Reinforcement Learning: Deep-Q + REINFORCE

CSCI 1470/2470
Spring 2024

Deep Learning

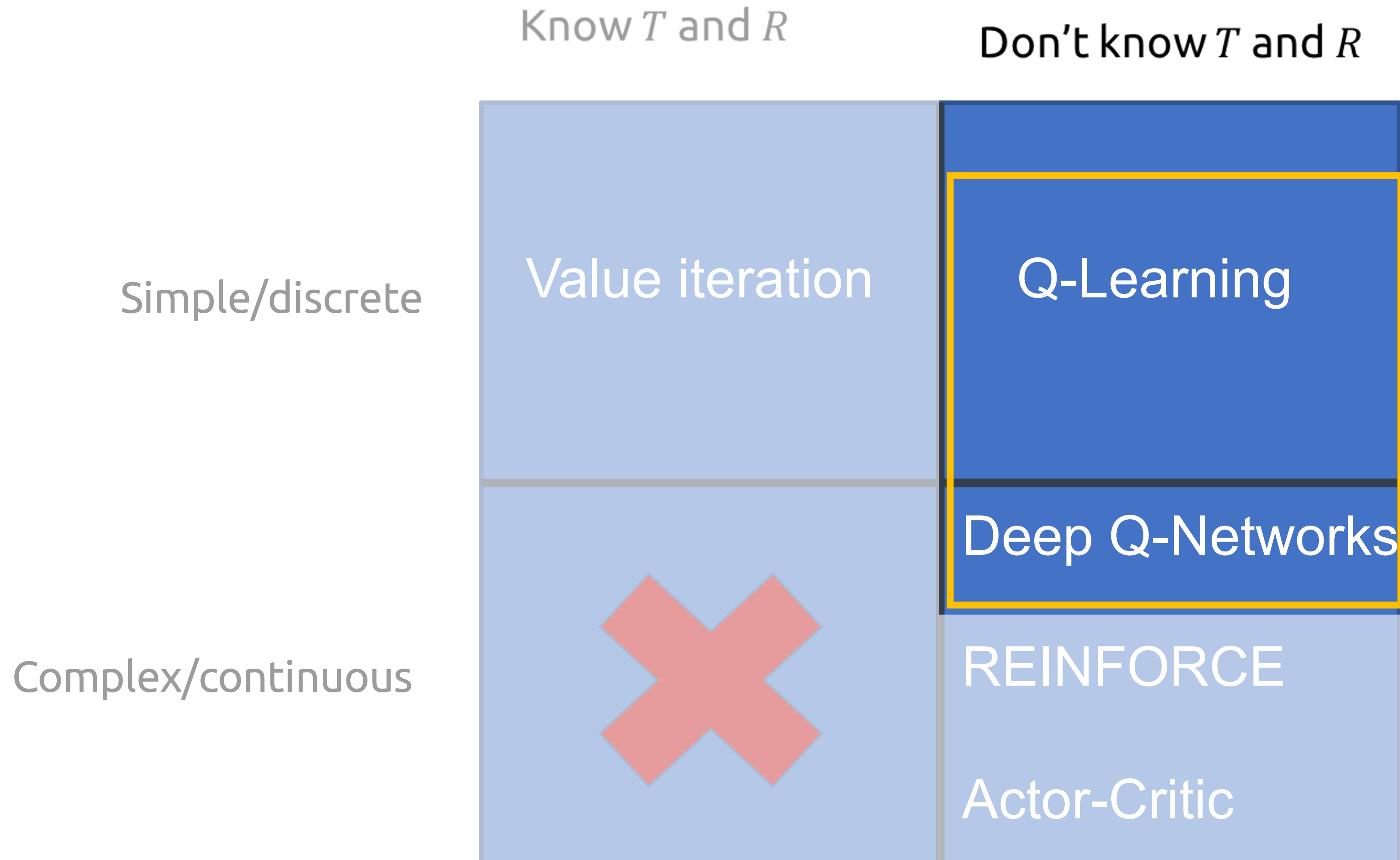
Ritambhara Singh

April 22, 2024
Monday

Students can access the Course Feedback System through:
1) Canvas
2) https://brown.evaluationkit.com
3) Personalized login link in the reminder emails sent from course_feedback@brown.edu.

ChatGPT prompt “minimalist landscape painting of a deep underwater scene with a blue tang fish in the bottom right corner”

Organizing RL problems/algorithms

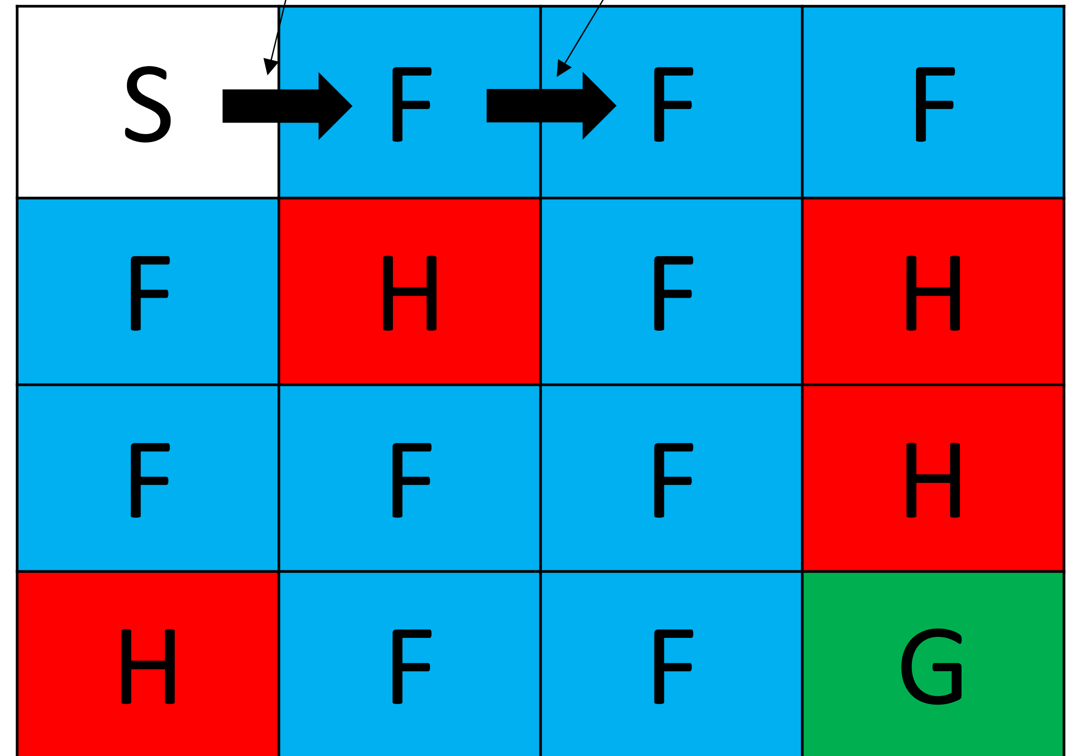


For a more complete taxonomy of RL algorithms, see https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below

Review: Q-Learning

Use Q to pick action a_0. Observe transition and get reward r_0. Use r_0 to update $Q(s_0, a_0)$ immediately

Repeat process for s_1. Update $Q(s_1, a_1)$ with r_1



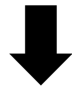
$$Q^{\pi}(s,a) = E[R(s,a,s') + \gamma V^{\pi}(s')]$$

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

Review: The Q-Learning Update Rule

Combining our new estimate for $Q(s, a)$ with the weighted average equation, the final update rule for Q-Learning becomes

$$Q(s, a) = (1 - \alpha)Q_{old}(s, a) + \alpha Q_{new}(s, a)$$

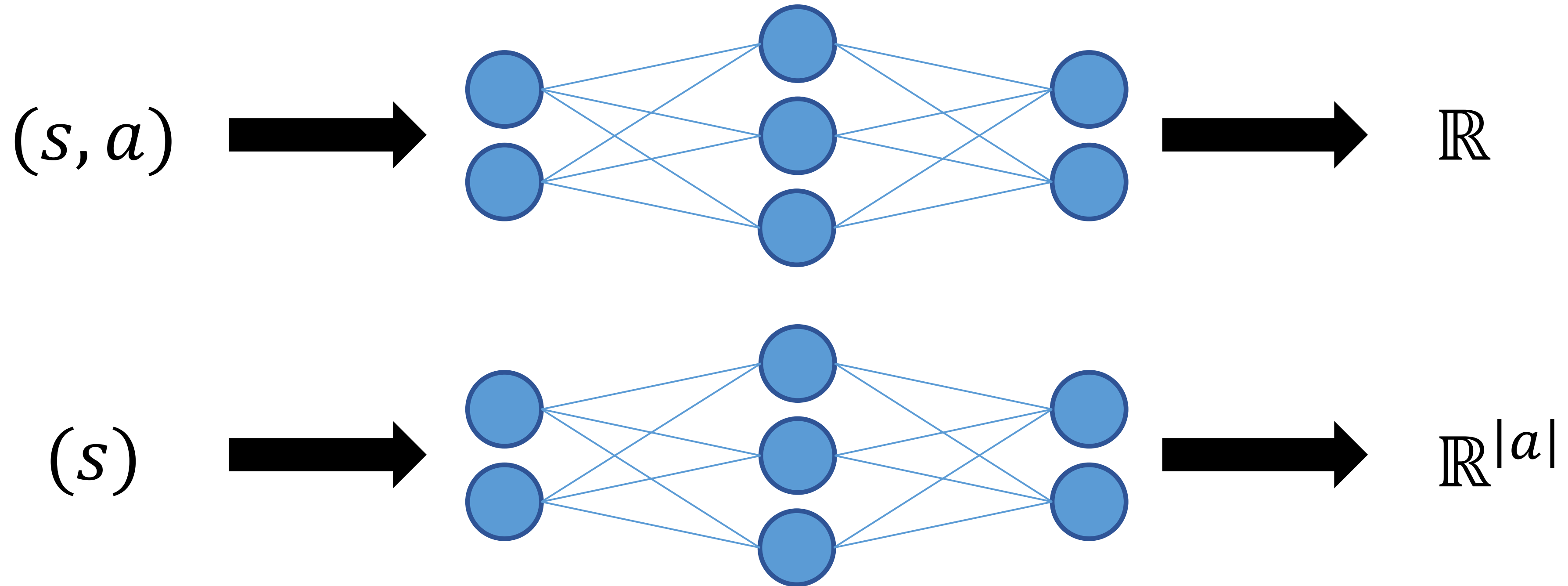


$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma V(s'))$$

Review: Q-Learning Update in Code

```
if np.random.rand(1) < epsilon:
     act = env.action_space.sample()
else:
     act = np.argmax(Q[st])
nst, rwd, done, _ = env.step(act)
Q[st][act] = (1-alpha)Q[st][act] + alpha(rwd + 
gamma*V[nst])
```

Review: Neural Nets as Function Approximators



Neural networks are excellent function approximators, so we can use a deep network to learn this parameterized function

Review: TF Example Code for Frozen Lake

```
# Weights for Q-Network
Q = tf.Variable(tf.random.uniform([16,4],0,0.01))
# Q-value function
def qVals(inptSt):
    oneH = tf.one_hot(inptSt,16)
    qVals = tf.matmul([oneH],Q)
    return qVals
# argmax over q-values to get estimated best action
action = tf.argmax(qVals(st),1)
```

Designing our Loss Function

The update rule looks similar to gradient descent

At what point is the change from the update rule zero? That would be our point of ‘zero loss’

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R(s,a,s') + \gamma V(s'))$$

This happens when $Q(s,a) = R(s,a,s') + \gamma V(s')$

Designing our Loss Function

We can then use the difference between our observed estimate and our old estimate as a loss function

$$L = Q(s,a) - (R(s,a,s') + \gamma(max_a Q(s,a))$$

$V(s)$ becomes $\max_a Q(s,a)$ because we only learn an estimate for Q, without explicitly learning V.

This loss is called the ‘temporal difference error’ or TD(0). The zero refers to how many steps we take before comparing our old estimate to the new.

Designing our Loss Function

In practice, we actually use the squared difference between our current estimate and new observation (to more heavily penalize large errors and to ensure loss is always positive)

$$L = \Big(Q(s,a) - \big(R(s,a,s') + \gamma max_a Q(s,a)\big)\Big)^2$$

TD(0) also isn't the only option for a loss function. You can wait any number of steps in the environment before updating your estimates of Q. If you waited one additional step, you would have the TD(1) error.

$$L_{TD(1)} = \Big(Q(s,a) - \big(R(s,a,s') + \gamma R(s',a',s'') + \gamma^2 max_a Q(s'',a)\big)\Big)^2$$

DQN in Tensorflow

Convert this into deep q-learning code

```
if np.random.rand(1) < epsilon:
      act = env.action_space.sample()
else:
      act = np.argmax(Q[st])
nst, rwd, done, _ = env.step(act)
Q[st][act] = (1-alpha)Q[st][act] + alpha(rwd +
gamma*V[nst])
```

- Start with original tabular Q-Learning Code
- Replace Q table with Q network
- Replace weighted Q update with loss function
- Add gradient update step

DQN in Tensorflow

```
if np.random.rand(1) < epsilon:
      act = env.action_space.sample()
else:
      act = np.argmax(Q[st])
nst, rwd, done, _ = env.step(act)
Q[st][act] = (1-alpha)Q[st][act] + alpha(rwd +
gamma*V[nst])
```

- Start with original tabular Q-Learning Code

DQN in Tensorflow

```
Q_values = qVals(st)
if np.random.rand(1) < epsilon:
        act = env.action_space.sample()
else:
        act = tf.argmax(Q_values)
nst, rwd, done, _ = env.step(act)
Q[st][act] = (1-alpha)Q[st][act] + alpha(rwd +
gamma*V[nst])
```

- Start with original tabular Q-Learning Code
- Replace Q table with Q network

DQN in Tensorflow

```
Q_values = qVals(st)
if np.random.rand(1) < epsilon:
    act = env.action_space.sample()
else:
    act = tf.argmax(Q_values)
nst, rwd, done, _ = env.step(act)
nextQ = qVals(nst)
target_Q = Q_values.numpy();
target_Q[act] = rwd + np.max(nextQ)
loss = tf.reduce_sum(tf.square(Q_values - target_Q))
```

- Start with original tabular Q-Learning Code
- Replace Q table with Q network
- Replace weighted Q update with loss function

DQN in Tensorflow

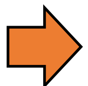
```
Q_values = qVals(st)
if np.random.rand(1) < epsilon:
      act = env.action_space.sample()
else:
      act = tf.argmax(Q_values)
nst, rwd, done, _ = env.step(act)
nextQ = qVals(nst)
target_Q = Q_values.numpy();
target_Q[act] = rwd + np.max(nextQ)
loss = tf.reduce_sum(tf.square(Q_values - target_Q))
```

- Start with original tabular Q-Learning Code
- Replace Q table with Q network
- Replace weighted Q update with loss function

Note that we treat the next-step estimate of Q as our optimization "target" and do not differentiate through it

DQN in Tensorflow

```
Q_values = qVals(st)
if np.random.rand(1) < epsilon:
        act = env.action_space.sample()
else:
        act = tf.argmax(Q_values)
nst, rwd, done, _ = env.step(act)
nextQ = qVals(nst)
target_Q = Q_values.numpy();
target_Q[act] = rwd + np.max(nextQ)
loss = tf.reduce_sum(tf.square(Q_values – target_Q))
optimizer.apply_gradients(tape.gradient(loss,Q),Q)
```



- Start with original tabular Q-Learning Code
- Replace Q table with Q network
- Replace weighted Q update with loss function
- Add gradient update step

Code Demo

https://drive.google.com/open?id=1cKkuEdoqnwe99dhxnBUz5KcRqRUqoFnk

Beyond Deep Q Networks (DQN)

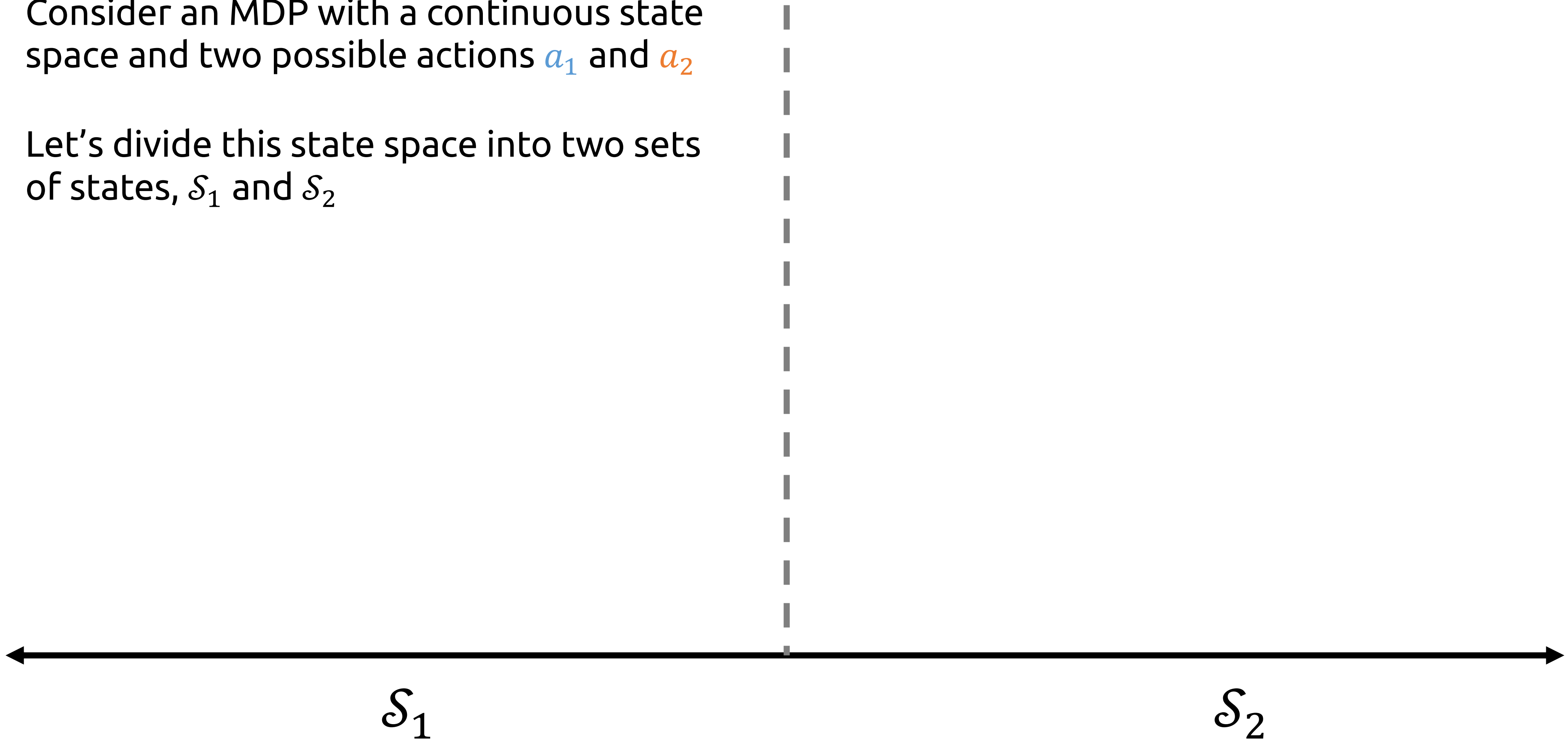
- DQN is amazing!
  - Can learn optimal play for Breakout, other Atari games given only raw pixels as input

- ***Does it have any weaknesses?***
  - DQN uses a neural net to learn an approximation of the Q function
  - Could that ever be a hard learning problem?

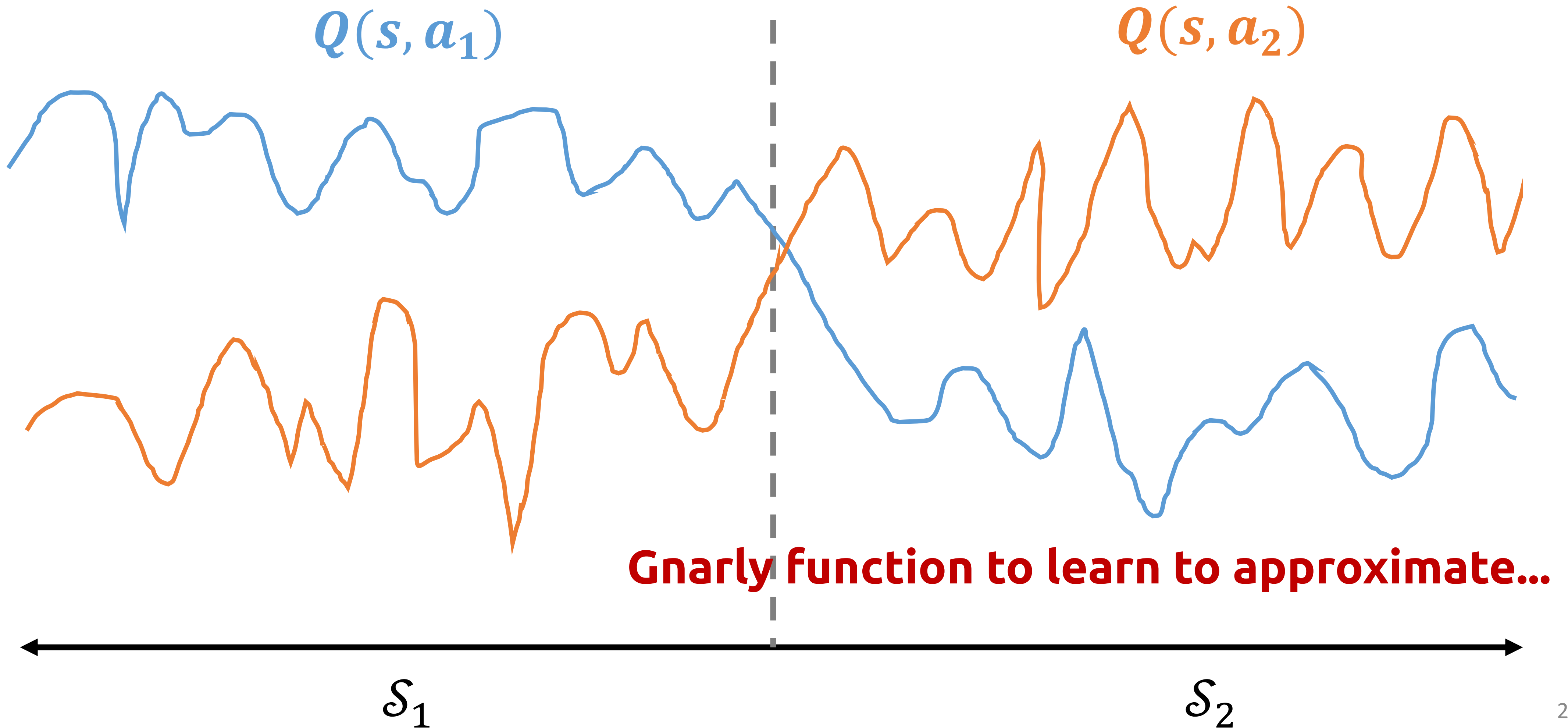
https://www.youtube.com/watch?v=TmPfTpjtdgg

Q Functions can be complex...

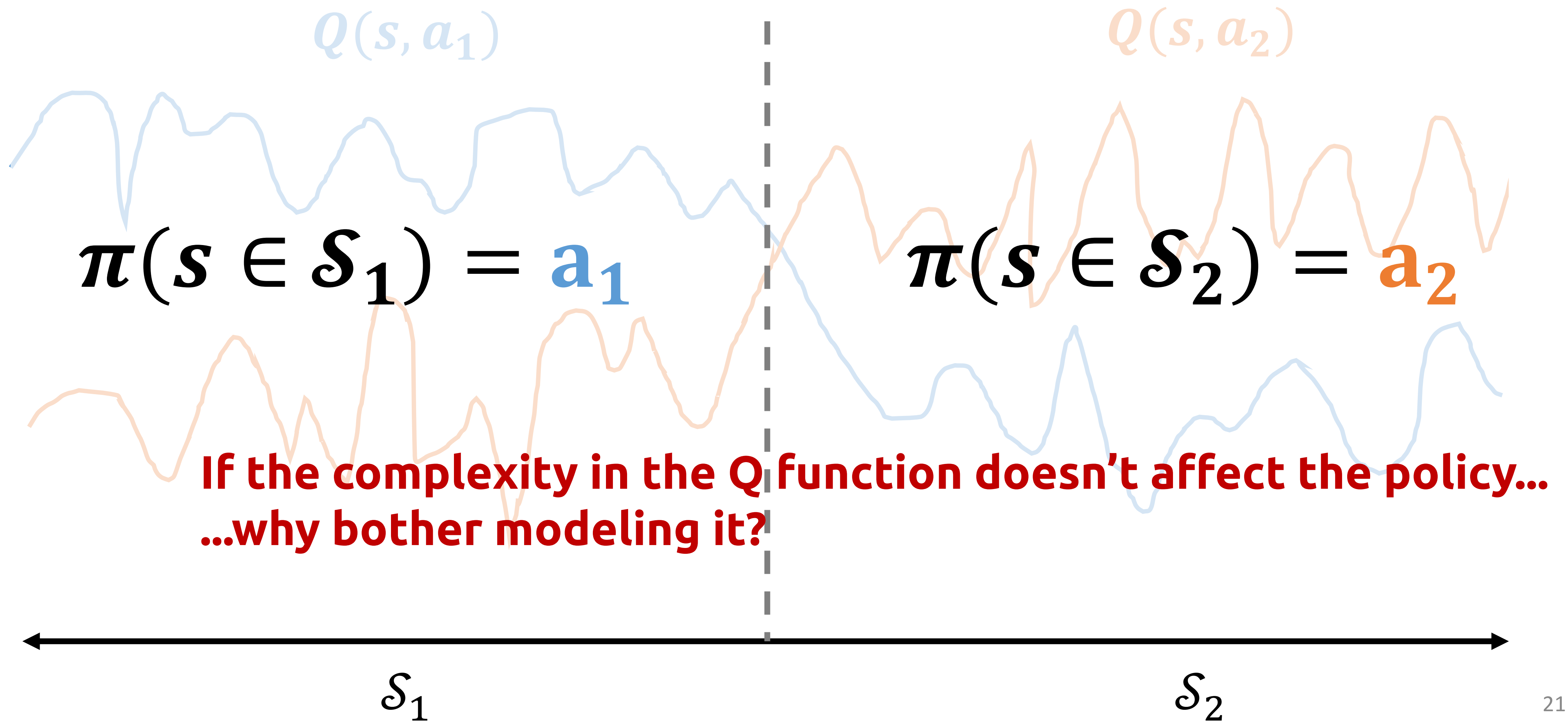
- Consider an MDP with a continuous state space and two possible actions a_1 and a_2
- Let's divide this state space into two sets of states, $\mathcal{S}_1$ and $\mathcal{S}_2$



Q Functions can be complex...

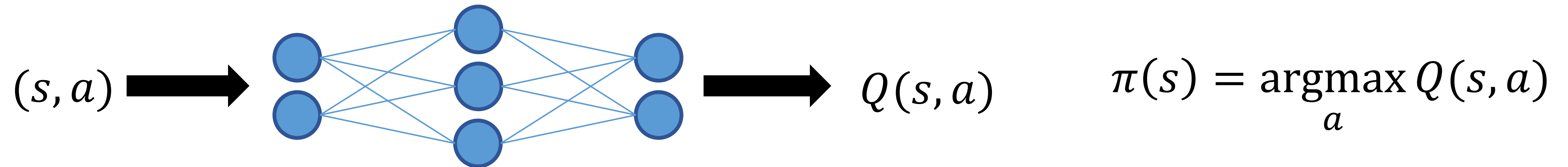


...but policies can still be simple

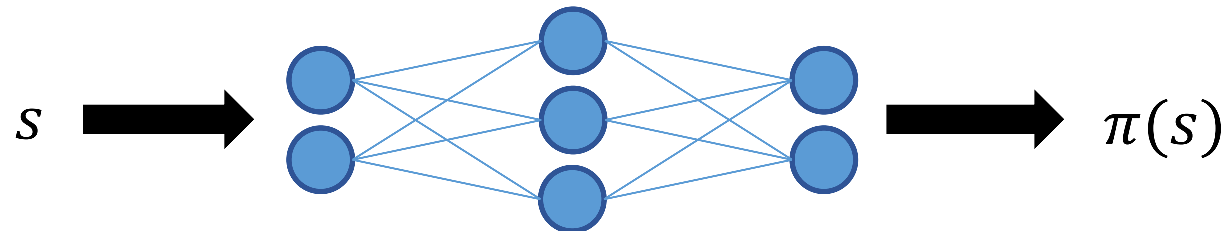


An Idea:

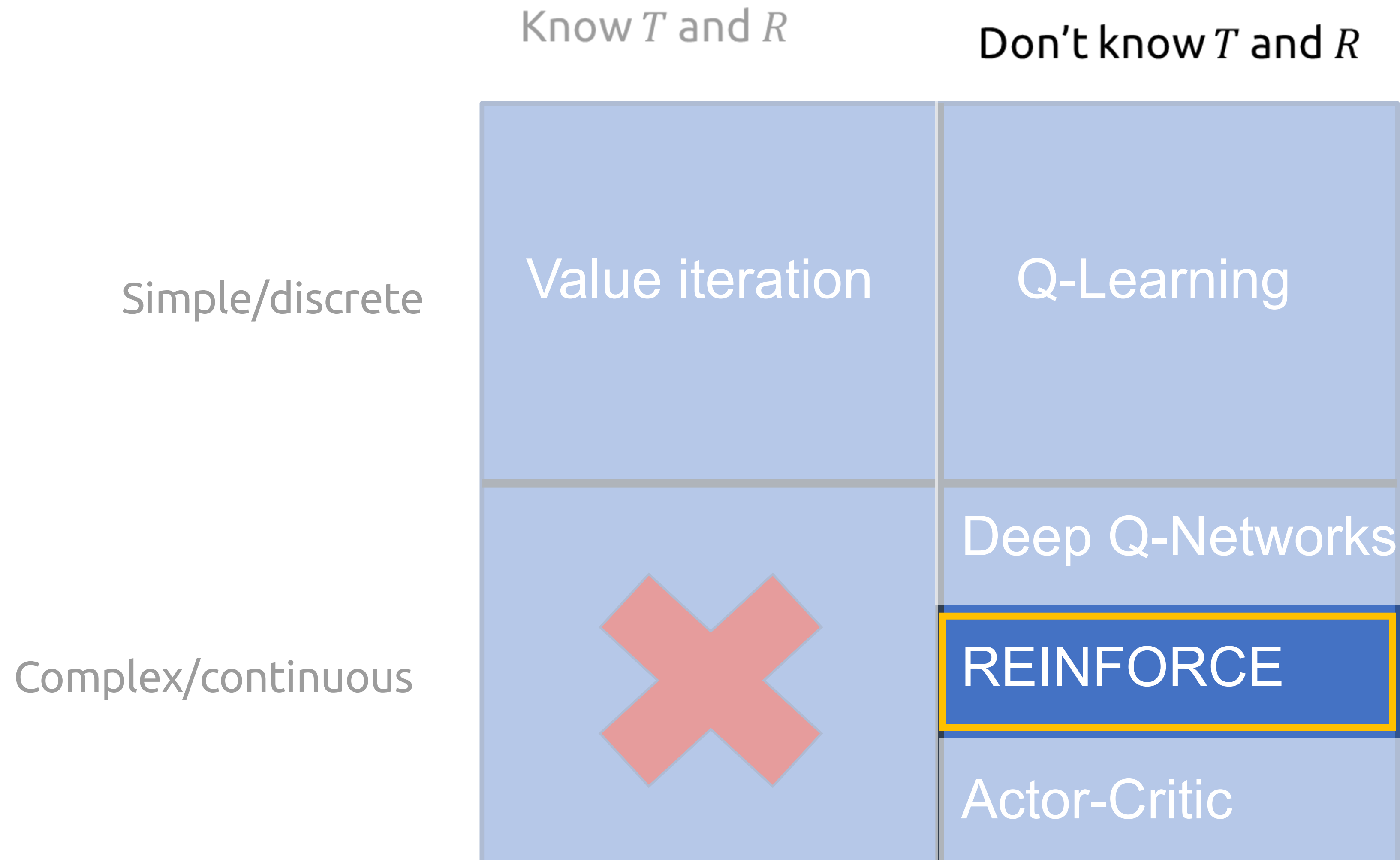
- Instead of learning a Q Network, and then extracting the policy from it:



- ...why don't we just directly learn a ***Policy Network?***
  - i.e. have a neural net that takes in a state and outputs an action

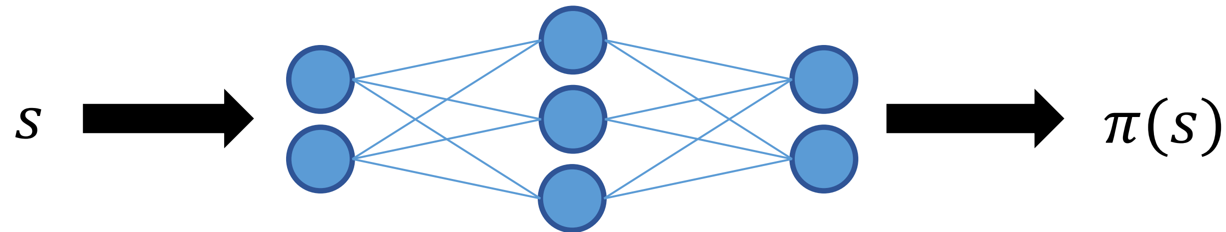


Organizing RL problems/algorithms

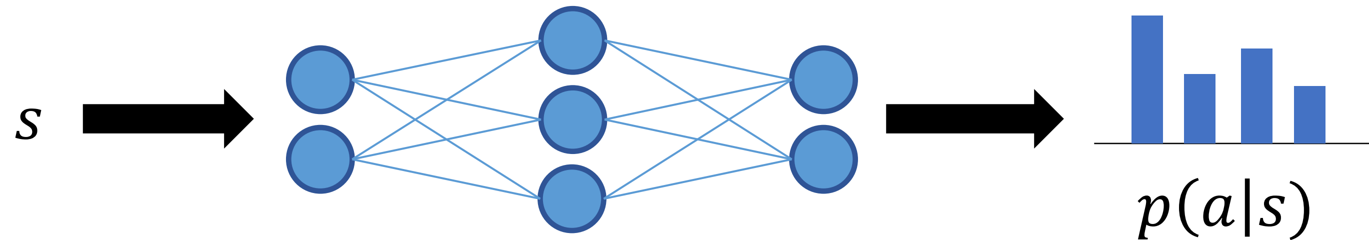


For a more complete taxonomy of RL algorithms, see https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below

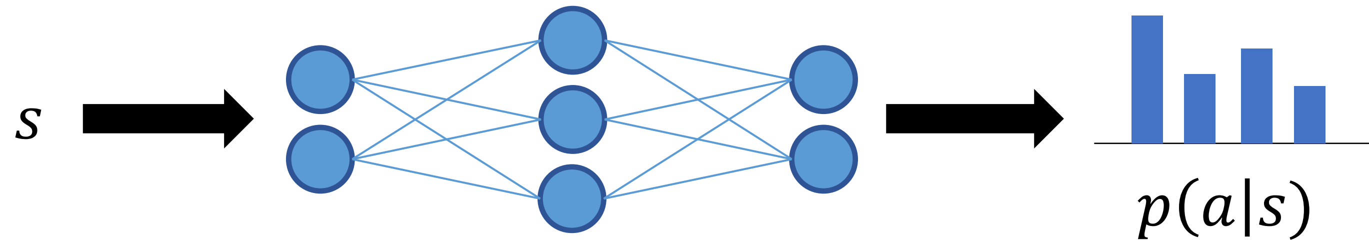
Policy Networks



- Q: $\pi(s)$ is a discrete action...how to make the network output that?
- A: Treat it like a classification problem—have the network output a ***probability distribution*** over actions

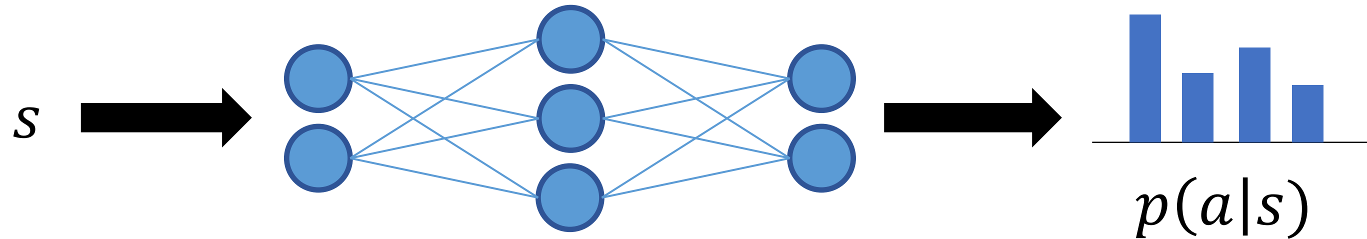


Using a Policy Network



- Q: How to get a discrete action out of this distribution?
- A: Two possibilities:
  1. $\pi(s) = \operatorname{argmax}_a p(a|s)$ → Deterministic policy (just like Q learning)
  2. $\pi(s) = \text{sample}(p(a|s))$ → ***Stochastic*** policy
     - Don't always take the same action in the same situation
     - Arguably, more "naturalistic" behavior

Training Policy Networks



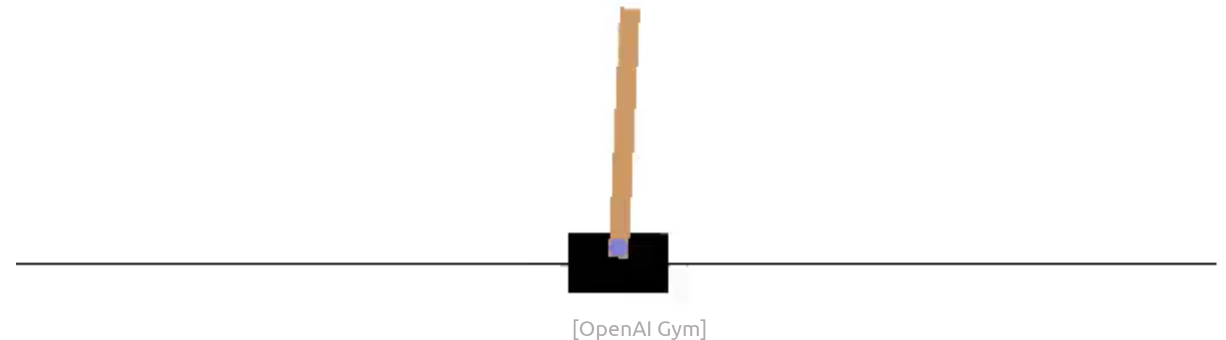
- How do we train a network like this?
- We can’t just “adapt Q learning” somehow—this is a fundamentally different beast
- The study of how to learn policy networks lies at the core of most modern deep reinforcement learning research
- Family of learning algorithms known as ***Policy Gradient*** methods

- Let’s make this concrete via a specific example...

The “Cart Pole” Environment

Cart Pole

- Attempt to keep a pole vertically balanced on a moving cart
- Continuous-state MDP
  - Not solvable with tabular Q-learning
- Still a "toy problem"
  - This is an instance of a dynamic equilibrium problem in classical robotics / control theory.
  - There exist [closed-form solutions](#) to the problem.
  - But it's also a fun test-case for RL ☺



[OpenAI Gym]

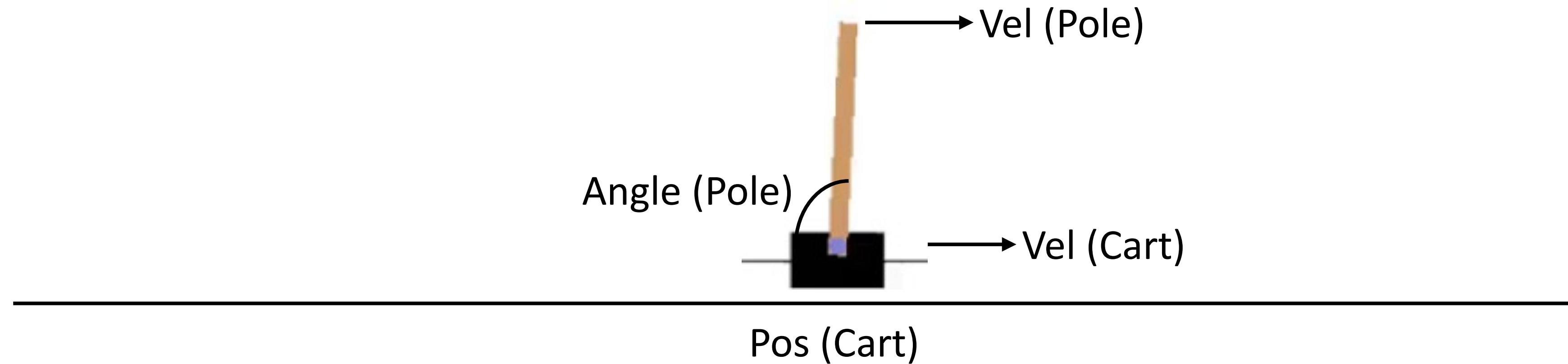
Note: the 'jumps' in the video are from the agent failing and the simulation restarting again

*A *closed-form solution* (or *closed form* expression) is any formula that can be evaluated in a finite number of standard operations.

Let's define S, A,T, R

Cart Pole MDP Formulation

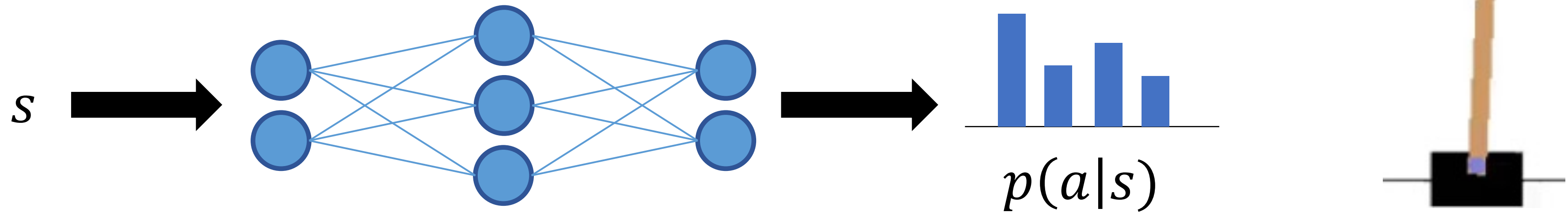
- State: cart position, cart velocity, pole angle, pole tip velocity



- Actions: push cart to **left** or **right**
- Transition function: (deterministic) simulation of Newtonian physics
- Reward function: 1 for every step taken
  - i.e. rewards keeping the pole balanced for as many steps as possible

Training a Policy Network for Cart Pole

- Would be easy to do with supervised learning (i.e. if we had a ground-truth expert demonstration to follow)

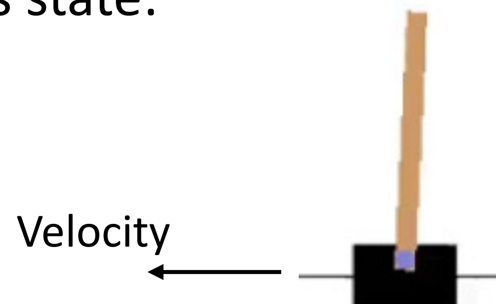


- Just use cross-entropy loss on the ground-truth “correct” action at every time step
- But we don’t have supervision in RL...so what do we do instead?

What should we do?

Training a Policy Network for Cart Pole

- Naïve loss function: Play an episode of the simulation, record the states/actions taken $(\mathbf{s}, \mathbf{a}) = (s_1 \dots s_T, a_1 \dots a_T)$, maximize the reward received at each timestep
  - i.e. $L(s_t, a_t) = -r(s_t, a_t, s_{t+1})$
- ***Why is this not a good loss function?***
  - Just because an action keeps the pole up for one more timestep, doesn't mean it will lead to keeping the pole up for the long term
  - E.g. consider this state:



  - Moving the cart the left will not make the pole tip over immediately (so you'll get a reward of 1), but it will hasten the pole's eventual tipping

Training a Policy Network for Cart Pole

- Better loss function: maximize the expected future return that you'll get from taking an action (not the immediate reward)
  - i.e. $L(s_t, a_t) = -\mathbb{E}[G_t \mid a_t]$
- What's another name for the expected future return?
  - The Q function! $L(s_t, a_t) = -\mathbb{E}[G_t \mid a_t] = -Q(s_t, a_t)$
- We don't know Q, though (we're trying to ***avoid*** estimating it)
- But, we can play an entire simulation episode to completion, and then see what future reward we got ***in that single episode***.
  - i.e. if the episode lasts T steps, then $L(s_t, a_t) = -\sum_{i=t}^{T} \gamma^{i-1} r(s_i, a_i, s_{i+1})$

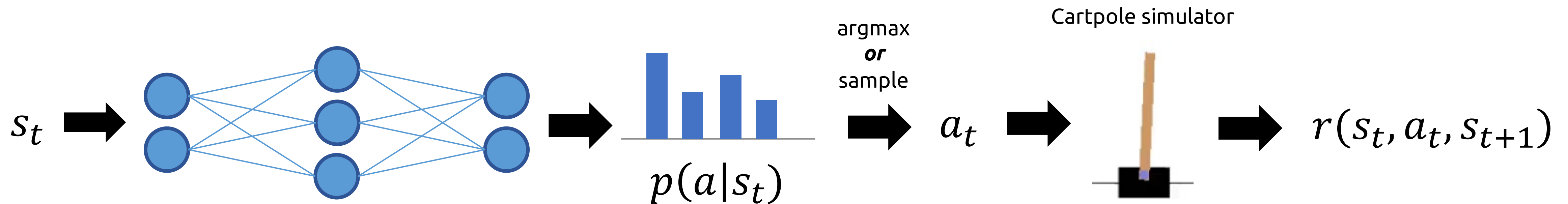
Training a Policy Network for Cart Pole

- Let's call this the discounted future reward function:
  - $D(s_t, a_t) = \sum_{i=t}^{T} \gamma^{i-t}\, r(s_i, a_i, s_{i+1})$
- This gives us a good idea for our ideal loss function: across every step of our simulated training episode $(\mathbf{s}, \mathbf{a})$, maximize the discounted future reward:
  - $L(\mathbf{s}, \mathbf{a}) = -\sum_{t=1}^{T} D(s_t, a_t)$
- Brilliant! Let's simulate some episodes, throw them at our favorite SGD optimizer, and call it a day :)

Not so fast...

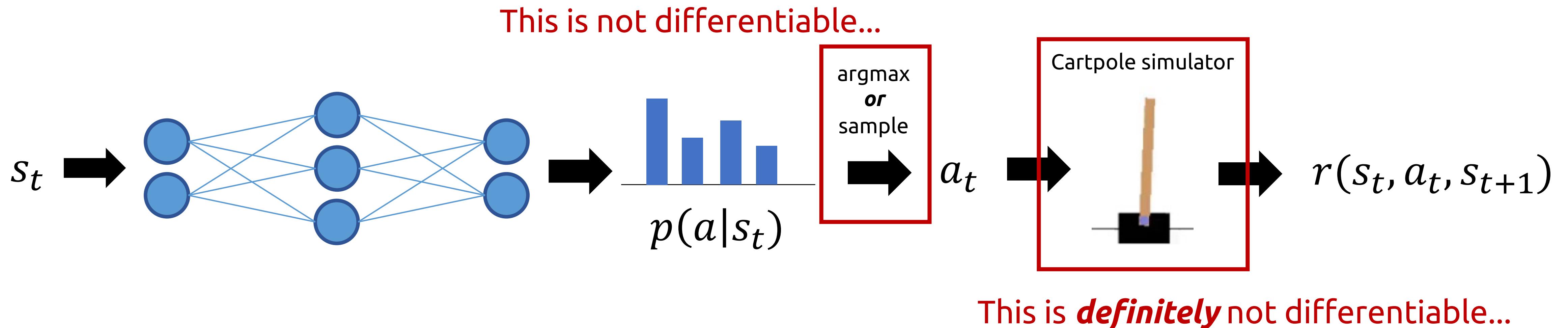
Do we anticipate any issues running SGD?

- Let's take a look at the computation graph for a single term of the discounted future reward function $D(s_t, a_t) = \sum_{i=t}^{T} \gamma^{i-t}\, r(s_i, a_i, s_{i+1})$



Not so fast...

- Let's take a look at the computation graph for a single term of the discounted future reward function $D(s_t, a_t) = \sum_{i=t}^{T} \gamma^{i-t}\, r(s_i, a_i, s_{i+1})$



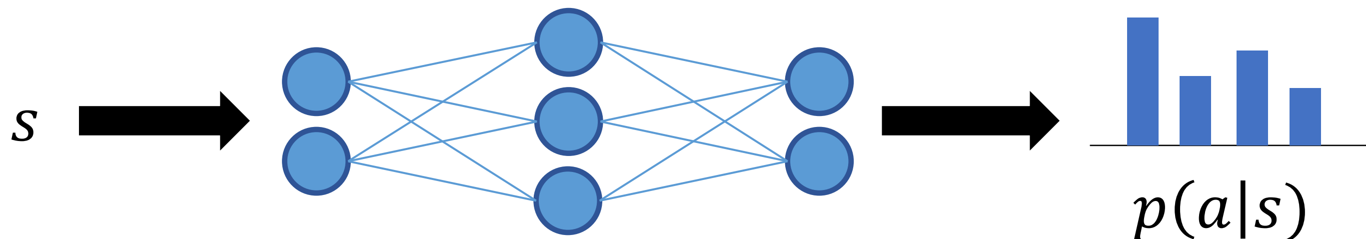
- Uh oh...it looks like we can't use SGD because we don't have an end-to-end differentiable function!

The Policy Gradient Theorem to the Rescue

- Fortunately, it turns out that we can get the behavior we want by running SGD with the following gradient:

$$-\sum_{t=1}^{T} \nabla \log p(a_t|s_t) D(s_t, a_t)$$

We only need the gradient of this part, which is our (fully differentiable) policy network!



Just like computing gradients through a classification network

Policy Gradient: Why It Works

$$-\sum_{t=1}^{T} \nabla \log p(a_t|s_t) D(s_t, a_t)$$

- It's possible to rigorously prove that this gradient does the right thing...
- ...but instead, we're going to focus on the ***intuition*** behind what it does

Policy Gradient: Why It Works

$$-\sum_{t=1}^{T} \nabla \log p(a_t|s_t)\, D(s_t, a_t)$$

- This part says "maximize the probability of taking this action"
- If the sequence of actions $\mathbf{a} = a_1 \ldots a_T$ from our episode were given by a ground truth demonstration, then this would be all we need.
- But they're not. So, some of these actions that we took in our episode might not be so good, so we shouldn't just blindly maximize them.

Policy Gradient: Why It Works

$$-\sum_{t=1}^{T} \nabla \log p(a_t|s_t)\, D(s_t, a_t)$$

- This part says "weight how much we maximize the probability of this action by how good that action was in the long term"
  - If it led to positive reward in the long term, we try to maximize the probability
  - If it led to zero reward in the long term, we leave the probability unchanged
  - If it led to ***negative*** reward in the long term, we try to ***minimize*** the probability

Policy Gradient: Why It Works

Any questions?

$$-\sum_{t=1}^{T} \nabla \log p(a_t|s_t) D(s_t, a_t)$$

- There are, in fact, many different approaches that fall under the umbrella of "policy gradient methods" and which look something like this
- This particular one is the simplest, and is known as ***REINFORCE***
  - No, it's not an acronym for anything. The authors of the original paper just thought that shouting their algorithm name in all-caps would be a good idea...

REINFORCE: Pseudo Code

Initialize model weights θ

Repeat until done (converge, time limit expired, etc.):

Run N episodes of environment simulation, each for T timesteps

For each episode

For $t = 1$ to $t = T$

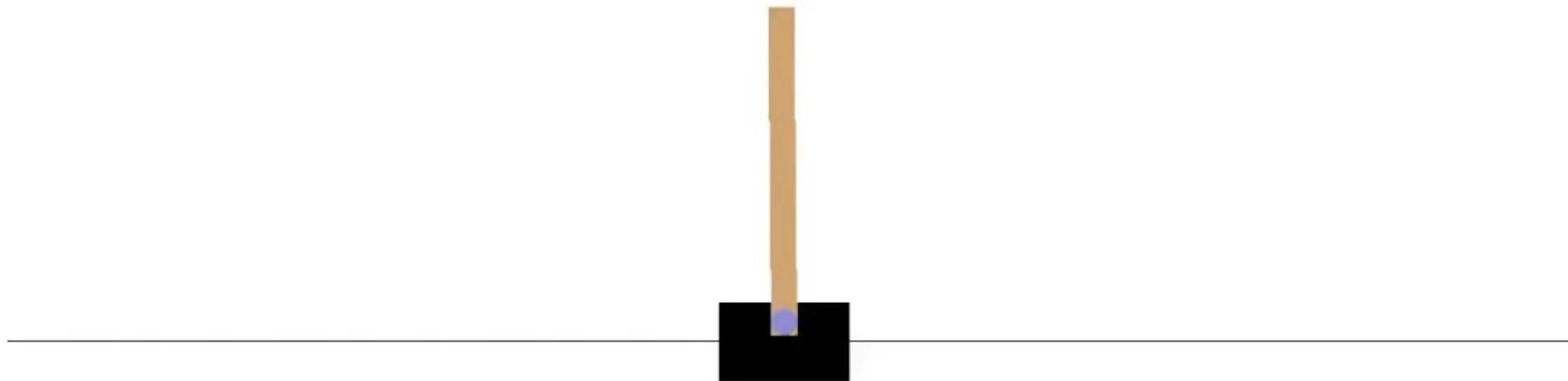
$$\theta \leftarrow \theta + \text{OptimizerStep}(\nabla \log p(a_t|s_t) D\ (s_t, a_t))$$

Your favorite optimizer (SGD, Adam, …)

Return θ

REINFORCE in action on Cart Pole

Episode= 1
Episode reward= 10.0
Average of last 500 rewards= 10.0
Average of last 100 rewards= 10.0



https://www.youtube.com/watch?v=qx-KNh0I4CM

Reinforce vs DQN

Pros

Cons

What are the pros and cons of using REINFORCE over Deep Q-Network?

Reinforce vs DQN

Pros

- Policy often easier to learn than Q function
- Automates explore vs. exploit tradeoff
  - Policy network starts off random and gradually becomes better as it is trained for more and more episodes
- Can learn stochastic policies
  - More naturalistic behavior
- In practice, can converge faster than DQN

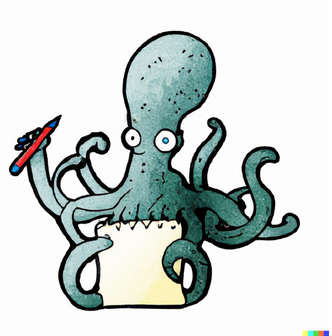
Cons

- Finds local optima more often than DQN...
- Unstable training
- Gradient updates only at end of each game (DQN updates after every step)

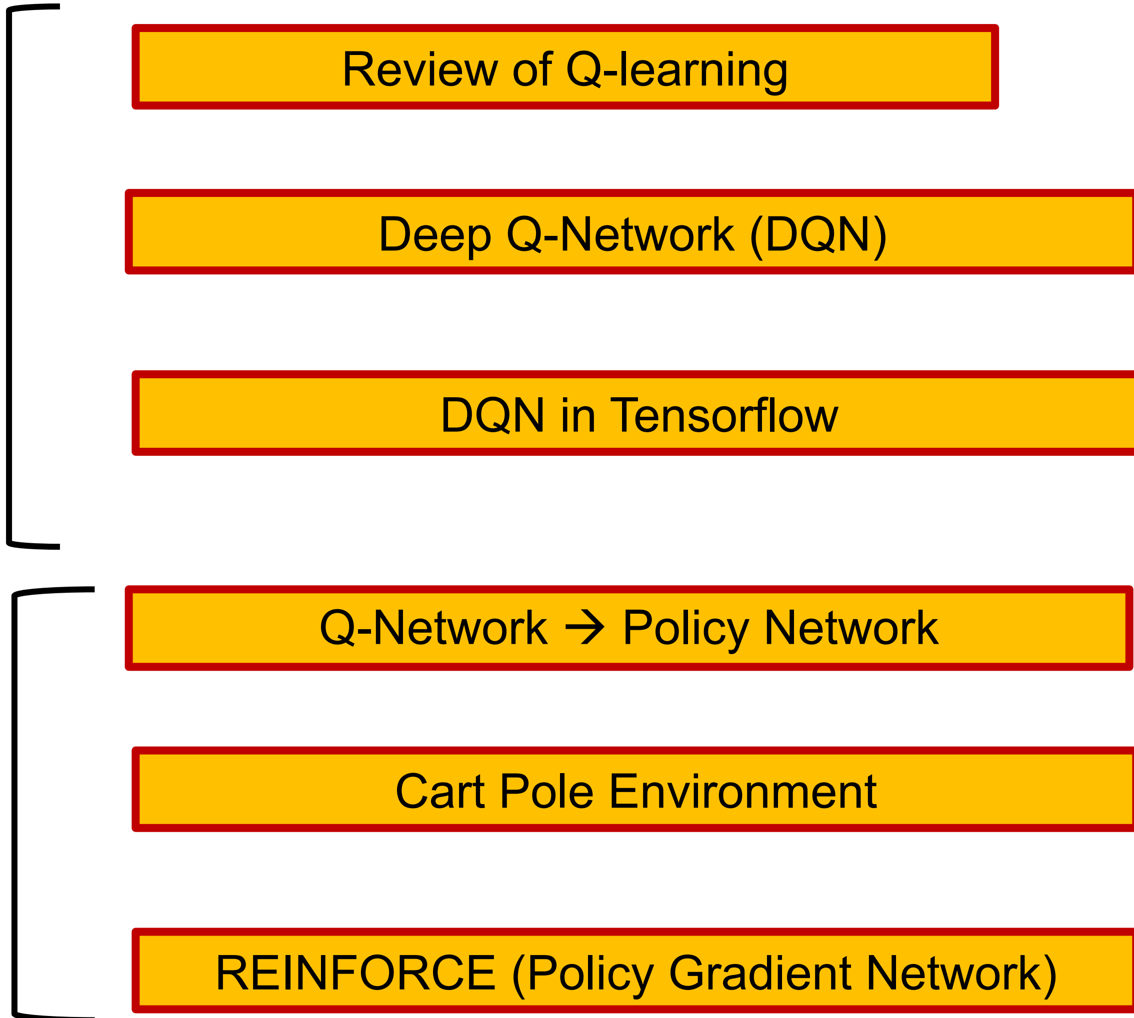
We'll see how to fix these two issues in the next lecture...

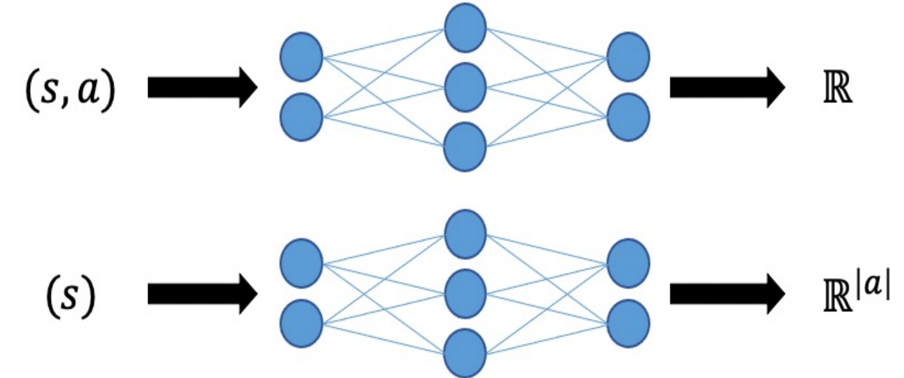
Recap

Deep Q-learning



Policy gradient learning





Cartpole simulator