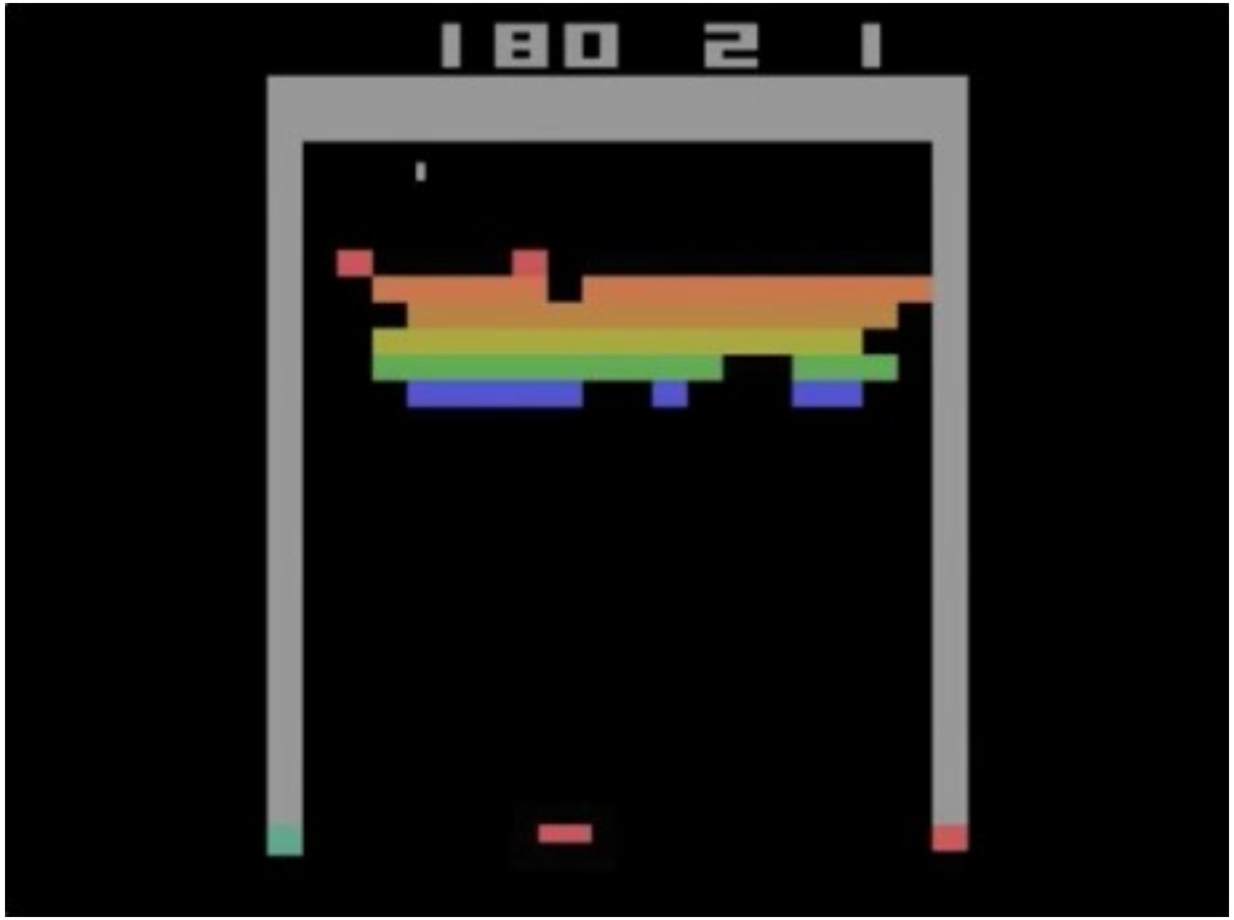
Policy Gradient Methods

Brown CSCI 1470/2470: Deep Learning

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?

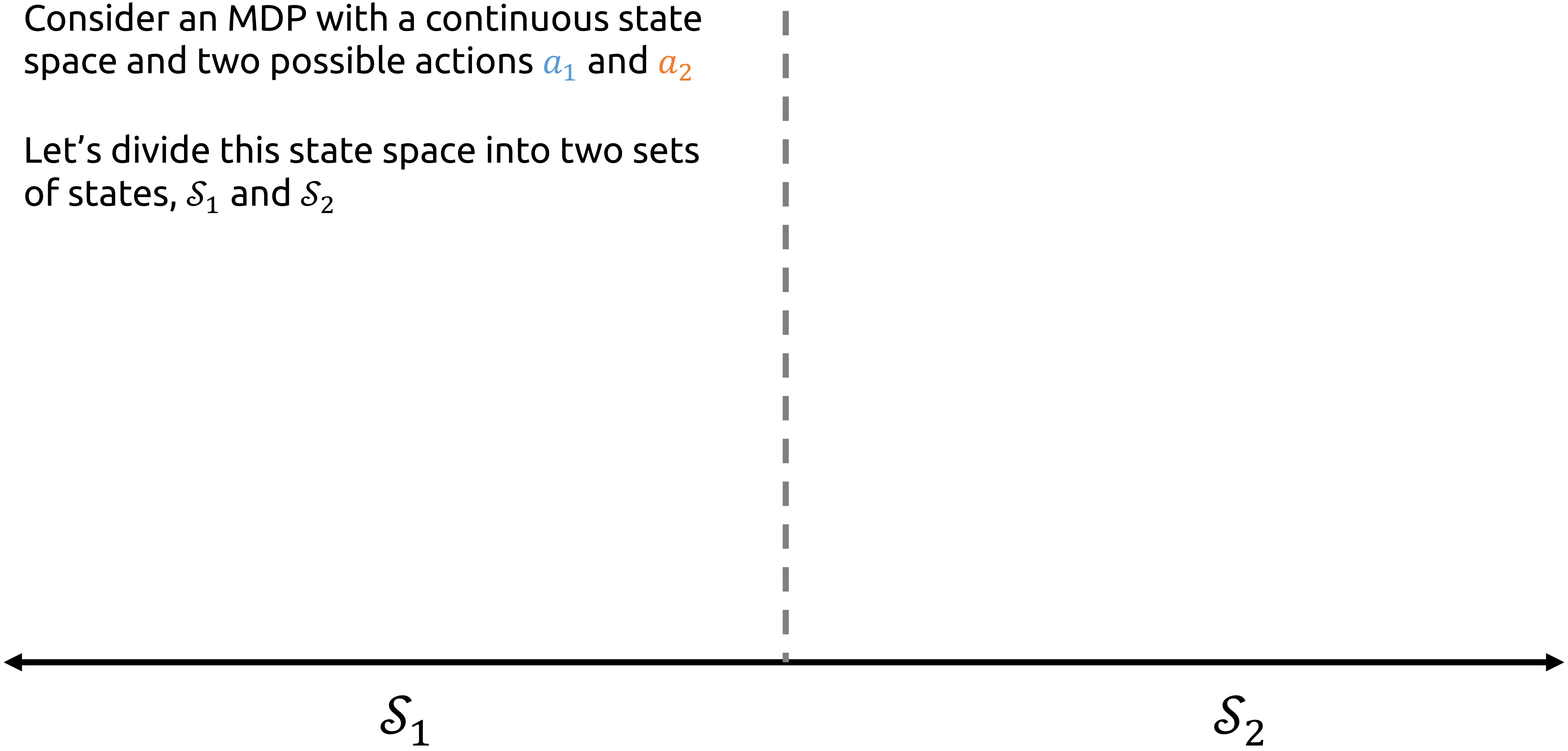


https://www.youtube.com/watch?v=TmPfTpjtdgg

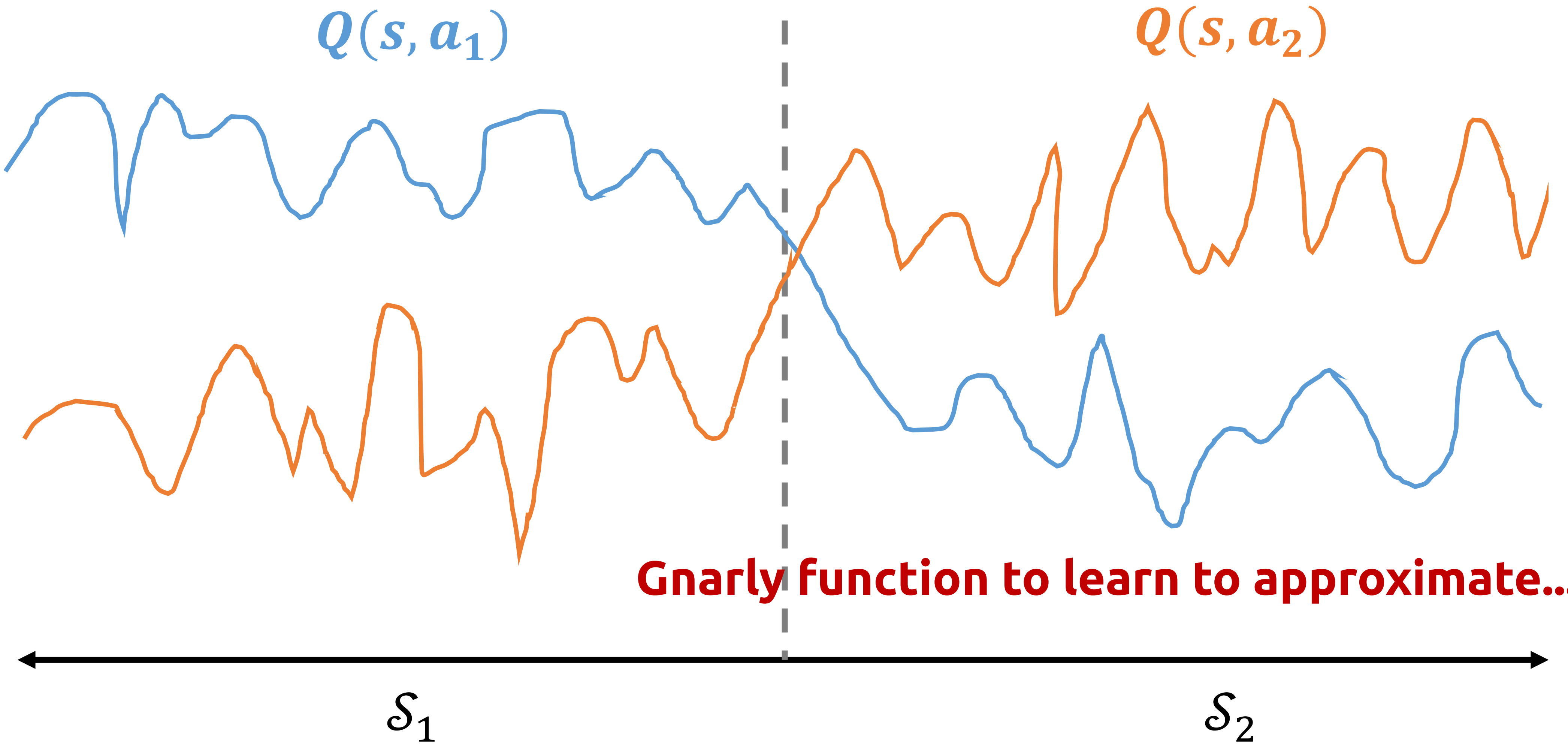
Q Functions can be complex…

- Consider an MDP with a continuous state space and two possible actions a_1 and a_2

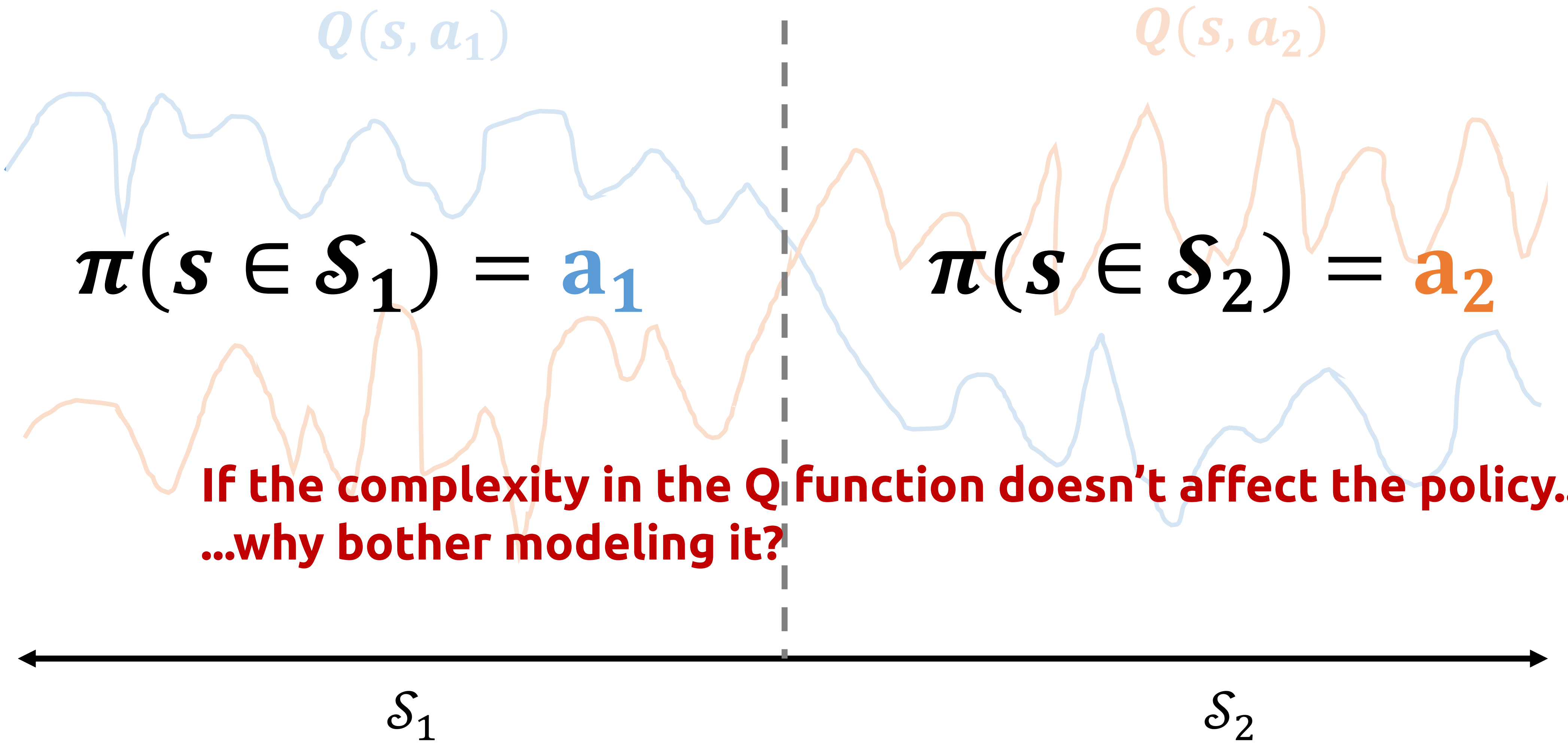
- Let's divide this state space into two sets of states, $\mathcal{S}_1$ and $\mathcal{S}_2$



Q Functions can be complex…



...but policies can still be simple



$$\boldsymbol{\pi}(\boldsymbol{s} \in \boldsymbol{\mathcal{S}}_1) = \mathbf{a}_1$$

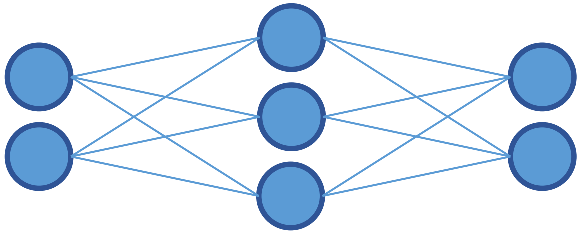
$$\boldsymbol{\pi}(\boldsymbol{s} \in \boldsymbol{\mathcal{S}}_2) = \mathbf{a}_2$$

If the complexity in the Q function doesn't affect the policy...
...why bother modeling it?

An Idea:

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

$(s,a) \rightarrow$

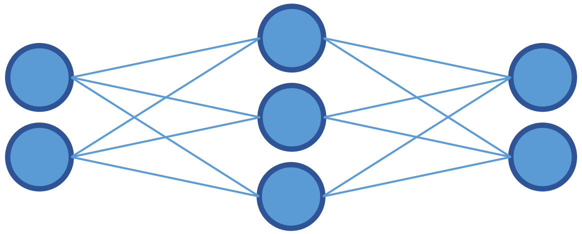


$\rightarrow Q(s,a)$

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

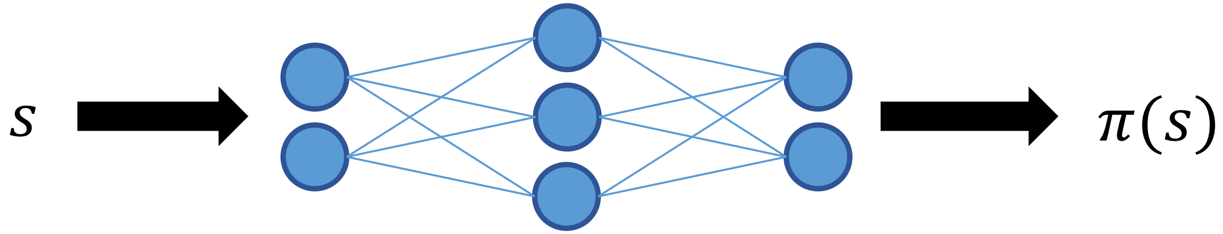
- …why don't we just directly learn a ***Policy Network?***
  - i.e. have a neural net that takes in a state and outputs an action

$s \rightarrow$

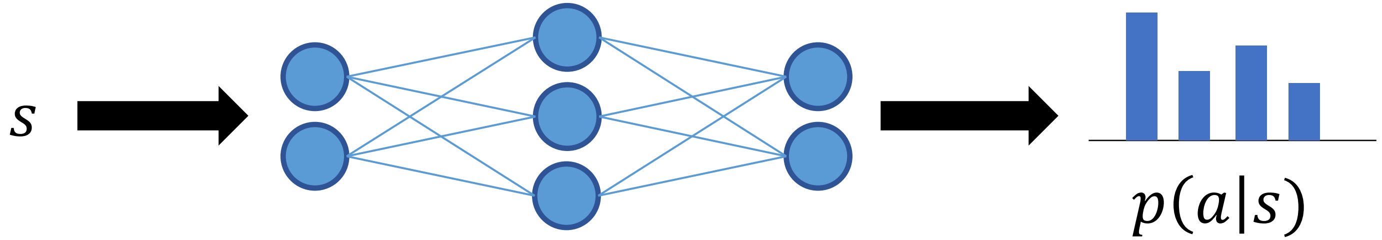


$\rightarrow \pi(s)$

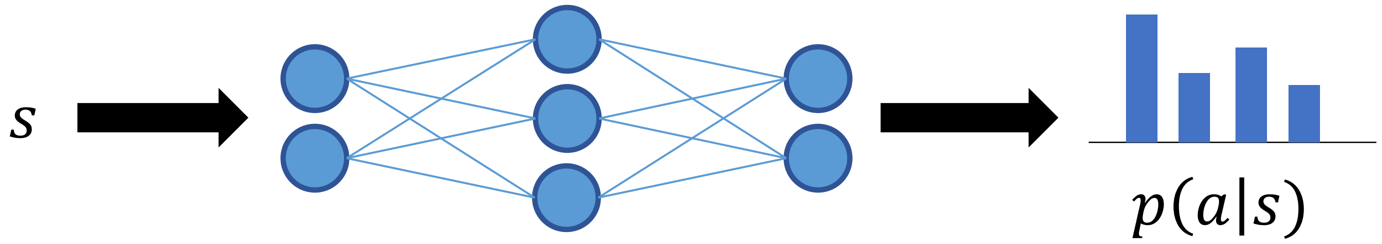
Policy Networks



- 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

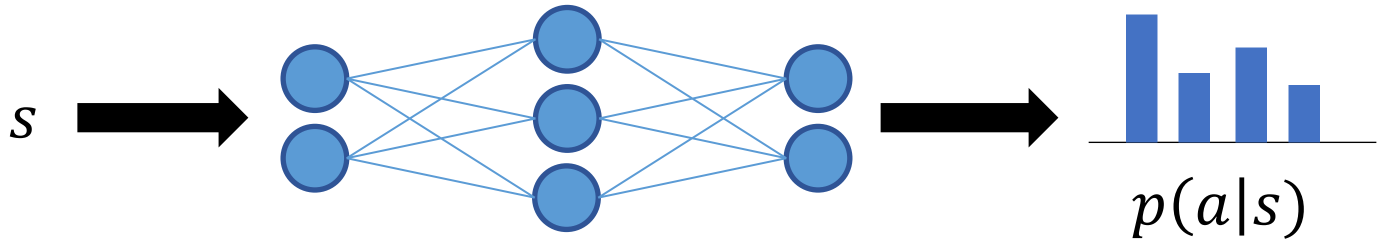


Using a Policy Network



- Q: How to get a discrete action out of this distribution?
- A: Two possibilities:
  1. $\pi(s) = \underset{a}{\operatorname{argmax}}\, p(a|s)$ → 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



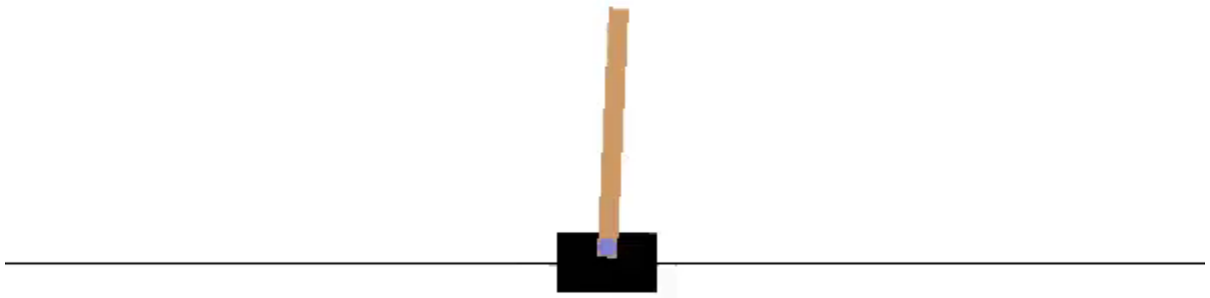
- How do we train a network like this?
- We can't just "adapt Q learning" somehow—this is a fundamentally different beast
- The study of how to learn policy networks lies at the core of all modern deep reinforcement learning research
- Family of learning algorithms known as ***Policy Gradient*** methods

- 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 a 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 ☺

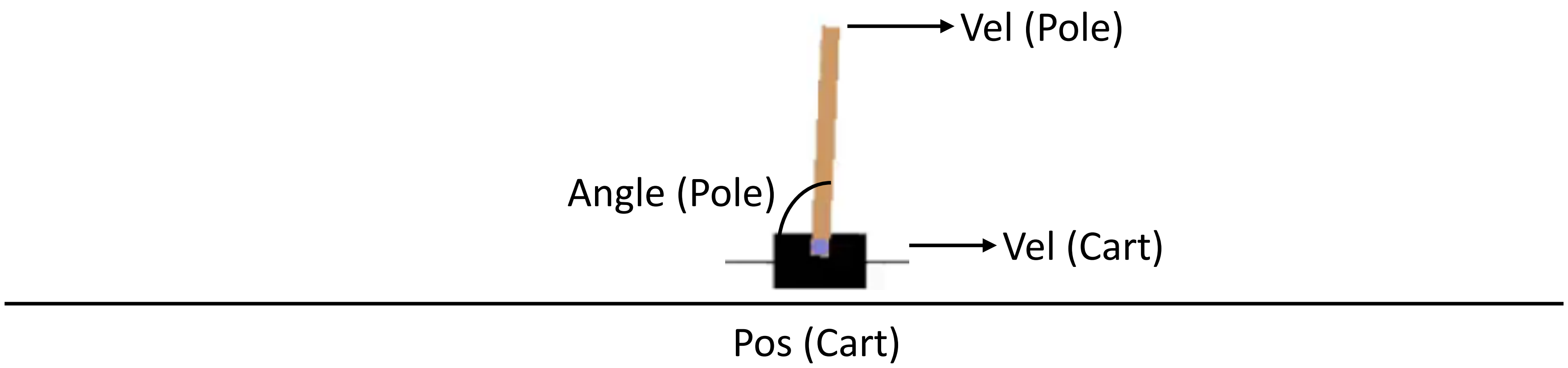


[OpenAI Gym]

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

Cart Pole MDP Formulation

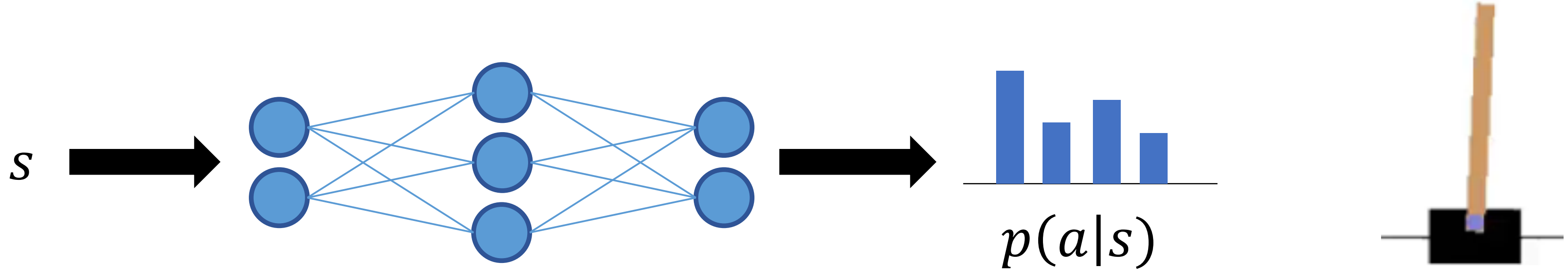
- State: cart position, cart velocity, pole angle, pole tip velocity



- 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

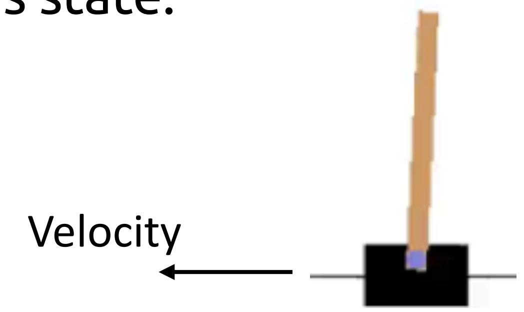
- Would be easy to do with supervised learning (i.e. if we had a ground-truth expert demonstration to follow)



- 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?

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

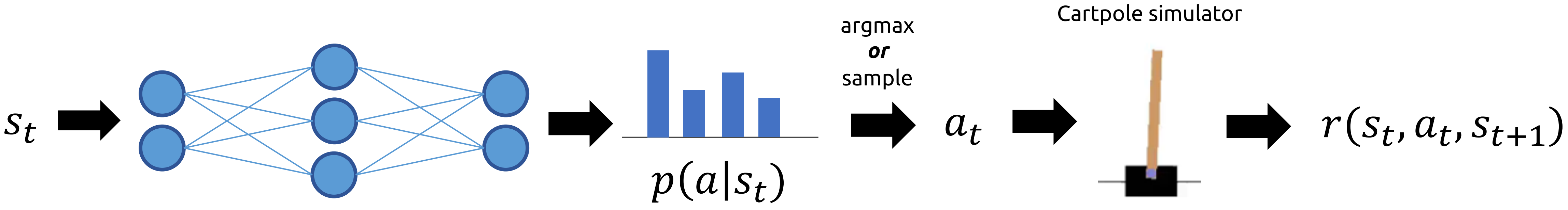
- Better loss function: maximize the expected future return that you'll get from taking an action (not the immediate reward)
  - i.e. $L(s_t, a_t) = \mathbb{E}[G_t \mid a_t]$
- What's another name for the expected future return?
  - The Q function! $L(s_t, a_t) = \mathbb{E}[G_t \mid a_t] = Q(s_t, a_t)$
- 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:
  - $D(s_t, a_t) = \sum_{i=t}^{T} \gamma^{i-1} r(s_i, a_i, s_{i+1})$
- 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 :)

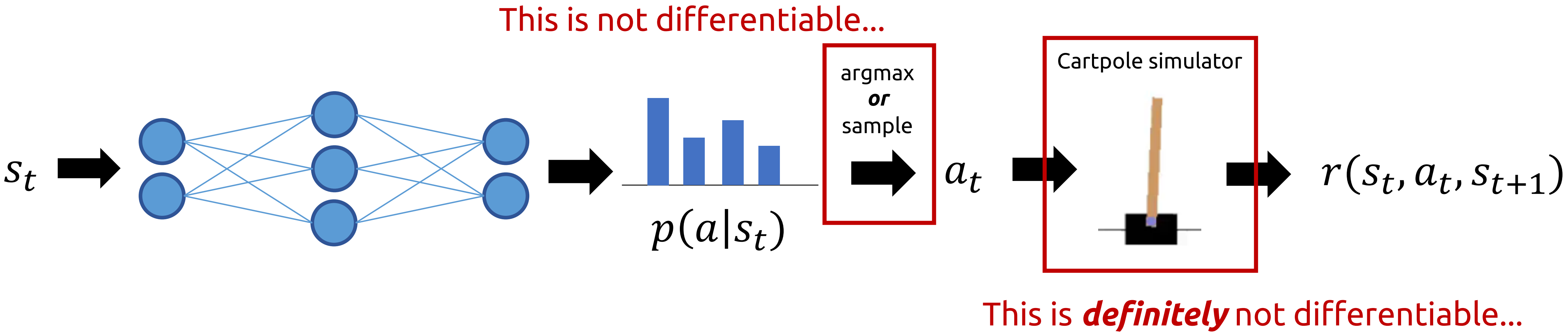
Not so fast...

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



Not so fast…

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



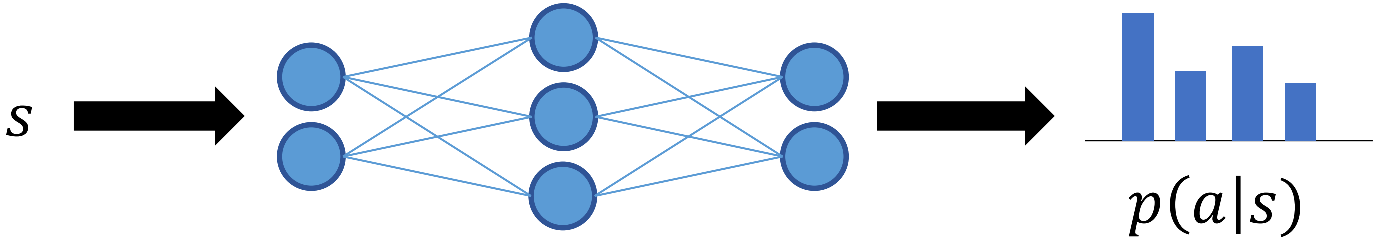
- Uh oh…it looks like we can't use SGD because we don't have an end-to-end differentiable function!

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:

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

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



Just likely 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 $\mathbf{a} = a_1 \dots a_T$ 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

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

- 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

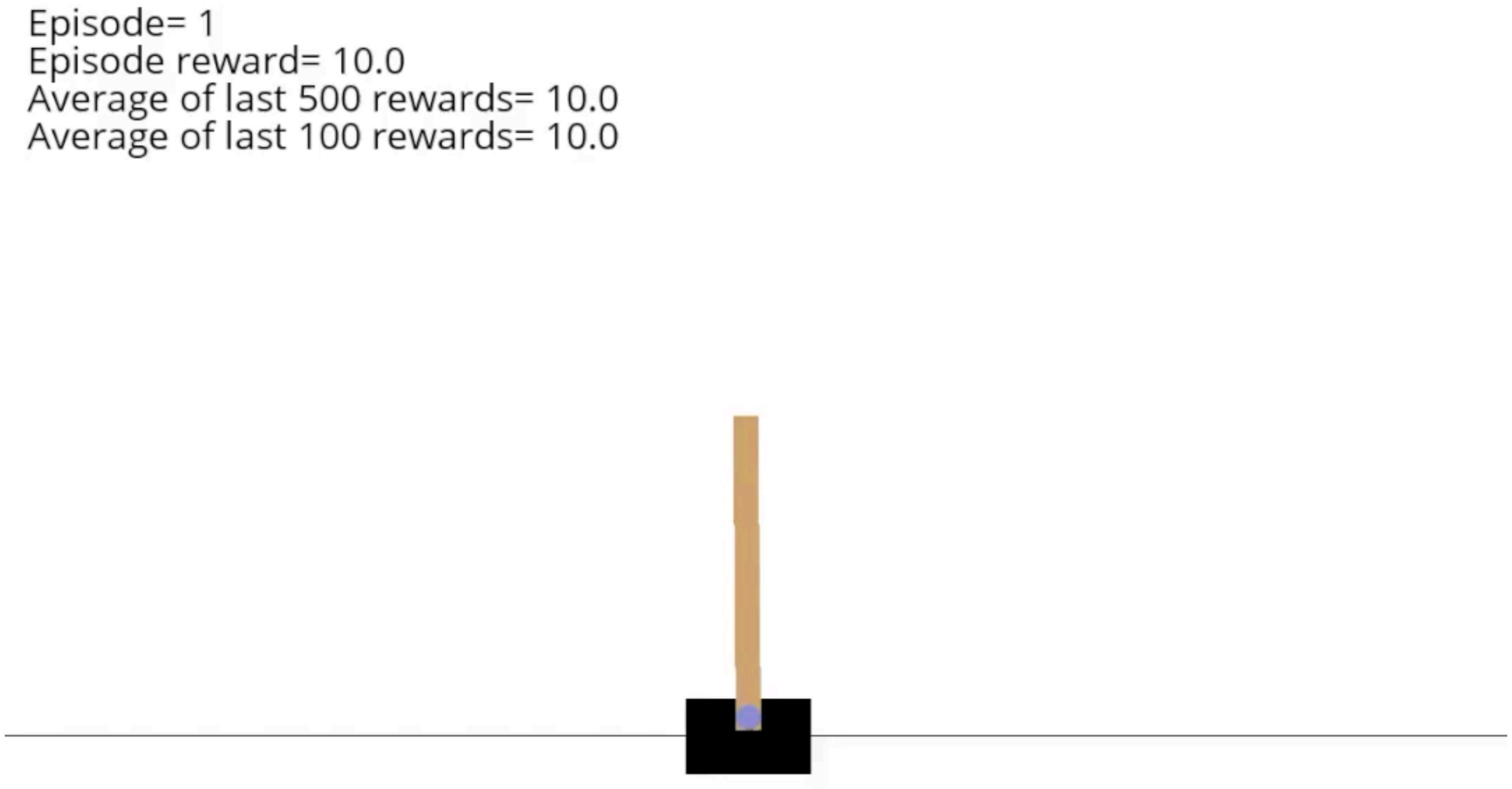
      For $t = 1$ to $t = T$

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

Return θ

Your favorite optimizer (SGD, Adam, …)

REINFORCE in action on Cart Pole



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

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…

Acknowledgments

- The following people contributed to these slides:
  - Josh Roy (UTA Fall ‘19)