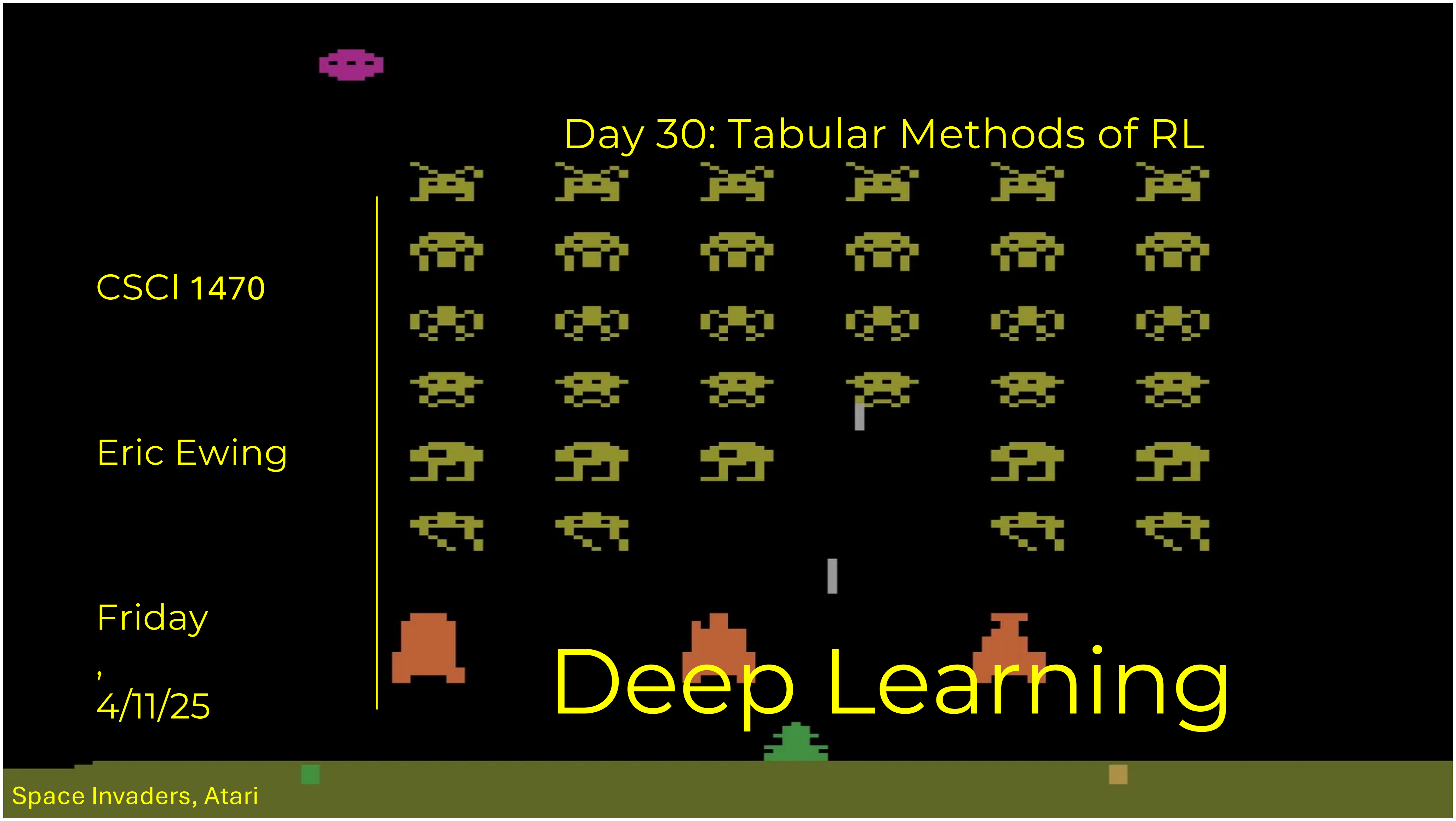


# Day 30: Tabular Methods of RL

# Deep Learning

CSCI 1470

Eric Ewing

Friday
,
4/11/25

Space Invaders, Atari

## RL vs. Other Deep Learning

- This class has mostly focused on data (e.g., framing as supervised or unsupervised problem, adapting to new modalities) and architectures
- In RL, the data and architectures matter less than the algorithms we use to update our networks
  - Sometimes RL algorithms compute the gradients directly
  - Determining the proper loss function is harder than just choosing between MSE or Cross Entropy.

More time on the blackboard than other topics

## Markov Decision Process (MDP) Review

Consist of:

- States: Set of all possible ways the world can be
- Actions: Set of all actions the agent can take
- Reward: Function mapping states to reward values
- Transitions: Function mapping (state, action) pairs to a distribution over next states.

Seek a policy $\pi$ that maps states to actions

## Gamma

One more term to add to MDPs: a discount factor $\gamma \in [0, 1]$

Some MDPs have no terminal state (or otherwise can have an agent take infinitely many actions)

We care about the total reward the agent gets, how do we reason about that when we need to sum infinitely many things?

The discount factor is a helpful mathematical trick: Each step into the future, we care about reward a little less

This sum is never infinite if *r* is bounded

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

## Key Terms

Episode: (For *Episodic* MDPs with defined start and terminal states)  a single run through of the MDP from a start state to a terminal state (or until a cutoff time *T*)

Trajectory: state, action, reward for every timestep in an episode

$\tau = (s_0, a_0, r_0, \dots, s_T, a_T, r_T)$

Return: Cumulative discounted rewards from timestep *t* for a single episode

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

$$G_t = \sum_{i=0}^{T-t} \gamma^i r^{i+t}$$

The return $G_0$ (sometimes just denoted $G$) is total discounted reward of the entire episode

## Breakout Example

- Episode: From start of game until player loses (or wins)
- Trajectory: list of all states, actions, and rewards from that episode
- Return: Cumulative discounted reward of that trajectory (if $\gamma = 1$, then it is the sum of all rewards)

## Key Terms

Value of a state: the **expected** returns from a state

$$V(s_t) = \mathbb{E}[G_t]$$

Q-Values: The expected returns of being in a state and **taking an action**

$$Q(s_t, a_t) = \mathbb{E}_{s' \sim T(s_t, a_t)}[V(s')]$$

## Value Function

A value function is defined for a specific policy $\pi$!

(If you have a bad policy, you expect your values to be smaller)

$$V^\pi(s_t) = \mathbb{E}[G_t]$$

$$V^\pi(s_t) = \mathbb{E}\left[\sum_{i=0}^{T-t} \gamma^i r^{i+t}\right]$$

$$V^\pi(s_t) = r_t + \gamma \cdot \mathbb{E}\left[\sum_{i=1}^{T-t} \gamma^i r^{i+t}\right]$$

$$V^\pi(s_t) = r_t + \gamma \sum_{s' \in S} \Pr(s' | s_t, a_t)\, V^\pi(s')$$

This is called policy evaluation

We can the value function as a recursive formula:
How good it is to be in a state is the immediate reward
for being in that state + the expected returns for future states

## Value Function → Policy

What if we don't have a policy already and want to find one?

If we already have a value function:

```
For every state
    iterate over all possible actions
        calculate the expected value if the agent takes that action
        $Q(s,a) = \sum_{s'} \Pr(s'|s,a)\,[R(s') + \gamma V(s')]$
    set $\pi(s)$ to be the action with highest expected Q-value
```

This is called *policy improvement*

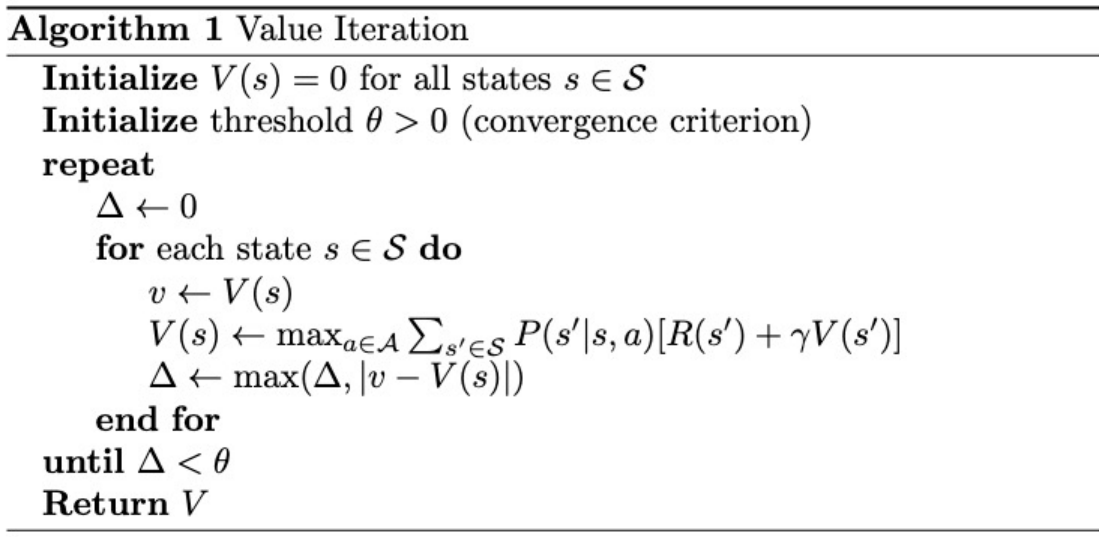
## Value Iteration

1. Start with a random Value function V
2. Run Policy Improvement to determine best actions at each state
3. Run Policy Estimation to determine the new values with the updated policy
4. Repeat

## Value Iteration

Repeatedly apply Policy Estimation and Policy improvement steps

Run until convergence (i.e., estimates of V no longer changes)



**Algorithm 1** Value Iteration

```
**Initialize** $V(s) = 0$ for all states $s \in \mathcal{S}$
**Initialize** threshold $\theta > 0$ (convergence criterion)
**repeat**
    $\Delta \leftarrow 0$
    **for** each state $s \in \mathcal{S}$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s') + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    **end for**
**until** $\Delta < \theta$
**Return** $V$
```

## Tabular Value Iteration

Value iteration is typically a dynamic programming algorithms

A table of values is constructed (one row for each state) and then updated according to the Bellman Equation:

$$V(s) = r + \gamma \max_a \sum_{s'} \Pr(s'|s, a)\, V(s')$$

## Q-Learning

Q-Learning is our first actual RL algorithm

- Reinforcement Learning algorithms actually simulate episodes, gather trajectories, and learn from experiences.

- Collect experiences (i.e.: $(s, a, r, s')$ tuples)

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

## Q-Learning

Q-Learning is our first actual RL algorithm

- Reinforcement Learning algorithms actually simulate episodes, gather trajectories, and learn from experiences.

- Collect experiences (i.e.: $(s, a, r, s')$ tuples)

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

Important:
How do we collect experiences (i.e., how do we select what action to take)?

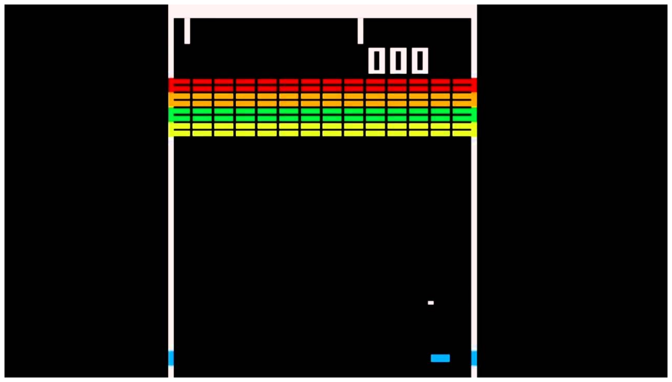
How do we update our estimates of Q?

# Collecting Experiences

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

What if we always took the action go-right?

- We'd update our estimates for go-right, but never go-left



What if we take uniform random actions?

- We'd update estimates for both right and left, but we'd be unlikely to get too far into the game

What if we find a happy middle ground between fully deterministic and fully random?

- With probability $\epsilon$ take a random action
- With probability $1 - \epsilon$ take the best action (action with highest Q-value)

$\epsilon$-greedy Algorithm for balancing exploration in RL

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

$$0 = [r + \gamma \max_{a'} Q(s',a')] - Q(s,a)$$

## Updating estimates of Q-values

Q-learning:

Maintain estimates of Q(s, a) for all (s, a) pairs

Collect experiences, update Q estimates with:

$$Q(s,a) \leftarrow \underset{\substack{\uparrow \\ \text{Current estimate}}}{Q(s,a)} + \underset{\substack{\uparrow \\ \text{Learning rate}}}{\alpha}\Big[\underbrace{r + \gamma \max_{a'} Q(s',a') - Q(s,a)}_{\substack{\uparrow \\ \text{Error in estimate (Temporal} \\ \text{Difference Error)}}}\Big]$$

## Tabular Q-Learning

**Algorithm 2** Q-Learning

```
**Initialize** $Q(s, a) = 0$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
**Initialize** learning rate $\alpha \in (0, 1]$ and discount factor $\gamma \in [0, 1)$
**Initialize** exploration parameter $\epsilon \in (0, 1)$
**for** each episode **do**
    Initialize state $s$
    **repeat**
        With probability $\epsilon$: choose a random action $a \in \mathcal{A}$
        Otherwise: choose $a = \arg\max_{a'} Q(s, a')$
        Take action $a$, observe reward $r$ and next state $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        (Or $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$)
        $s \leftarrow s'$
    **until** $s$ is terminal
**end for**
**Return** $Q$
```

## Where's the Deep Learning part of this?

- Neural Networks are Function approximators and we have some functions...
  - $V: S \rightarrow \mathbb{R}$
  - $Q: S \times A \rightarrow \mathbb{R}$
  - $\pi: S \rightarrow A$
- Deep Reinforcement Learning seeks to approximate these functions with neural networks

## Deep Q-Learning

- Approximate Q-values with a neural network
- Always needed a loss function with neural networks before…
- Can we come up with a loss function here?
- We want this equality to hold: $0 = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$
- If we can force $[r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$ to be close to 0, we will have good approximations of Q-values

$$L = \left(\left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)\right)^2$$

# Q-Learning

How to update tabular Q-learning to be deep Q-learning

$$L = \left(\left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)\right)^2$$

**Algorithm 2** Q-Learning

```
**Initialize** $Q(s, a) = 0$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
**Initialize** learning rate $\alpha \in (0, 1]$ and discount factor $\gamma \in [0, 1)$
**Initialize** exploration parameter $\epsilon \in (0, 1)$
**for** each episode **do**
    Initialize state $s$
    **repeat**
        With probability $\epsilon$: choose a random action $a \in \mathcal{A}$
        Otherwise: choose $a = \arg\max_{a'} Q(s, a')$
        Take action $a$, observe reward $r$ and next state $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        (Or $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$)
        $s \leftarrow s'$
    **until** $s$ is terminal
**end for**
**Return** $Q$
```

Can't just update outputs of a NN directly...
Instead, compute loss and run a step of SGD

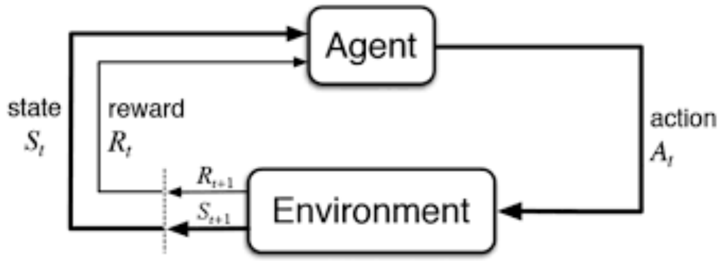
## Recap

Important RL Notation: V, Q, G, $\pi$

Value Iteration

Q-Learning

Up Next: Deep-Q Learning tips, tricks, and variants


Agent
state $S_t$
reward $R_t$
action $A_t$
$R_{t+1}$
$S_{t+1}$
Environment