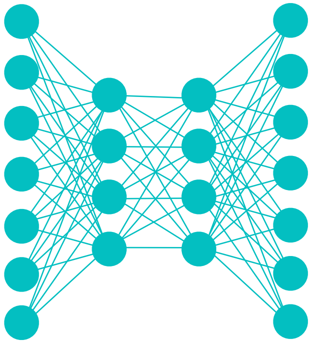
Lecture Notes for

Neural Networks and Machine Learning

Value Iteration and Q-Learning



Logistics and Agenda

- Logistics
    - Grading Update
    - Office Hours
- Agenda
    - Markov Building Blocks
    - Value Iteration (and demo)
        - Q-Function Variant
    - Q-Learning Approximation
    - Deep Q-Learning

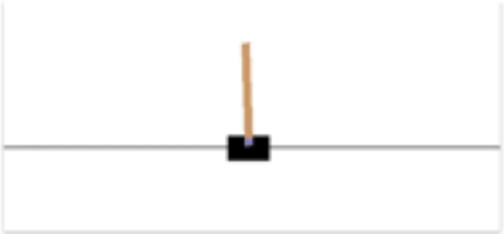
Last Time

```
import gym


if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```

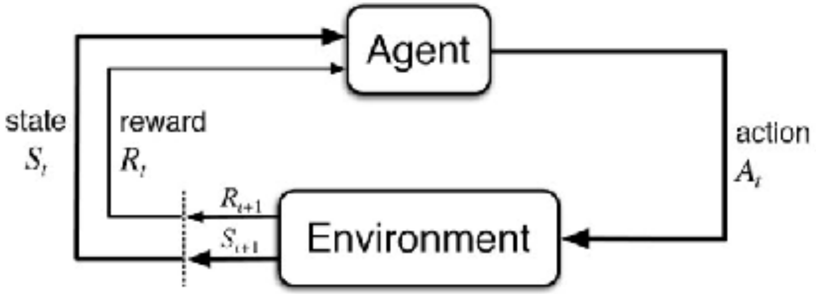


Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step

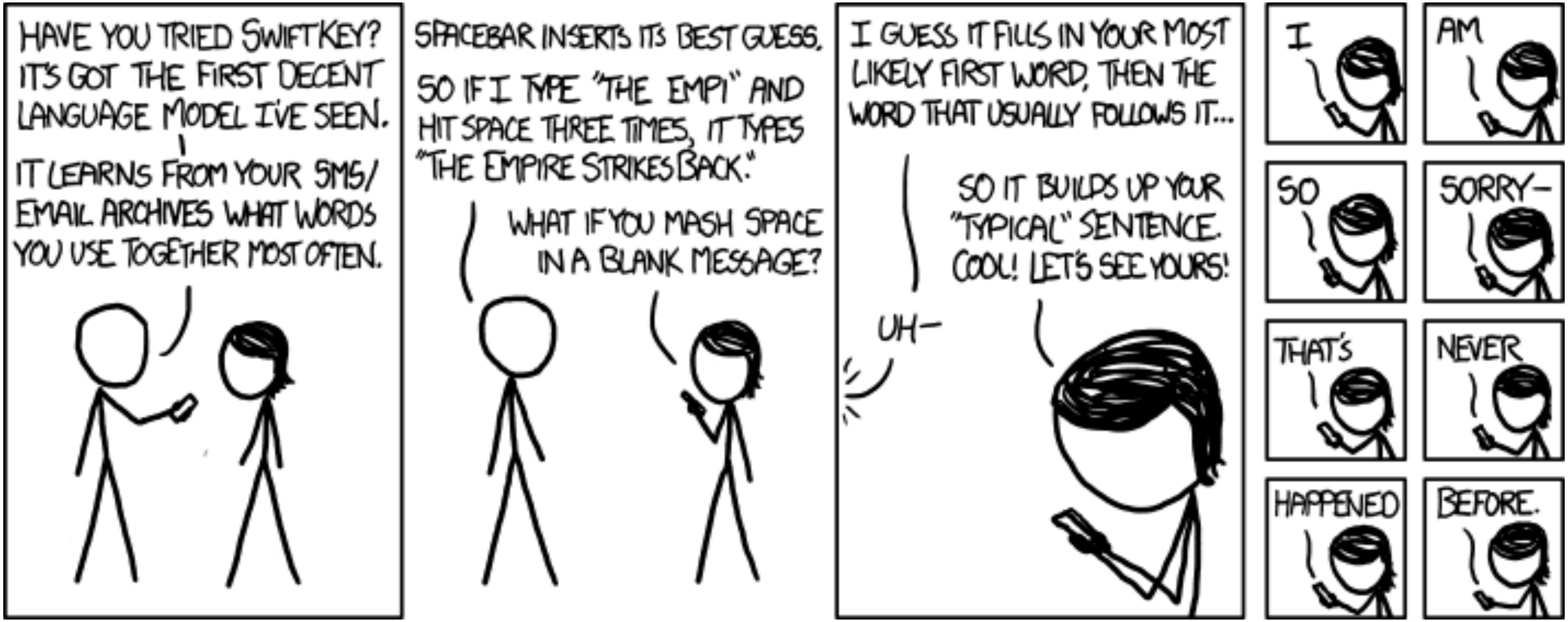


- **State**: Every square in grid
- **Action**: Move to make (l,r,u,d), *with probability*
- **Reward**: Goal, Death
- **Policy**: Given state, where should we move?
- **Optimal Policy**

$$\pi^* = \arg\max_{\pi} \mathrm{E}\left[\sum_k \gamma^k R_{t+k+1} \mid \pi\right]$$

Random Policy Another Policy Another Policy

Markov Building Blocks



Markov Processes (MP)

- **Definition**: Any process that can be explained (or simplified) through a sequential set of states that depend only on the previous state
- **Practical Meaning**: For N states, there will be the probability of transition to any other state, encoded through an NxN transition matrix of discrete probabilities
- State sequences are not deterministic, they are sampled from these distributions
- Despite **simplicity**, MP can model a number of real processes with **good enough** precision

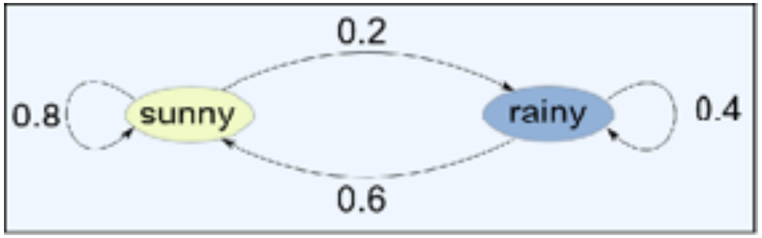
Next State, s_{t+1}

Current State, s_t

0.1	0.2	0.1	0.6	0.0
0.9	0.0	0.1	0.0	0.0
0.0	0.4	0.0	0.4	0.2
0.0	0.4	0.2	0.0	0.4
0.0	0.0	0.6	0.0	0.4

MP Example from Maxim Lapan

	Sunny’	Rainy’
Sunny	0.8	0.2
Rainy	0.6	0.4



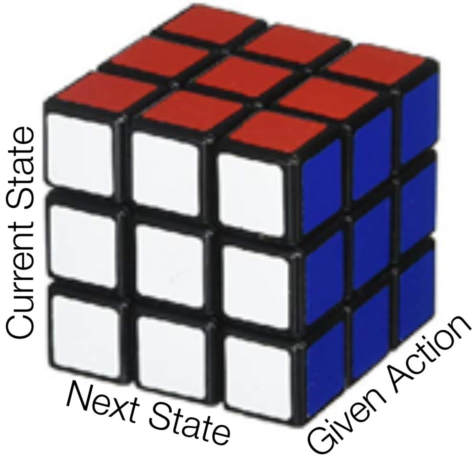
Sun+Summer				
Rainy+Summer				
Sun+Fall				
Rainy+Fall				
Sun+Else				
Rainy+Else				

...

Adding One Variable Can Have Drastic Effect on State Space Size

Markov Decision Processes (MDP)

- **New Definition**: any state to state transition can be altered by an action that is given by a Markov Process, so we can alter the MP with discrete actions (decisions)
- **Definition**: An MDP consists of:
  - Env. States, s_t
  - Actions set for each time $\boldsymbol{a}_t$
  - Reward function for each state, $\boldsymbol{r}(s_t)$
  - A transition model, $P(s_{t+1}, s_t \mid a)$ a matrix of probabilities
    - Not ***guaranteed*** next state by given action, probabilistic



Markov Reward Process (MRP)

- **Total reward:** weighted sum of future rewards in sequence

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_k \gamma^k R_{t+k+1}$$

- γ defines future reward far- and short-sightedness
  - Common values are **0** (short), **0.9**, **0.99**, and **1** (far)
- G: Want to estimate and maximize this reward!
- This reward calculation, G, can be used to estimate the "**Value**" of each state based upon the average total reward a state *should* give, $V(s) = \mathbf{E}[G \mid s_t{=}s]$
- Typically, this value must be estimated from the model over fixed sequences, otherwise some reward values can become arbitrarily large by looping actions

MDPs and MRPs

- The million dollar question:
  How do we select a good action given a current state?
- What we did with Cross Entropy: setup a comparison of different actions we might take (**policy**)
- A **policy** is defined as $\pi(a, s) = P(a_t = a \mid s_t = s)$
  - Given the current state, we have a certain probability of selecting each action
  - Action selection is **probabilistic**, but easy to discover **deterministic** actions (*set one action to 1.0, others to 0.0*)
- Try different policies, select one with best average reward
- What we will do now: iteratively interact with environment and get an estimate of $V(s) = \mathbf{E}[G \mid s_t = s]$ called **value iteration**

Value Iteration

When you first start Training with Reinforcement Learning

Value Iteration Overview

- Randomly initialize $V(s)$ values
- Interact with environment to estimate rewards from states
- Use $V(s)$ to select the next state (policy from state value)
- Estimate transition probabilities of actions that will take us to desired next state with the largest $V(s)$
- Update $V(s)$ values using recurrence relation
- Keep repeating, updating transition probabilities and $V(s)$ estimate until we get good rewards consistently
- $V(s)$ should follow the Bellman equation, keep updating it until it does
  - So what is this Bellman equation?

State Value Function (Review)

- Given: $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_k \gamma^k R_{t+k+1}$$
- $V(s_t) = \mathbf{E}[G_t \mid s_t{=}s]$, expected Value of a given state over all future iterations
- **Important**: we can only calculate this exactly if we know:
    - all the rewards for all the states, actions, next states
    - the probabilities of transitioning to a given state from selecting an action
    - likelihood of successful action

We can also define the following recurrence relation:

$$V(s) = \mathbf{E}[G_t \,|\, s_t = s]$$

$$V(s) = \mathbf{E}[R_{t+1} + \gamma G_{t+1} \,|\, s_t = s, s_{t+1} = s']$$

$$V(s) = \mathbf{E}[R_{t+1} + \gamma V(s') \,|\, s_t = s]$$

The Bellman Equation

- For the case when each action is successful and state is discrete, ideal V has property, $a \rightarrow s$:

$$V_s^{ideal} = \max_{a \in 1 \ldots A} (r_a + \gamma V_a)$$

current value is immediate reward plus value of next state with highest value because we will choose this next state and will be successful in reaching it

- In general, actions are probabilistic, we need to sum over possible transitions for ideal V, and property becomes:

$$V_s^{ideal} = \max_{a \in A} \mathbf{E}[r_{s,a,\hat{s}} + \gamma V_{\hat{s}}] \quad \approx \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V_{\hat{s}})$$

-probabilities of getting to next state x (current value is immediate reward plus value of next state)
-$p_{a,0 \rightarrow s}$ probability of getting to state s from state 0, given that you perform action a

- **Needs:** To select action with best value we need reward matrix, $r_{s,a,\hat{s}}$, action transition matrix $p_{a,s \rightarrow \hat{s}}$ and $V_{\hat{s}}$

Value Iteration (direct variant)

- **Direct**:
    - Initialize $V(s)$ to all zeros
    - Take a series of random steps, then follow policy
    - estimate $p_{a,s \to s'}$ via observed **Transitions**
    - Perform value iteration: $V(s) \leftarrow \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \to \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$
    - Repeat until $V(s)$ stops changing

With infinite time and exploration, this update will
Converge to Optimal Policy

Value Iteration Reinforcement Learning

M. Lapan Implementation for
and Frozen Lake

`Follow Along:`
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Value Iteration with Q-function Variant

$$V_s = \max_{a \in A} \mathbf{E}[r_{s,a,\hat{s}} + \gamma V_{\hat{s}}] = \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V_{\hat{s}})$$

- Define intermediate function Q

$$Q(s,a) = \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V_{\hat{s}})$$

- With some nice properties/relations:

$$V_s = \max_{a \in A} Q(s,a)$$

$$Q(s,a) = \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{\hat{a}} Q(\hat{s},\hat{a}))$$

Value Iteration via q-function

- **Q-Function Variant**:
  - Initialize Q(s,a) to all zeros
  - Take a series of random steps, then follow policy
  - estimate $p_{a,s \rightarrow s'}$ via observed **Transitions**
  - Perform value iteration with Q:

$$Q(s,a) \leftarrow \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q(\hat{s}, a'))$$

  - Repeat until Q is not changing

Q-Learning Iteration Reinforcement Learning

M. Lapan Implementation for and Frozen Lake

Follow Along:
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Value Iteration Limitations

- Q and V can get really big for **large states and action spaces**
- Transition matrix can get gigantic for large state and action spaces (and potentially intractable)
    - We will solve this by dropping the transition probabilities in Q function update
    - Helps make computation tractable, but optimization harder
- This Variant is known as Q-Learning
- (*not addressing yet…*) Q-table needs infinite inputs when the state spaces are **continuous**
    - We will solve this by using a neural network to **approximate** the Q function
    - Q function already has the transitions simplified, so this is already in a good form for learning from NN

Creating a computable Q approximation

- Assume Q function can incorporate this

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\to\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s},a'))$$

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\to\hat{s}} \cdot r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s},a') \cdot p_{a,s\to\hat{s}}$$

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\to\hat{s}} \cdot r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{new}(\hat{s},a')$$

Leftmost term actually just sums over $\hat{s}$ so that reward has no dependence

$$Q^{old}(s,a) = r_{s,a} + \sum_{\hat{s}\in S} \gamma \max_{a'} Q^{new}(\hat{s},a')$$

What happens if our state space is absolutely gigantic. Summing over all possible states seems like a bad idea.

$$Q^{new}(s,a) = r_{s,a} + \gamma \max_{a'} Q^{new}(s',a')$$

Can we approximate it more simply?

We now call this the **Bellman Approximation**

Tabular Q-Learning Algorithm

- In update, **ignore the transition probability**, making use of the iterative nature of Q, Bellman Update:

$$Q(s_t, a_t) = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 \dots$$

$$Q(s_t, a_t) = r_0 + \gamma(r_1 + \gamma^2 r_2 + \gamma^3 r_3 \dots)$$

$$Q(s_t, a_t) = r_0 + \gamma \max_a Q(s_{t+1}, a)$$

$$Q^{old}(s, a) \leftarrow \sum_{\hat{s} \in S} p_{a,s \to \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s}, a'))$$

$$Q^{new}(s, a) \leftarrow r_{s,a} + \gamma \max_{a' \in A} Q^{new}(s', a')$$

- For stability, add momentum to the **Bellman approximation update** equation

$$Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot [r_{s,a} + \gamma \max_{a' \in A} Q(s', a')]$$

- Algorithm, start with empty $Q(s,a)$:
  - Sample (with rand) from environment, (s, a, r, s')
  - Make Bellman Update with Momentum
  - Repeat until desired performance

Tabular Q-Learning Reinforcement Learning

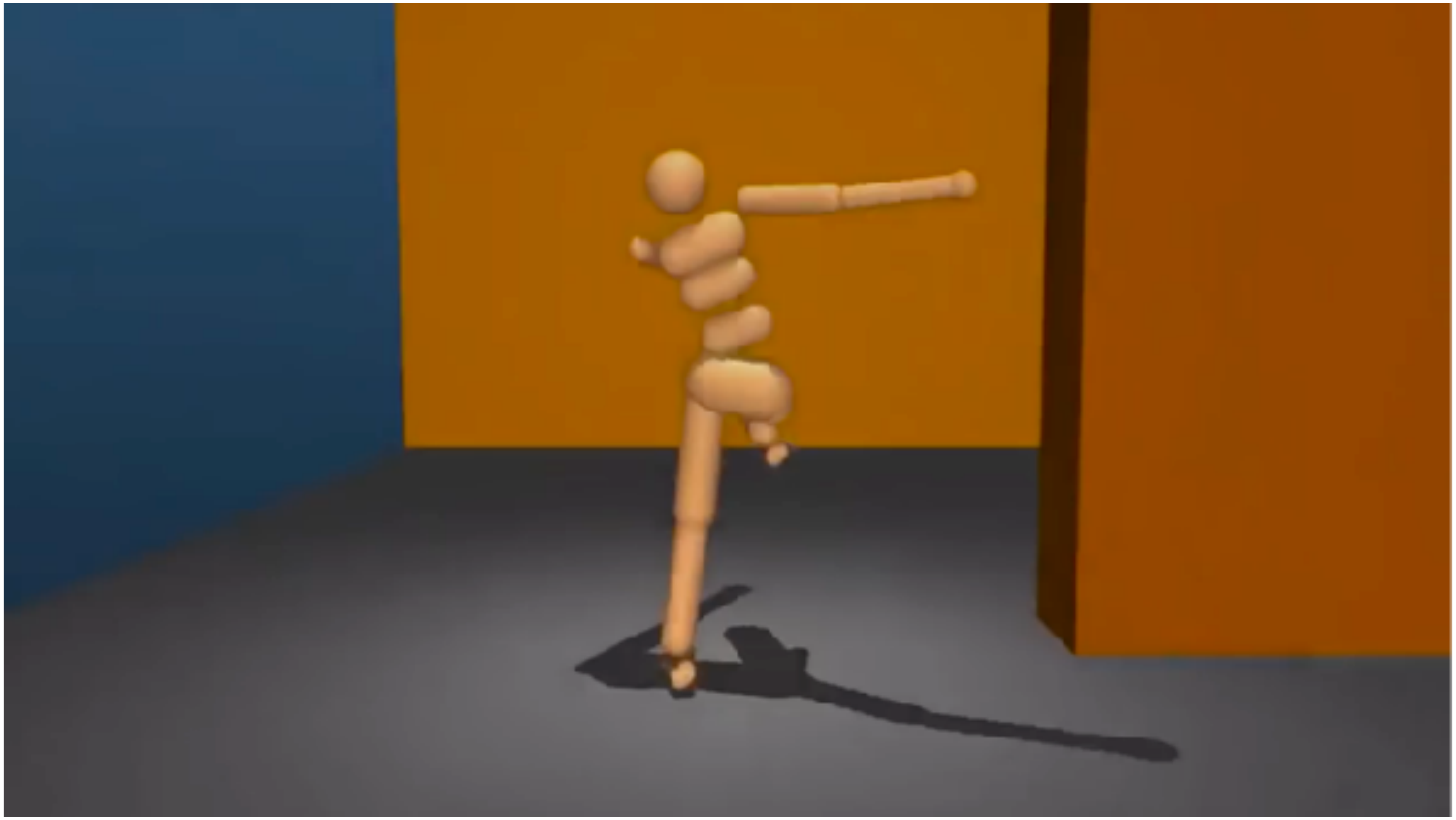
M. Lapan Implementation for
and Frozen Lake

Conclusion:

- It still works, but wow it takes much longer to converge!!!
- Placing so much emphasis on the Q-function (to learn all variability) makes the optimization much more difficult and the update to Q noisy (because its an approximation)

`Follow Along:`
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Deep Q-Learning



Q-Learning with a Neural Network

- Want to approximate $Q(s,a)$ when the state space is potentially large. Given s_t (could be continuous), we want the network to give us a row of actions from $Q(s,a)$ table that we can choose from:

[… other states…]

→ [$Q(s_t,a_1)$, $Q(s_t,a_2)$, $Q(s_t,a_3)$, … $Q(s_t,a_A)$] ←

[… other states…]

- How to train network to be Q? Make a loss function which incentives the actual Q-function behavior we desire from a sampled tuple (s, a, r, s')

$$\mathcal{L} = \left[\underbrace{Q(s,a)}_{\text{from current network params}} - [r_{s,a} + \gamma \underbrace{\max_{a' \in A} Q^*(s',a')}_{\text{from older network params (better stability)}}]\right]^2$$

Periodically Update

Params of Q^* from Q

$$\mathcal{L} = \left[Q(s,a) - [r_{s,a}]\right]^2$$

if no next state (env is done)