CIS 7000-04 / ESE 6800 Spring 2025: Intro To Imitation and Reinforcement Learning Part III

Thu, Jan 28, 2025

Instructor: Dinesh Jayaraman

Assistant Professor, CIS

Where We Left Off: Policy Gradients for RL

The objective of RL is

$$\mathbb{E}_{\tau_i \sim \pi_\theta(\tau)}\left[\sum_t \gamma^t r_{i,t}\right]$$

Its gradient w.r.t. policy parameters θ is the "policy gradient":

$$\mathbb{E}_{\tau_i \sim \pi_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(a_{i,t} \middle| s_{i,t}\right)\left(\sum_{j=t,t+1,\dots} \gamma^{j-t} r_{i,j}\right)\right]$$

Policy-generated trajectories

Policy-generated actions

Reward returns: "how good was this action?"

We generate our own data during learning … this is trial-and-error learning!

"Make good stuff more likely, and bad stuff less likely"

For intuition, we started to derive a version of the policy gradient:

Today’s Plan

- Finish policy gradient proof
- Monte Carlo estimates of Policy Gradient
- Understanding “On-Policy” RL and Its Sample Complexity
- Exploration in Policy Gradient algorithms

- An Introduction to “Actor-Critic” algorithms
- An Introduction to Value-Function-Based RL
- An Introduction to Model-Based RL

Step 1 of 2: Writing the gradient as an expectation

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau$$

Taking the gradient,

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau$$

To make this look like an expectation, we used the log-gradient trick:

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau)\, r(\tau) d\tau = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\boxed{\nabla_\theta \log \pi_\theta(\tau)}\, r(\tau)]$$

Do we know this quantity?

(Recall: Expectations are easy to approximate, as sample averages)

Step 2 of 2: Gradient of trajectory log-likelihood

$$\nabla_\theta \log \pi_\theta(\tau) = ?$$

Let $\tau = [s_0, a_0, s_1, a_1, \dots, s_T, a_T]$ *(PS: finite trajectory case, just to make derivation a little easier, will generalize)*

$$\log \pi_\theta(\tau) = \log \mu(s_0) + \log \pi_\theta(a_0|s_0) + \log P(s_1|s_0, a_0) + \ \dots$$

$$\log \pi_\theta(\tau) = \log \mu(s_0) + \sum_t \log \pi_\theta(a_t|s_t) + \sum_t \log P(s_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log \pi_\theta(\tau) = \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Plugging into the policy gradient expression!

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \, r(\tau) \right]$$

Final Touch: Adding Causality

The policy gradient that we have derived:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_i \sim \pi_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{j=0,1,2,\ldots} \gamma^j r_{i,j}\right)\right]$$

Add causality, and this becomes the most commonly used form of vanilla policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_i \sim \pi_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{j=t,t+1,\ldots} \gamma^{j-t} r_{i,j}\right)\right]$$

We skip derivation, but the core idea is: actions can only affect future rewards, so they should be evaluated based on future rewards alone.
Reduces variance in estimates.

"Vanilla Policy Gradient": Monte Carlo Estimate

$$\mathbb{E}_{\tau_i \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) \left(\sum_{j=t,t+1,\ldots} \gamma^{j-t} r_{i,j} \right) \right]$$

Estimated through sampling trajectories τ by executing policy π in the environment.

$$\frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) \left(\sum_{j=t,t+1,\ldots} \gamma^{j-t} r_{i,j} \right) \right)$$

Policy-generated trajectories

Policy-generated actions

Reward returns: "how good was this action?"

Good estimates often require large N!

Policy Gradient Algorithms | Lil'Log   Williams 1992 REINFORCE   Sutton, McAllester, Singh, Mansour 1999

Vanilla Policy Gradient vs. Reward-Wtd Regression

Reward-weighted regression gradient:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{\tau=t,t+1,\dots}\gamma^{\tau-t}\,r_{i,\tau}\right)\right)$$

(Non-optimal) demonstrated data

(Non-optimal) demonstrated actions

Reward returns: "how good was this action?"

Policy gradient:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{\tau=t,t+1,\dots}\gamma^{\tau-t}\,r_{i,\tau}\right)\right)$$

Identical expression, Only difference is the distribution

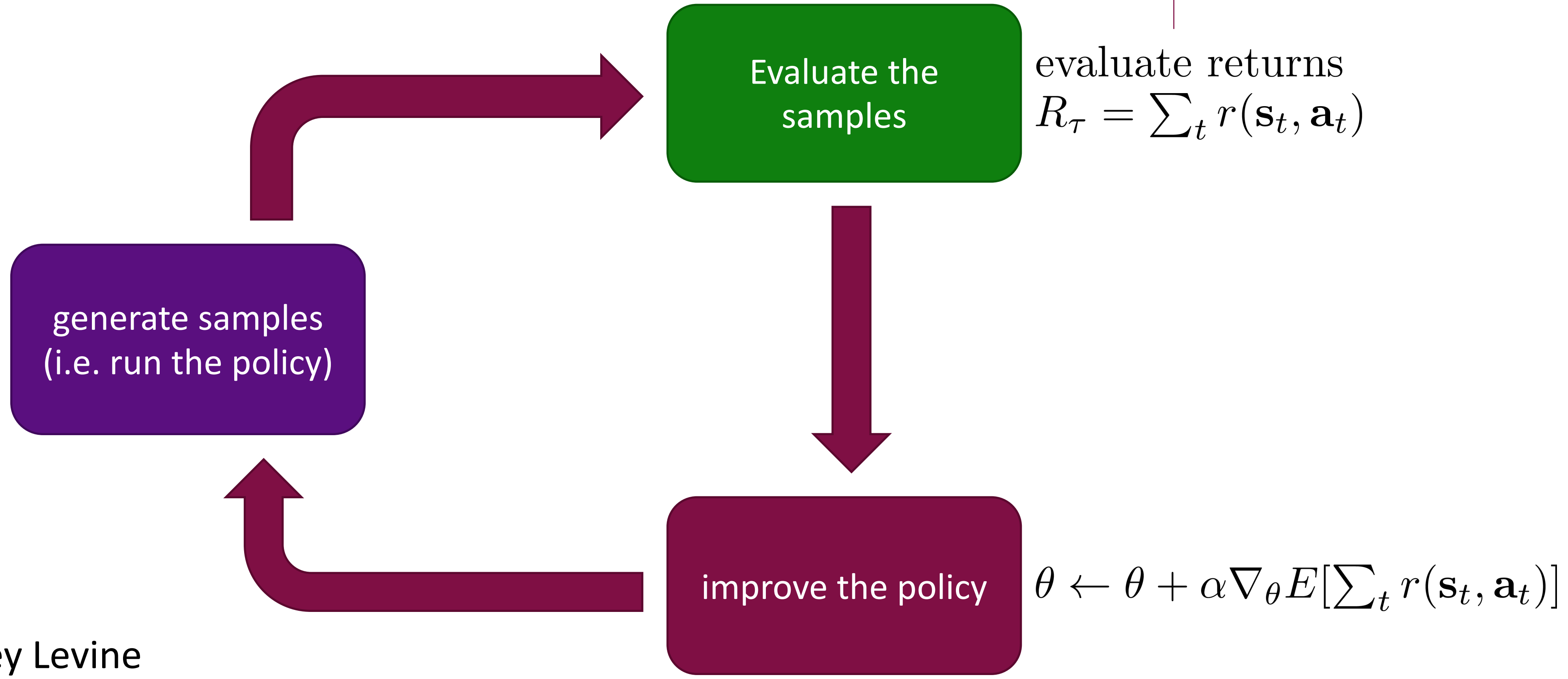
Policy-generated trajectories

Policy-generated actions

Reward returns: "how good was this action?"

The basic policy gradients algorithm

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right) \left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$


Evaluate the samples
evaluate returns
$R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta E[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$
generate samples (i.e. run the policy)

Slide: Sergey Levine

Note: Reward function need not be differentiable!

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$

In supervised learning, when we optimized an objective using gradient descent, we needed the objective to be differentiable w.r.t. to the parameters θ.

In RL, this is not true any more. See how the update term involves no derivative of the reward function!

In supervised learning, when we optimized an objective using gradient descent, we needed the objective to be differentiable w.r.t. to the parameters θ.

In RL, this is not true any more. See how the update term involves no derivative of the reward function!

2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$

“On-Policy” Learning

$$\nabla_\theta J_\theta = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{\tau=t,t+1,\ldots}\gamma^{\tau-t}\, r_{i,\tau}\right)\right)$$

- The policy gradient increases the likelihood of past actions that yielded good returns ***when later actions were generated from the current policy.***
- This means you can only ever compute the policy gradient update on data that is generated *from the current policy.*
  - “On-policy” learning.
  - Expensive in terms of amount of experience required in the environment, because old experience, generated from old policies, is no longer relevant. Need to keep generating fresh new experiences.
  - Also, recall, large N is required for getting good gradient estimates i.e. *lots of fresh new experiences.*

The “Sample Complexity” of Online RL

- Recall that in supervised learning, we often update a neural network ~10^5-10^6 times on mini-batches to find a solution.
- Policy gradient requires *generating a whole new dataset for each update*, generated by executing the current policy many times.
  - **Often too sample-inefficient to run on real-world robots, and instead deployed in highly parallelized simulators.** More on this when we discuss “sim-to-real RL” methods later in the course.
- Gradient descent works best when step sizes are small, but if each gradient step is very expensive (as in online RL), one might consider more aggressive updates. Tweaking learning rates like one might in supervised learning doesn’t work too well.
  - **TRPO** [Schulman et al 2015]**, PPO** [Schulman et al 2017]**, etc.** propose ways to set the step size at each update as large as possible without breaking optimization.

Recap: Exploration vs Exploitation

- What happens if you execute only your current-best policy all the time?
  - Might not explore enough to discover other solutions.
  - For example, you might never discover a shortcut if you only stick to a known route to a target.
- What happens if you only execute random actions all the time?
  - Wasteful. You mainly care about states and actions encountered by the optimal policy.
  - For example, if you keep exploring the city randomly, it will take a really long time for you to learn any meaningful route to your target.

Exploration vs. Exploitation Tradeoff

Based on slide by Dan Klein

Whither Exploration In Policy Gradient?

- **Exploration in RL**: Which actions to execute in the world to most efficiently learn an optimal policy?
  - But with on-policy RL, do we really have a choice? Remember, our updates can only be computed from trajectories sampled from the current policy π_θ at each stage of training!
- **Two solutions:**
  - π_θ is inherently stochastic, because it is probabilistic, so it does automatically perform different actions each time it is executed, and therefore induces some exploration.
  - Explicitly add an **"exploration bonus"** to the reward, e.g. entropy

$$r_t \leftarrow r_t + \lambda H\big(\pi_\theta(a_t|s_t)\big)$$

which incentivizes more uncertain policies, inducing more exploration. $\lambda \rightarrow 0$ during training.

Popular Implementations

- RL implementation details can be hard to get right. Good to start with **popular repositories: OpenAI stable-baselines, CleanRL etc.**

Reducing Policy Gradient Variance with Critics

Improving the policy gradient "critic"

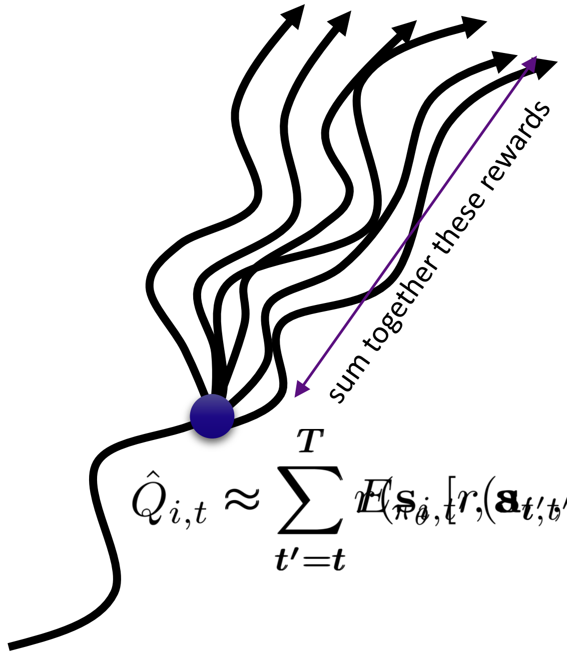
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\underbrace{\left(\sum_{t'=t}^{T} r(s_{i,t'},a_{i,t'})\right)}_{\text{"reward to go"}}$$

$\hat{Q}_{i,t}$

$\hat{Q}_{i,t}$: estimate of expected reward if we take action $\mathbf{a}_{i,t}$ in state $\mathbf{s}_{i,t}$

can we get a better estimate?

$Q(\mathbf{s}_t,\mathbf{a}_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'},\mathbf{a}_{t'})|\mathbf{s}_t,\mathbf{a}_t\right]$: true *expected* reward-to-go



$$\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} r(\mathbf{s}_{i,t'},\mathbf{a}_{i,t'})$$

$$\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'},\mathbf{a}_{t'})|\mathbf{s}_t,\mathbf{a}_t\right]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})Q(\mathbf{s}_{i,t},\mathbf{a}_{i,t})$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})(Q(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) - V(\mathbf{s}_{i,t}))$$

$$V(\mathbf{s}_t) = E_{\mathbf{a}_t\sim\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}[Q(\mathbf{s}_t,\mathbf{a}_t)]$$

Slide based on: Sergey Levine

Fitting a Q value function

- How to "fit" Q^π (and what to fit to?)

- If we could arbitrarily reset to any state infinite times and rollout π, we could just compute the average from the last slide.

- Better way to do this: by maintaining a table* of Q values for all s, a and then making use of **consistency properties of $\boldsymbol{Q(s,a)}$.**
    - Enter Bellman Equation.

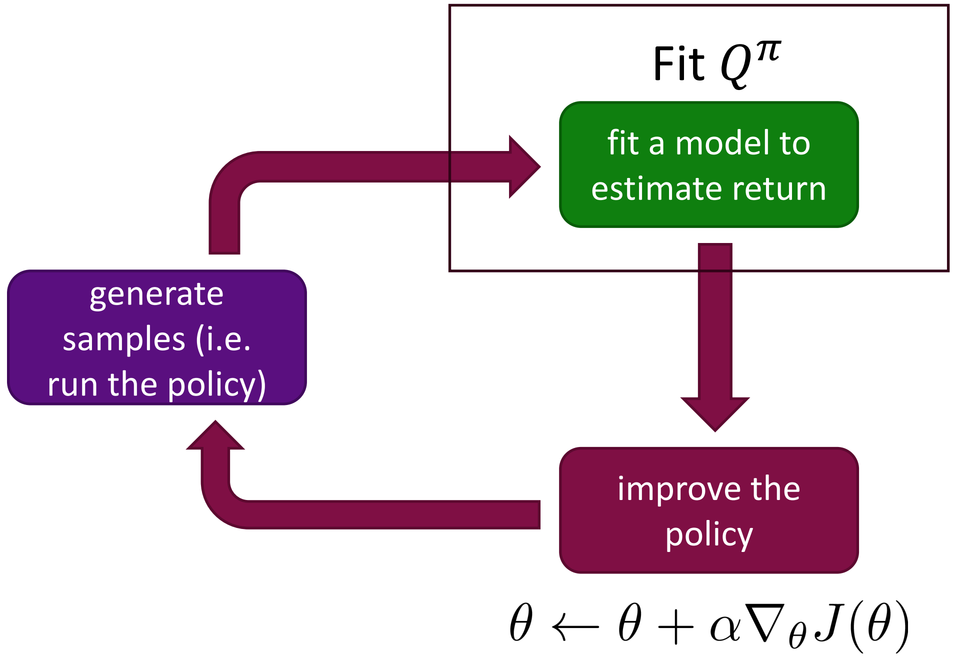
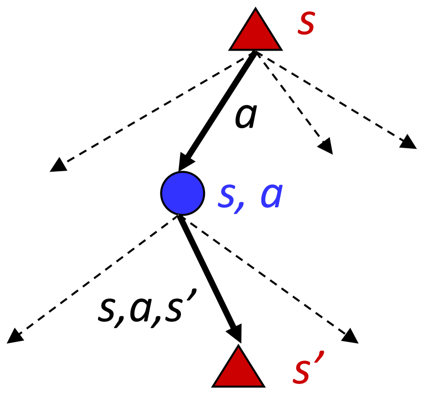

Fit Q^π
fit a model to estimate return
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
generate samples (i.e. run the policy)

Image based on: Sergey Levine

Bellman Equation for Q^π

$$Q^\pi(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_t | S_0 = s, A_0 = a\right)$$

- By definition, action-value function for a policy π obeys:



$$Q^\pi(s,a) = \underbrace{\sum_{s' \in S} P(s'|s,a)}_{\text{expected value over successor state s'}} \underbrace{[R(s,a,s') + \gamma Q^\pi(s',\pi(s'))]}_{\text{current reward + discounted future reward}}$$

$$Q^\pi(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)}[R(s,a,s') + \gamma Q^\pi(s',\pi(s'))]$$

$$\approx R(s,a,s') + \gamma Q^\pi(s',\pi(s'))$$

Calculating Q^π: (Approximate) Policy Evaluation

For all (s, a)

$Q_0^\pi(s,a) \leftarrow 0$ (could also be initialized differently)

Update $Q_{j+1}^\pi(s,a)$ to move towards $R(s,a,s') + \gamma Q_j^\pi(s', \pi(s'))$

The "fixed point" of this recursive update is indeed the correct Q^π.
We will skip proof.

"actor-critic architecture"

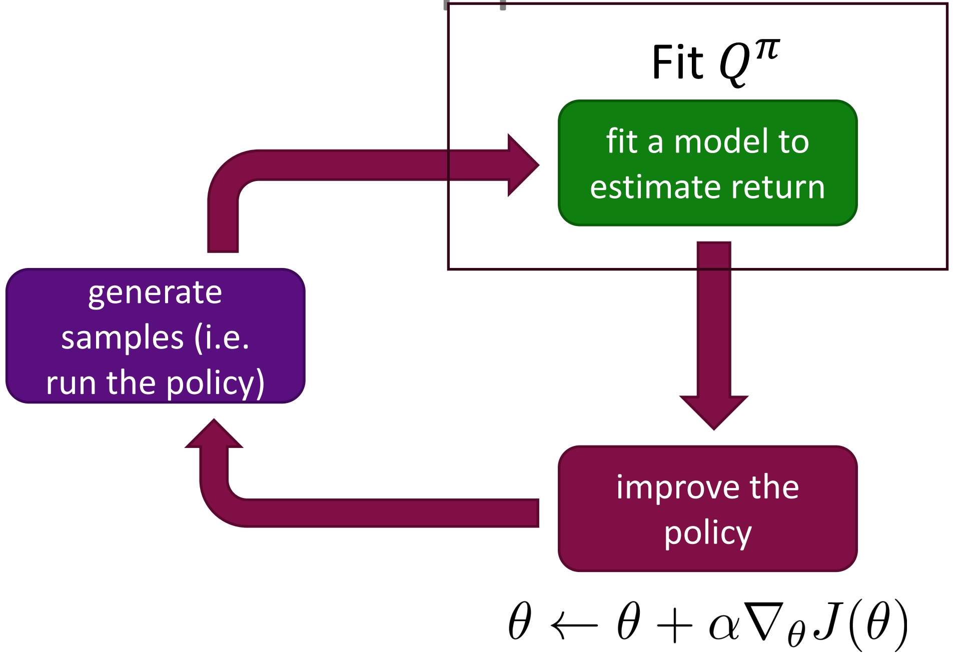

Fit Q^π
fit a model to estimate return
improve the policy
generate samples (i.e. run the policy)
$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Image based on: Sergey Levine

Policy gradient with automatic differentiation

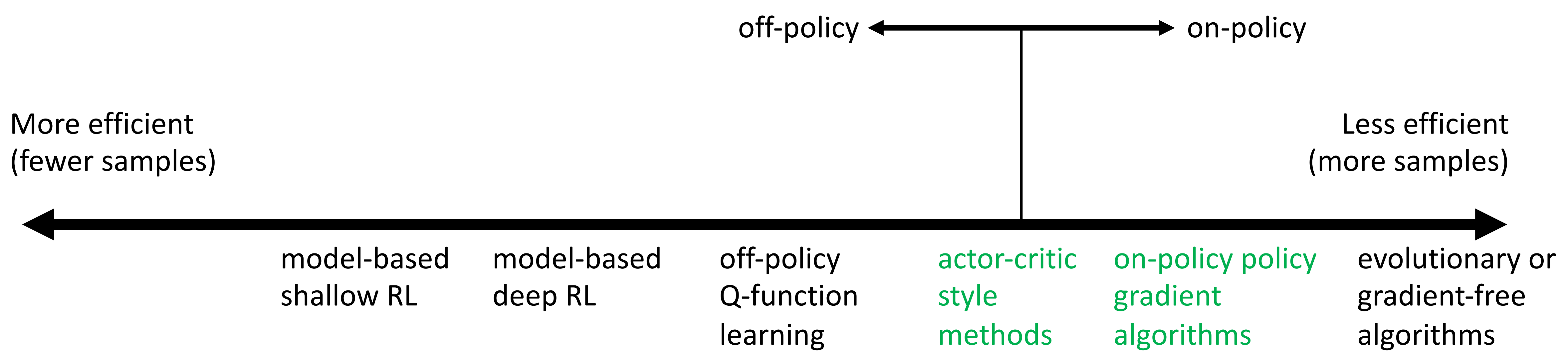
Pseudocode example (with discrete actions):

Policy gradient:

```
# Given:
# actions - (N*T) x Da tensor of actions
# states - (N*T) x Ds tensor of states
# q_values – (N*T) x 1 tensor of estimated state-action values
# Build the graph:
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits
negative_likelihoods = softmax_cross_entropy_with_logits(labels=actions, logits=logits)
weighted_negative_likelihoods = multiply(negative_likelihoods, q_values)
loss = reduce_mean(weighted_negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

$$\tilde{J}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\underbrace{\hat{Q}_{i,t}}_{\text{q\_values}}$$

Other RL Algorithms



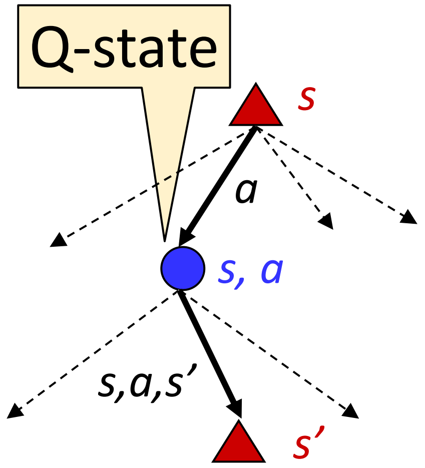
off-policy
on-policy
More efficient (fewer samples)
Less efficient (more samples)
model-based shallow RL
model-based deep RL
off-policy Q-function learning
actor-critic style methods
on-policy policy gradient algorithms
evolutionary or gradient-free algorithms

But policy gradients and (off-policy) actor-critic approaches are among the most *stable* approaches that work most broadly, and take limited wall clock time even though many samples.

Next: Q learning and actor-critic approaches!

An Alternative Paradigm: Value-Function-Based RL

$Q^{\pi}(s,a)$ For *Optimal* π



Q-value of taking action a in state *s* then following policy π :

$Q^{\pi}(s,a)$ = expected return when taking *a* in *s* and then following π

$$Q^{\pi}(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_{t+1} | S_0 = s, A_0 = a\right)$$

Optimal Q-value: $Q^*(s,a) = Q^{\pi^*}(s,a)$

Given Q*, can you select optimal actions?

Yes, π^* can be **greedily** determined from Q^*: $\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$

In other words, knowing/learning Q^* would be sufficient to act optimally (assuming you can solve the argmax)!

Bellman Equation for optimal Q^* functions

$$\boxed{Q^\pi(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)}[R(s,a,s') + \gamma Q^\pi(s', \pi(s'))]}$$

Optimal values are what we get by picking the optimal action

$$Q^*(s,a) = Q^{\pi^*}(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

Recall:
$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$

“Q Learning”

$$Q^*(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

Idea: As we did earlier for policy evaluation in actor-critic, it is once again possible to treat the single sample you get as a rough estimate of the expectation, and apply an *incremental* update to reduce the “Bellman error”:

- Execute a single action a from state s and observe s' and R:

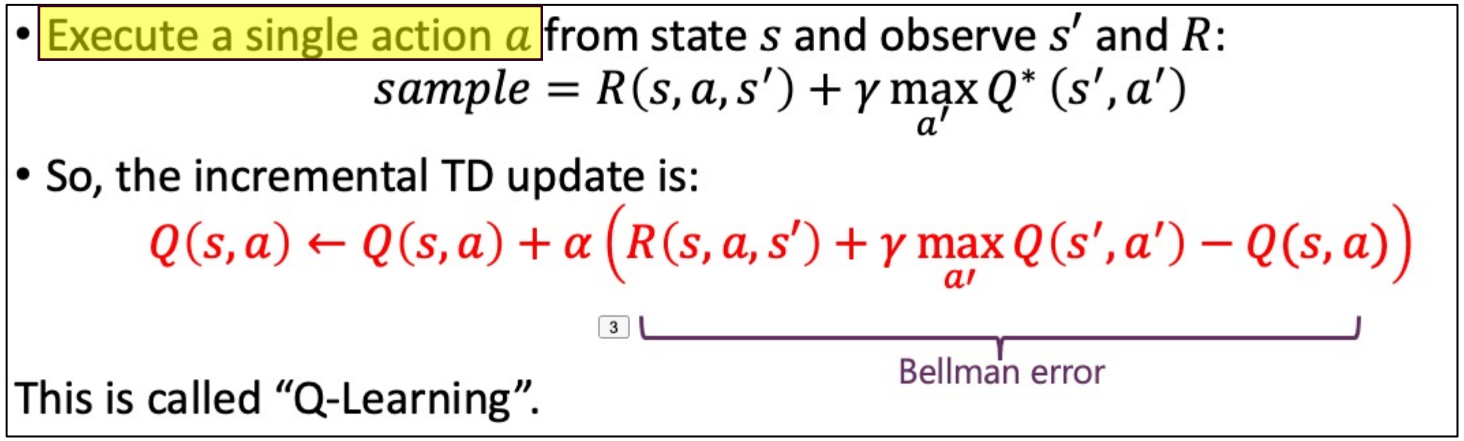
$$sample = R + \gamma \max_{a'} Q_{old}(s',a')$$

- Now, compare this sample to the LHS, and apply the *incremental* update:

$$Q(s,a) \leftarrow Q_{old}(s,a) + \alpha \underbrace{\left(R + \gamma \max_{a'} Q_{old}(s',a') - Q_{old}(s,a)\right)}_{\text{Bellman error}}$$

Thus, we can now get the optimal Q from the agent’s trial-and-error experience. **This is called “Q-Learning”.** Q iteration + 1-sample-based incremental update.

How to Act During Q-Learning: “Off-policy” vs “On-Policy”



- Execute a single action a from state s and observe s' and R:

$$sample = R(s,a,s') + \gamma \max_{a'} Q^*(s',a')$$

- So, the incremental TD update is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \underbrace{\left(R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)}_{\text{Bellman error}}$$

This is called “Q-Learning”.

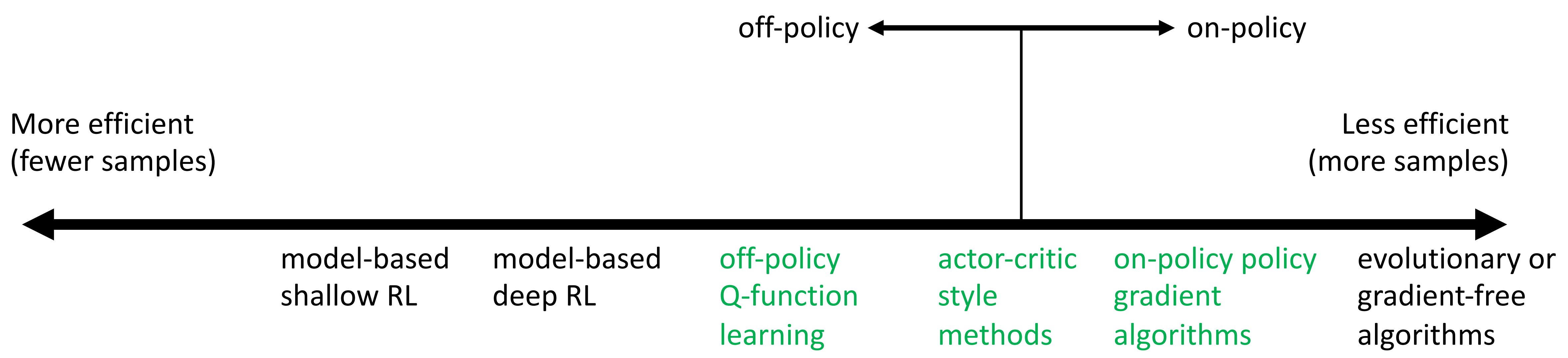
Note that for Q-Learning, we have said nothing about *which* actions to sample.

- You can act any way you want, and as long as you “explore” the environment well, Q-Learning will eventually converge to the optimal $Q^*(s,a)$. “Off-policy”.
- This is different from policy gradients or the kind of actor-critic approach* that we saw: they rely on updates based exclusively on data generated from the current best policy. Those are “on-policy” approaches.

Some Simple Schemes for Balancing Explore-Exploit

- ϵ −greedy:
  - At every state,
    - With small probability ϵ, perform a random action
    - Otherwise, follow current best $a^* = \text{argmax}_a[Q(s,a)]$
  - Can anneal ϵ over time
    - Intuition: should explore more when you know very little about the city. After having lots of experience navigating it, there isn't much value to exploration any more.
- Track Visitation Counts:
  - Maintain a running count of the number of times $N(s,a)$ that you have tried executing a from state s.
  - Select $a^* = \text{argmax}_a[Q(s,a) + 1/N(s,a)]$, inflating the return of states that you have not visited.

Other RL Algorithms


off-policy
on-policy
More efficient (fewer samples)
Less efficient (more samples)
model-based shallow RL
model-based deep RL
off-policy Q-function learning
actor-critic style methods
on-policy policy gradient algorithms
evolutionary or gradient-free algorithms

But policy gradients and (off-policy) actor-critic approaches are among the most *stable* approaches that work most broadly, and take limited wall clock time even though many samples.

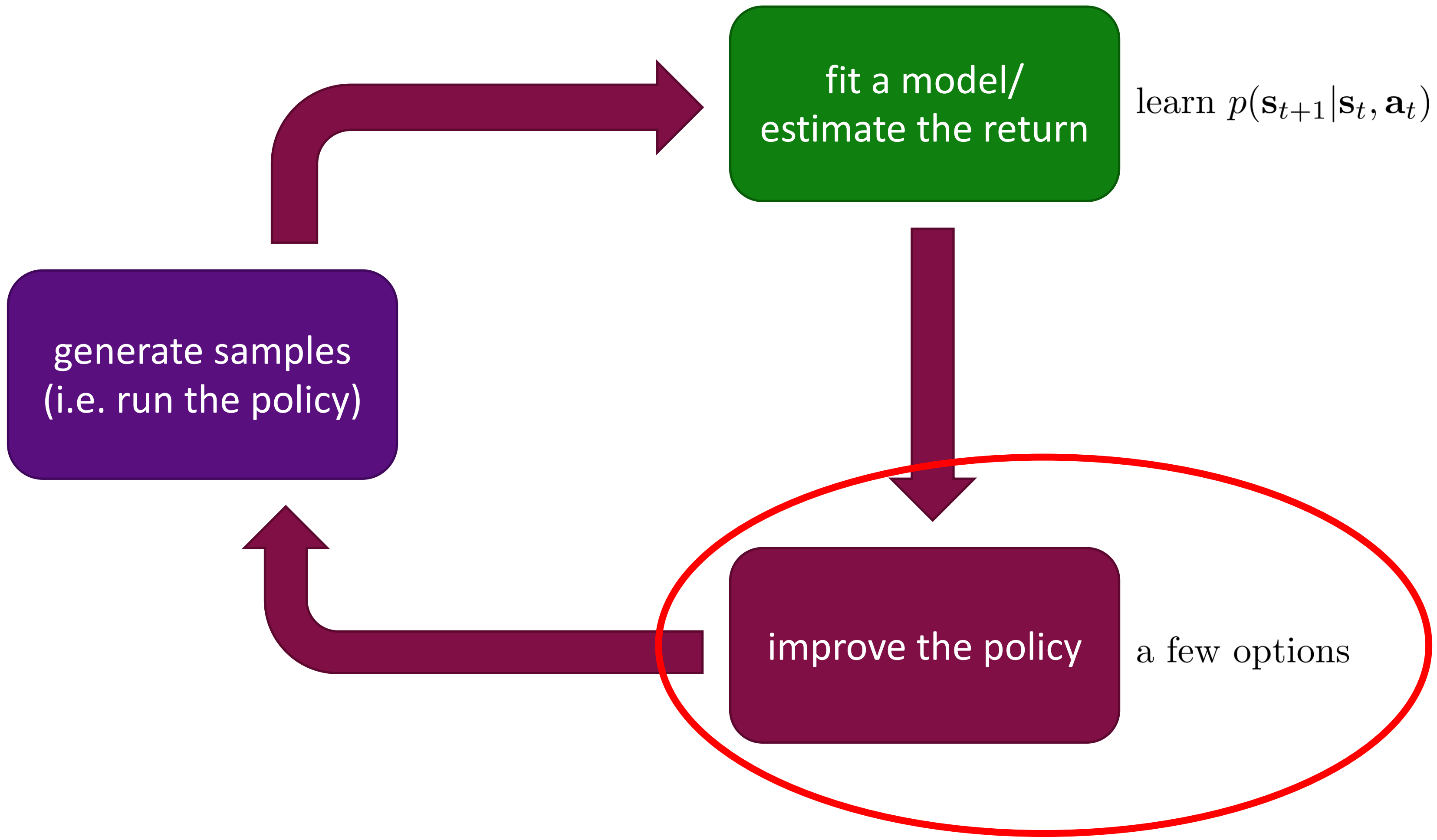
Next: Q learning and actor-critic approaches!

Types of RL algorithms

$$\theta^\star = \arg\max_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

- Policy gradients: directly differentiate the above objective
- Value-based: estimate value function or Q-function of the optimal policy (no explicit policy)
- Actor-critic: estimate value function or Q-function of the current policy, use it to improve policy
- Model-based RL: estimate the transition model, and then…
  - Use it for planning (no explicit policy)
  - Use it to improve a policy
  - Something else

Model-based RL algorithms



Model-based RL algorithms

improve the policy

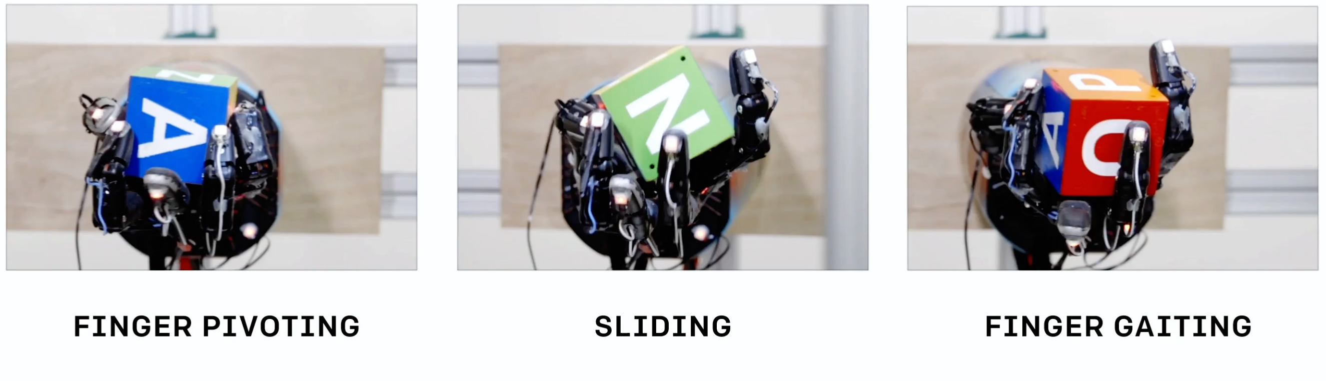
a few options

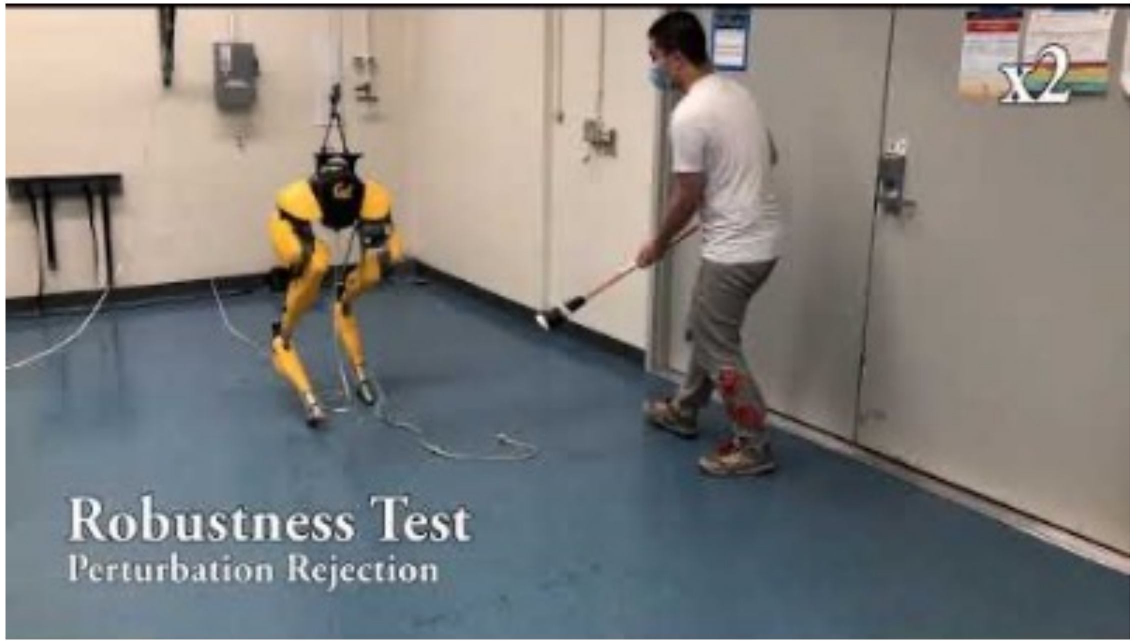
- Just use the model to plan (no policy)
  - Trajectory optimization/optimal control (primarily in continuous spaces) – essentially backpropagation to optimize over actions
  - Discrete planning in discrete action spaces – e.g., Monte Carlo tree search
- Backpropagate gradients into the policy
  - Requires some tricks to make it work
- Use the model to learn a value function
  - Dynamic programming
  - Generate simulated experience for model-free learner (Dyna)

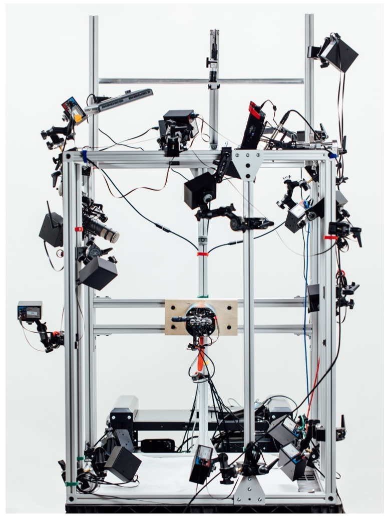
Examples of RL On Robots

Robotics

Robotics







Open AI Dactyl (2018)

Reinforcement Learning for Robust Parameterized Locomotion Control of Bipedal Robots, 2022

More RL for Robotics

- Guiness world record in 100 meters by biped robots (Oregon State University)
- Learned quadrupedal locomotion in challenging environments (ETH Zurich)
- Autonomous Navigation of Stratospheric Balloons (Google AI), blog (was real, just Google canceled the whole project.. sadly..)
- Not yet perching (article), but soon? Just for inspiration..
- Video games; car racing in video games, competing with humans
- Vision-based autonomous drone racing (video, UZH RPG)
- Commanding robots using natural language to perform tasks (SayCan project, Google)
- behavioral cloning/imitation learning (not RL) is doing well with transformers in the kitchen (Google)
- Yet it is not enough to learn to drive well
- Quadruped learns to walk in the park in 20 minutes, model-free (UC Berkeley)
  - More of this
- Still, dexterous manipulation is not easy.. (Berkeley, Meta, UW)
- Visual Navigation (Berkeley)
- In the need for resets (Berkeley)

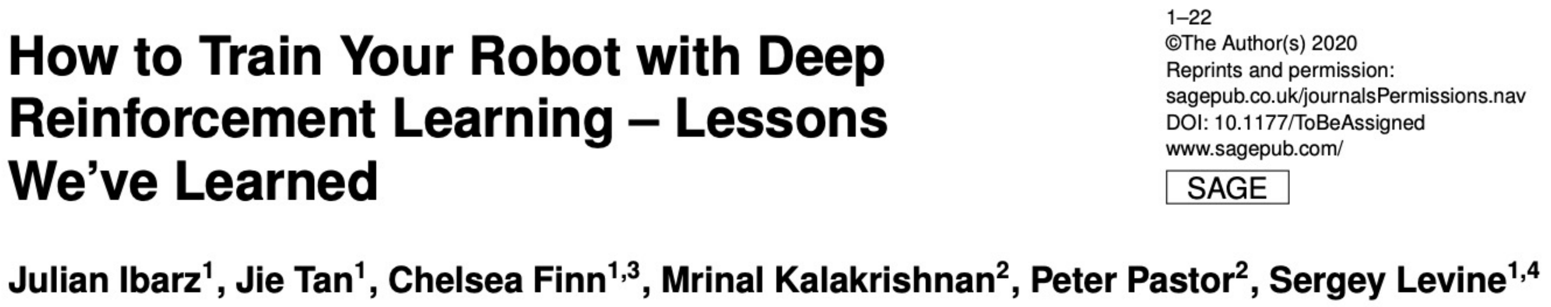
Credit: Csaba Szepasvari

RL For Robots: Challenges

- How sample-efficient and stable is RL optimization?
- Does it make sense to ignore that we may know dynamics / physics $P(s'|s,a)$?
  - Simulation, residual learning, robot-aware learning etc.
- Where do you get episode resets from?
  - Reset-free RL etc.
- Where do you get rewards from?
  - Learning from examples, demonstrations etc.
- Is it fair to treat time as discrete?
- Is it fair to treat $s_t, and\ a_t = \pi(s_t)$ as happening at the same instant?
  - Delay-aware methods

RL For Robots: Challenges

- Partial observability: is the Markov state actually available to the agent?
  - No! This is particularly important in the context of this class!
- How are demonstrations provided?
- Will learning by trial and error damage my robot / other equipment / me?
- Do I have to learn from scratch for each robot?
- Non-stationarity, e.g. my robot deteriorates over time?



How to Train Your Robot with Deep Reinforcement Learning – Lessons We've Learned

1–22
©The Author(s) 2020
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Julian Ibarz[1], Jie Tan[1], Chelsea Finn[1,3], Mrinal Kalakrishnan[2], Peter Pastor[2], Sergey Levine[1,4]

arxiv.org/pdf/2102.02915.pdf