leftarrow \gamma\lambda e(s'', a'')$ ▷ $e(s'', a'')$ decays over time
    - $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action
- **return** learned policy π, with actions chosen greedily for each state

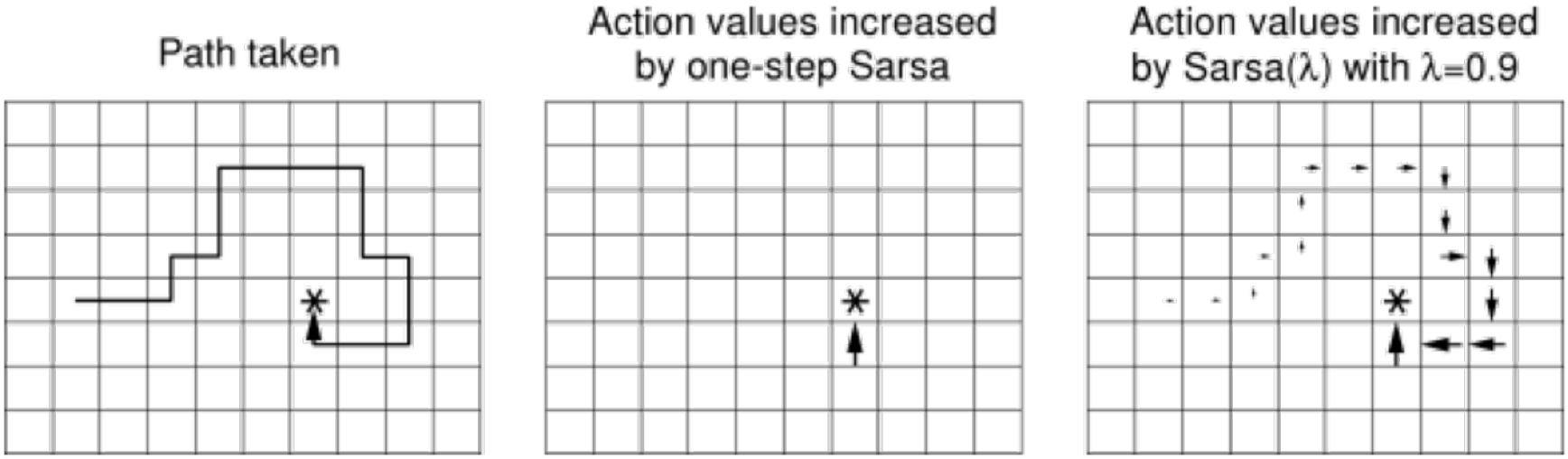
Using Eligibility Traces



- The algorithm sends the TD-error back over prior time-steps, with a "distance" that is affected by the choice of λ
- If we choose $\lambda = 0$, the algorithm is simply doing a ***single-state backup*** as before
- As λ nears (but never equals) 1, the algorithm pushes the updates further and further back in time

Diagram taken from: Sutton & Barto, 1998

Possible Advantages of Using Traces



- In many cases, using Eligibility Traces can speed the learning process, since whole chains of states can be updated at once
- This works especially well for path-planning problems, and things similar to it, since the value updates can affect multiple locations along the path to a goal at the same time

Example from: Sutton & Barto, 1998

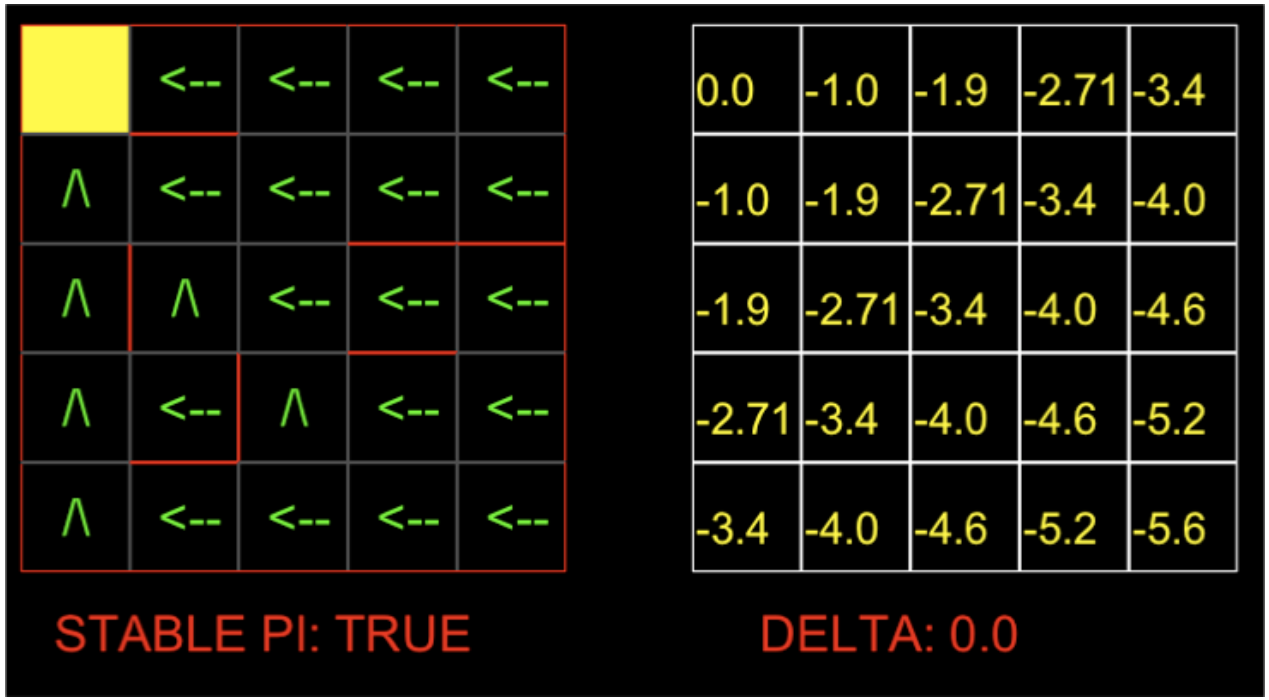
Q-Learning with Eligibility Traces

Watkins' Q(λ): Q-Learning with Eligibility Traces

procedure Q-λ(MDP M; γ; α; ϵ; λ)
  for each $(s, a) \in S \times A$: $Q(s, a) \leftarrow 0$
  for each episode **do**
    for each $(s, a) \in S \times A$: $e(s, a) \leftarrow 0$ ▷ Reset eligibility values at start of each episode
    $s \leftarrow s_0$ ▷ s is current state, s_0 is fixed start state of M
    $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$ ▷ a is next action to take, chosen ϵ-greedily
    while episode has not ended **do**
      Execute action a
      Observe next state s' and one-step reward $r = R(s, a, s')$
      $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$ ▷ a' is what the agent will do next
      $a^* \leftarrow \arg\max_{a'' \in A} Q(s', a'')$ ▷ a^* is the optimal next action
      $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$ ▷ Compute TD error
      $e(s, a) \leftarrow e(s, a) + 1$ ▷ Update eligibility of current state-action pair
      for each $(s'', a'') \in S \times A$ **do**
        $Q(s'', a'') \leftarrow Q(s'', a'') + \alpha\delta e(s'', a'')$
        $e(s'', a'') \leftarrow \begin{cases} \gamma\lambda e(s'', a'') & \text{if } a' = a^* \\ 0 & \text{if } a' \neq a^* \end{cases}$ ▷ Reset eligibility if a' is random
      $s \leftarrow s'$, $a \leftarrow a'$ ▷ Update current state and next action
  return learned policy π, with actions chosen greedily for each state

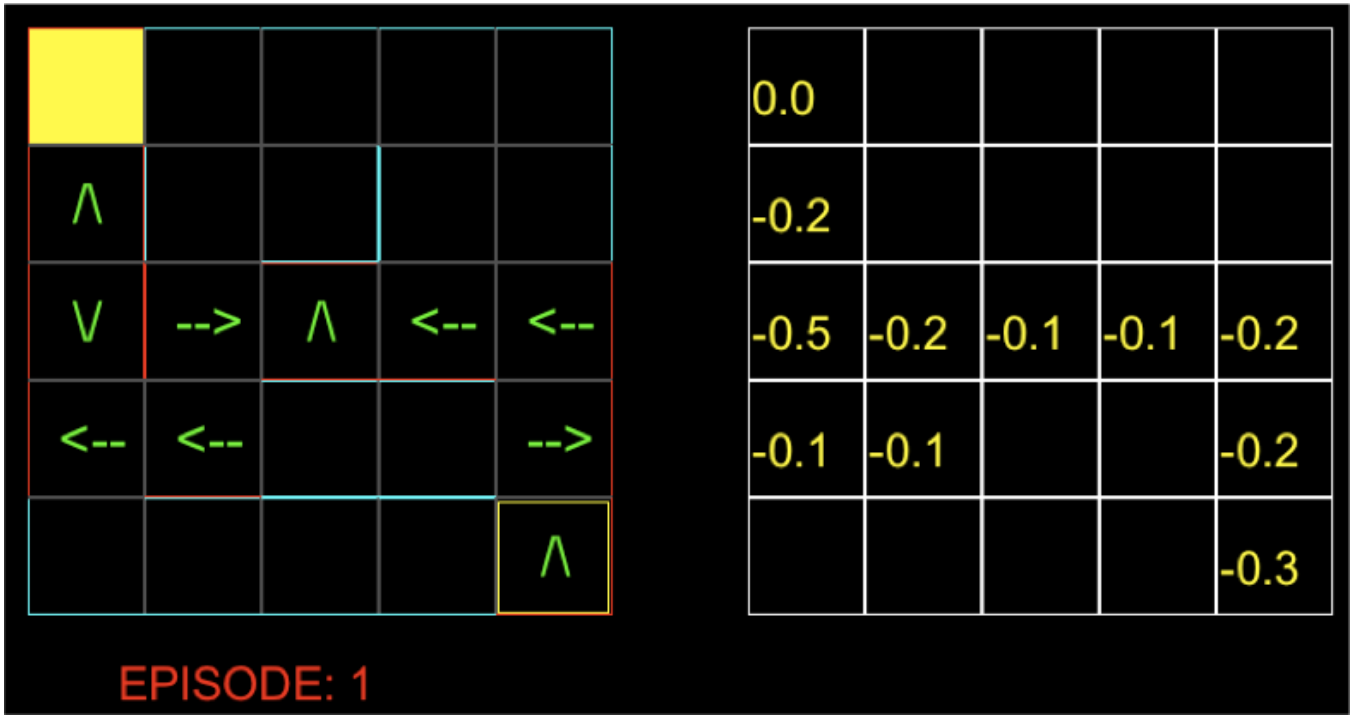
Lecture 09-4b:
Planning and Learning in RL

Full-Fledged Planning



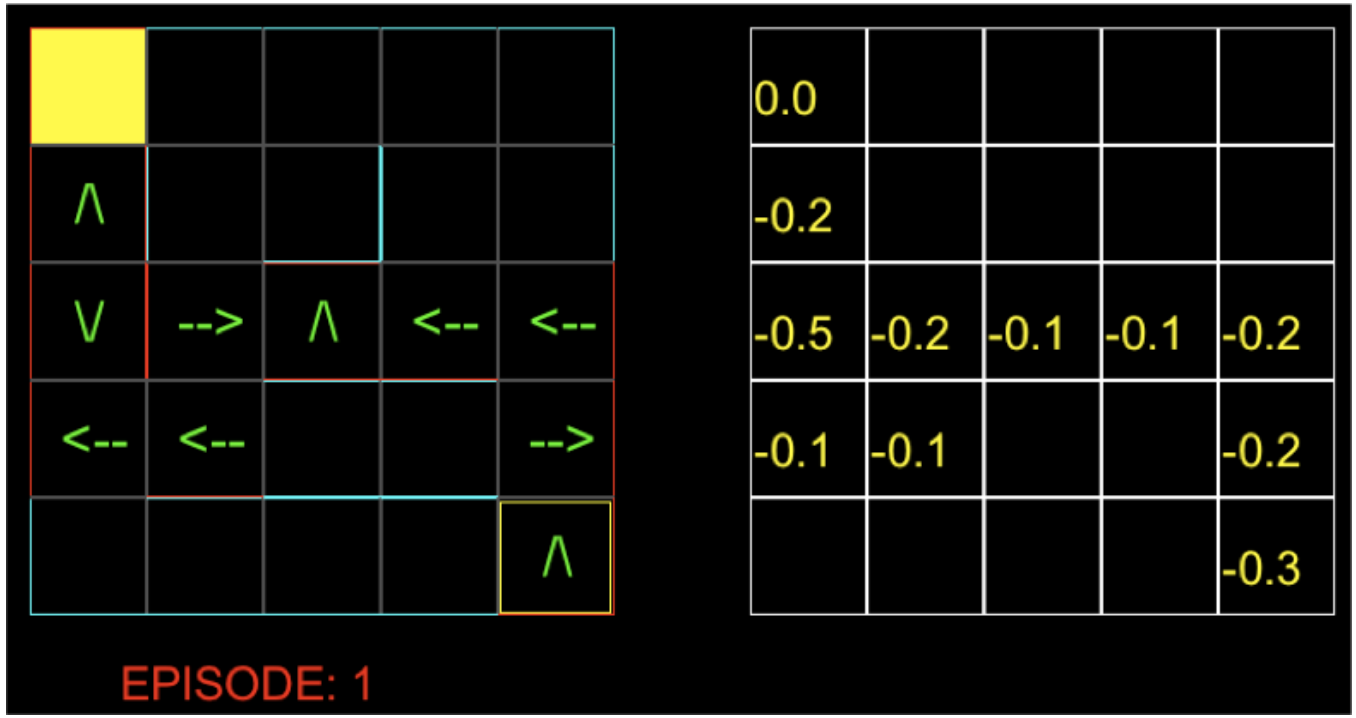
- Planning provides a full solution:
  - We have a model of the entire domain
  - We derive a solution for that whole model
- What are some potential benefits? Problems?

Learning our Model



- Learning allows us to ***find*** a correct model
  - Don't need to create model ahead of time
  - Can avoid incorrect models
- Is there ***anything wrong*** with the plan learned so far?

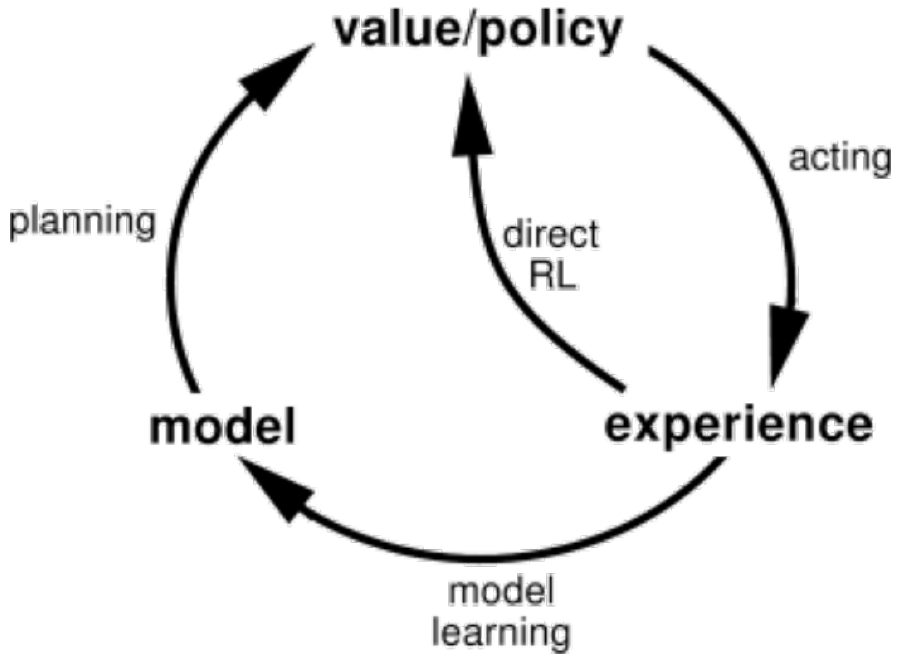
Sub-Optimal Learning



- Basic TD methods build model of the environment as they go
- Early on, model is ***partial***, and action-policies are ***sub-optimal***
- Even based on what we have learned about the environment so far, we are not doing the right thing!

Combining Planning & Learning

- What if we combine the two ideas together?
  - Use "direct RL" to ***build*** model
  - Use "indirect RL" to ***test policies***, based on that model
  - Then do ***planning*** to get best current solution for the model
  - ***Repeat*** to get to optimal policy



The *Dyna-Q* Algorithm

function DYNA-Q (mdp) **returns a policy**

inputs: mdp, an MDP, and π, a policy to be evaluated

$\forall s \in S, \forall a \in A, Q(s,a) = 0$ **and** $Model(s,a) = (s,0)$

repeat forever :

$s \leftarrow$ current state

take action a, chosen ϵ-greedily based on $Q(s,a)$

observe next state s', one-step reward r

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

$Model(s,a) \leftarrow (s',r)$ (***assumes determinism***)

repeat N **times** :

$s \leftarrow$ a randomly previously observed state

$a \leftarrow$ an random action previously taken in state s

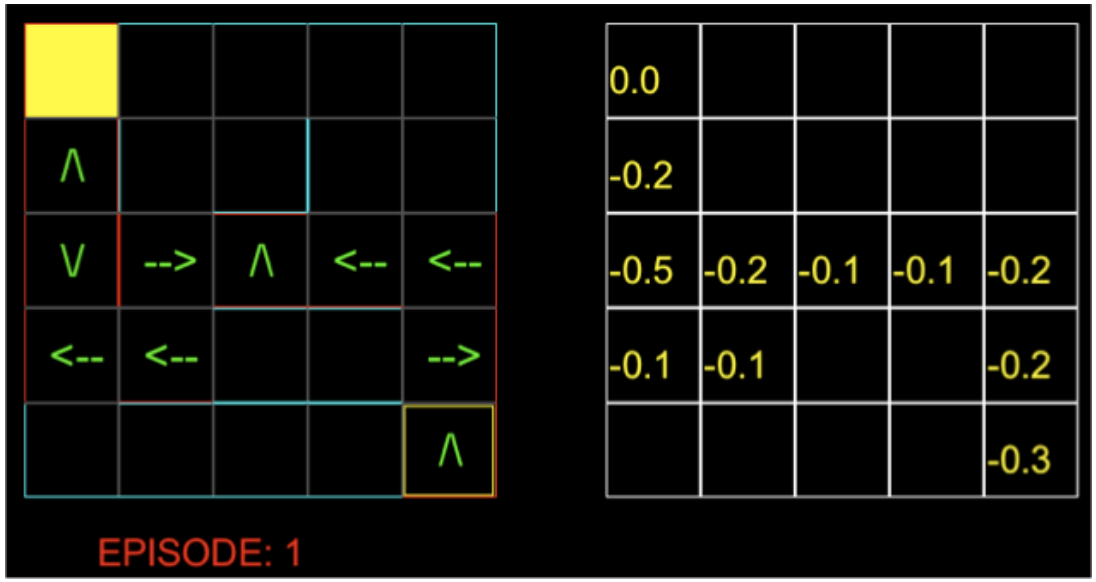
$(s',r) \leftarrow Model(s,a)$

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

return policy π, set greedily for every (s,a) based upon $Q(s,a)$

- This method stores a separate set of results, the *Model*, based on what successor states and rewards have actually been seen while learning
- These are used to ***simulate outcomes***, and build better value estimates for the states and actions we have actually seen
- What happens when $N = 0$?

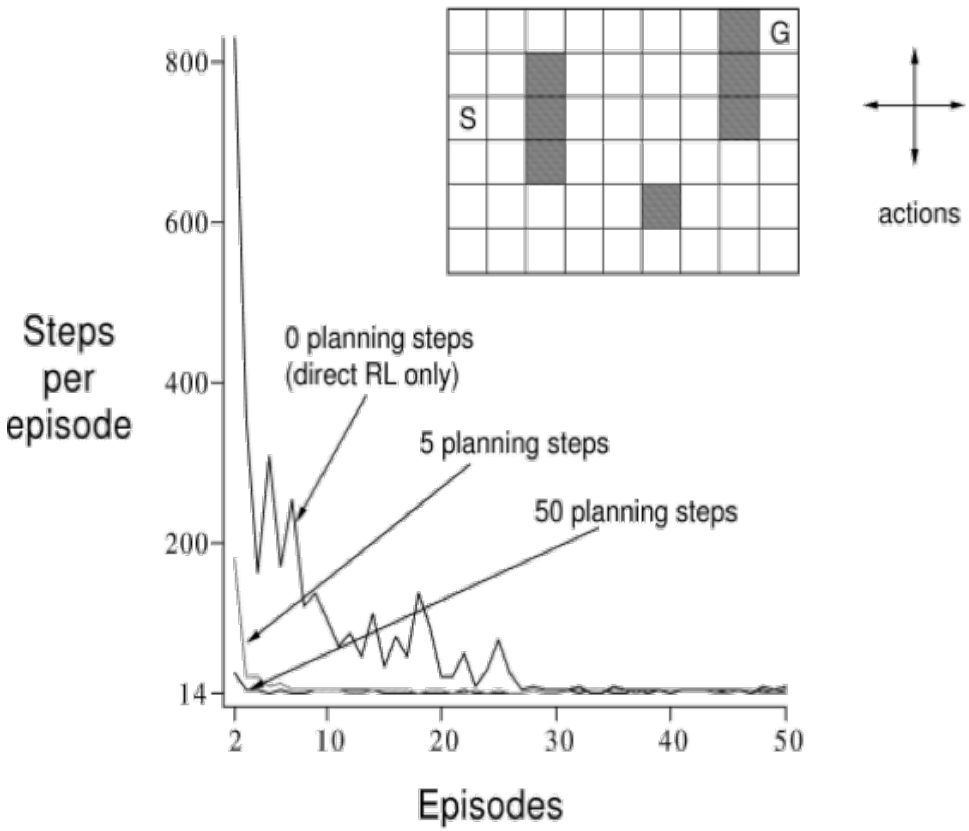
$N = 0$: Regular TD-Learning



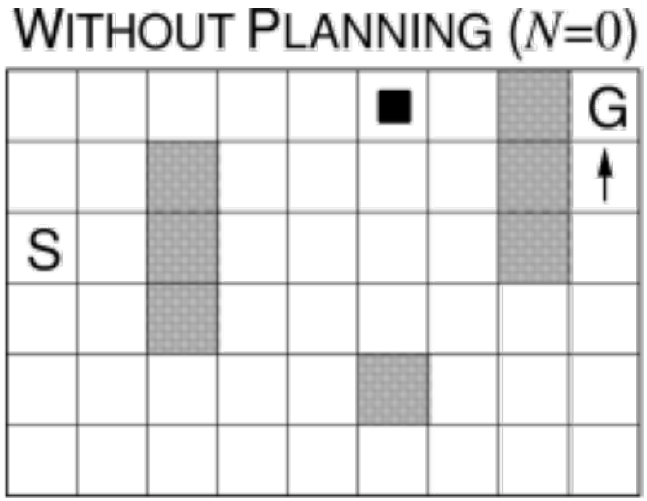
- After a single episode in the maze where we find the goal, ***one-step policies*** (the last arrow before goal) are already ***correct***
- Remaining policy is not really very good in general
- Dyna-Q can fix this problem

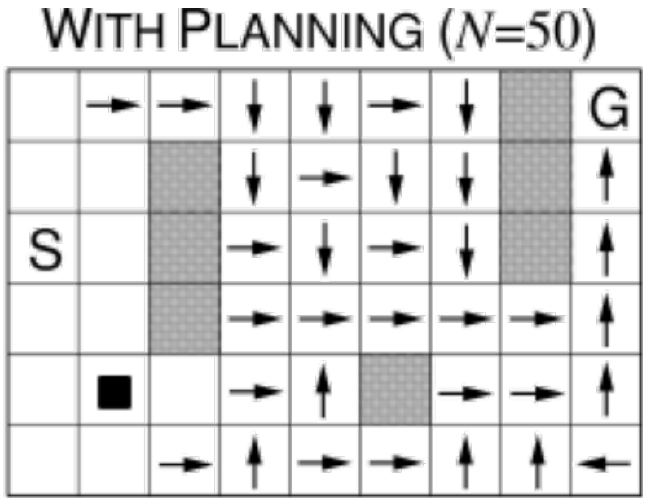
Using Dyna-Q Methods

- In this Maze Problem, agents get 0 reward in all states, except the goal, where there is a positive reward (+1)
- We can see effects of different values of *N*
- As *N* increases, the agent learns better policies in fewer episodes
- Why does this happen?



Advantages of Dyna-Q

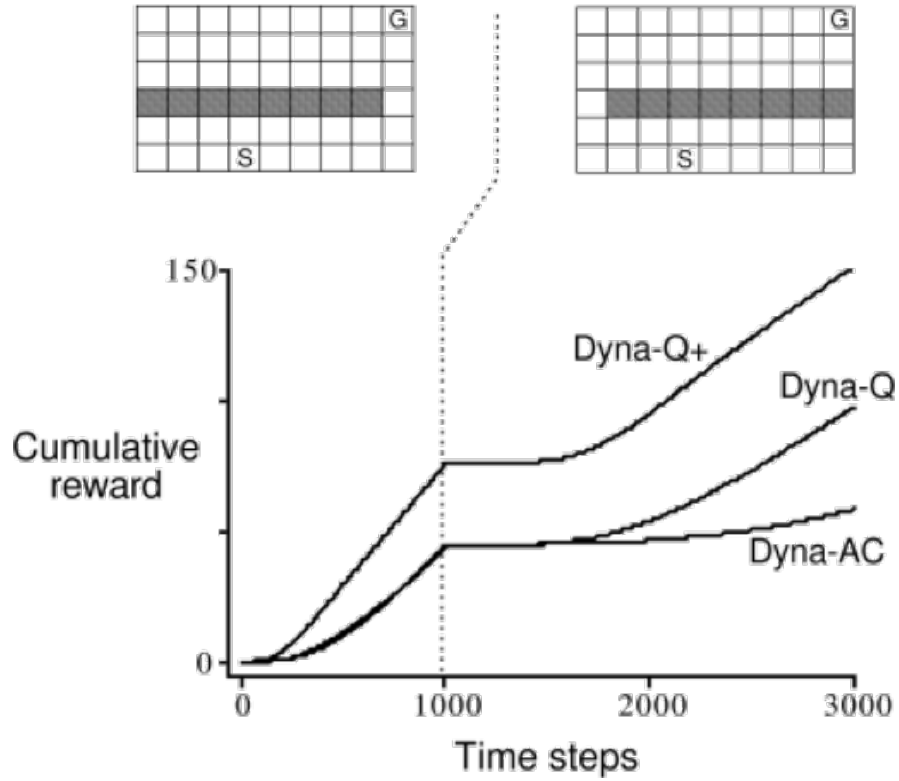




- The policies learned after one episode
  - If all actions have same value, we don't show any arrow
  - Without planning (N = 0), only 1-step positive policy is really learned
  - With increased planning (N = 50), actions are chosen correctly for much more of the state space
  - Simulations keep updating values, finding paths from all visited states
  - Further exploitation of the learned model of the maze
  - Those states not visited yet still need to be learned through exploration

Model Recovery

- What happens if the world changes after learning?
- Dyna-Q adapts to new maze structures
- Here, we see that Dyna-Q+, which uses greater values of ε for its almost-greedy policy, figures things out more quickly, and gets more value overall



Lecture 09-4c:
Generalization in RL

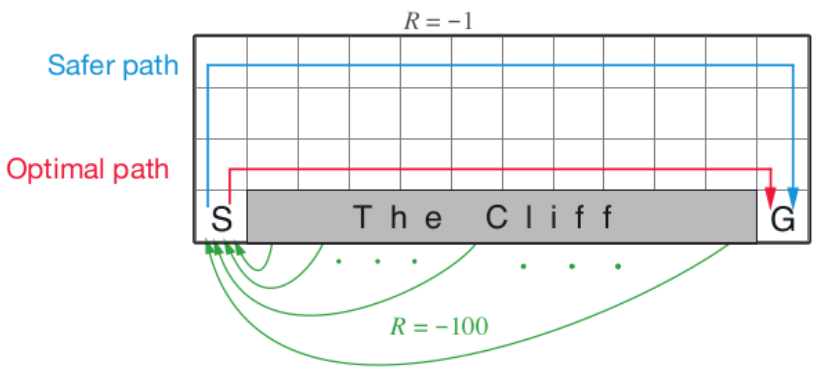
Limitations of Basic Reinforcement Learning

One **limitation** of the basic TD(0) update is that it just updates the value of **one** action at a time:

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right].$$

With eligibility traces and/or planning, some of these limitations can be overcome. However, there is still another issue...

Recall the cliff problem:

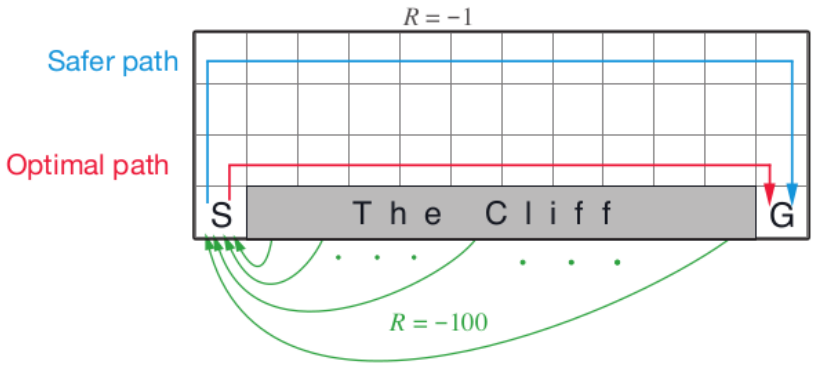


In order to learn to **not jump** off the cliff, **the agent** first needs to **try jumping** off the cliff.

- This lesson needs to be **re-learned in every state** along the cliff!

Lack of Generalization

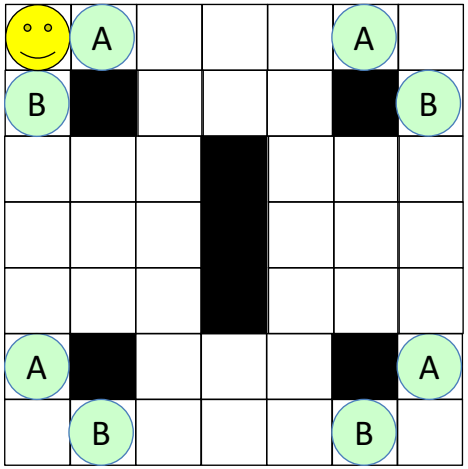
In basic reinforcement learning, **the agent** treats every **state-action** pair as **unique**, and it maintains **separate** action values for each pair.



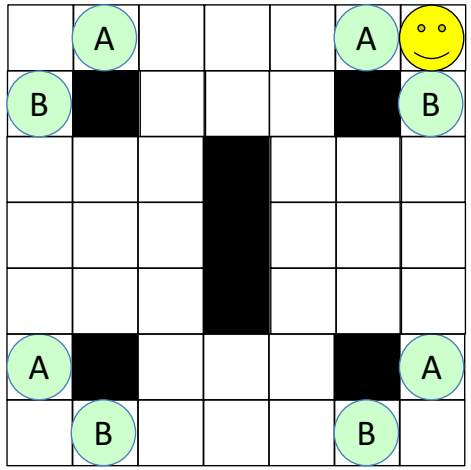
For cliff-adjacent states $s_1 = (2, 2)$ and $s_2 = (3, 2)$, **the basic agent** can learn that $Q(s_1, D)$ is **bad** through trial and error, but it has **no way** to **generalize** from this knowledge and apply it to **estimate** $Q(s_2, D)$!

Lack of Generalization

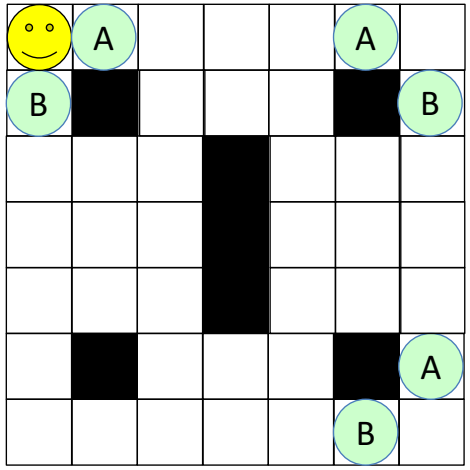
- When doing learning, each state is treated as unique, and must be repeated over and over to learn something about its value



Even if we learn that going right here is best, because type-A objects are better than type-B ones…



But this really tells us nothing about what to do in a ***similar state*** like this one…



Might even say nothing about what to do in a ***different environment*** with some states ***exactly the same!***

States and Features

In order to **generalize** from one state to another, **the agent** needs to be able to identify **commonalities** between states!

This can be done by **characterizing** each state with a set of **features**!

For any state $s \in S$, let $f_j(s)$ denote the **value** that s has for feature j, with $j \in \{1, 2, \ldots, p\}$. Then the **feature vector** for state s is given by

$$\mathbf{f}(s) = (f_1(s), f_2(s), \ldots, f_p(s)) .$$

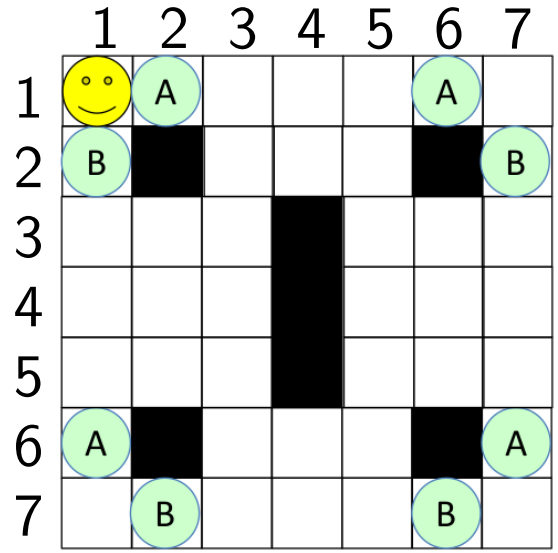
For any two states s_1 and s_2, if $\mathbf{f}(s_1) \approx \mathbf{f}(s_2)$, then **the agent** can try to use knowledge gained in s_1 to inform its actions in s_2.

- This is **good** if s_1 and s_2 are **actually similar** in ways that matter
- This might be **bad** if these similarities are only superficial (i.e., the features do not capture all relevant aspects of these states)

Creating Features

For now, **we** will need to specify **features** for **the agent** to use.

For example, given the grid world to the right, we might include the following features for any state $s \in S$:



$f_X(s) = x$ coordinate of s

$f_Y(s) = y$ coordinate of s

$f_A(s) = L^1$ distance to closest A object from s

$f_B(s) = L^1$ distance to closest B object from s

Then:

- For $s_1 = (1, 1)$, we have $\mathbf{f}(s_1) = (1, 1, 1, 1)$
- For $s_2 = (7, 1)$, we have $\mathbf{f}(s_2) = (7, 1, 1, 1)$
- For $s_3 = (2, 7)$, we have $\mathbf{f}(s_3) = (2, 7, 2, 0)$

Creating Features, Take 2: Normalization

We want to make **one minor adjustment** to these features, however!

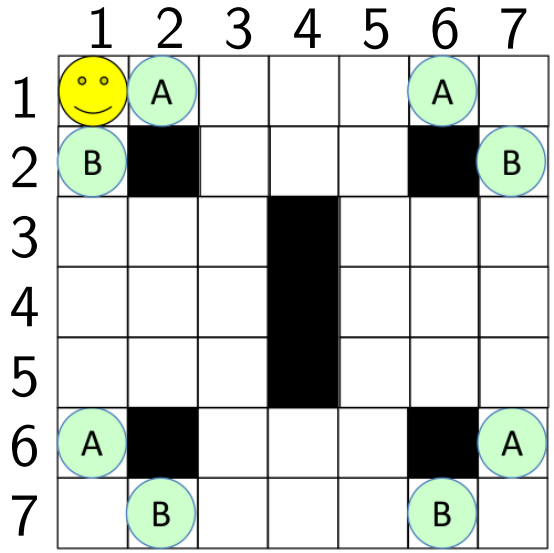
It has been shown that performance is better if features are **normalized**, so that each feature value is in the same range (e.g., $[0, 1]$):

$$f_X(s) = \frac{x \text{ coordinate of } s}{x_{\max}}$$

$$f_Y(s) = \frac{y \text{ coordinate of } s}{y_{\max}}$$

$f_A(s) = \frac{1}{1+d_A}$ where d_A is L^1 dist. to closest A

$f_B(s) = \frac{1}{1+d_B}$ where d_B is L^1 dist. to closest B



Then:

- For $s_1 = (1, 1)$, we have $\mathbf{f}(s_1) = \left(\frac{1}{7}, \frac{1}{7}, \frac{1}{2}, \frac{1}{2}\right)$
- For $s_2 = (7, 1)$, we have $\mathbf{f}(s_2) = \left(\frac{7}{7}, \frac{1}{7}, \frac{1}{2}, \frac{1}{2}\right)$
- For $s_3 = (2, 7)$, we have $\mathbf{f}(s_3) = \left(\frac{2}{7}, \frac{7}{7}, \frac{1}{3}, \frac{1}{1}\right)$

Lecture 09-4d:
Using Features in RL

Using Features of States

Given a set of **features** that **characterize** the states, **the agent** can **use** these features to (try to) **estimate** the **values** of states.

How? The **simplest option** is to assume that the **value** of any state s can be written as a **linear combination** of the feature values of s, i.e.,

$$V(s) \approx \sum_{j=1}^{p} w_j f_j(s).$$

We've seen this idea before, e.g., in linear regression. The above is a **large simplifying assumption**, but it gives us a place to get started.

- If feature j is **important**, then w_j should have a large magnitude
- If feature j is **unimportant**, then w_j should be close to 0

Now we just need **the agent** to **learn the appropriate weights!**

Using Features of Actions (in States)

One issue that we saw earlier was that knowing $V(s)$ didn't provide **the agent** with an immediate way to improve its **policy** because it did not have a model for the (likely) effects of its actions.

To remedy this, we had **the agent** learn the **action value function** $Q(s, a)$ instead of $V(s)$. We can do the same thing with **features**, except that features will be defined for **state-action pairs** instead of **states**.

- For any $(s, a) \in S \times A$, let $f_j(s, a)$ denote the **value** that (s, a) has for feature j, with $j \in \{1, 2, \ldots, p\}$
- The **feature vector** for (s, a) is $\mathbf{f}(s, a) = \big(f_1(s, a), \ldots, f_p(s, a)\big)$
- **The agent** estimates $Q(s, a)$ as

$$Q(s, a) \approx \sum_{j=1}^{p} w_j f_j(s, a).$$

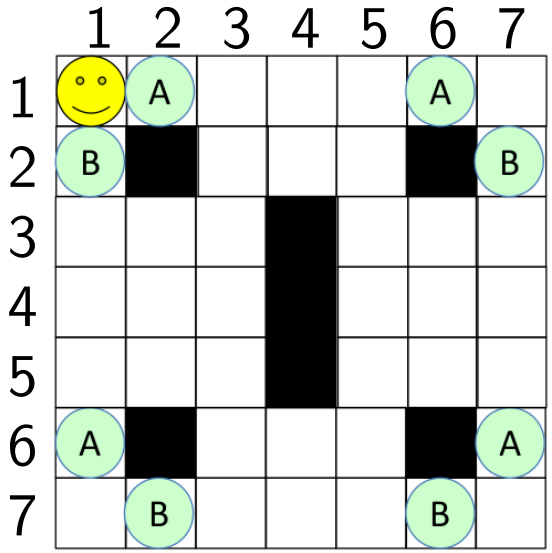
Creating Features for Actions (in States)

One way to create features for a state-action pair (s, a) is to use features of the (likely) next state s' (assuming **we** know what s' might be).

For any $(s, a) \in S \times A$, let $s' \in S$ be the expected next state after executing action a in state s. Then some **action features** might be

$$f_X(s, a) = f_X(s'), \qquad f_Y(s, a) = f_Y(s'),$$
$$f_A(s, a) = f_A(s'), \qquad f_B(s, a) = f_B(s'),$$

where the $f_X(s')$, $f_Y(s')$, etc. are the **state features** that we defined earlier.



Then:

- For $s_1 = (1, 1)$, we have $\mathbf{f}(s_1, R) = \left(\frac{2}{7}, \frac{1}{7}, \frac{1}{1}, \frac{1}{3}\right) = \mathbf{f}(s'_1)$
- For $s_2 = (7, 1)$, we have $\mathbf{f}(s_2, L) = \left(\frac{6}{7}, \frac{1}{7}, \frac{1}{1}, \frac{1}{3}\right) = \mathbf{f}(s'_2)$

Lecture 09-4e:
Learning Weights

Learning Weights: Initialization

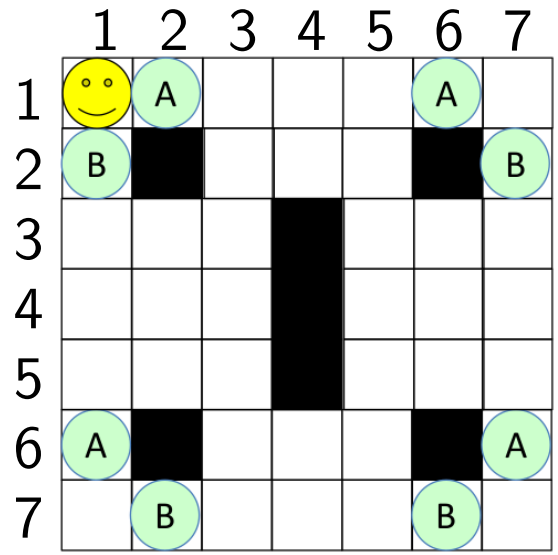
Given features **characterizing** **state-action** pairs, **the agent** needs to **learn the appropriate weights** for each feature to **estimate** Q.

Initially, **the agent** might start with default weights, e.g., $w_X = w_Y = w_A = w_B = 1$.

Then for $s_1 = (1, 1)$ and $s_2 = (7, 1)$:

$$\begin{aligned} Q(s_1, R) &= w_X \cdot f_X(s_1, R) + w_Y \cdot f_Y(s_1, R) \\ &\quad + w_A \cdot f_A(s_1, R) + w_B \cdot f_B(s_1, R) \\ &= 1 \cdot \tfrac{2}{7} + 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{1}{1} + 1 \cdot \tfrac{1}{3} = \tfrac{37}{21} \approx 1.76 \\ Q(s_1, D) &= 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{2}{7} + 1 \cdot \tfrac{1}{3} + 1 \cdot \tfrac{1}{1} = \tfrac{37}{21} \approx 1.76 \\ Q(s_2, L) &= 1 \cdot \tfrac{6}{7} + 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{1}{1} + 1 \cdot \tfrac{1}{3} = \tfrac{49}{21} \approx 2.33 \end{aligned}$$

For default weights, many actions may have the same value initially; **the agent** needs to adjust the weights to identify **good** and **bad** actions.



$$\mathbf{f}(s_1, R) = \left(\tfrac{2}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$$

$$\mathbf{f}(s_1, D) = \left(\tfrac{1}{7}, \tfrac{2}{7}, \tfrac{1}{3}, \tfrac{1}{1}\right)$$

$$\mathbf{f}(s_2, L) = \left(\tfrac{6}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$$

Learning Weights: Weight Updates

In normal **Q-Learning**, **the agent** updates its value for a **state-action pair** (s, a) based on the next state s' and one-step reward r:

$$\delta = r + \gamma \max_{a'' \in A} Q(s', a'') - Q(s, a)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta$$

(Recall that δ is the **TD error**; **SARSA** computes δ using $Q(s', a')$.)

To update **weights** for features, **the agent** adjusts each weight w_j based on the **TD error** and the **value** of feature j for (s, a):

$$\delta = r + \gamma \max_{a'' \in A} Q(s', a'') - Q(s, a)$$

$$w_j \leftarrow w_j + \alpha\delta f_j(s, a) \quad \forall j \in \{1, 2, \ldots, p\}$$

Learning Weights: Update Example

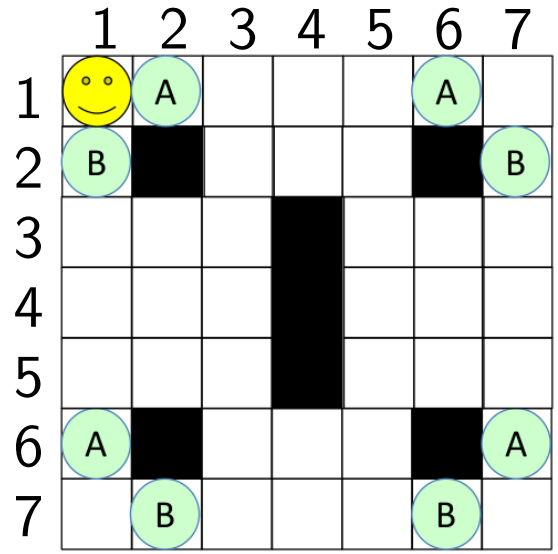
Example: Suppose **the agent** starts with default weights of 1 for all features and executes the action R in state $s_1 = (1, 1)$.

Assume that the **TD error** observed by **the agent** is

$$\delta = r + \gamma \max_{a'' \in A} Q(s', a'') - Q(s, a) = 10.$$

With $\alpha = 0.9$, the **the agent** updates the weights as:

$$\begin{aligned}
w_X &\leftarrow w_X + \alpha\delta f_X(s_1, R) \\
&= 1 + 0.9 \cdot 10 \cdot \tfrac{2}{7} && = \tfrac{25}{7} \approx 3.57 \\
w_Y &\leftarrow w_Y + \alpha\delta f_Y(s_1, R) \\
&= 1 + 0.9 \cdot 10 \cdot \tfrac{1}{7} && = \tfrac{16}{7} \approx 2.29 \\
w_A &\leftarrow w_A + \alpha\delta f_A(s_1, R) \\
&= 1 + 0.9 \cdot 10 \cdot \tfrac{1}{1} && = 10 \\
w_B &\leftarrow w_B + \alpha\delta f_B(s_1, R) \\
&= 1 + 0.9 \cdot 10 \cdot \tfrac{1}{3} && = 4
\end{aligned}$$



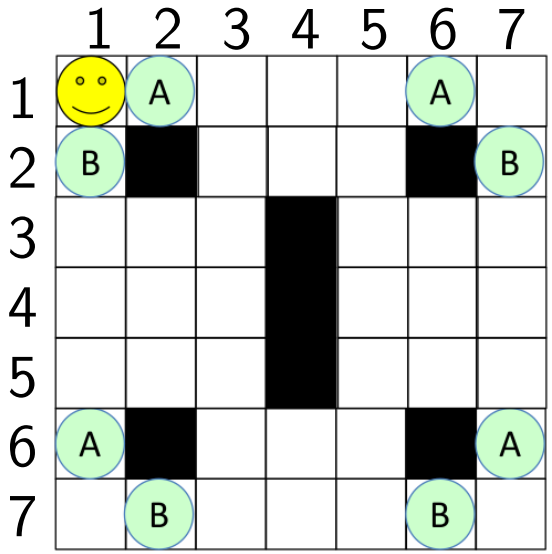
$$\mathbf{f}(s_1, R) = \left(\tfrac{2}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$$

Learning Weights: Effects of Updates

Example (continued): Let's look at what happens to **the agent's** value of $Q(s_1, R)$ after this update:

Recall that with **default weights**, we had:

$$\begin{aligned} Q(s_1, R) &= w_X \cdot f_X(s_1, R) + w_Y \cdot f_Y(s_1, R) \\ &\quad + w_A \cdot f_A(s_1, R) + w_B \cdot f_B(s_1, R) \\ &= 1 \cdot \tfrac{2}{7} + 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{1}{1} + 1 \cdot \tfrac{1}{3} = \tfrac{37}{21} \approx 1.76. \end{aligned}$$



$\mathbf{f}(s_1, R) = \left(\tfrac{2}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$

With the **updated weights** $w_X = \frac{25}{7}$, $w_Y = \frac{16}{7}$, $w_A = 10$, and $w_B = 4$, we now have:

$$\begin{aligned} Q(s_1, R) &= w_X \cdot f_X(s_1, R) + w_Y \cdot f_Y(s_1, R) \\ &\quad + w_A \cdot f_A(s_1, R) + w_B \cdot f_B(s_1, R) \\ &= \tfrac{25}{7} \cdot \tfrac{2}{7} + \tfrac{16}{7} \cdot \tfrac{1}{7} + 10 \cdot \tfrac{1}{1} + 4 \cdot \tfrac{1}{3} \\ &= \tfrac{1864}{147} \approx 12.68. \end{aligned}$$

Learning Weights: Effects of Updates, continued

A TD update in normal Q-Learning changes the **value** of **one** (s, a) **pair**, but a **weight update** can change the **values** of **all state-action pairs**!

Recall that with **default weights**, we had:

$$Q(s_1, R) = 1 \cdot \tfrac{2}{7} + 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{1}{1} + 1 \cdot \tfrac{1}{3} \approx 1.76$$

$$Q(s_1, D) = 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{2}{7} + 1 \cdot \tfrac{1}{3} + 1 \cdot \tfrac{1}{1} \approx 1.76$$

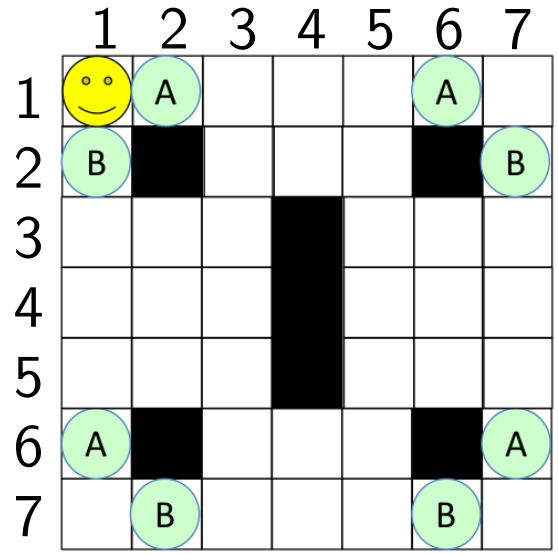
$$Q(s_2, L) = 1 \cdot \tfrac{6}{7} + 1 \cdot \tfrac{1}{7} + 1 \cdot \tfrac{1}{1} + 1 \cdot \tfrac{1}{3} \approx 2.33$$

With the **updated weights**, we now have:

$$Q(s_1, R) = \tfrac{25}{7} \cdot \tfrac{2}{7} + \tfrac{16}{7} \cdot \tfrac{1}{7} + 10 \cdot \tfrac{1}{1} + 4 \cdot \tfrac{1}{3} \approx 12.68$$

$$Q(s_1, D) = \tfrac{25}{7} \cdot \tfrac{1}{7} + \tfrac{16}{7} \cdot \tfrac{2}{7} + 10 \cdot \tfrac{1}{3} + 4 \cdot \tfrac{1}{1} \approx 8.50$$

$$Q(s_2, L) = \tfrac{25}{7} \cdot \tfrac{6}{7} + \tfrac{16}{7} \cdot \tfrac{1}{7} + 10 \cdot \tfrac{1}{1} + 4 \cdot \tfrac{1}{3} \approx 17.39$$



$$\mathbf{f}(s_1, R) = \left(\tfrac{2}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$$

$$\mathbf{f}(s_1, D) = \left(\tfrac{1}{7}, \tfrac{2}{7}, \tfrac{1}{3}, \tfrac{1}{1}\right)$$

$$\mathbf{f}(s_2, L) = \left(\tfrac{6}{7}, \tfrac{1}{7}, \tfrac{1}{1}, \tfrac{1}{3}\right)$$

Learning Weights: Summary

As **the agent** adjusts the weights after each step, it **learns** connections between **feature values** and **rewards** (both one-step and longer-term) and how these features can be used to estimate action values.

- Each **weight update** potentially changes the action values of **all** state-action pairs, which can speed up the overall learning process
- Over time, features that are **important** will have their weights increased (in magnitude), while features that are **unimportant** will have their weights pushed towards 0

In order for this process to **work well**, **the agent** needs to be given **features** that capture **relevant aspects** of the environment.

- With **uninformative** features, **the agent** might make spurious connections between features and action values that **do not generalize well**

Lecture 09-4Xa:
(Optional) Finding Features

Linear Functions over Features

- Suppose we have a set of features that describe states, or state-action pairs, of an MDP
- What we want is to learn the set of weights needed to properly calculate our *U*- or *Q*-values

$$U(s) = w_1\, f_1(s) + w_2\, f_2(s) + \cdots + w_n\, f_n(s)$$

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

- We learn by adjusting weights, e.g., in a *Q*-learning way:

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

$$\forall i,\ w_i \leftarrow w_i + \alpha\, \delta\, f_i(s,a)$$

Where Do Features Come From?

- A variety of approaches can be used to generate useful features for a given learning problem
  - Sometimes we have good intuitions about what features are useful, and sometimes we don't
- For example, suppose we have a problem in which states are characterized by (x, y) points in the plane:
  1. We might just use the features (x, y) themselves, with 2 matching weights…
  2. Or a simple polynomial combination, like $(1, x, y, xy)$, with 4 weights to go along with those…
  3. Or a more complex polynomial, like:

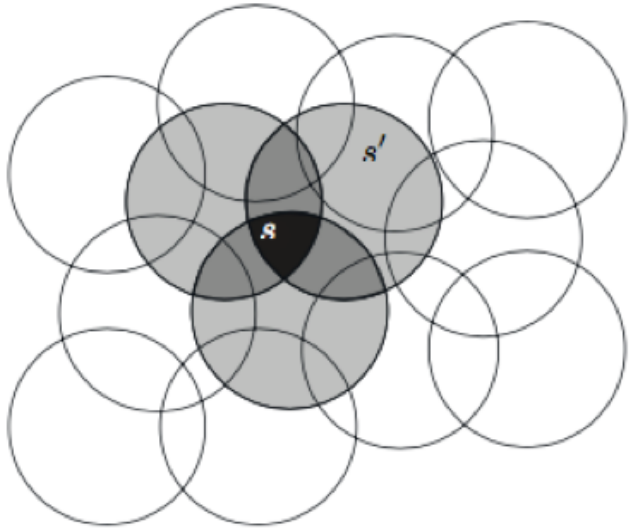
$$(1, x, y, xy, x^2, y^2, xy^2, x^2y, x^2y^2)$$

Various Feature Functions

- Experimentation has been performed with many different functional forms for features:
  1. Polynomial features: linear weights over polynomial combinations of numeric features
  2. Fourier features: combinations of sine and cosine functions over the underlying numbers
  3. Radial basis functions: real-valued features based on computed distances from chosen points in the state-space
- A common issue is the complexity growth of the features as the dimensionality of the state space increases
  - A variety of techniques exist to use sparser, binary features

Lecture 09-4Xb: (Optional) Finding Features: Geometric Approaches

Coarse Coding



- Suppose we have a state-space consisting of points in the plane
- A binary coding scheme is to use features that each correspond to circles in the state-space
  - A point has a given feature if it lies inside the circle
  - Two points share all the features that overlap them both

Image: Sutton & Barto, 2018

Advantages of Binary Features

- Given a coarse coding scheme (set of n circles), we get a feature-vector, and corresponding weight-vector for state s:

$$\mathbf{x}_s = (c_1, c_2, \ldots, c_n)$$

$$\mathbf{w}_s = (w_1, w_2, \ldots, w_n)$$

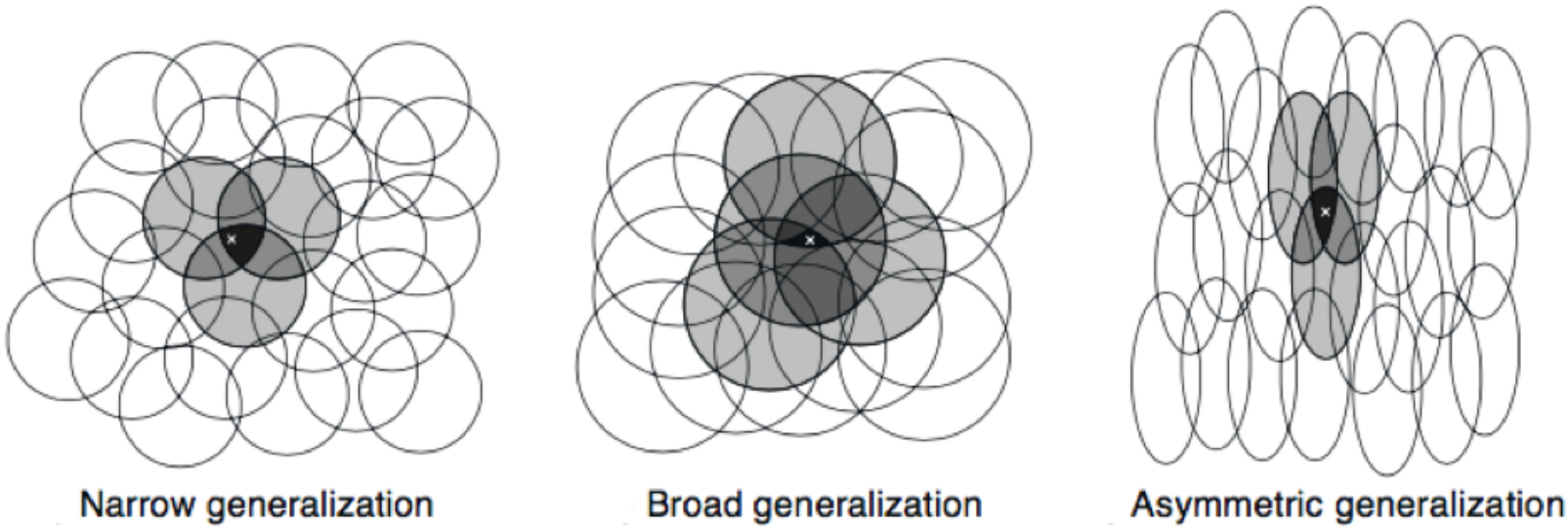
- Each feature is a binary value:

$$c_i = \begin{cases} 1 & \text{if } s \text{ lies inside circle } i \\ 0 & \text{else} \end{cases}$$

- If we use these features to compute a utility-value for s, rather than a series of multiplications and additions, we get the simpler summation:

$$U(s) = \mathbf{w}_s \cdot \mathbf{x}_s = \sum_i \{w_i \,|\, c_i = 1\}$$

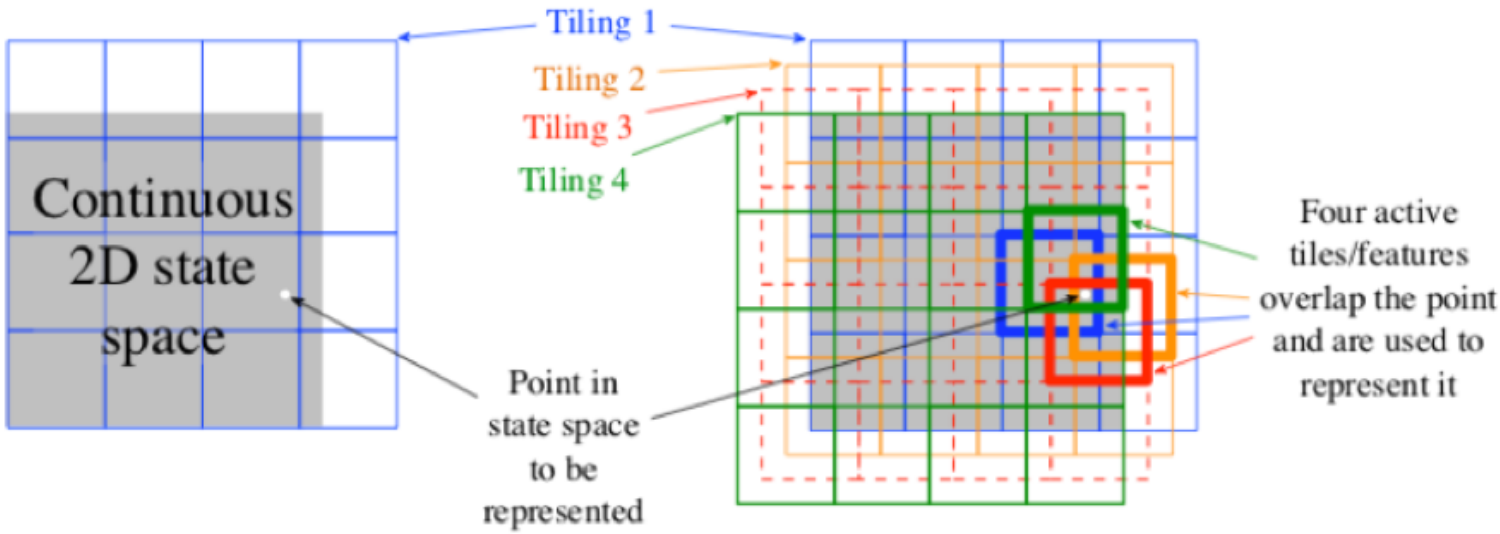
Feature Variation



- A simple idea, circular coarse coding can still generalize over domains in a variety of ways
  - We get generalization, since the ***same*** weight vector is used for every state (only the ***particular features*** change)

Image: Sutton & Barto, 2018

Tile Coding



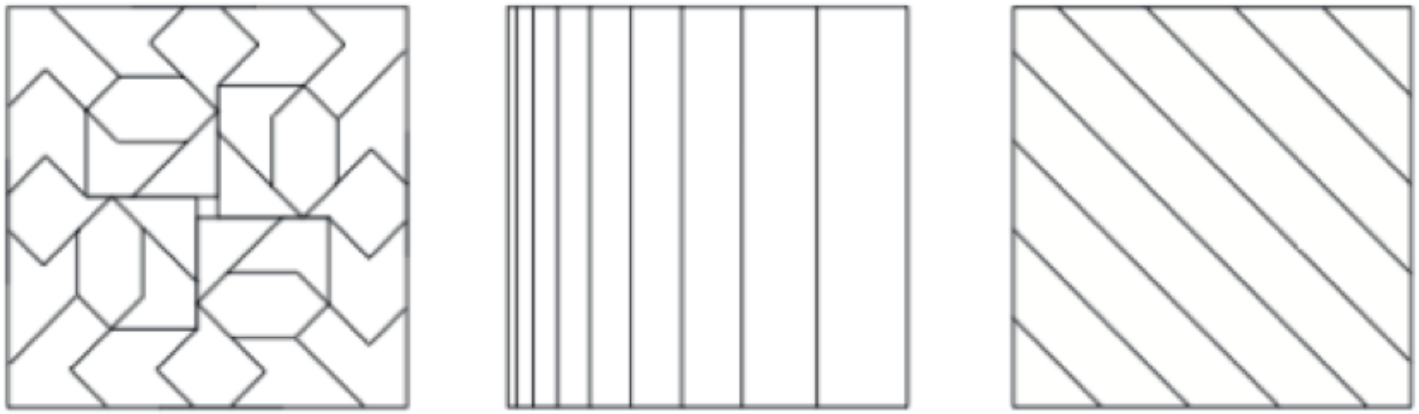
- Another form of coarse coding is to tile the state space
  - A single, simple tiling is just a partition, dividing the state space into uniform regions
  - A more complex tiling uses ***multiple overlapping*** partitions
  - Again, each individual tile corresponds to a binary feature

Image: Sutton & Barto, 2018

Placement of Tiles

- Much research has gone into good ways to choose how many tilings to use, and how they should overlap
  - It has been found that choosing ***uniform offsets*** of the tiles (e.g. moving each new tiling over by 1 unit in each direction) can introduce certain numerical biases
- Best practices, as recommended by Miller and Glanz (96):
  1. For a state space of dimensionality k, use $2^n \geq 4k$ tilings
  2. Offset each dimension using numbers $(1, 3, 5, \ldots, 2k-1)$
- For example, in 2 dimensions, use 8 or more tilings, offsetting each by 1 unit in the x dimension, and 3 in y
- In 3 dimensions, use at least 12 tilings, offsetting by 1 unit in x, 3 in y, 5 in z …

Other Approaches to Tiling



- Any number of tiling patterns and offsets can be used
  - Tilings need not be regular in shape or size
  - "Striped" tilings generalize along only certain dimensions
  - Different sizes of tiles allow finer/coarser discrimination in certain parts of the state space

Image: Sutton & Barto, 2018

Lecture 09-4Xc:
(Optional) Kanerva Coding

Sparse Distributed Memory

- Developed by NASA Ames researcher Pentti Kanerva while working on a model of human long-term memory
- In RL, often now called Kanerva coding
- A model that generalizes over states based on a similarity measure
- State-features are represented again as binary vectors, which can be regarded as lists of ***other*** states to which they are or are not similar

Basic Kanerva Coding for Q-Learning

- Rather than save Q-values for all state-action pairs, a Kanerva coding selects a subset of prototypes:

$$P = \{p_1, p_2, ..., p_n\} \subsetneq S$$

- For any state s and prototype p_i, we say the two are adjacent if s and p_i differ by at most 1 feature (other such similarity measures are possible)

- For example, if we have two state feature-variables:

$$f_1 \in \{1, 2\} \qquad f_2 \in \{a, b, c\}$$

- Then the state $s = (1, a)$ would be:
  1. Adjacent to prototypes $p_i = (2, a), (1, b)$, or $(1, c)$
  2. ***Not*** adjacent to $p_j = (2, b)$ or $(2, c)$

From Prototypes to Binary Features

- For our vector of prototypes we then get a vector of binary features for every state:

$$P = \{p_1, p_2, ..., p_n\}$$

$$\mathbf{x}_s = (f(s)_1, f(s)_2, ..., f(s)_n)$$

$$f(s)_i = \begin{cases} 1 & \text{if } s \text{ is adjacent to } p_i \\ 0 & \text{else} \end{cases}$$

- Our *Q*-learning algorithm then learns weight values over prototypes only:

$$\theta(p_i, a), \forall p_i \in P, \forall a \in A$$

Adjusting Prototype Weights

- For any state-action pair, we can now compute an approximate Q-value, based only on adjacent prototypes:

$$\hat{Q}(s,a) = \sum_i \theta(p_i, a)\, f(s)_i$$

- Furthermore, when our learning algorithm takes action a in state s_1, receives reward r, and ends up in state s_2 we update:

$$\theta(p_i, a) \leftarrow \theta(p_i, a) + f(s)_i\, \alpha(r + \gamma \max_{a_2} \hat{Q}(s_2, a_2))$$

- Doing it this way also means that we only update those weights on pairs featuring prototypes that are adjacent to s, since otherwise $f(s)_i = 0$

Lecture 09-4Xd:
(Optional) Adaptive Kanerva Coding

Choosing Prototypes

- In the limit, we could use ***all*** states as prototypes ($P = S$), and impose strict identity on the measure of adjacency:

$$f(s)_i = \begin{cases} 1 & \text{if } s = p_i \\ 0 & \text{else} \end{cases}$$

  - In this case, the algorithm is just normal *Q*-learning, without any generalization at all

- If we make the set of prototypes very small, on the other hand, then most states will not be adjacent to any prototype, and we won't learn ***anything*** about those states at all

Choosing Prototypes

- In between the extremes, the usefulness of Kanerva encoding is maximized when:
  1. No prototype is visited too much (such a state is effectively ***too abstract***)
  2. No prototype is visited too little (such a state is effectively ***too specific***)
- Achieving this balance with an initial set or prototypes, which is often chosen randomly, or according to some heuristic, is challenging, leading to interest in adaptive Kanerva Coding, where we ***change*** the prototype set over time as we learn

Adaptive Kanerva Coding

- We can modify the basic approach as follows:

1. We start with a set of randomly chosen prototypes
2. For each prototype, we keep track of how many times we visit a state that is adjacent to it
3. Periodically, we modify the prototype set in two ways:
   a. Deletion: if a prototype has been visited t times, we decide whether or not to delete it randomly, with probability:

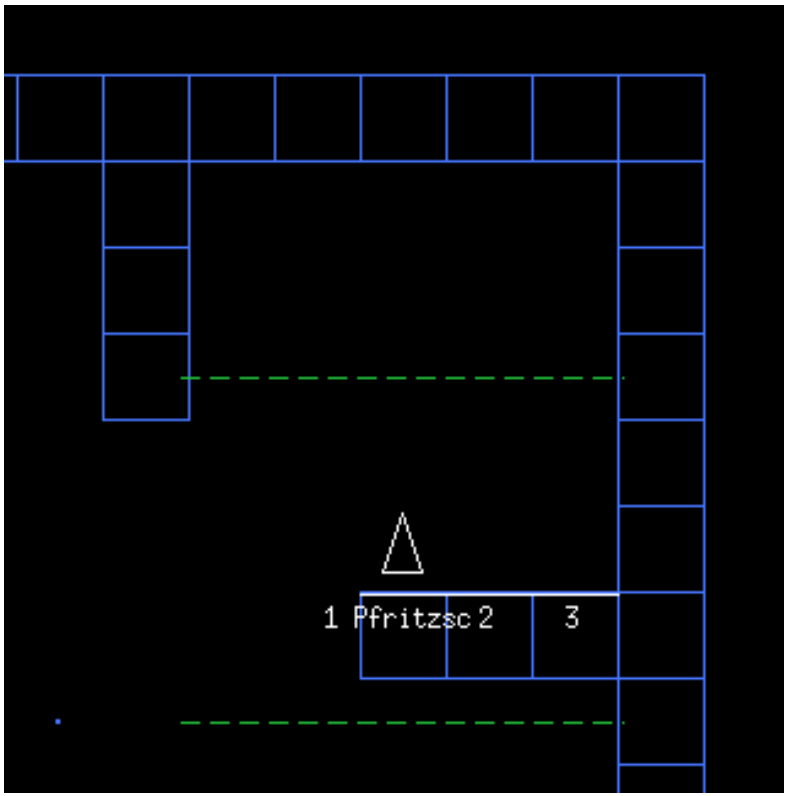
$$P_{del} = e^{-t}$$

   b. Splitting: whenever some prototypes are deleted, they are replaced by taking the **most-visited** prototypes and generating new, adjacent ones by changing one of their feature-values

Lecture 09-4Xe:
(Optional) Kanerva Coding Application

An Application: Xpilot Navigation

- Xpilot is a space-shooter game with the ability to add complex physics and environments
- Basic Q-learning is thwarted by the enormous state-space of the full game (even when only trying to learn to navigate successfully without crashing)



- Even on a small map of size (500 x 500), a ship with 10 possible speeds and full rotation will correspond to approximately 3.24×10^{10} states!

An Application: Xpilot Navigation

TABLE I
STATE VARIABLES FOR XPILOT, WITH RANGES. THE STATE SPACE IS MADE UP OF 5 VARIABLES AND 31104 STATES. THESE COMBINE WITH 4 ACTIONS FOR 124416 STATE-ACTION PAIRS (s, a). 1024 STATES ARE CHOSEN AS PROTOTYPES, FOR 4096 PROTOTYPE-ACTION PAIRS (p, a).

State Variable	Range
Heading	1–36
Tracking	1–36
Speed	1–6
Near Wall	$\{1, 0\}$
Near Corner	$\{1, 0\}$

TABLE II
POSSIBLE AGENT ACTIONS

Action	Effect
Avoid Wall	Turn 10°: away from nearest wall; thrust once
Avoid Corner	Turn 10°: direction 180° opposite **Tracking**; thrust once
Thrust	If speed $s < 6$, thrust once
Do Nothing	`null`

TABLE III
THE REWARD-STRUCTURE.

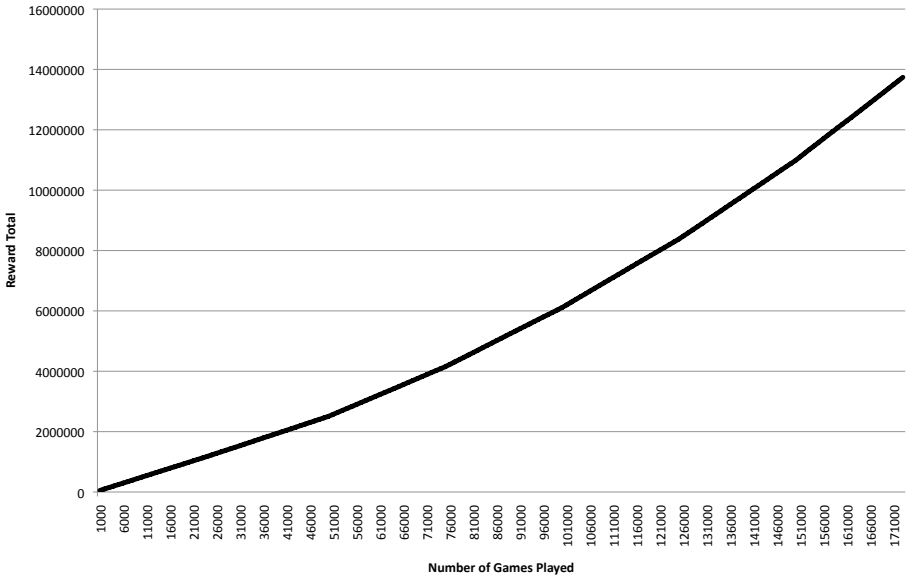
Reward Value	Condition
-10	Agent crashes
$+1$	Agent is alive for one frame

Even with a simplified and discretized state-action space, things are still far too complex for effective use of basic RL algorithms

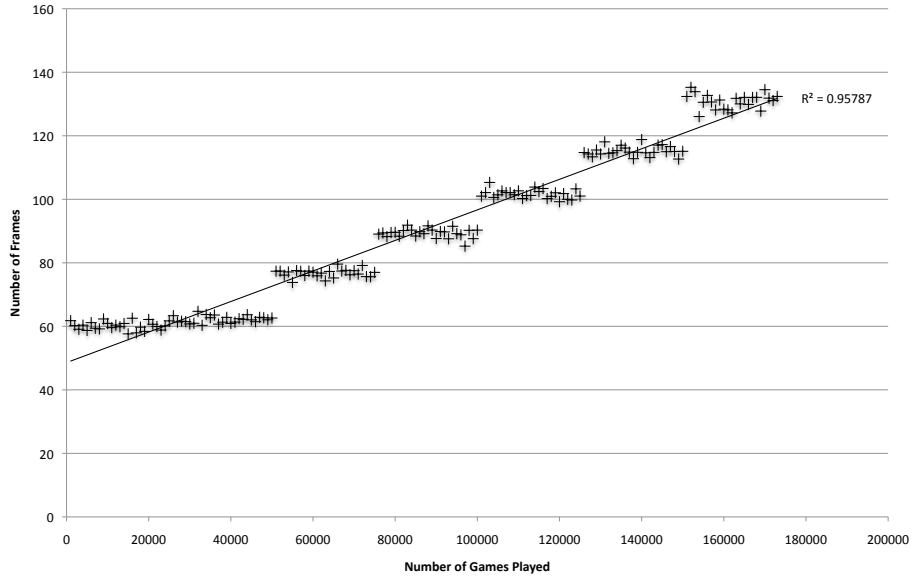
Other Parameters

1. # of prototypes: $1{,}024$ initial prototypes (at random)
   - A total of $4{,}096$ (prototype, action) pairs
   - ~3.3% of overall possible (state, action) pairs
2. Learning updates: $\gamma = 0.9$, $\alpha = 0.1$
3. Policy randomness: initially $\varepsilon = 0.9$
   - Every $25{,}000$ games we update: $\varepsilon = \dfrac{0.9}{\lfloor \texttt{totalGames}/25{,}000 \rfloor + 1}$
4. Updating prototypes: every time any prototype is visited (by encountering an adjacent state) 50 times, we:
   a. Delete m of them using randomness based upon number of visits
   b. Split each of the most-visited m prototypes, returning to $1{,}024$
   c. Re-set all counts of how many times each is visited

Learning Performance



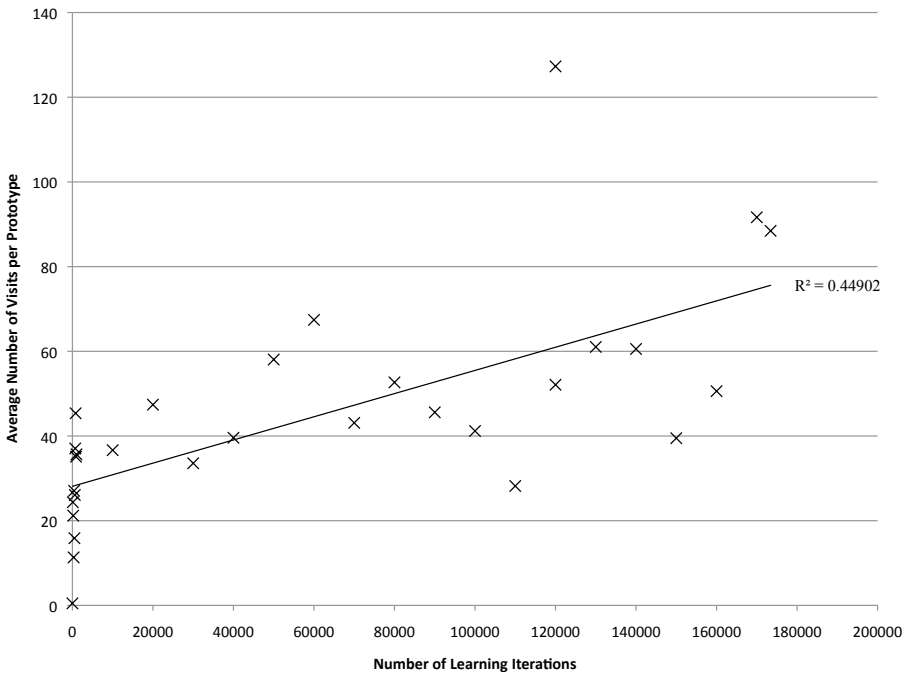
(a) Total Reward Over Time



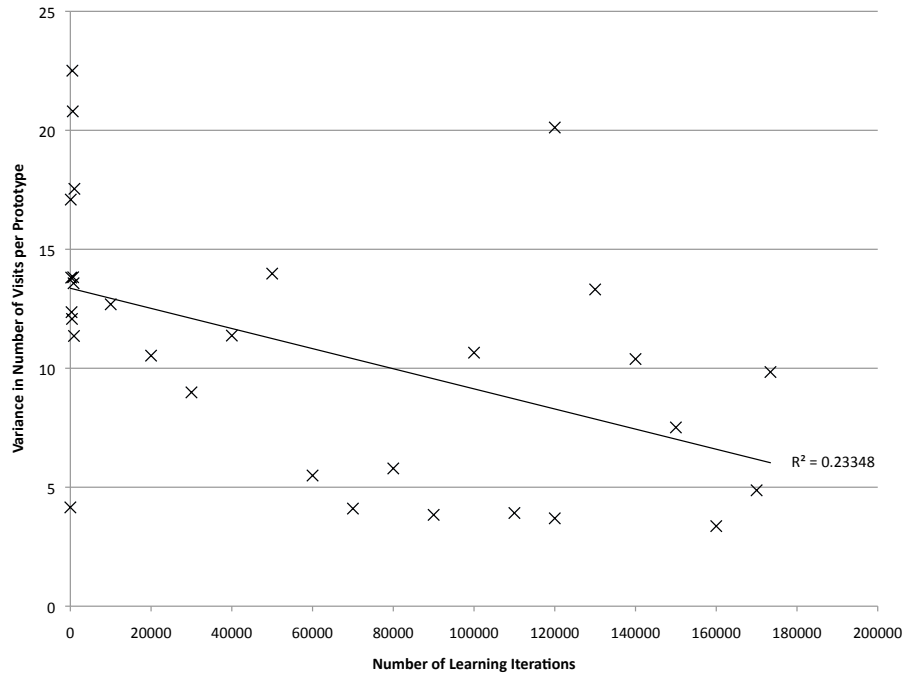
(b) Average Frames Alive Over Time

- The agent continues to improve performance over many hours of learning, comprising over 170,000 episodes
  - Note: as they get better, each episode of learning gets longer as they survive more frames

Prototype Visits



(a) Average Visits per Prototype Over Time



(b) *Variance*: Average Visits per Prototype Over Time

- As rarely visited prototypes are replaced by more useful ones, overall rate of visits increases
  - At same time, variance of visits to any prototype decreases

Adding More Prototypes

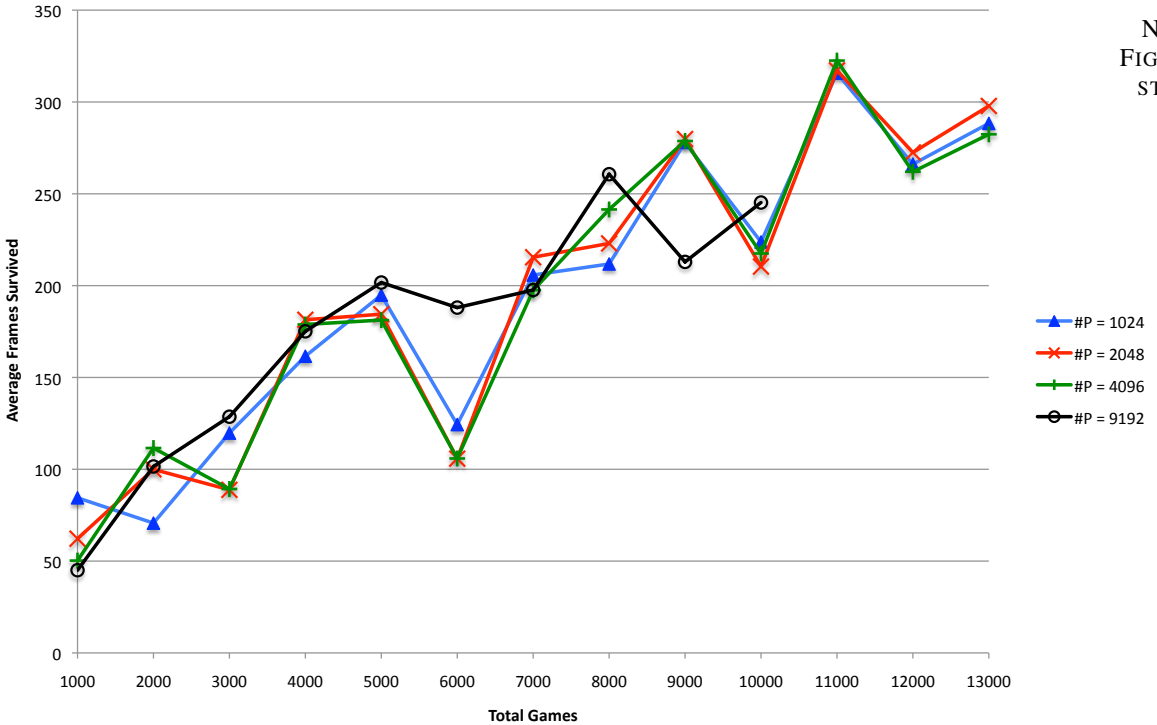


TABLE IV
NUMBERS OF PROTOTYPES USED IN THE EXPERIMENTS DESCRIBED IN FIGURE 4. EACH COMBINES WITH 4 ACTIONS FOR THE GIVEN NUMBER OF STATE-ACTION PAIRS (s, a). ALSO SHOWN IS THE PERCENTAGE OF THE ENTIRE STATE-ACTION SPACE (124416 PAIRS) REPRESENTED.

Prototypes	(s, a)	% Total
1024	4096	3.3%
2048	8192	6.6%
4096	16384	13.2%
9192	36768	29.6%

- Using 2/4/8 times as many prototypes has little to no effect, meaning that the smaller number is as good as any other
- For the largest size of prototype set, learning was much slower, and experiments had to be curtailed somewhat

Lecture 09-5a:
Deep Reinforcement Learning

Review: Q-Learning with Function Approximation

- Q-learning: we evaluate a state-action pair (s,a) based on the results we get (r and s') and update the ***single pair value***:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \, \delta$$

- The Q-values we learn can simply be stored in a data-structure of our choice, for look-up whenever we need them

Review: Q-Learning with Function Approximation

- If we decide to do function approximation instead, we can assume that the value of any given state-action pair is a linear function of certain features of the state, just as when we looked at various different classification algorithms:

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

- Learning updates these weights, instead of Q-values directly

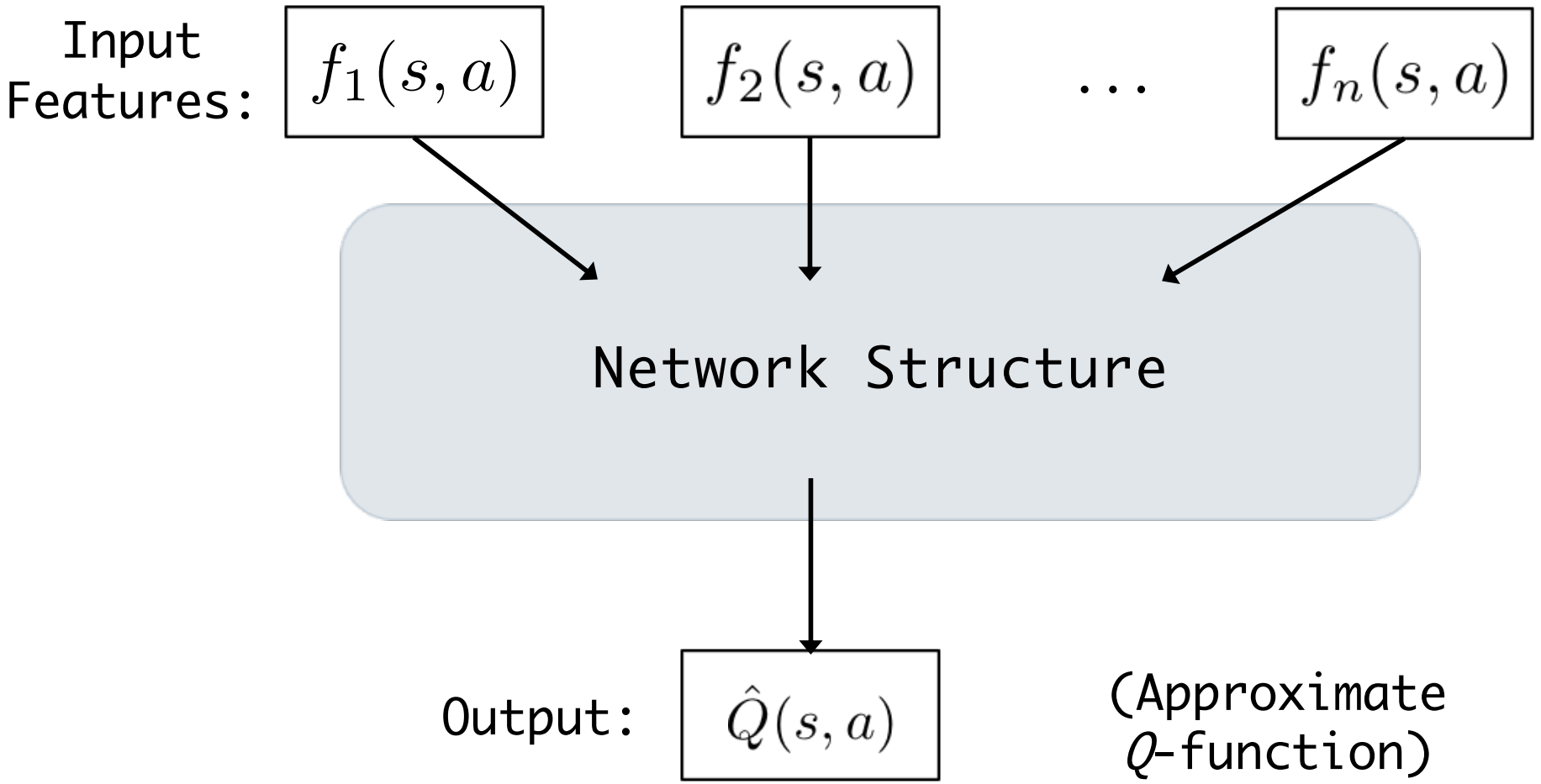
$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

$$\forall i,\ w_i \leftarrow w_i + \alpha\, \delta\, f_i(s,a)$$

- Why limit ourselves to ***linear*** functions of features?

Combining Neural Nets and Reinforcement Learning

- Basic idea: use a NN as our function approximator for Q



Problems with the Basic Idea

- The use of NNs (and other non-linear function approximators) in RL has been considered at length
- It faces some challenges, including:

1. The temporal nature of an MDP process: since actions and states proceed in a (cause/effect) sequence, there are significant correlations between different inputs seen, which is a problem for reliable NN training
2. The relationship between values and policies: since changes in *Q*-value affect what actions are chosen, the input distribution can change every time we update

- As a result, *Q*-learning may no longer converge to stable values over time, and may in fact be impossible
  - For more information see: Tsitsiklis & Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control* 42 (1997).

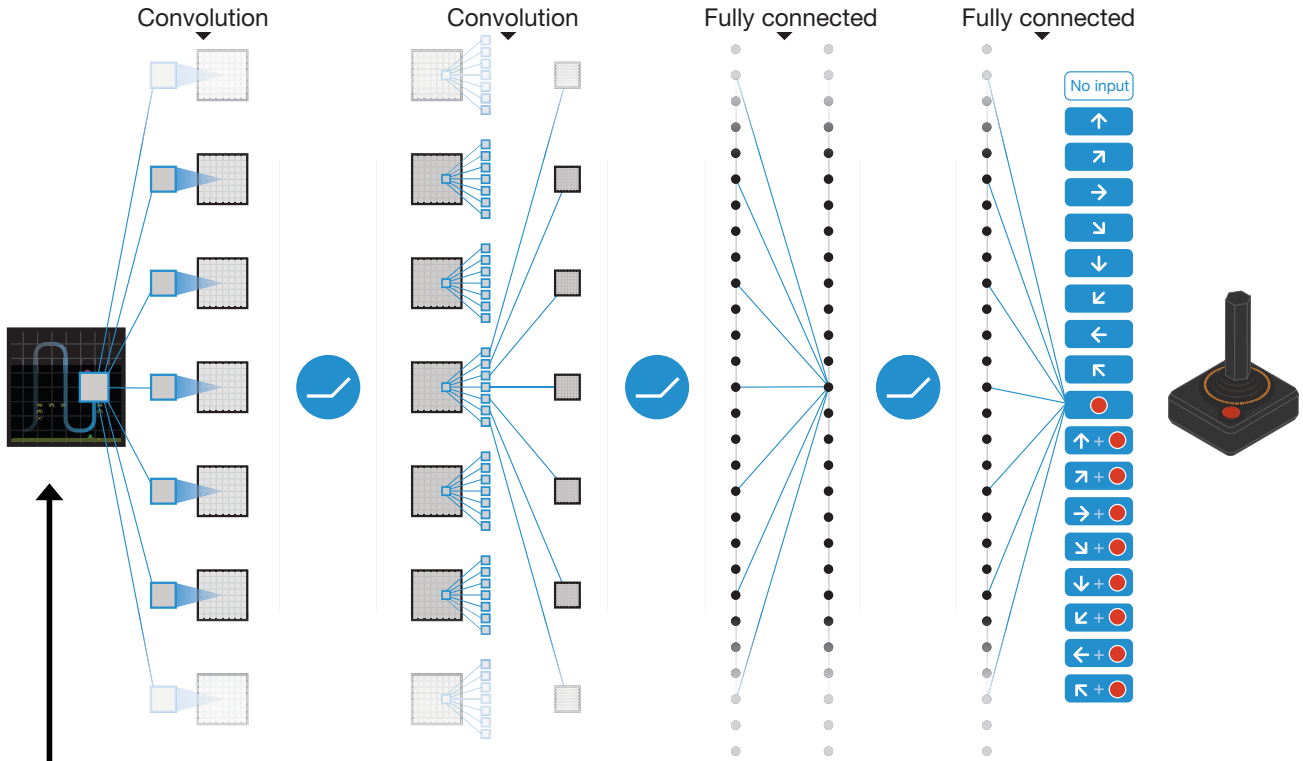
Lecture 09-5b:
Deep Q-Networks

Deep Q-Networks (DQN)

- Mnih, et al., "Human-level control through deep reinforcement learning," *Nature* 518 (2015)
- A recent breakthrough in combining neural nets and reinforcement learning, achieving strong results for a test-bed of Atari 2600 games (played via emulator)

A CNN for Atari Control

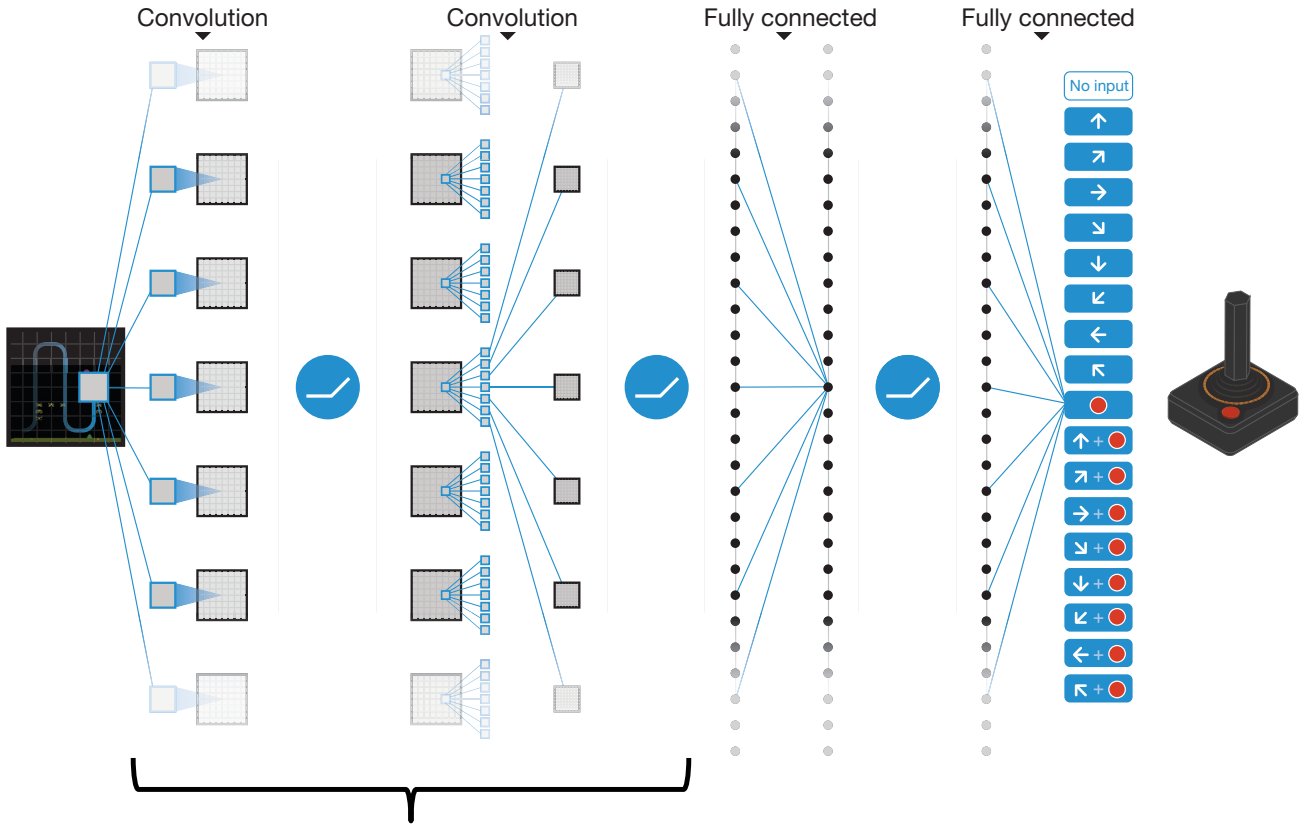
Mnih, et al. (2015)



Inputs: (84 x 84 x 4)
4 frames of graphics from a game,
scaled down to (84 x 84) pixels

A CNN for Atari Control

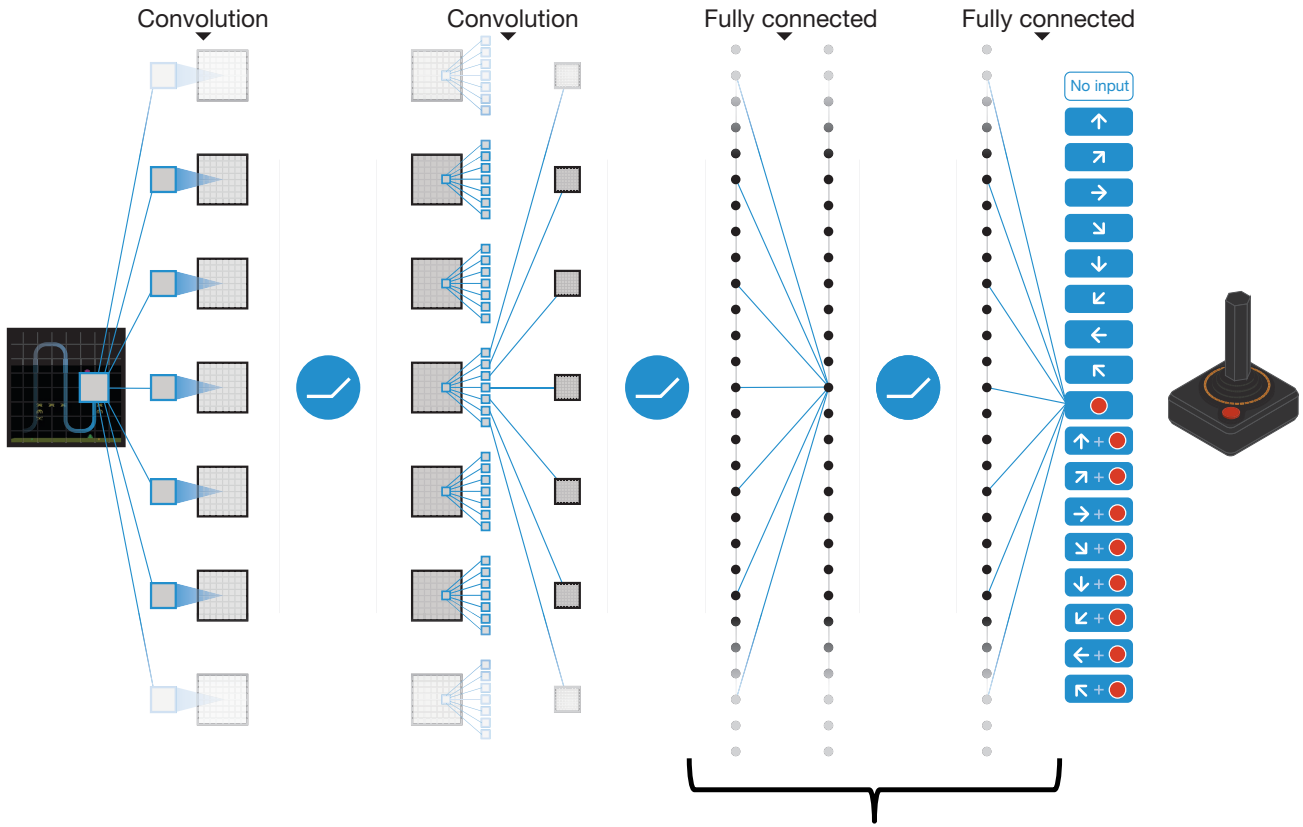
Mnih, et al. (2015)



3 Convolution Layers, with RELU in between each:
[1] (32 x 8 x 8), stride = 4
[2] (64 x 4 x 4), stride = 2
[3] (64 x 3 x 3), stride = 1

A CNN for Atari Control

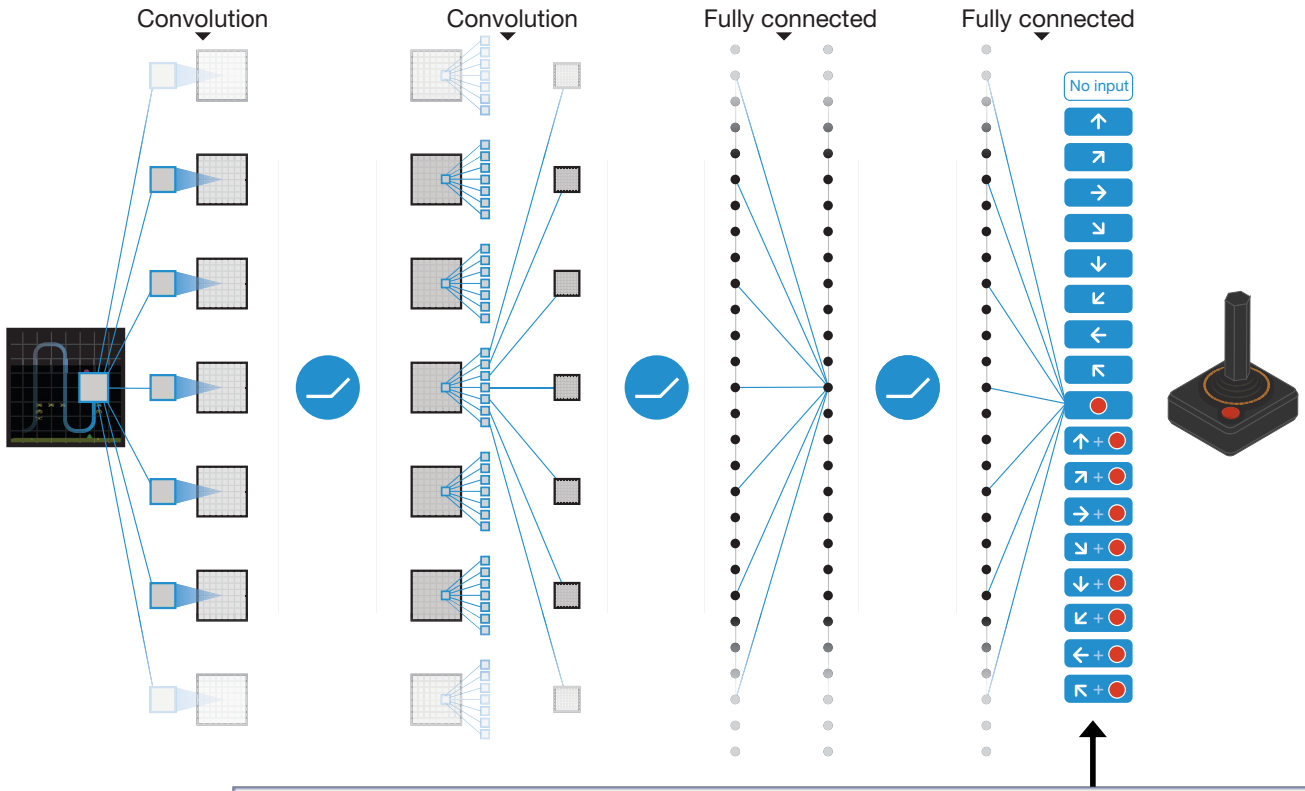
Mnih, et al. (2015)



2 Fully-Connected Layers:
512 RELU layer, followed by
a linear output layer

A CNN for Atari Control

Mnih, et al. (2015)



Outputs: one output node per action (rather than single Q-value output), varying from 4 to 18 nodes, depending on the game

Lecture 09-5c:
DQN Details

Deep Q-Learning with Replay

Let D be a *replay memory*, with capacity N
Initialize two neural networks with identical random weights, $\Theta_1 = \Theta_2$
for episode $e = 1 \ldots E$ **do**
  Set state $s_t = s_0$, the start state
  for time-step $t = 1 \ldots T$ **do**
    Choose action a_t using an ϵ-greedy policy, based on Θ_1
    Receive reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in memory D
    for simulation $m = 1 \ldots M$ **do**
      Sample a random transition $(s_i, a_i, r_i, s_{i+1}) \in D$
      Set $y_i = \begin{cases} r_i & \text{if } s_{i+1} \text{ is end of game} \\ r_i + \gamma \max_{a'} \Theta_2(s_{i+1}, a') & \text{otherwise} \end{cases}$
      Do gradient descent, using error $(y_i - \Theta_1(s_i, a_i))^2$
    endfor
    Every C steps, set $\Theta_2 = \Theta_1$
  endfor
endfor

Features of the Algorithm

- The deep-*Q* algorithm uses a replay memory, which stores state-action transitions encountered during play
  - Periodically, the algorithm samples random past transitions from this memory and does weight updates based upon those
- This avoids the first problem for methods that combine neural nets and RL (temporal association): since episodes are sampled at random, correlations between states of game-play no longer bias the learning
  - Mnih, et al. use a memory of $1{,}000{,}000$ past transitions
  - At the start of the algorithm, $50{,}000$ purely random actions are chosen to initially populate the memory

Features of the Algorithm

- The algorithm uses ***two*** neural nets, one to generate sample outputs, and one updated by back-propagation

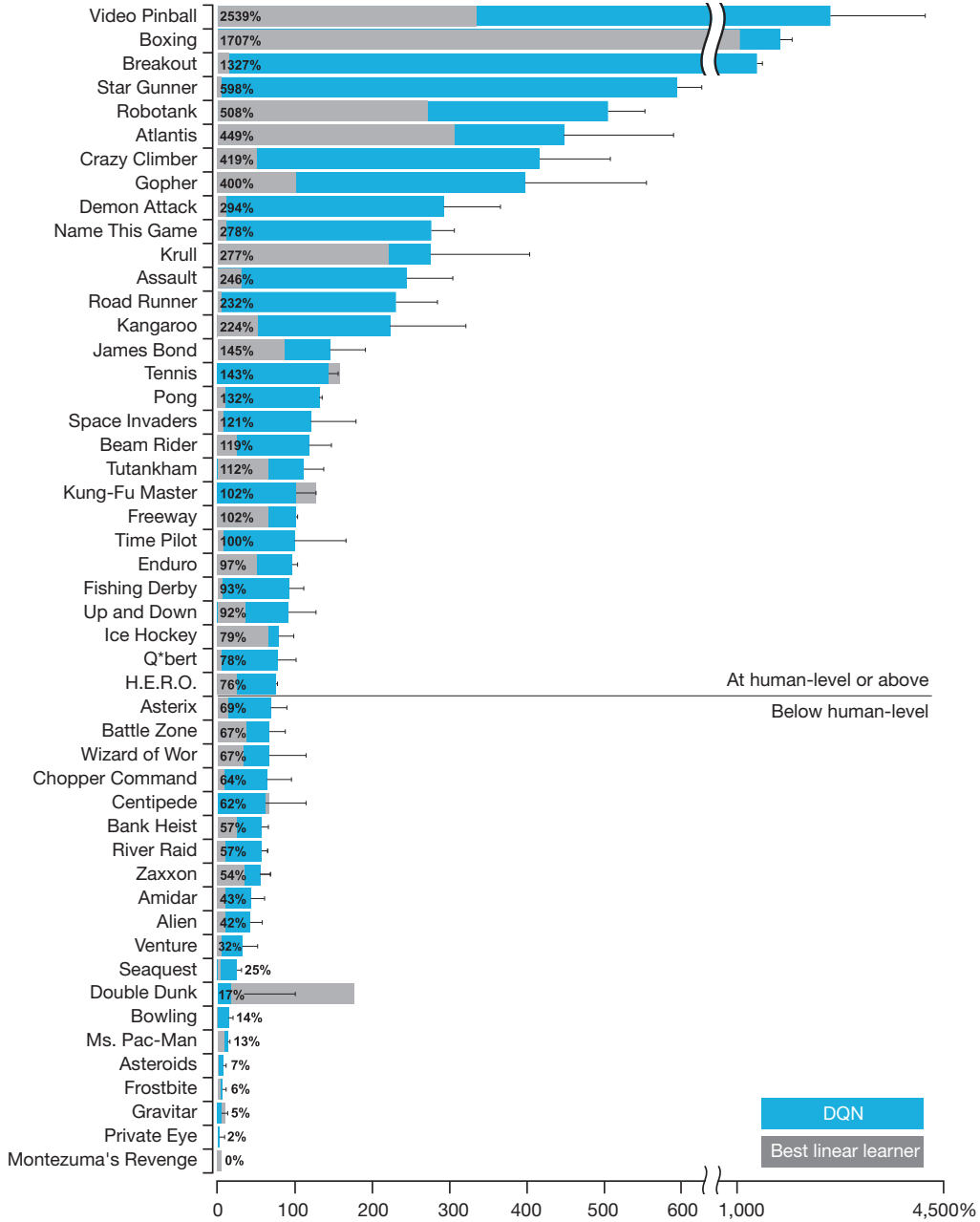
for simulation $m = 1 \ldots M$ **do**
  Sample a random transition $(s_i, a_i, r_i, s_{i+1}) \in D$
  Set $y_i = \begin{cases} r_i & \text{if } s_{i+1} \text{ is end of game} \\ r_i + \gamma \max_{a'} \Theta_2(s_{i+1}, a') & \text{otherwise} \end{cases}$
  Do gradient descent, using error $(y_i - \Theta_1(s_i, a_i))^2$
endfor
Every C steps, set $\Theta_2 = \Theta_1$

Output of Θ_2 used

Net Θ_1 updated

- This avoids second problem (value/policy dependence): updates to weights don't change output right away
  - Mini-batches of 32 simulated transitions are sampled
  - The sampling network is updated every $C = 10{,}000$ steps

Lecture 09-5d:
DQN Results



Learned Performance

The same algorithm was run on all games in the Atari test-bed (only changing number of available actions).

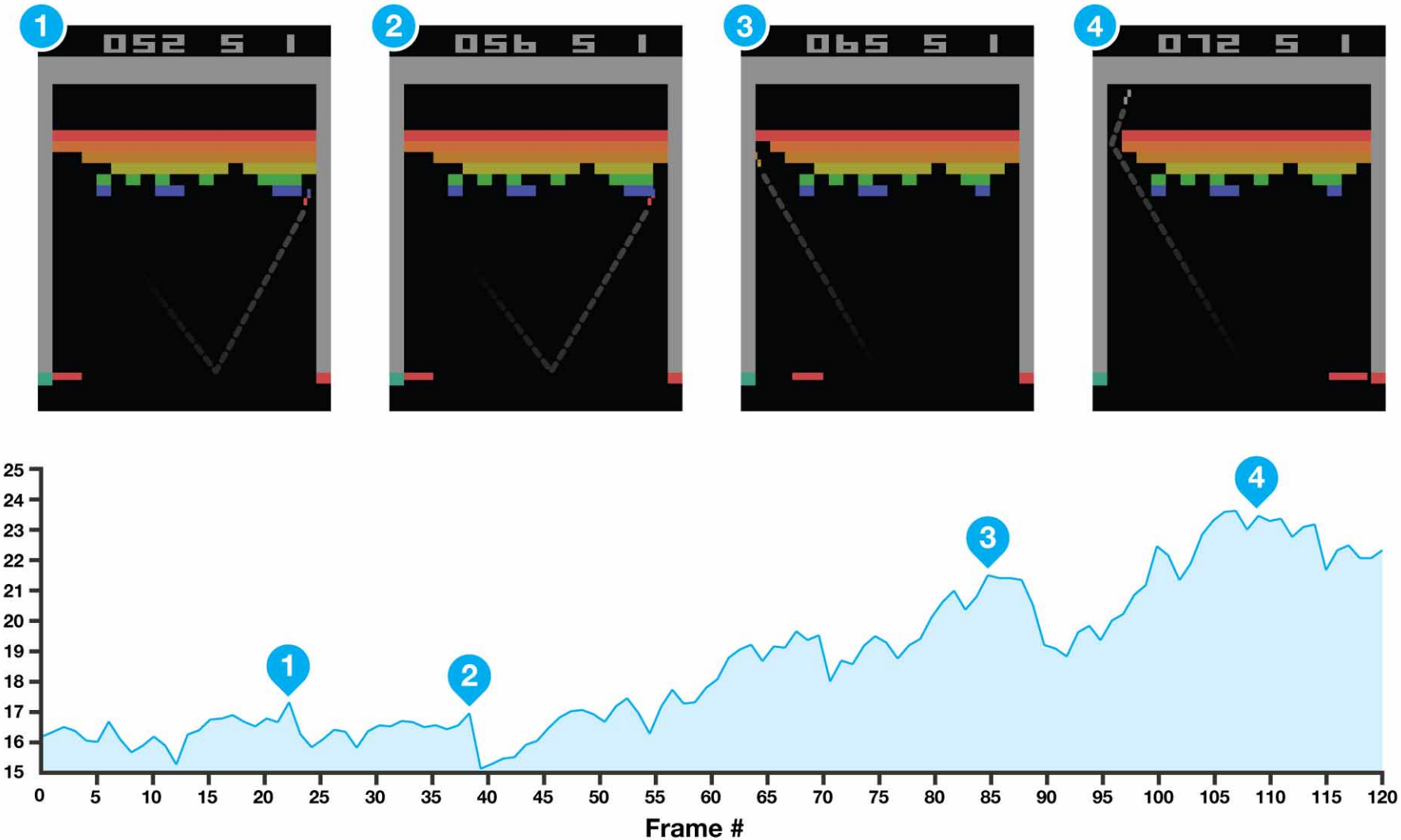
Compared to both the best-performing linear RL algorithm and to an expert human player (allowed to practice for 2 hours on each game).

Outperformed linear in all but two cases, and matched or outperformed the human player in a majority of games, even when limited to choosing an action every 6th frame.

Mnih, et al. (2015)

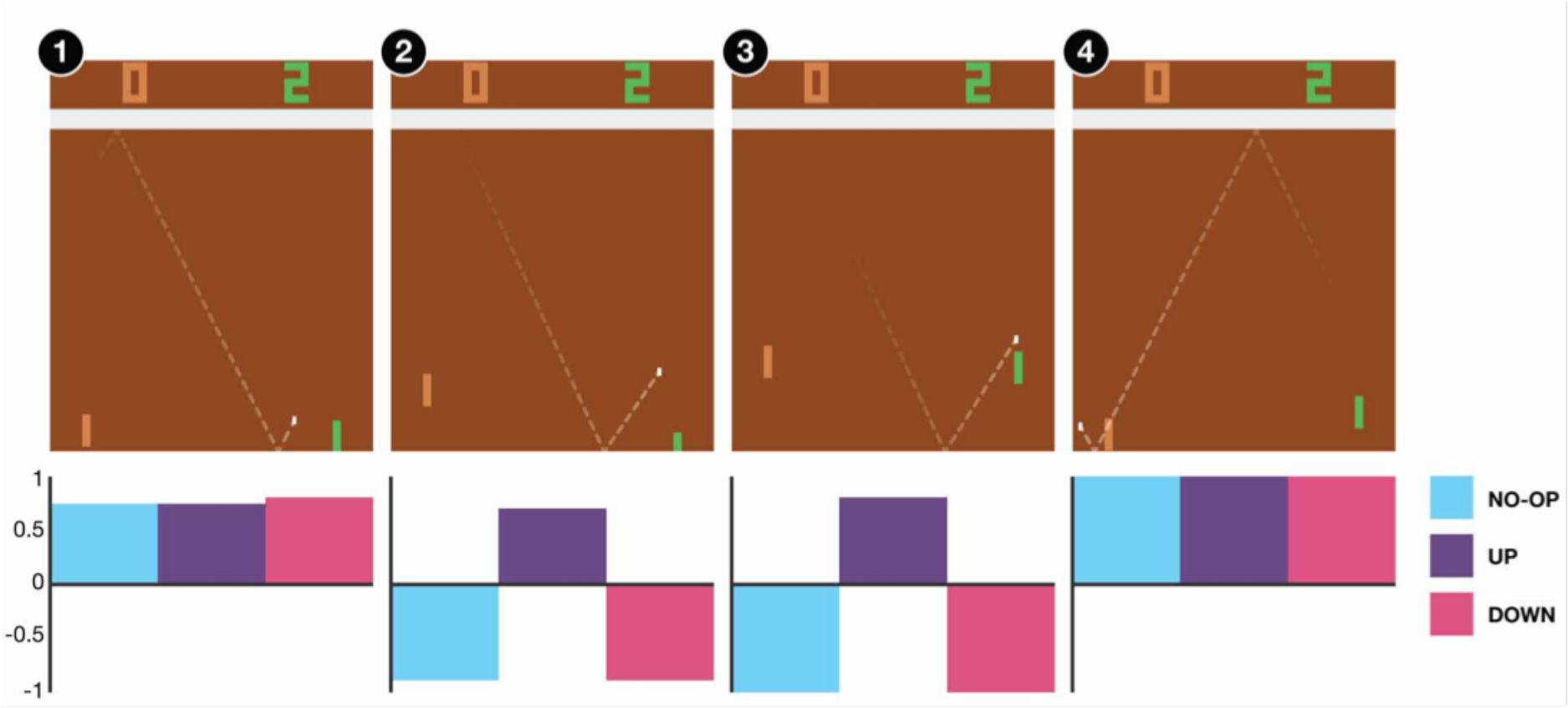
Sample Value Function

Mnih, et al. (2015)



Sample Q-Values

Mnih, et al. (2015)



Lecture 09-5e:
Additional Notes

Some Cool Videos

- OpenAI plays Hide and Seek
  `https://www.youtube.com/watch?v=Lu56xVlZ40M`
  `https://openai.com/blog/emergent-tool-use/`
  `https://arxiv.org/abs/1909.07528`
- AlphaGo Movie
  `https://www.youtube.com/watch?v=WXuK6gekU1Y`