# Reinforcement Learning

CMPT 726
Mo Chen
SFU Computing Science
4/11/2020

# Outline

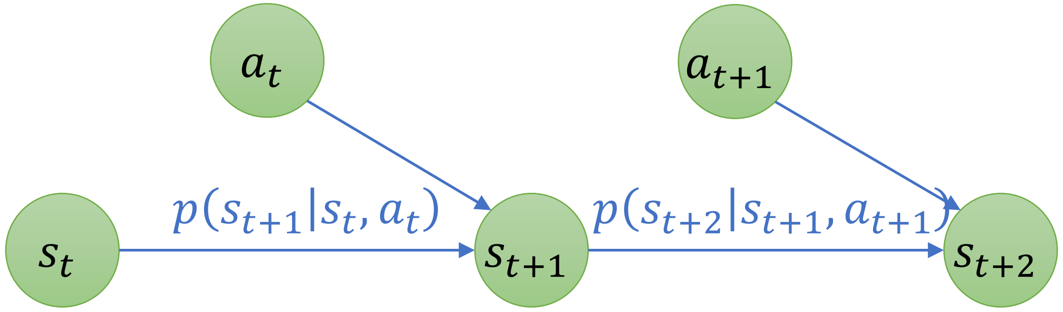
- Reinforcement learning problem setup
- Imitation learning
- Basic ideas in RL
- Model-free value-based RL
- Policy-based and actor-critic RL

# Outline

- **Reinforcement learning problem setup**
- Imitation learning
- Basic ideas in RL
- Model-free value-based RL
- Policy-based and actor-critic RL

# Markov Decision Process

- Probabilistic model of robots and other systems
- State: $s \in \mathcal{S}$, discrete or continuous
- Action (control): $a \in \mathcal{A}$, discrete or continuous
- Transition operator (dynamics): $\mathcal{T}$
  - $\mathcal{T}_{ijk} = p(s_{t+1} = i | s_t = j, a_t = k)$ ← a tensor (multidimensional array)



# State in MDPs and Reinforcement Learning

- State includes the internal states of an agent, but often also include
  - State of other agents
  - State of the environment
  - Sensor measurements

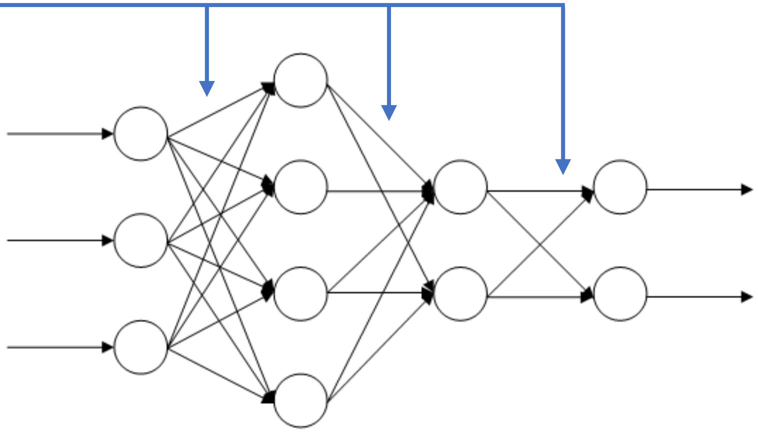
- Distinction between state and observation can be blurred

- In general, the state contains all variables other than actions that determine the next state through the transition probability $p(s_{t+1}|s_t, a_t)$

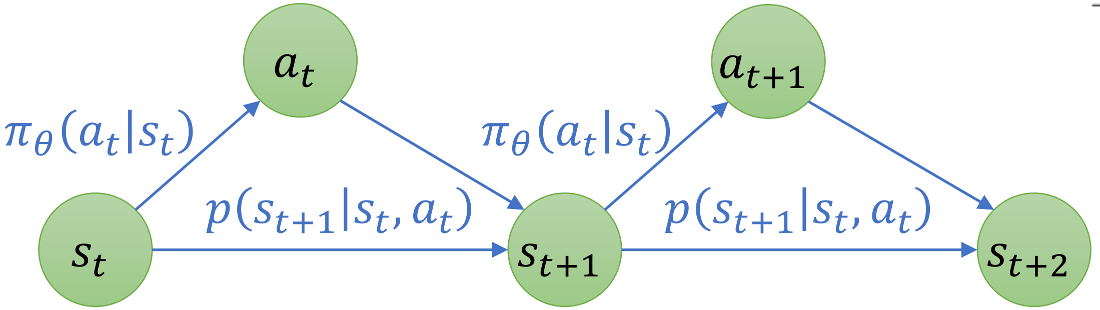
# Policy and Reward

- Control policy (feedback control): $\pi(a|s)$
  - Parametrized by $\theta$

$$\theta : \pi_\theta(a|s) \coloneqq p(a|s)$$



  - Can be stochastic: probability of applying action $a$ at state $s$

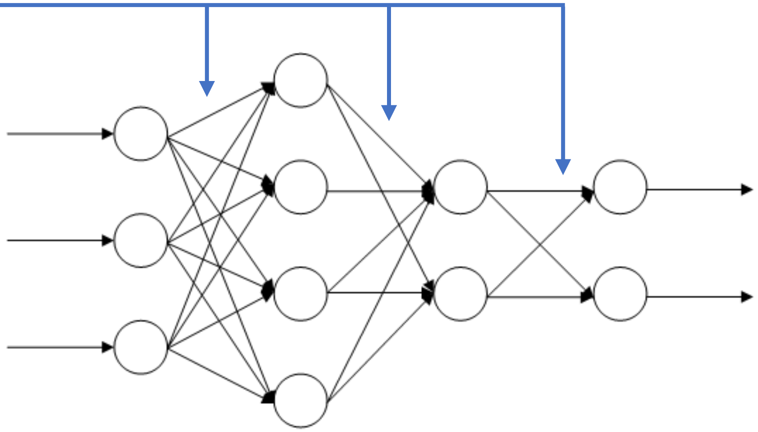


# Policy and Reward

- Control policy (feedback control): $\pi(a|s)$
  - Parametrized by $\theta$

$$\theta: \pi_\theta(a|s) \coloneqq p(a|s)$$

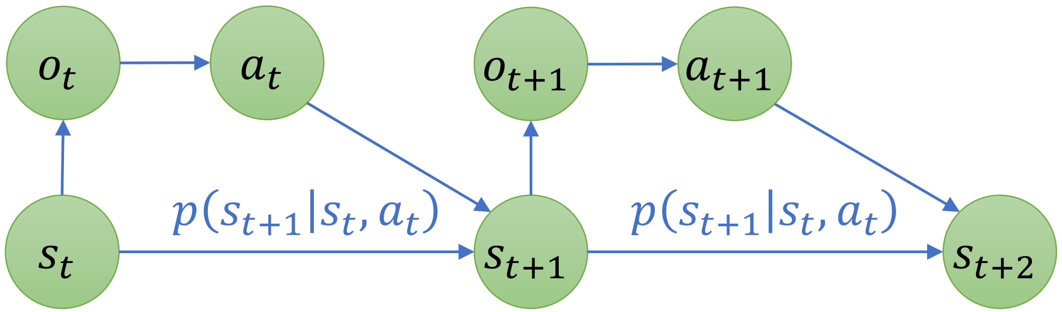
  - Can be stochastic: probability of applying action $a$ at state $s$



- Reward function: $r(s_t, a_t)$
  - Reward received for being at state $s_t$ and applying action $a_t$

# Extensions of Problem Setup

- Partially observability
  - Partially Observable Markov Decision Process (POMDP)
  - State not fully known; instead, act based on observations




  - Policy: $\pi_\theta(a|o)$
- In this class, state $s$ will be synonymous with observation $o$.

# Reinforcement Learning Objective

- Given: an MDP with state space $\mathcal{S}$, action space $\mathcal{A}$, transition probabilities $\mathcal{T}$, and reward function $r(s, a)$

- Objective: Maximize discounted sum of rewards (“return”)

$$\underset{\pi_\theta}{\text{maximize}}\, \mathbb{E} \sum_t \gamma^t r(s_t, a_t)$$

  - $\gamma \in (0,1]$: discount factor – larger roughly means “far-sighted”
    - Prioritizes immediate rewards
    - $\gamma < 1$ avoids infinite rewards; $\gamma = 1$ is possible if all sequences are finite

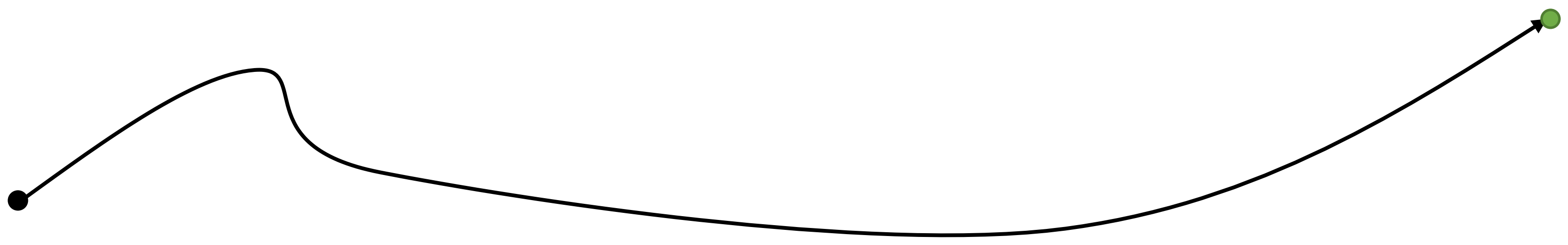
- Constraints: often implicit, and part of the objective
  - Subject to transition matrix $\mathcal{T}$ (system dynamics)

# Outline

- Reinforcement learning problem setup
- **Imitation learning**
- Basic ideas in RL
- Model-free value-based RL
- Policy-based and actor-critic RL

# Imitation Learning

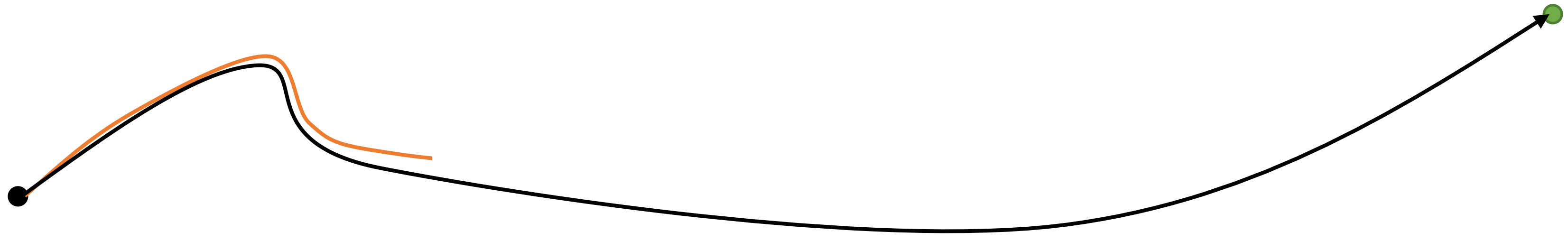
- Collect data through expert demonstration – sequence of states and actions, $\{s_0, a_0, s_1, a_1, \dots, s_{N-1}, a_{N-1}, s_N\}$
  - Note: Expert may not be solving $\underset{\pi}{\text{maximize}}\, \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$



- Learn $\pi_\theta(a_t|s_t)$ from data via regression

# Imitation Learning

- Collect data through expert demonstration – sequence of states and actions, $\{s_0, a_0, s_1, a_1, \dots, s_{N-1}, a_{N-1}, s_N\}$
  - Note: Expert may not be solving $\underset{\pi}{\text{maximize}}\, \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$



- Learn $\pi_\theta(a_t|s_t)$ from data via regression

# Imitation Learning

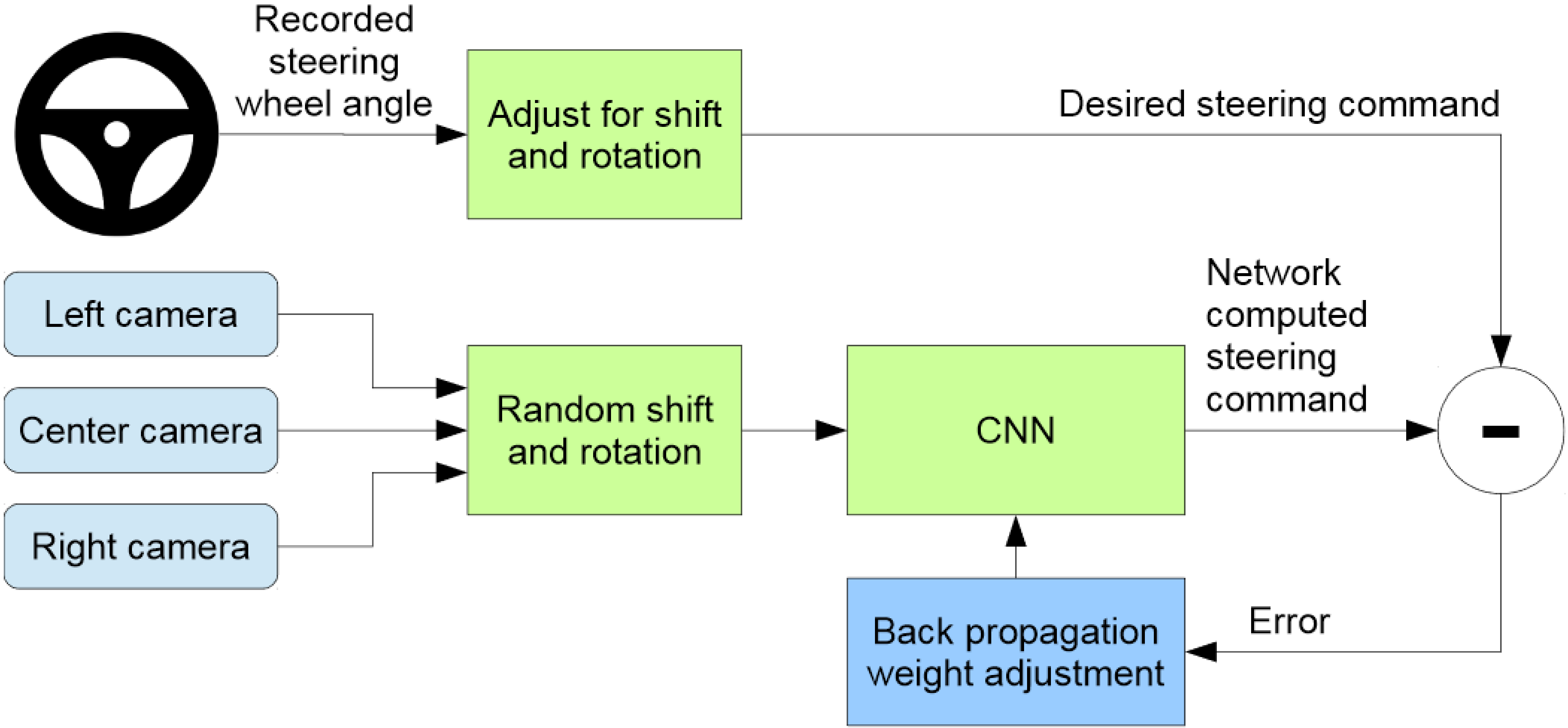
- Collect data through expert demonstration – sequence of states and actions, $\{s_0, a_0, s_1, a_1, \dots, s_{N-1}, a_{N-1}, s_N\}$
  - Note: Expert may not be solving $\underset{\pi}{\text{maximize}}\ \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$

- Learn $\pi_\theta(a_t|s_t)$ from data via regression

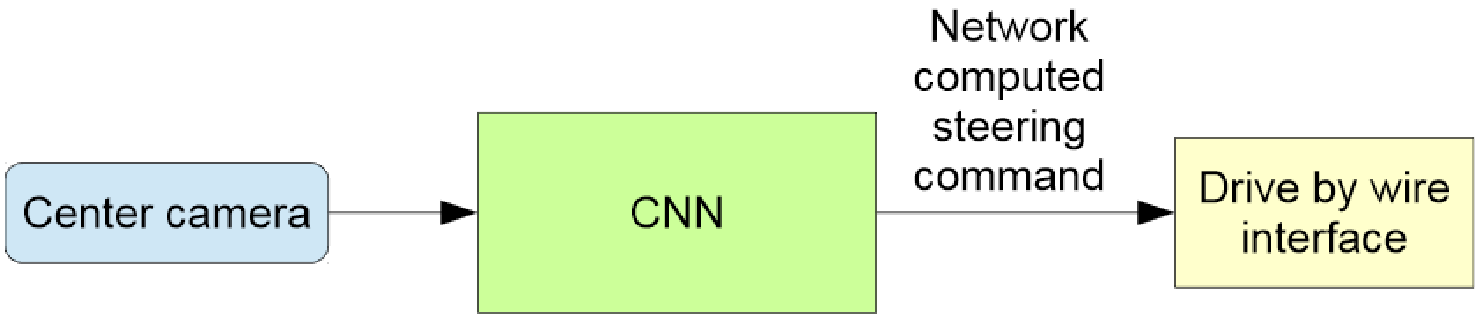
- Usually doesn't work due to "drift": small mistakes add up, and takes the system far from trained states
  - Sometimes, there can be "tricks" to make imitation learning work!

# Autonomous Driving Through Imitation

## Training:



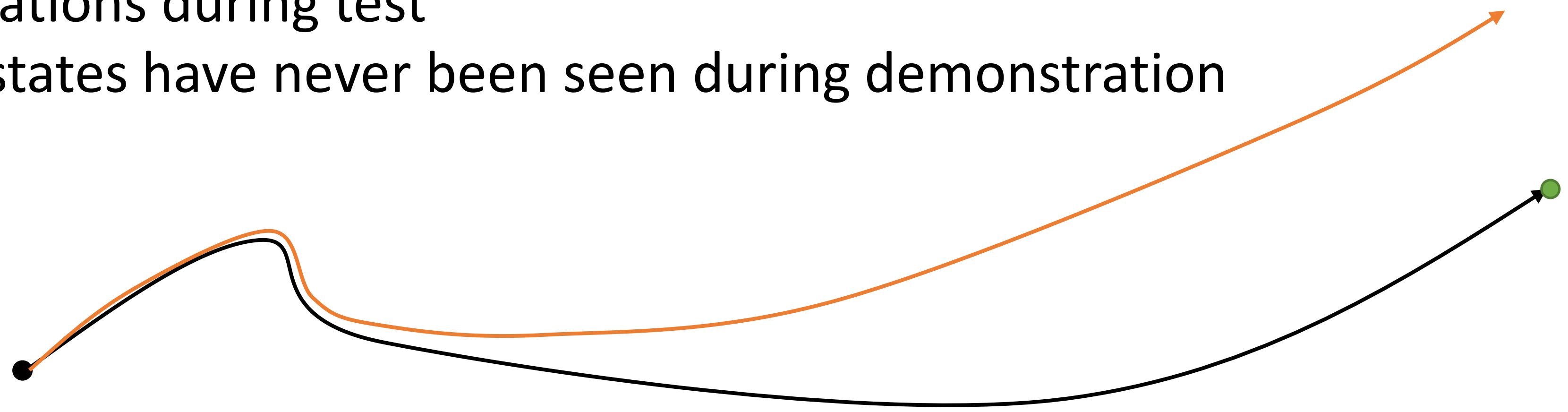

## Testing:




Bojarski et al. 2016. “End to End Learning for Self-Driving Cars,” CVPR 2016

# Dataset Aggregation

- Imitation learning drawback:
  - Distribution of observations in training is different from distribution of observations during test
  - Some states have never been seen during demonstration



- How to make the distributions equal?
  - Train perfect policy
  - Change data set → DAgger (Dataset Aggregation)

# Dataset Aggregation (DAgger) Algorithm

1. Train policy from some initial data, $\mathcal{D}_i = \{s_0, a_0, s_1, a_1, \dots, s_{N-1}, a_{N-1}, s_N\}$
2. Run policy to obtain new observations $\{s_{N+1}, s_{N+2}, \dots, s_{N+M}\}$
   - Note: time indices and states here may not continue from initial data
3. Use humans to label data by providing actions for new observations, $\{a_{N+1}, \dots, a_{N+M-1}\}$
   - This creates another data set, $\overline{\mathcal{D}}_i = \{s_{N+1}, a_{N+1}, s_{N+2}, a_{N+2} \dots, a_{N+M-1}, s_{N+M}\}$
4. Combine two datasets, $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \overline{\mathcal{D}}_i$
   - Go back to first step

# Challenges

- Non-Markovian behaviour
  - Perhaps augment state/observation space to include some history
  - Use neural networks that implicitly capture time series data: RNNs/LSTMs

- Unnatural data collection
  - Humans are probably not very good at collecting correction data in this manner

- Inconsistencies in human action

# Addressing Drift

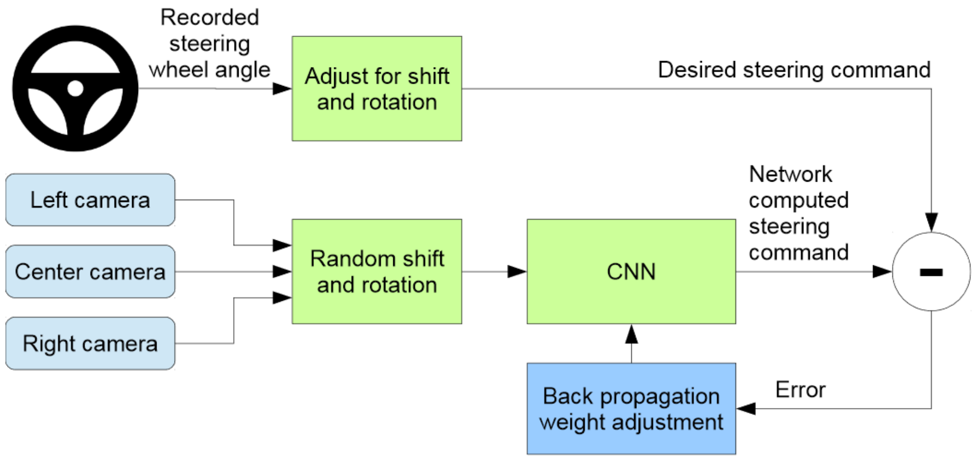

- Main goal: Teach system to correct errors
- Explicitly demonstrate corrections (DAgger)
- During demonstration, add noise to “force” mistakes, and see how humans correct them
- Ask humans to intentionally make mistakes
- Prior knowledge and heuristics
  - Example: Learn from stabilizing controller

# Imitation Learning Tricks

- Common neural network architectures
  - LSTM – since we have time-series data
  - CNN – usually in combination with LSTM, if the observations are images

- Simplify action space:
  - Driving example: action space simplified to {left, centre, right}

- Clever data collection
  - Driving example: side cameras




- Inverse reinforcement learning
  - Learn goal, instead of policy, from data
  - Use reinforcement learning to learn to achieve the same goal

# Imitation Learning Drawbacks

- Very small amount of data – challenging for training deep neural networks

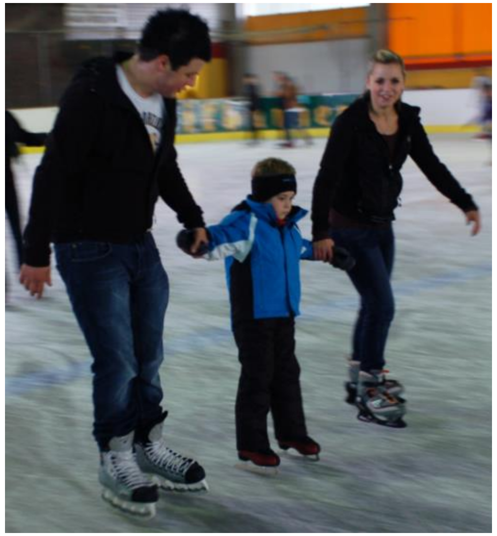
- Humans are not very good at providing some kinds of actions
  - Quadrotor motor speed
  - Non-humanoid machines

- Hard to perform better at tasks humans are not very good at

# Outline

- Reinforcement learning problem setup
- Imitation learning
- **Basic ideas in RL**
- Model-free value-based RL
- Policy-based and actor-critic RL

# Reinforcement Learning



- Humans can learn without imitation
  - Given goal/task
  - Try an initial strategy
  - See how well the task is performed
  - Adjust strategy next time

- Reinforcement learning agent
  - Given goal/task in the form of reward function $r(s, a)$
  - Start with initial policy $\pi_\theta(a|s)$; execute policy
  - Obtain sum of rewards, $\sum_t r(s_t, a_t)$
  - Improve policy by updating $\theta$, based on rewards

# Reinforcement Learning Objective

- Given: an MDP with state space $\mathcal{S}$, action space $\mathcal{A}$, transition probabilities $\mathcal{T}$, and reward function $r(s, a)$

- Objective: Maximize expected discounted sum of rewards ("return")

$$\underset{\pi_\theta}{\text{maximize}}\, \mathbb{E} \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$$

  - $\gamma \in (0,1]$: discount factor – larger roughly means "far-sighted"
    - Prioritizes immediate rewards
    - $\gamma < 1$ avoids infinite rewards; $\gamma = 1$ is possible if all sequences are finite

- Constraints: now incorporated into the reward function
  - Only constraint (usually implicit): subject to transition matrix $\mathcal{T}$ (system dynamics)

# RL vs. Other ML Paradigms

- No supervisor
  - But we will often draw inspiration from supervised learning
- Sequential data in time
- Reward feedback is obtained after a long time
  - Many actions combined together will receive reward
  - Actions are dependent on each other
- In robotics/health sciences/etc.: lack of data

# Reinforcement Learning Categories

- Model-based
  - Explicitly involves an MDP model
- Model-free
  - Does not explicitly involve an MDP model

- Value based
  - Learns value function, and derives policy from value function
- Policy based
  - Learns policy without value function
- Actor critic
  - Incorporates both value function and policy

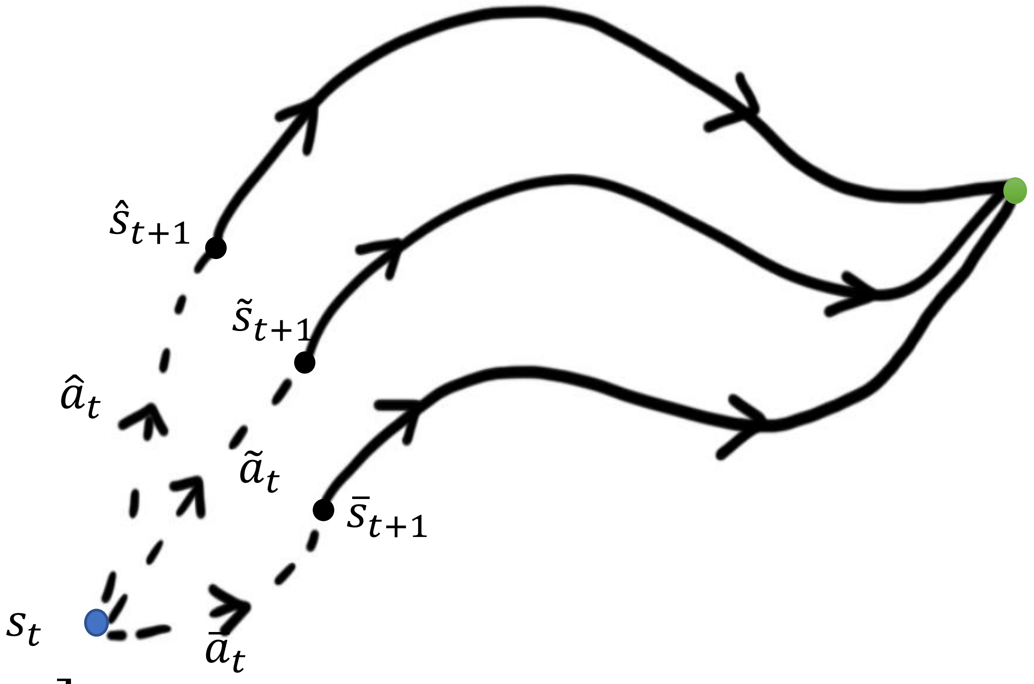
# Value Functions

- "**State-value function**": $V_\pi(s)$ -- expected return starting from state $s$ and following policy $\pi$
  - $V_\pi(s) = \mathbb{E}_{a_t \sim \pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s]$
  - Expectation is on the random sequence $\{s_0, a_0, s_1, a_1, \dots\}$

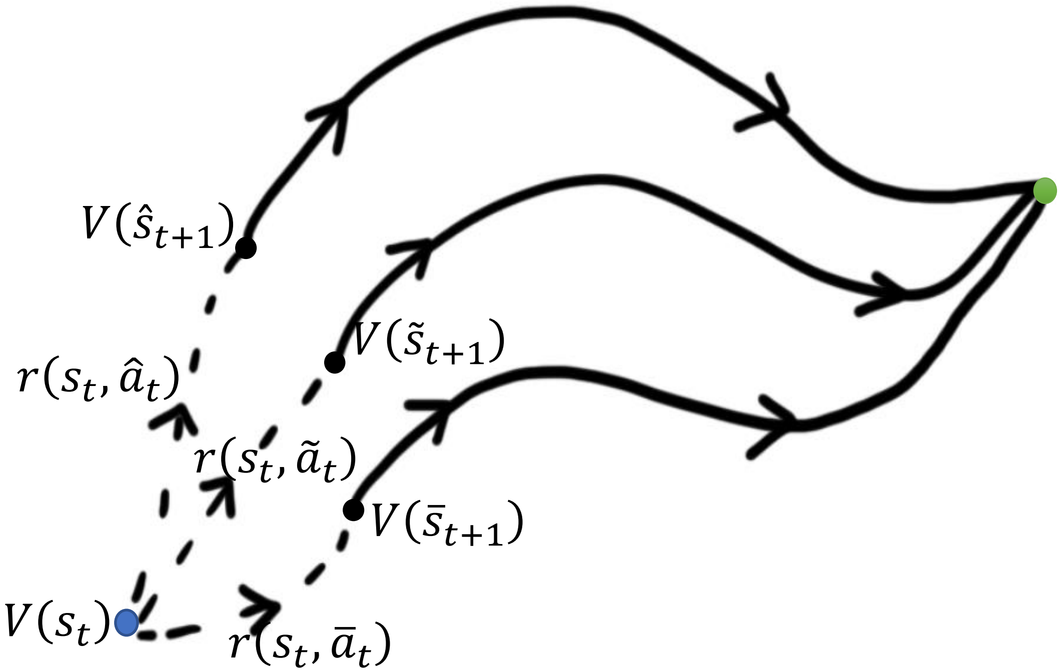
- "**Action-value function**", or "***Q*** **function**": $Q_\pi(s, a)$ -- expected return starting from state $s$, taking action $a$, and then following policy $\pi$
  - $Q_\pi(s, a) = \mathbb{E}_{a_t \sim \pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s, a_0 = a]$

# Principal of Optimality

- Optimal discounted sum of rewards:
  - $V_{\pi^*}(s) = \max_{\pi} \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s]$
- Dynamic programming:
  - $V_{\pi^*}(s) = \max_{a_t} \mathbb{E}[r(s_t, a_t) + \gamma V_{\pi^*}(s_{t+1})|s_t = s]$



# Principal of Optimality



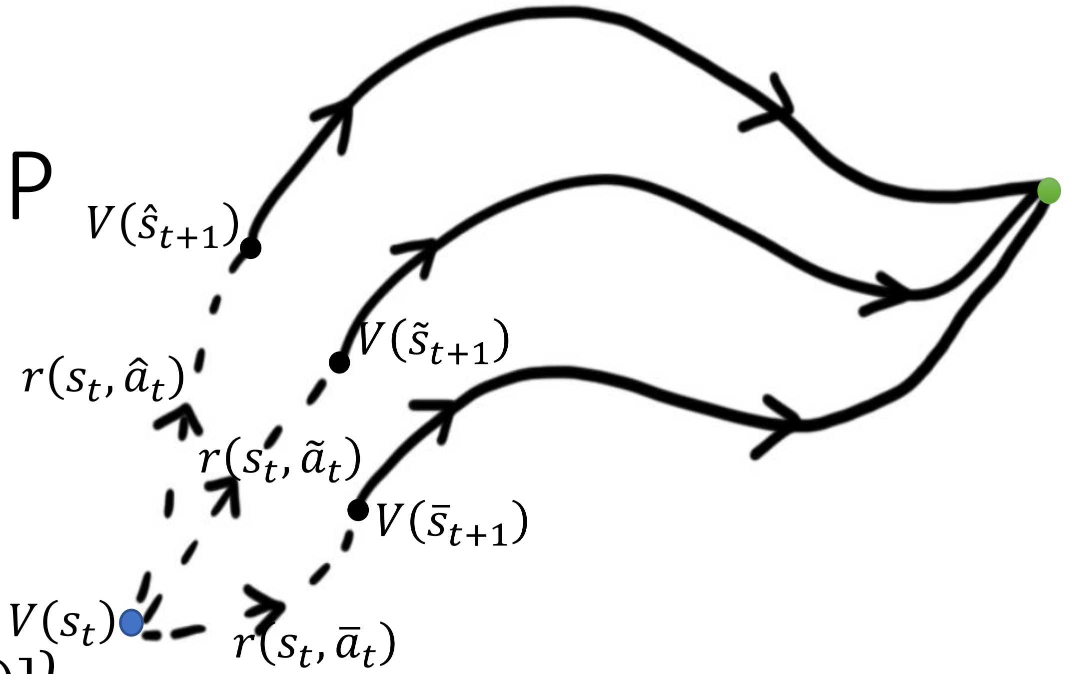

- Optimal discounted sum of rewards:
  - $V_{\pi^*}(s) = \max_{\pi} \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s]$
- Dynamic programming:
  - $V_{\pi^*}(s) = \max_{a_t} \mathbb{E}[r(s_t, a_t) + \gamma V_{\pi^*}(s_{t+1}) | s_t = s]$
  - $Q_{\pi^*}(s, a) = \mathbb{E}[r(s_t, a_t) + \gamma V_{\pi^*}(s_{t+1}) | s_t = s, a_t = a]$

- Actually, recurrence is true even without maximization
  - $V_{\pi}(s) = \mathbb{E}[r(s_t, \overset{\pi(s_t)}{a_t}) + \gamma V_{\pi}(s_{t+1}) | s_t = s]$
  - $Q_{\pi}(s, a) = \mathbb{E}[r(s_t, a_t) + \gamma V_{\pi}(s_{t+1}) | s_t = s, a_t = a]$

# Basic Properties of Value Functions

- $V_{\pi^*}(s) = \max_{\pi} V_{\pi}(s)$
- $Q_{\pi^*}(s,a) = \max_{\pi} Q_{\pi}(s,a)$
- $V_{\pi^*}(s) = \max_{a} Q_{\pi^*}(s,a)$
- For now, value functions are stored in multi-dimensional arrays
- DP leads to deterministic policies – we will come back to stochastic policies

# Optimizing the RL Objective via DP

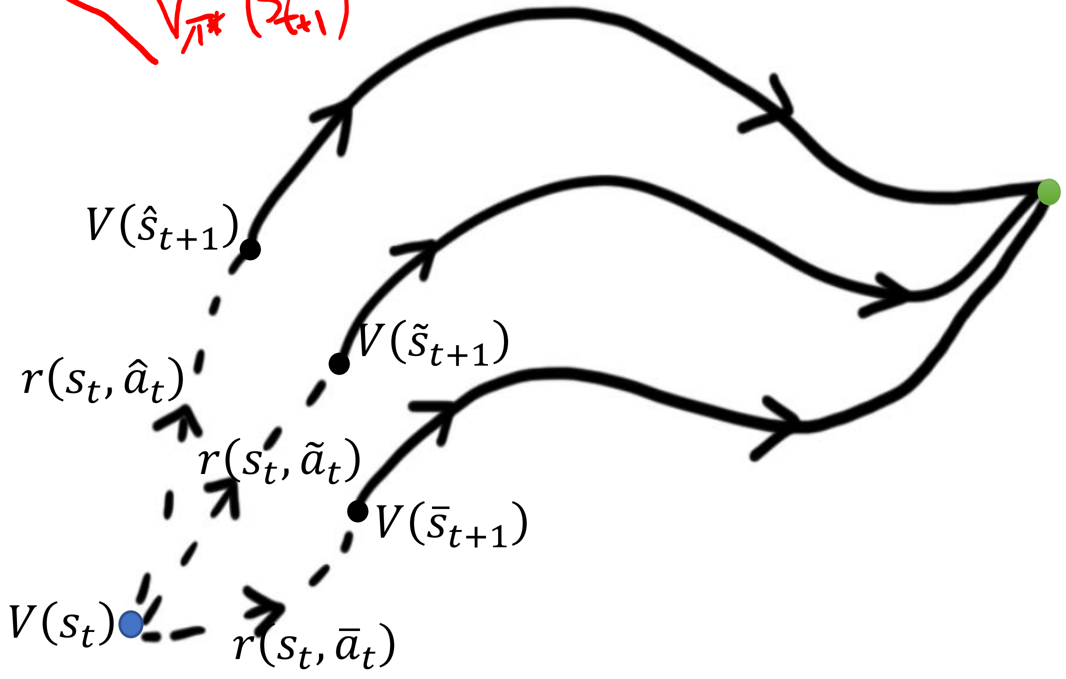



- State-value function
  - $V_{\pi^*}(s) = \max_{a_t} \mathbb{E}[r(s_t, a_t) + \gamma V(s_{t+1}) | s_t = s]$
  - $V_{\pi^*}(s) = \max_{a} \{r(s, a) + \gamma \mathbb{E}[V(s_{t+1}) | s_t = s]\}$
  - $V_{\pi^*}(s) = \max_{a} \{r(s, a) + \gamma \sum_{s'} [p(s' | s, a) V_{\pi^*}(s')]\}$
  - "**Bellman backup**": $V(s) \leftarrow \max_{a} \{r(s, a) + \gamma \sum_{s'} [p(s' | s, a) V(s')]\}$
    - This is done for all $s$
    - Iterate until convergence

- Optimal policy: $a = \arg\max_{a'} \{r(s, a') + \gamma \sum_{s'} [p(s' | s, a') V(s')]\}$
  - Deterministic

# Optimizing the RL Objective via DP

- Action-value function
  - $Q_{\pi^*}(s,a) = \mathbb{E}\left[r(s_t,a_t) + \gamma \max_{a_{t+1}} Q_{\pi^*}(s_{t+1},a_{t+1}) \,|s_t = s, a_t = a\right]$
  - $Q_{\pi^*}(s,a) = r(s,a) + \gamma \mathbb{E}\left[\max_{a_{t+1}} Q_{\pi^*}(s_{t+1},a_{t+1}) \,|s_t = s, a_t = a\right]$
  - $Q_{\pi^*}(s,a) = r(s,a) + \gamma \sum_{s'} [p(s'|s,a) V_{\pi^*}(s')]$ $V_{\pi^*}(s_{t+1})$
  - "Bellman backup":
    - $V(s) \leftarrow \max_a Q(s,a)$
    - $Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} [p(s'|s,a) V(s')]$
    - This is done for all $s$ and all $a$
    - Iterate until convergence




- Optimal policy: $a = \arg\max_{a'} Q(s,a')$
  - Deterministic

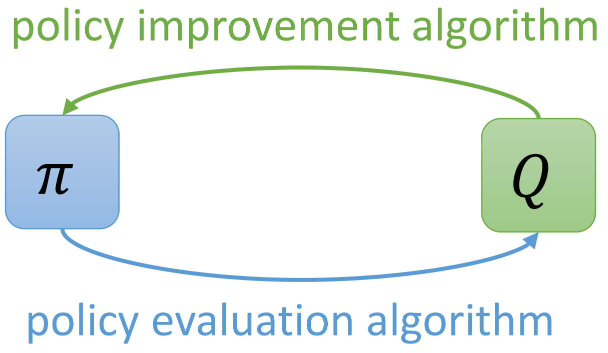
# Approximate Dynamic Programming

- Use a function approximator (eg. neural network) $\hat{V}(s; w)$, where $w$ are weights, to approximate $V$
  - $V(s)$ is no longer stored at every state
  - Weights $w$ are updated using Bellman backups
- Basic algorithm: (We will learn about other variants too)
  - Sample some states, $\{s_i\}$
  - For each $s_i$, generate $\tilde{V}(s_i) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} \left[ p(s' | s_t, a) \hat{V}(s'; w) \right] \right\}$
  - Using $\{s_i, \tilde{V}(s_i)\}$, update weights $w$ via regression (supervised learning)

# Generalized Policy Evaluation and Policy Improvement

- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use policy $\pi$ to update $V$ or $Q$: $a = \pi(s)$

DP
  - $V(s) \leftarrow r(s,a) + \gamma \sum_{s'}[p(s'|s,a)V(s')]$
  - $Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'}[p(s'|s,a)V(s')]$
  - In general, any policy evaluation algorithm

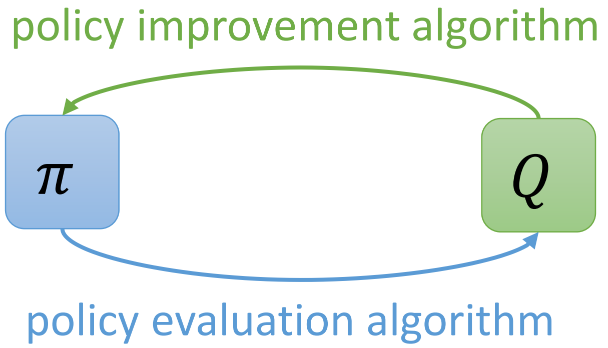



- Use $V$ or $Q$ to update policy $\pi$:

DP
  - Given $V(\mathrm{s}), \pi(s) = \arg\max_a \{r(s,a) + \gamma \sum_{s'}[p(s'|s,a)V(s')]\}$
  - Given $Q(s,a), \pi(s) = \arg\max_a Q(s,a)$
  - In general, any policy improvement algorithm

# Convergence

- At convergence, the following are simultaneously satisfied:
  - $V(s) = r(s,a) + \gamma \sum_{s'} [p(s'|s,a)V(s')]$
  - $\pi(s) = \arg\max_{a'} \{r(s,a') + \gamma \sum_{s'} [p(s'|s,a')V(s')]\}$
- This is the principle of optimality
- Therefore, the value function and policy are optimal



# Key Definitions So Far

- "**State-value function**": $V_\pi(s)$ -- expected return starting from state $s$ and following policy $\pi$
  - $V_\pi(s) = \mathbb{E}_{a_t \sim \pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s]$
  - Expectation is on the random sequence $\{s_0, a_0, s_1, a_1, \dots\}$

- "**Action-value function**", or "***Q*** **function**": $Q_\pi(s, a)$ -- expected return starting from state $s$, taking action $a$, and then following policy $\pi$
  - $Q_\pi(s, a) = \mathbb{E}_{a_t \sim \pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,|s_0 = s, a_0 = a]$

# Key Definitions So Far

- "**Value iteration**": The process of iteratively updating value function
  - With DP, we only need to keep track of value function $V$ or $Q$, and the policy $\pi$ is implicit – determined from value function
- "**Policy iteration**": The process of iteratively updating policy
  - This is done implicitly with Bellman backups
- "**Greedy policy**": the policy obtained from choosing the best action based on the current value function
  - If the value function is optimal, the greedy policy is optimal

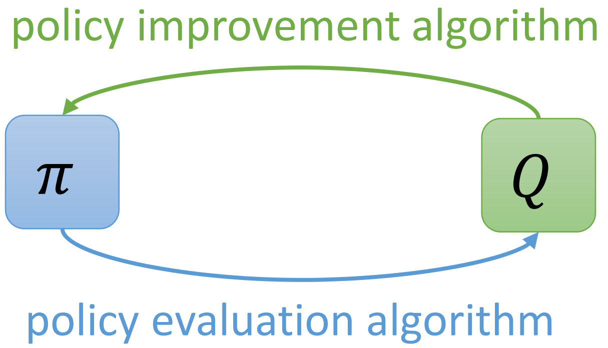
# Towards Model-Free Learning

- Policy evaluation
  - Monte-Carlo (MC) Sampling
  - Temporal-difference (TD)

- Policy improvement
  - $\epsilon$-greedy policies

# Actually Learning from Experience: Monte-Carlo Policy Evaluation

- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use policy $\pi$ to update $V$: $a = \pi(s)$
  - Apply $\pi$ to obtain trajectory $\{s_0, a_0, s_1, a_1, \ldots\}$
  - Compute return: $R \coloneqq \sum \gamma^t r(s_t, a_t)$
  - Repeat for many episodes to obtain empirical mean
    - "**Episode**": a single "try" that produces a single trajectory




- Use $V$ or $Q$ to update policy $\pi$

# Actually Learning from Experience: Monte-Carlo Policy Evaluation

- To obtain empirical mean, we record $N(s)$, # of times $s$ is visited for every state
  - Start at $N(s) = 0$ for all $s$
  - Note that this means storing $N$ (and $S$ below) at every state

- First-visit MC Policy Evaluation:
  - At the first time $t$ that $s$ is visited in an episode,
    - Increment $N(s) \leftarrow N(s) + 1$
    - Record return $R(s) \leftarrow R(s) + \sum \gamma^t r(s_t, a_t)$
    - Repeat for many episodes
  - Estimate value: $V(s) = \frac{R(s)}{N(s)}$

# Actually Learning from Experience: Monte-Carlo Policy Evaluation

- To obtain empirical mean, we record $N(s)$, # of times $s$ is visited for every state
  - Start at $N(s) = 0$ for all $s$
  - Note that this means storing $N$ (and $S$ below) at every state

- Every-visit MC Policy Evaluation:
  - Every time $t$ that $s$ is visited in an episode,
    - Increment $N(s) \leftarrow N(s) + 1$
    - Record return $R(s) \leftarrow R(s) + \sum \gamma^t r(s_t, a_t)$
    - Repeat for many episodes
  - Estimate value: $V(s) \approx \frac{R(s)}{N(s)}$

# Incremental Updates

- Instead of estimating $V_\pi(s)$ after many episodes, we can update it incrementally after every episode after receiving return $R$
  - $N(s) \leftarrow N(s) + 1$
  - $V(s) \leftarrow V(s) + \frac{1}{N(s)}\big(R - V(s)\big)$

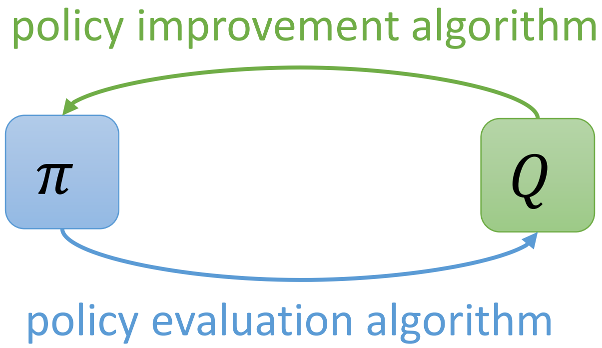
- More generally, we can weight the second term differently
  - $V(s) \leftarrow V(s) + \alpha\big(R - V(s)\big)$

"step size" "target"

# Monte-Carlo Policy Evaluation

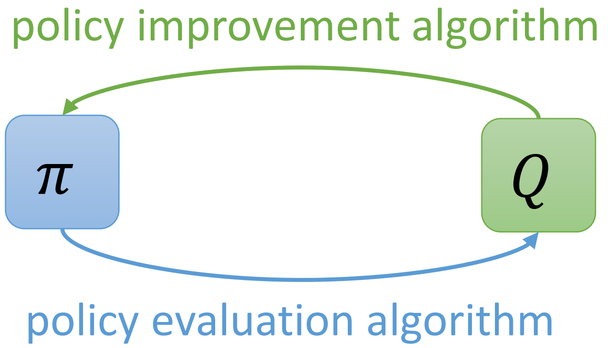
- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use policy $\pi$ to update $V$: $a = \pi(s)$
  - MC policy evaluation provides estimate of $V_\pi$
  - Many episodes are needed to obtain accurate estimate
  - Model-free with MC!

- Use $V$ or $Q$ to update policy $\pi$
  - Greedy policy?



# Monte-Carlo Policy Evaluation

- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use policy $\pi$ to update $V$: $a = \pi(s)$
  - MC policy evaluation provides estimate of $V_\pi$
  - Many episodes are needed to obtain accurate estimate
  - Model-free with MC!




- Use $V$ or $Q$ to update policy $\pi$
  - ~~Greedy policy?~~
    - Greedy policy lacks exploration, so $V_\pi$ is not estimated at many states
  - $\epsilon$-greedy policy

# $\epsilon$-Greedy Policy

- Also known as $\epsilon$-greedy exploration
- Choose random action with probability $\epsilon$
  - Typically uniformly random
  - If $a$ takes on discrete values, then all actions will be chosen eventually
- Choose action from greedy policy with probability $1 - \epsilon$
  - $a = \arg\max_{a'}\{r(s, a') + \gamma \sum_s [p(s|s_t, a')V(s)]\}$
  - Still requires model, $p(s|s_t, a)$…
  - Solution: $Q$ function

# Monte-Carlo Policy Evaluation for $Q$ Functions

- To obtain empirical mean, we record $N(s,a)$, # of times $s$ is visited for every state and action $a$ is applied
  - Start at $N(s,a) = 0$ for all $s$ and $a$
  - Note that this means $N$ (and $S, A$ below) must be stored for every $s$ and $a$

- First-visit MC Policy Evaluation:
  - At the first time $t$ that $s$ is visited and action $a$ is applied in an episode,
    - Increment $N(s,a) \leftarrow N(s,a) + 1$
    - Record return $R(s,a) \leftarrow R(s,a) + \sum \gamma^t r(s_t, a_t)$
    - Repeat for many episodes
  - Estimate action-value function: $Q(s,a) = \frac{R(s,a)}{N(s,a)}$

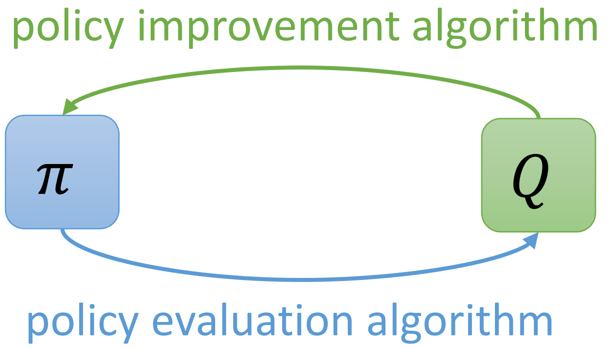
# Incremental Updates

- Instead of estimating $Q(s,a)$ after many episodes, we can update it incrementally after every episode after receiving return $R$
  - $N(s,a) \leftarrow N(s,a) + 1$
  - $Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}\big(R - Q(s,a)\big)$

- More generally, we can weight the second term differently
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(R - Q(s,a)\big)$

# Basic $Q$ Learning

- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use policy $\pi$ to update $Q$: $a = \pi(s)$
  - Repeat for many episodes:
    - $N(s, a) \leftarrow N(s, a) + 1$
    - $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}\big(R(s, a) - Q(s, a)\big)$
- Use $Q$ to update policy $\pi$
  - $\epsilon$-greedy policy
    - With probability $\epsilon$, choose random control
    - With probability $1 - \epsilon$, choose $a = \arg\max_{a'}\{Q(s, a')\}$
  - Pick $\epsilon = \frac{1}{k}$, where $k$ is the # of algorithm iterations
    - Explore less as value function becomes more accurate



# Outline

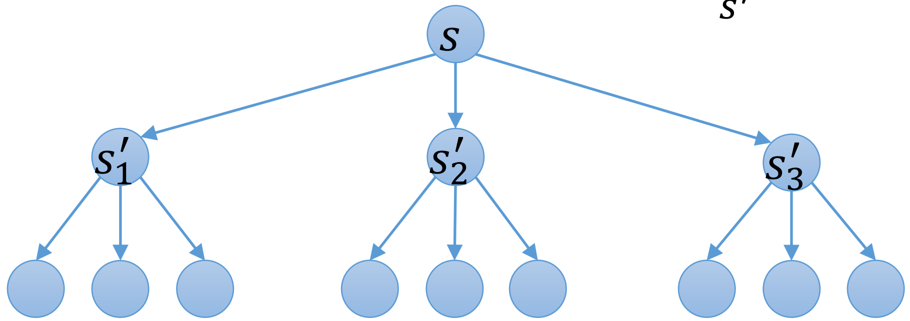
- Reinforcement learning problem setup
- Imitation learning
- Basic ideas in RL
- **Model-free value-based RL**
- Policy-based and actor-critic RL

# DP vs. MC Policy Evaluation

- Suppose the policy $\pi$ is given
  - Dynamic Programming

$$V(s) \leftarrow \max_a Q(s,a)$$

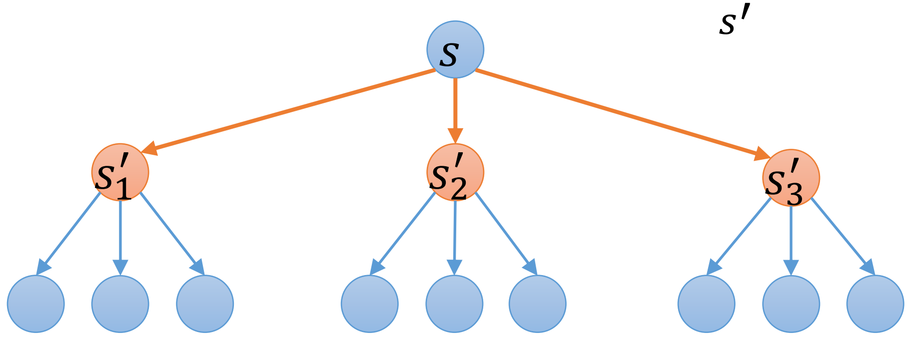
$$Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} [p(s'|s,a)V(s')]$$



# DP vs. MC Policy Evaluation

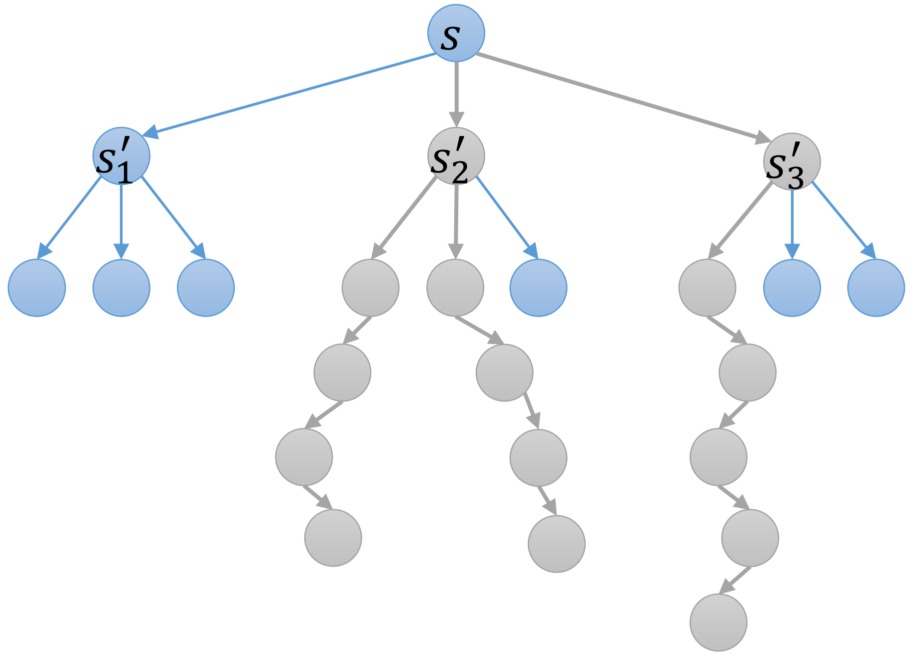
- Suppose the policy $\pi$ is given
  - Dynamic Programming

    $$V(s) \leftarrow \max_a Q(s,a)$$
    $$Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} [p(s'|s,a)V(s')]$$




- Monte-Carlo
  - Repeat for many episodes:

    $$N(s,a) \leftarrow N(s,a) + 1$$
    $$Q(s,a) \leftarrow Q(s,a) + \alpha\big(R - Q(s,a)\big)$$



# Temporal-Difference (TD) Policy Evaluation

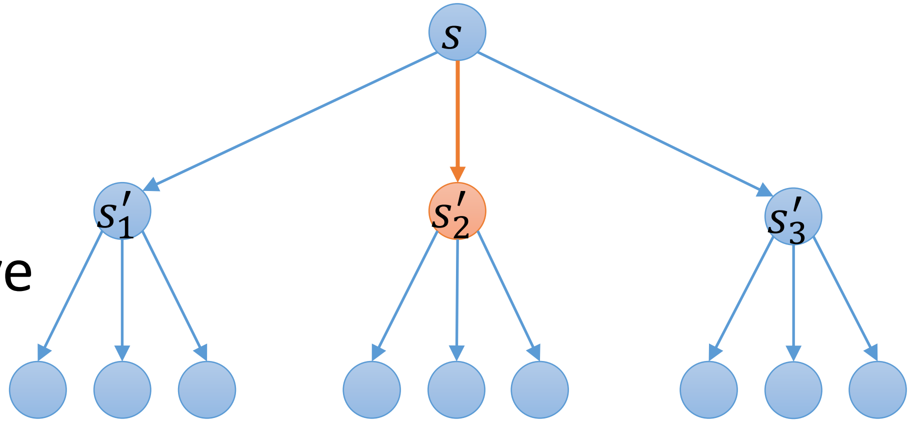
- Temporal-difference: a class of policy evaluation techniques TD($\lambda$)
- Most basic version: TD(0)
  - From any state $s$, apply policy $a = \pi(s)$ for one time step, obtain reward $r(s, a)$
  - Get to next state $s'$, and estimate return from then on using $Q$ function
    - Note: next action is also from the same policy, $a' = \pi(s')$
    - $Q(s, a) \leftarrow Q(s, a) + \alpha\big(r(s, a) + \gamma Q(s', a') - Q(s, a)\big)$
  - Repeat for many episodes to obtain $Q(s, a)$ estimates at many states $s$ and actions $a$

# Temporal-Difference (TD) Policy Evaluation

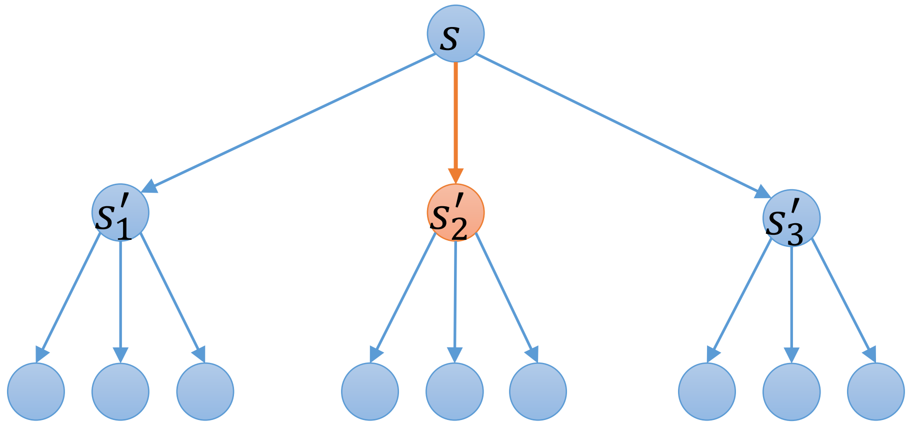
- Most basic version: TD(0)

  $$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$$

- Advantages:
  - Online algorithm: $Q$ can be updated during an episode
  - Does not require complete episodes
- Disadvantages:
  - System may not be Markov
  - Initial $Q$ can be very bad and $Q$ may never improve enough

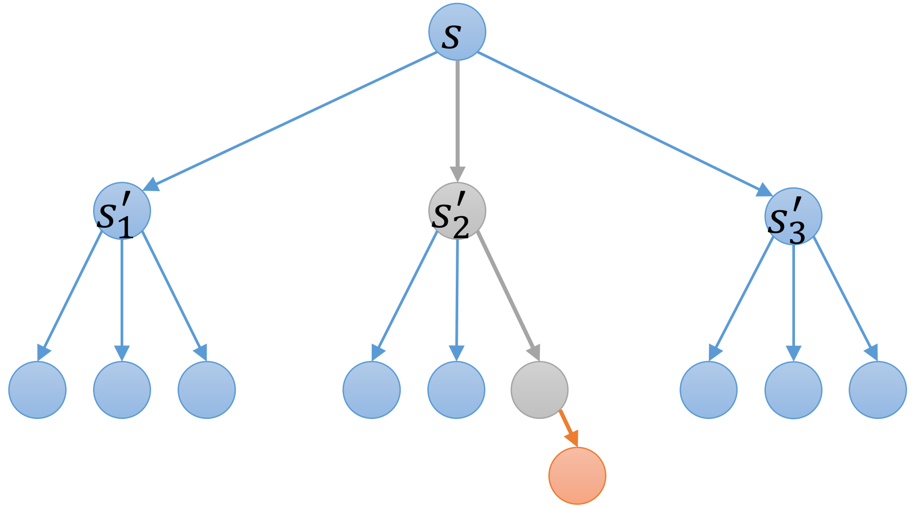
# $n$-step TD




- TD: Look ahead one step
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$

- $n$-step TD: look ahead $n$ steps

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(\underbrace{r(s,a) + \gamma r(s_{+1},a_{+1}) + \cdots \gamma^{n-1} r\big(s_{+(n-1)},a_{+(n-1)}\big) + \gamma^n Q(s_{+n},a_{+n})}_{:= R_n} - Q(s,a)\Big)$$




- MC: Look ahead until the end of the episode

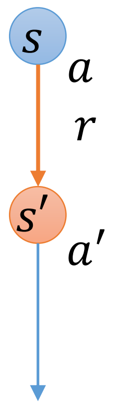
# TD($\lambda$)

- $n$-step return estimate:
  - $R_n = r(s,a) + \gamma r(s_{+1}, a_{+1}) + \cdots \gamma^{n-1} r\big(s_{+(n-1)}, a_{+(n-1)}\big) + \gamma^n Q(s_{+n}, a_{+n})$

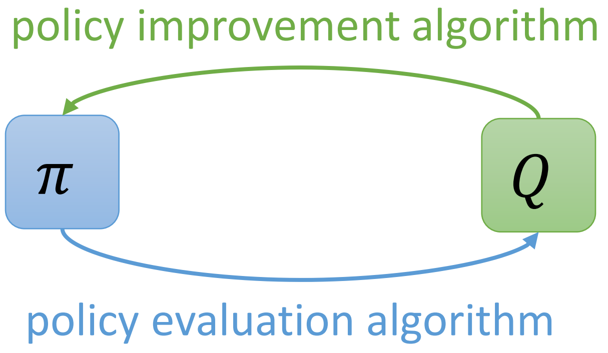
- $\lambda$-return: weighted average of different $n$-step returns
  - Weights: $(1-\lambda)\lambda^{n-1}$
  - Estimated return: $(1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_n$
  - Small $\lambda$ → near-future rewards are more important
  - Large $\lambda$ → far-future rewards are more important

- TD($\lambda$) policy evaluation:
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\Big(\overbrace{(1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_n}^{\text{target}} - Q(s,a)\Big)$

# SARSA Algorithm




- Start with initial policy $\pi$ and value function $V$ or $Q$
- Use $\epsilon$-greedy policy to update $Q$: $a, a' \sim \pi(s)$, $\pi$ is $\epsilon$-greedy
  - Repeat for many episodes:
    - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$
- New policy $\pi$ is derived from new $Q$
  - $\epsilon$-greedy policy
    - With probability $\epsilon$, choose random control
    - With probability $1 - \epsilon$, choose $a = \arg\max_{a'}\{Q(s,a')\}$
- If $\epsilon, \alpha \propto \frac{1}{k}$, then $Q(s,a) \rightarrow Q_{\pi^*}(s,a)$



# On-Policy and Off-Policy Learning

- From SARSA:
  - Use $\epsilon$-greedy policy to update $Q$: $a, a' \sim \pi(s)$, $\pi$ is $\epsilon$-greedy
    - Repeat for many episodes: $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$

- "**Behaviour policy**": policy used to collect rewards -- $a \sim \pi_B(s)$
- "**Target policy**": policy used to estimate future rewards -- $a' \sim \pi_T(s)$

- "**On-policy learning**": $\pi_B = \pi_T$
  - SARSA is an on-policy learning algorithm

- "**Off-policy learning**": $\pi_B \neq \pi_T$

  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$, where $a \sim \pi_B(s)$, $a' \sim \pi_T(s)$

# Off-Policy Learning

- Off-policy learning: Behaviour and target policies are different
  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$, where $a \sim \pi_B(s), a' \sim \pi_T(s)$

- Advantages:
  - Learn from observing another agent (eg. human) execute a different policy
  - Learn from experience generated from old policies
  - Improve two policies at once, while following one policy

- Example: Q-Learning algorithm
  - $\pi_B$ is $\epsilon$-greedy with respect to $Q$
  - $\pi_T$ is greedy with respect to $Q$

# Q-Learning Algorithm

- Start with initial policy $\pi$ and value function $V$ or $Q$
- Update $Q$:
  - Repeat for many episodes with $\epsilon$-greedy policy $a \sim \pi_B(s)$:
    - $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$
- Both the $\epsilon$-greedy $\pi_B$ and the greedy $\pi_T$ are derived from $Q$

- If $\epsilon, \alpha = \frac{1}{k}$, then $Q(s,a) \rightarrow Q_{\pi^*}(s,a)$

# Function Approximation

- So far, $Q(s, a)$ is stored in a multi-dimensional array
  - Model-free, but cannot solve large problems
- Parametrize value functions with parameters (or weights) $w$
  - $\hat{Q}(s, a; w) \approx Q(s, a)$
  - Update parameters $w$ using MC- or TD-based learning
  - Hopefully, $Q$ is generalizable to different states $s$ and actions $a$

# Fitting to a Known $Q_\pi$

- Fit $\hat{Q}(s, a; w)$ to $Q_\pi(s, a)$

$$\underset{w}{\text{minimize}} \left\| Q_\pi(S, A) - \hat{Q}(S, A; w) \right\|_2^2$$

  - Training data: $\{(s_i, a_i), Q_\pi(s_i, a_i)\}$
  - The collection of states and actions in training data is denoted $S$ and $A$

- Gradient with respect to $w$:
  - $\frac{\partial}{\partial w} \left\| \hat{Q}(S, A; w) - Q_\pi(S, A) \right\|_2^2 = 2 \left( Q_\pi(S, A) - \hat{Q}(S, A; w) \right) \frac{\partial \hat{Q}(S,A;w)}{\partial w}$

- Gradient descent:
  - $w \leftarrow w - \alpha \left( Q_\pi(S, A) - \hat{Q}(S, A; w) \right) \frac{\partial \hat{Q}(S,A;w)}{\partial w}$
  - In practice, use stochastic gradient descent

# Monte-Carlo Incremental Weight Updates

- First-visit MC policy evaluation
  - At the first time $t$ that $s$ is visited in an episode,
    - Increment $N(s,a) \leftarrow N(s,a) + 1$
    - Record return $R(s,a) \leftarrow R(s,a) + \sum \gamma^t r(s_t, a_t)$
    - Repeat for many episodes
  - Before: estimate action-value function at visited $s$ and $a$. $Q(s,a) \approx \frac{R(s,a)}{N(s,a)}$
- Now: Above procedure produces "training data" $\{S, A, R\}$
  - Storing a set of $S, A, R$, etc. is called "**experience replay**"
  - This is as opposed to updating $w$ as data is being collected
- Update weights:
  - $w \leftarrow w - \alpha \left( R - \hat{Q}(S,A;w) \right) \frac{\partial \hat{Q}(S,A;w)}{\partial w}$
- Guaranteed to converge to local optimum

# Temporal-Difference Incremental Weight Updates

- Most basic version: TD(0)
  - From any state $s$, apply policy $a = \pi(s)$ for one time step, obtain reward $r(s,a)$
  - Before: get to next state $s'$, and estimate return from then on using $Q$ function
    - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$
    - Repeat for many episodes to obtain $Q(s,a)$ estimates at many states $s$ and actions $a$
- Above procedure produces a collection of current and next states and actions, $S, A, R, S', A'$

  reward

- Update weights using TD target:
  - $w \leftarrow w - \alpha\left(R + \gamma\hat{Q}(S',A';w) - \hat{Q}(S,A;w)\right)\frac{\partial\hat{Q}(S,A;w)}{\partial w}$
- Not always guaranteed to converge to local minimum

# Q-Learning With Function Approximation

Goal: Given a set of weights $w^-$, find the next set of weights $w$ in $\hat{Q}(s,a;w)$

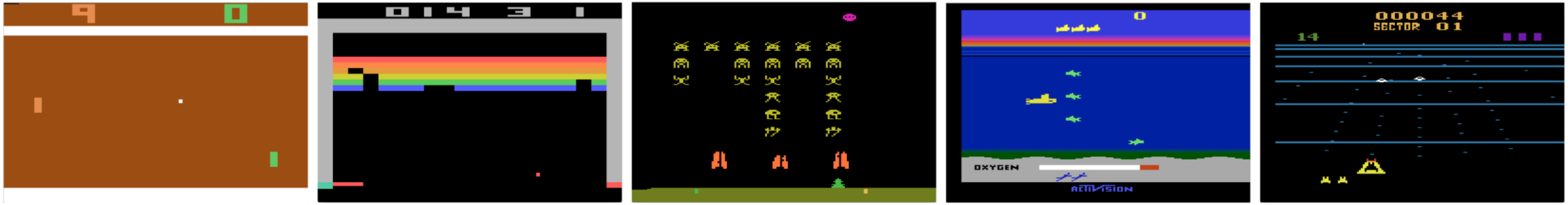
1. From any state $s$, apply $\epsilon$-greedy policy with respect to $\hat{Q}(s,a;w^-)$
   - This produces a collection $S, A, R, S'$
2. Sample from the above collection to obtain a smaller data set $\tilde{S}, \tilde{A}, \tilde{R}, \tilde{S}'$
3. Update weights using stochastic gradient descent

$$\underset{w}{\text{minimize}} \left\| \tilde{R} + \gamma \max_{a'} \hat{Q}\left(\tilde{S}', a'; w^-\right) - \hat{Q}\left(\tilde{S}, \tilde{A}; w\right) \right\|_2^2$$

- Use deep $Q$-network (DQN) for $\hat{Q}\left(\tilde{S}, \tilde{A}; w\right)$ → deep Q-learning

# Deep Q-Learning Example: Atari Games

- Mnih et al. “Playing Atari with Deep Reinforcement Learning,” 2013




- States: pixels from last few frames
- Actions: controls in the game
- Reward: game score
- Deep Q network: convolutional and fully connected layers

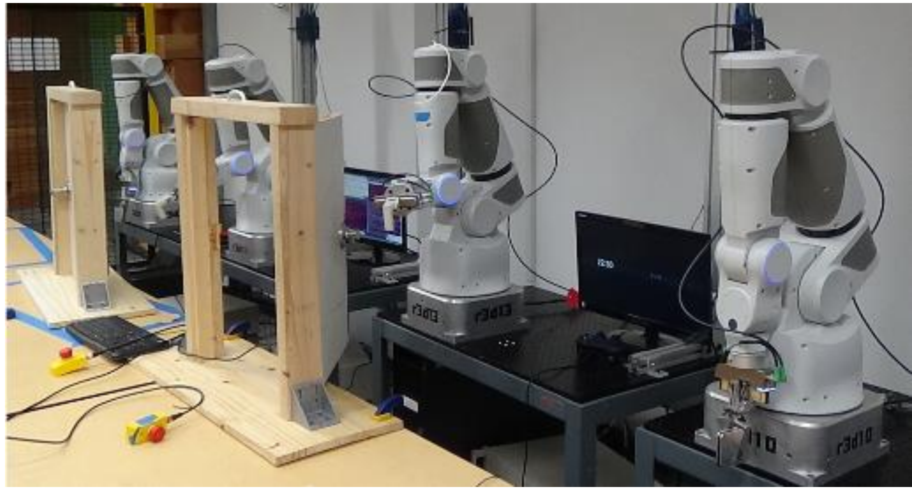
## Starting out - 10 minutes of training

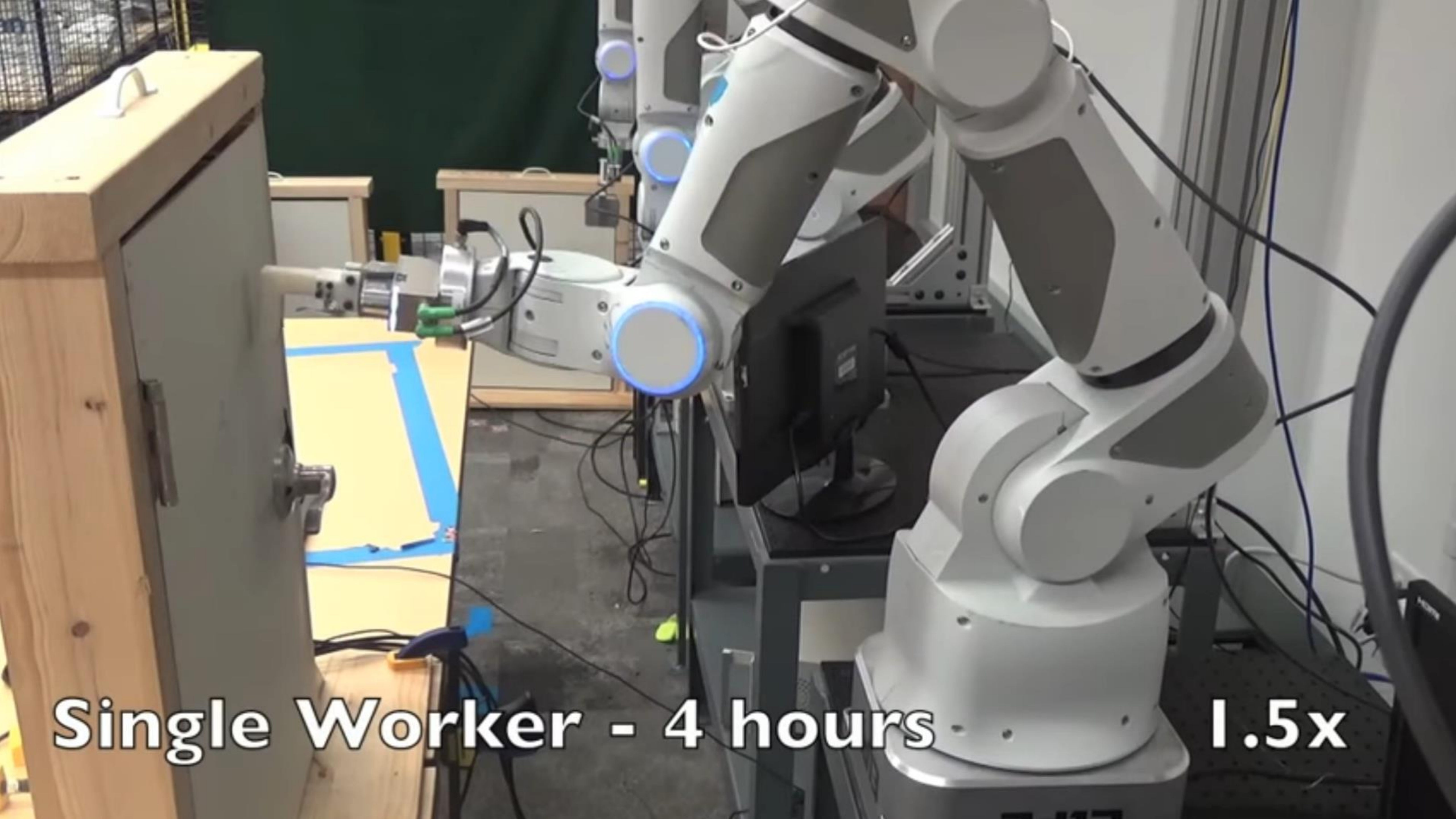
The algorithm tries to hit the ball back, but it is yet too clumsy to manage.

# Deep Q-Learning: Robotic Arms

- Gu et al. "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates," 2017.

- States: joint angles, end-effector positions, and their time derivatives, target position
- Actions: joint velocities of arm, torque of fingers
- Task: open door, pick up object and place it elsewhere
- Deep Q network: two fully connected hidden layers, 100 units each
- Main challenge: use multiple robots to learn at the same time and share knowledge




Single Worker - 4 hours
1.5x

# Outline

- Reinforcement learning problem setup
- Imitation learning
- Basic ideas in RL
- Model-free value-based RL
- **Policy-based and actor-critic RL**

# Categories of RL

- Model-based
  - Explicitly involves an MDP model
- Model-free
  - Does not involve an MDP model

- Value based
  - Learns value function, and derives policy from value function
- **Policy based**
  - **Learns policy without value function**
- **Actor critic**
  - **Incorporates both value function and policy**

# Policy Gradients

- If we executed a policy $\pi_\theta$ from state $s_0$, we obtain a trajectory
  - $\tau \coloneqq (s_0, a_0, s_1, a_1, \dots)$
  - Note: this is a random variable

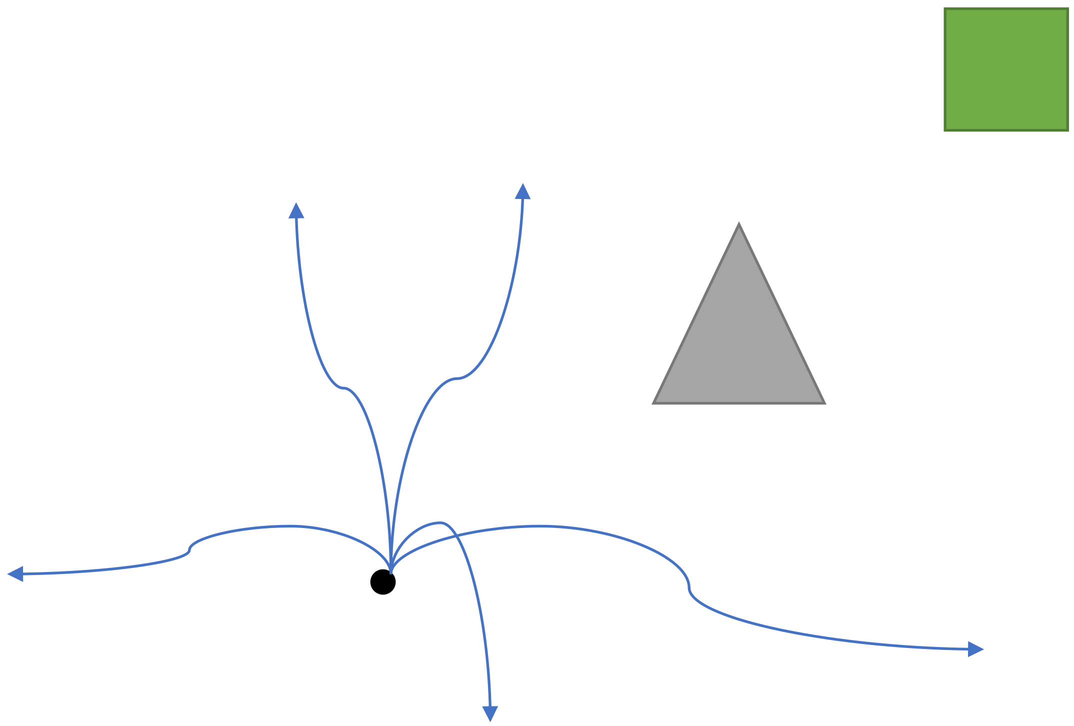
- The return is given by $R(\tau) \coloneqq \sum_{t \geq 0} \gamma^t r(s_t, a_t)$
  - Also a random variable

- Expected return given parameters $\theta$: $J(\theta) \coloneqq \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau)]$

- Parameters for the optimal policy:
  - $\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau)]$

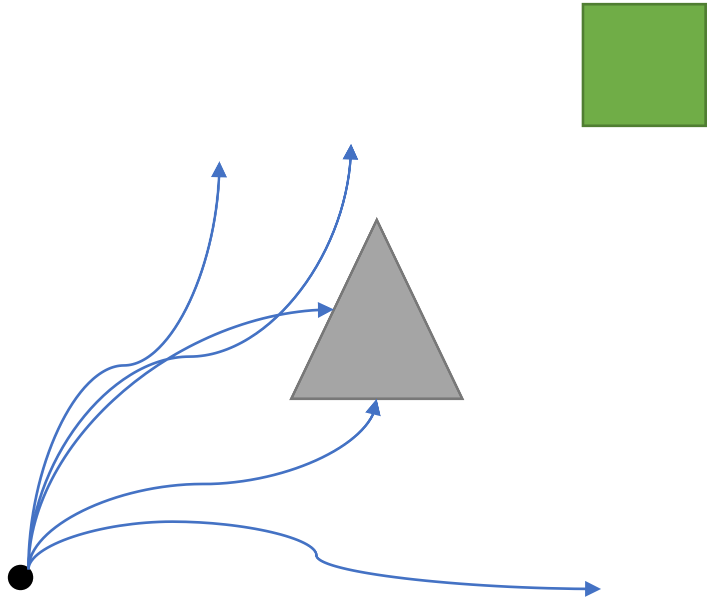
# Policy Gradients

- Strategy: differentiate $J(\theta)$ w.r.t. $\theta$ and perform stochastic gradient ascent
  - Do this in a way that is model-free and computationally tractable



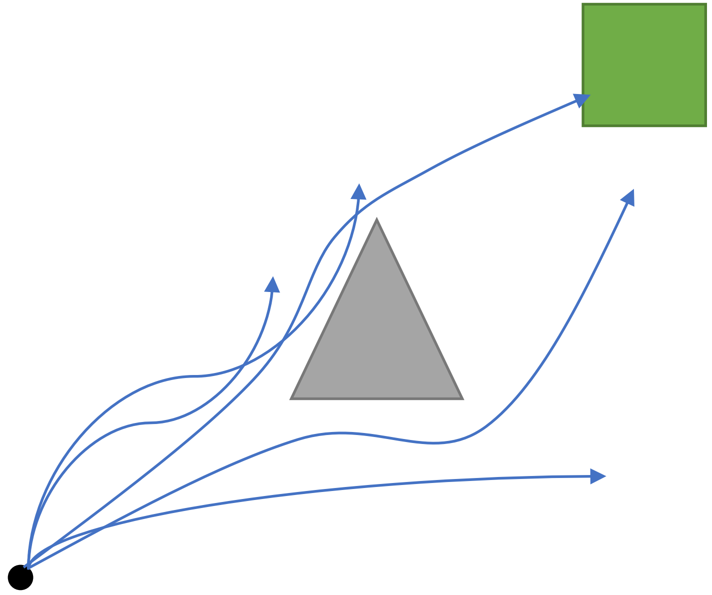
# Policy Gradients

- Strategy: differentiate $J(\theta)$ w.r.t. $\theta$ and perform stochastic gradient ascent
  - Do this in a way that is model-free and computationally tractable



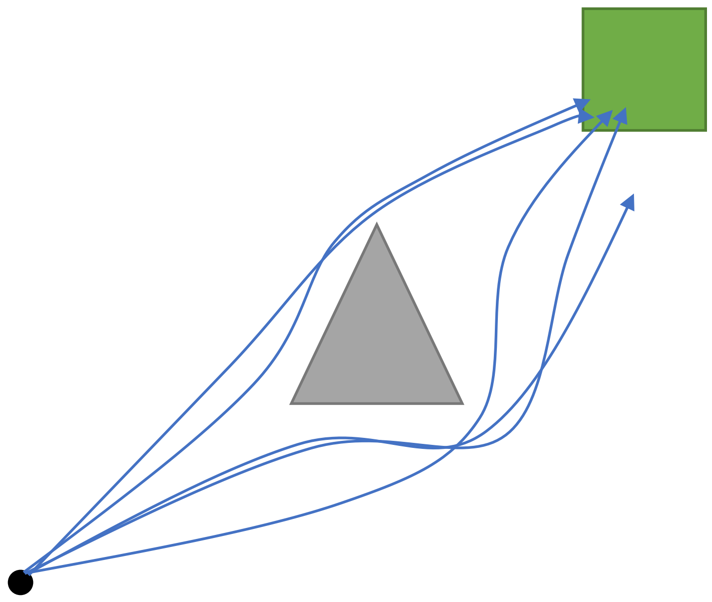
# Policy Gradients

- Strategy: differentiate $J(\theta)$ w.r.t. $\theta$ and perform stochastic gradient ascent
  - Do this in a way that is model-free and computationally tractable



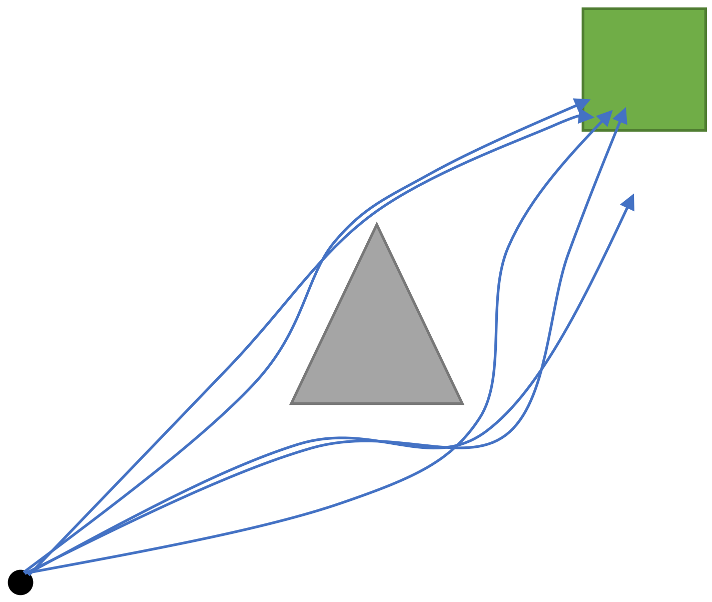
# Policy Gradients

- Strategy: differentiate $J(\theta)$ w.r.t. $\theta$ and perform stochastic gradient ascent
    - Do this in a way that is model-free and computationally tractable



# Policy Gradients

- Strategy: differentiate $J(\theta)$ w.r.t. $\theta$ and perform stochastic gradient ascent
  - Do this in a way that is model-free and computationally tractable
- To achieve this
  - Write out $J(\theta)$
  - Take gradient
  - Do a math trick
  - Obtain gradient expression that can be estimated easily



# Write Out $J(\theta)$ and Take Gradient

- $J(\theta) \coloneqq \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau)]$

- $J(\theta) = \int_\tau R(\tau) p(\tau;\theta) d\tau$

- $\nabla_\theta J(\theta) = \int_\tau R(\tau) \nabla_\theta p(\tau;\theta) d\tau$
    - Hard…

# Log Gradient Trick

- $\nabla_\theta J(\theta) = \int_\tau R(\tau) \nabla_\theta p(\tau; \theta) d\tau$

- Trick:
  - $\nabla_\theta p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau; \theta) \nabla_\theta \log p(\tau; \theta)$
  - $\nabla_\theta J(\theta) = \int_\tau R(\tau) p(\tau; \theta) \nabla_\theta \log p(\tau; \theta)\, d\tau$
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau) \nabla_\theta \log p(\tau; \theta)]$
  - Gradient is an expectation – can estimate this using techniques we learned before!

# Model-Free Estimate of Gradient

- $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau)\nabla_\theta \log p(\tau;\theta)]$

- $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$
- $\log p(\tau;\theta) = \sum_{t\geq 0}[\log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)]$
- $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$
  - Amazingly, model-free
  - Markov property is not used
  - $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is known: since the form of $\pi_\theta$ is known
    - Eg. Backprop if $\pi_\theta$ is a neural network

# Monte-Carlo Gradient Estimate

- Results so far:
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau)\nabla_\theta \log p(\tau;\theta)]$
  - $\nabla_\theta \log p(\tau;\theta) = \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

- Some more algebra to write out gradient of $\nabla_\theta J(\theta)$
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[R(\tau) \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)]$
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[\sum_{t \geq 0} R(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)]$
  - $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t \geq 0} R(\tau_i)\nabla_\theta \log \pi_\theta\left(a_{t,i}|s_{t,i}\right)\right]$

# REINFORCE Algorithm

- (Monte-Carlo Policy Gradient)

- Use policy $\pi_\theta(a|s)$ to obtain trajectories $\tau_i = \{s_{0,i}, a_{0,i}, \ldots\}$

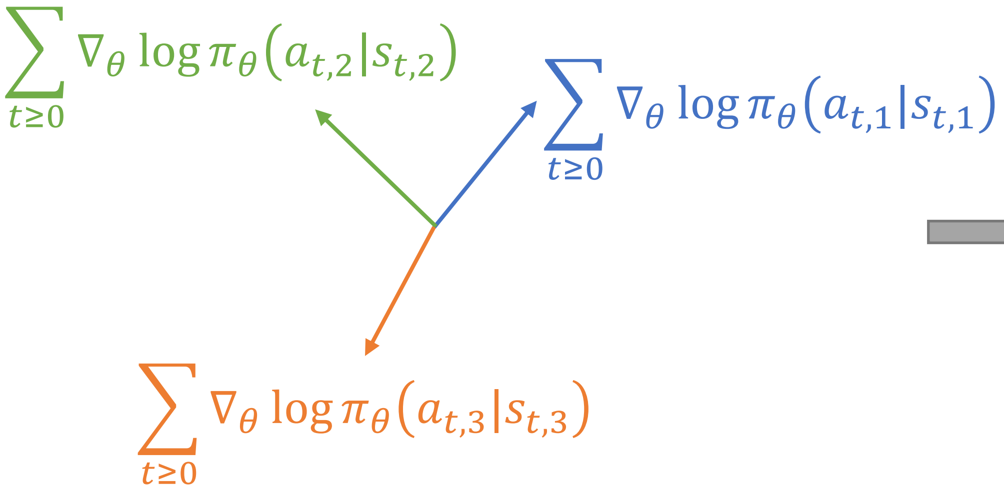
- Estimate the gradient of the reward
  - $\nabla_\theta J(\theta) \approx \sum_{i=1}^{N} \left[ \sum_{t \geq 0} R(\tau_i) \nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i}) \right]$

- Update policy parameters via (stochastic) gradient ascent
  - $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

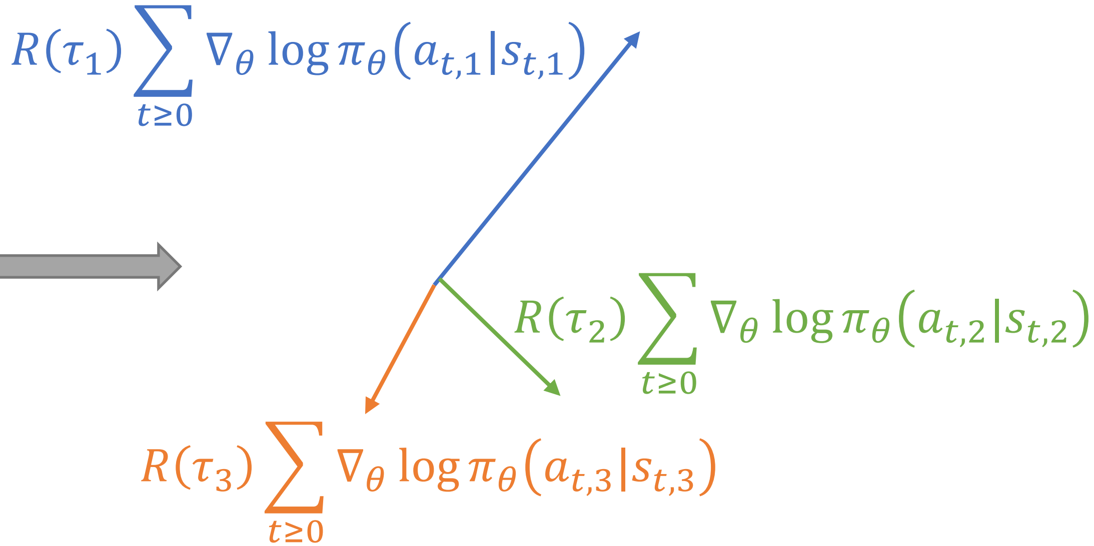
# Observation 1

- Gradient estimate:
  - $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[\sum_{t \geq 0} R(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)\right]$
  - $\nabla_\theta J(\theta) \approx \sum_{i=1}^{N}\left[\sum_{t \geq 0} R(\tau_i) \nabla_\theta \log \pi_\theta(a_{t,i} | s_{t,i})\right]$

- Gradient estimate also works for POMDPs without modification




Suppose
- $R(\tau_1) = 2$
- $R(\tau_2) = -1$
- $R(\tau_3) = 1$



# Observation 1

- Gradient estimate:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}[\sum_{t \geq 0} R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)]$
  - $\nabla_\theta J(\theta) \approx \sum_{i=1}^{N} \left[\sum_{t \geq 0} R(\tau_i) \nabla_\theta \log \pi_\theta\left(a_{t,i}|s_{t,i}\right)\right]$

- Gradient estimate also works for POMDPs without modification

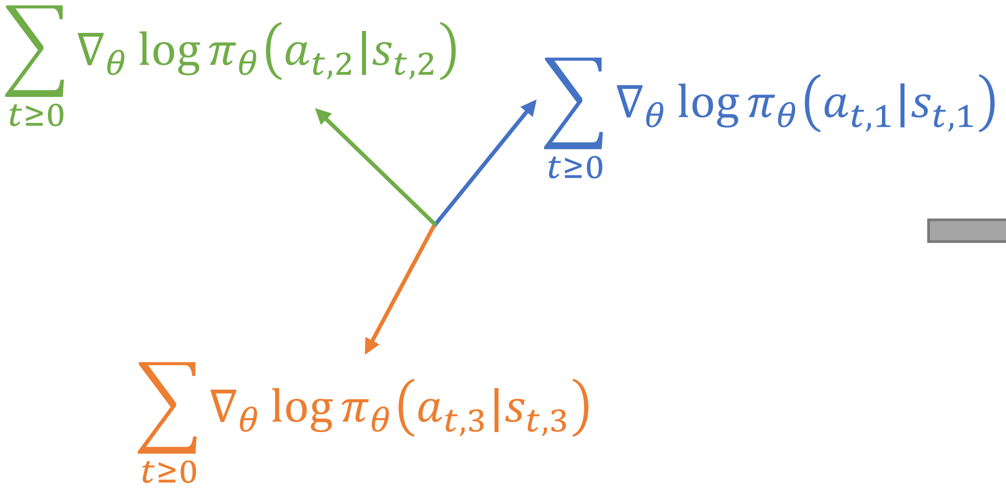
- Parameter updates: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
  - Trajectories have high reward will be made more likely
  - Trajectories with low reward will be made less likely
  - A high-reward trajectory has good actions… **on average**

# Observation 2

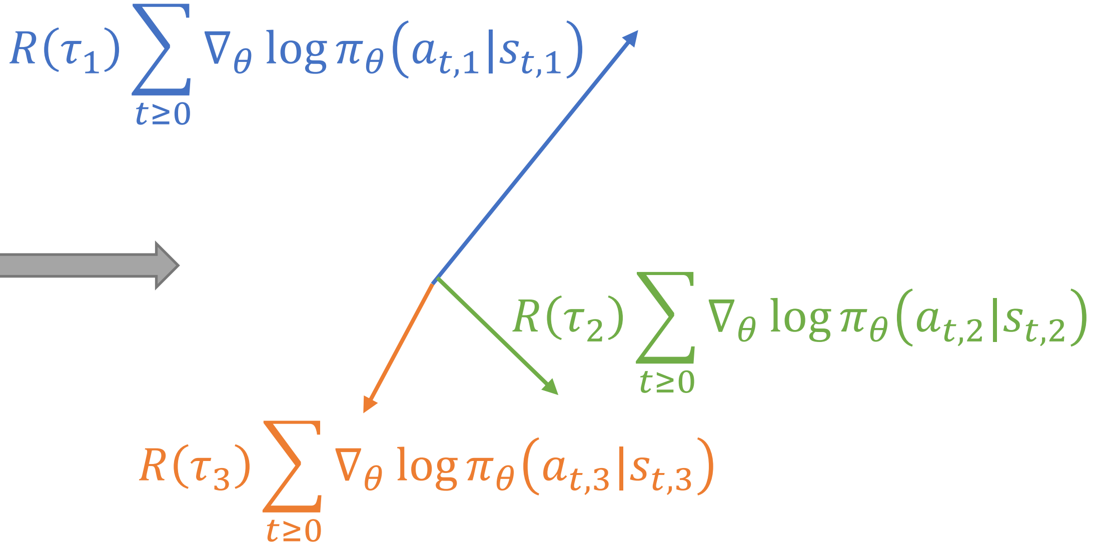
- Gradient estimate:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}[\sum_{t \geq 0} R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)]$
- Causality?
  - $R(\tau)$ is the reward of the entire trajectory
  - $R(\tau)$ is multiplied in every term of the sum
  - $\tau$ includes times before $t$
  - So, according to the above, the weight of $\nabla_\theta \log \pi_\theta(a_t|s_t)$ depends on times prior to $t$?
- Simple fix:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[\sum_{t \geq 0}\left[\left(\sum_{t' \geq t} \gamma^{t'-t} r(s_t, a_t)\right) \nabla_\theta \log \pi_\theta(a_t|s_t)\right]\right]$

# Observation 3

- Gradient estimate:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}[\sum_{t\geq 0} R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)]$

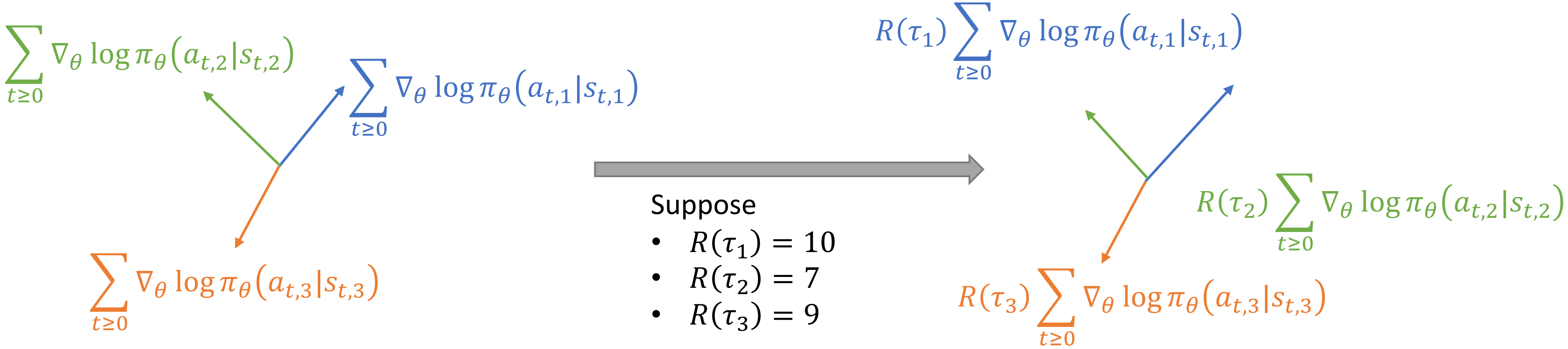



Suppose
- $R(\tau_1) = 2$
- $R(\tau_2) = -1$
- $R(\tau_3) = 1$



# Observation 3

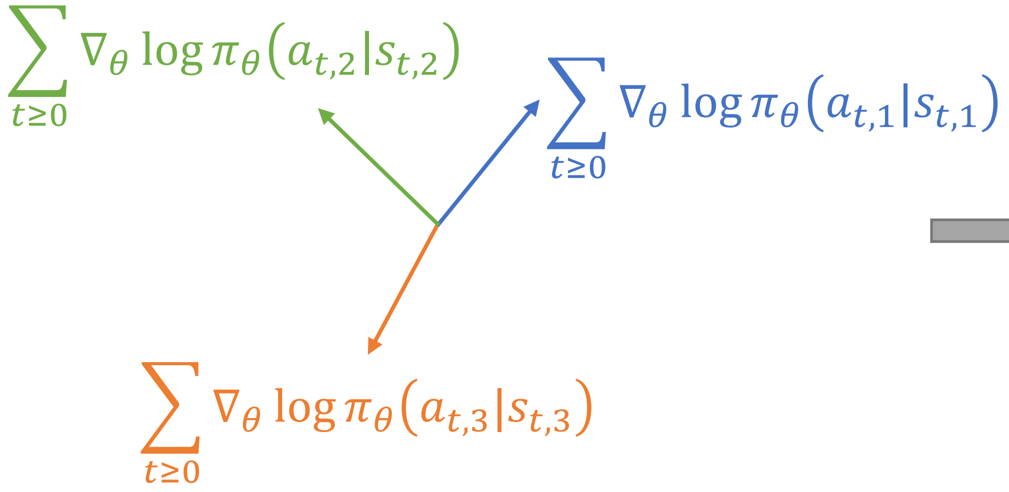
- Gradient estimate:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}[\sum_{t\geq 0} R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)]$




Suppose
- $R(\tau_1) = 10$
- $R(\tau_2) = 7$
- $R(\tau_3) = 9$

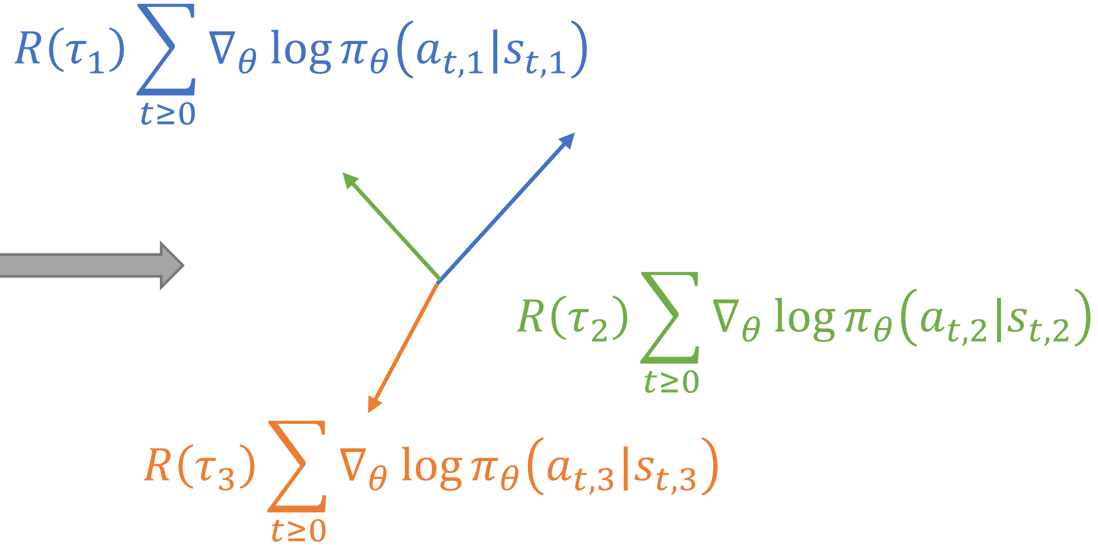
# Observation 3

- Gradient estimate:
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[\sum_{t\geq 0} R(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]$




Suppose

- $R(\tau_1) = 10$
- $R(\tau_2) = 7$
- $R(\tau_3) = 9$




- Performance is measured by reward $R(\tau)$
  - But what is considered "good"?
  - Need a baseline of comparison!
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[\sum_{t\geq 0} (R(\tau) - b)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]$
  - Fact: expectation is unchanged as long as $b$ does not depend on $\theta$

# Revised REINFORCE

- (Monte-Carlo Policy Gradient)

- Use policy $\pi_\theta(a|s)$ to obtain a trajectory $\tau = \{s_0, a_0, \dots\}$

- Estimate the gradient of the reward
  - $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[\sum_{t\geq 0}\left[\left(\sum_{t'\geq t} \gamma^{t'-t} r(s_t, a_t) - b\right)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]\right]$

- Update policy parameters via (stochastic) gradient ascent
  - $\theta \leftarrow \theta + \alpha\, \nabla_\theta J(\theta)$

# Picking a Baseline

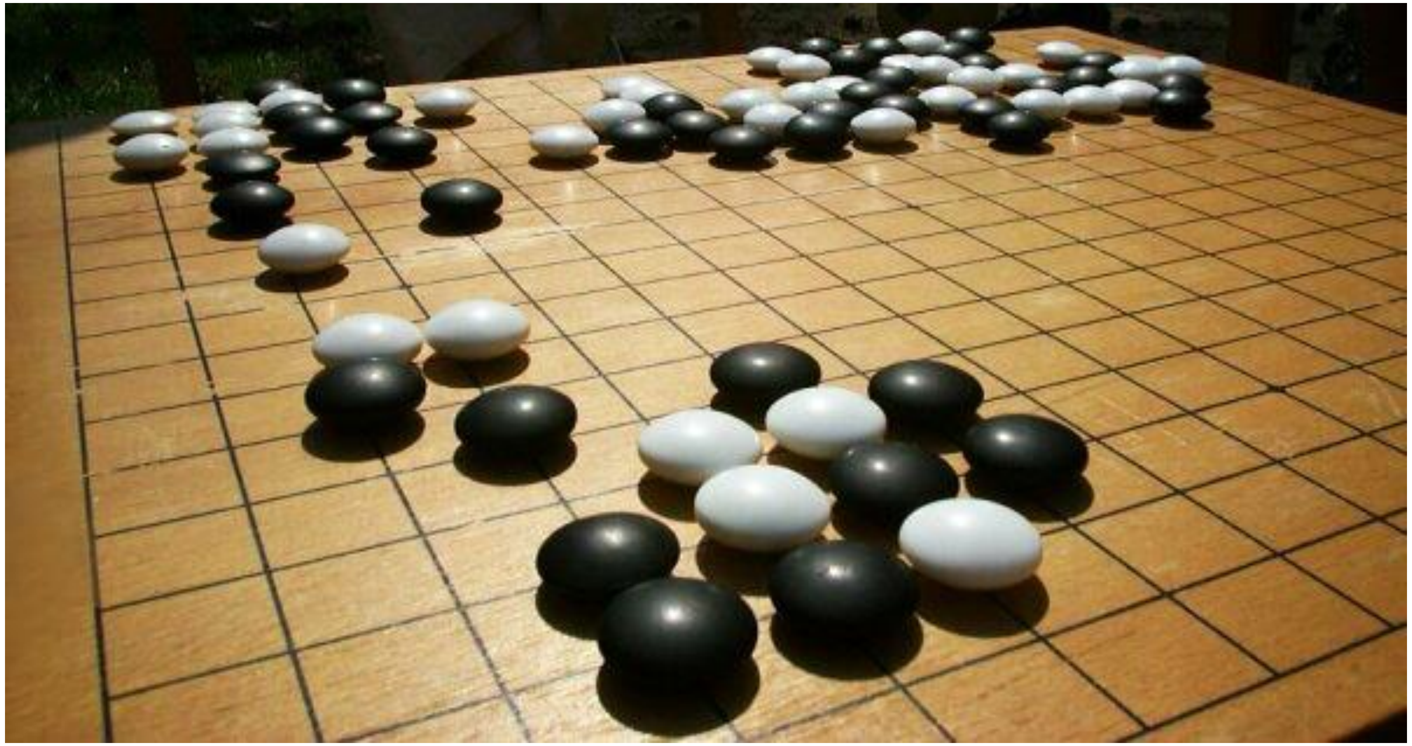
- Many choices

- One intuitive choice
  - $b = V_\pi(s)$
  - Define $\mathcal{A}_\pi(s, a) \coloneqq r(s, a) + \gamma V_\pi(s') - V_\pi(s)$
    - Good action: one that gives a return that is large relative to $V$
    - Bad action: one that gives a return that is small relative to $V$
  - $\mathcal{A}_\pi(s, a)$ -- "**advantage function**"

- But we don't know $V$…
  - Learn it!

# Actor-Critic Methods

- Actor (policy $\pi$) decides which actions to take
- Critic (value function $V$) decides how good the action is



# Actor-Critic Methods

- Basic algorithm, combining everything we've learned:
  1. Start with some initial policy $\pi_\theta$ and value function $\hat{V}(s; w)$
     - $\theta$ and $w$ are parameters
  2. Collect data $S, R, S'$ by executing policy
  3. Update $V_\phi$: $\underset{w}{\text{minimize}} \left\| \tilde{R} + \gamma \hat{V}\left(\tilde{S}'; w^-\right) - V\left(\tilde{S}; w\right) \right\|_2^2$
     - Many methods (eg. stochastic gradient descent)
  4. Estimate policy gradient: $\nabla_\theta J(\theta) \approx \mathbb{E}_{\tau \sim p(\tau;\theta)} \left[ \sum_{t \geq 0} \left( \tilde{R} + \gamma V_\pi\left(\tilde{S}'\right) - V_\pi\left(\tilde{S}\right) \right) \nabla_\theta \log \pi_\theta(a_t | s_t) \right]$
  5. Improve policy via gradient ascent: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
  6. Repeat 2-5 many times

# State-of-the-Art Policy Gradient Methods

- Trust region policy optimization (TRPO)
  - https://arxiv.org/abs/1502.05477
- Proximal policy optimization (PPO)
  - https://arxiv.org/abs/1707.06347