



CIS 4190/5190: Lec 22 Wed Nov 19,
2024

Reinforcement Learning: Review + Part 3

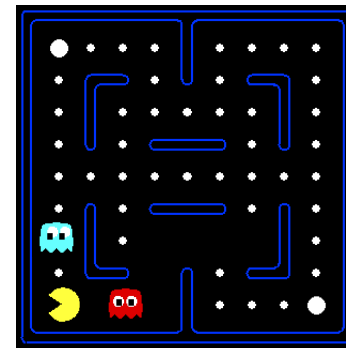
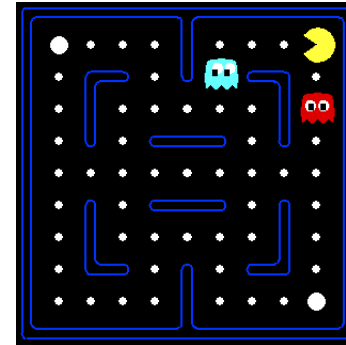
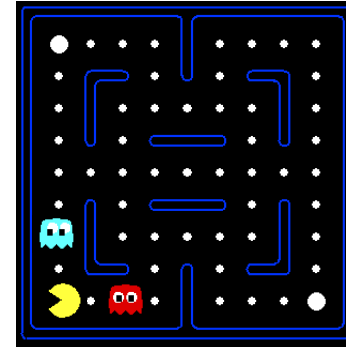
Recap

- Q-Learning: We can modify Q-value iteration when P and R are unknown:
 - Treat each sample from the distribution as a coarse proxy for the mean
 - Make updates *incremental*
- Deep Q-Learning v1:
 - To handle high-dimensional states, replace table by a deep network that maps (s, a) to $Q(s, a)$
 - Convert incremental update to gradient descent update

From Tabular to Deep Q Learning

High-dimensional states example: Pacman

- Let's say we discover through experience that this state is bad:
- In naïve Q-learning, we know nothing about this state or its Q states:
- Or even about this one!

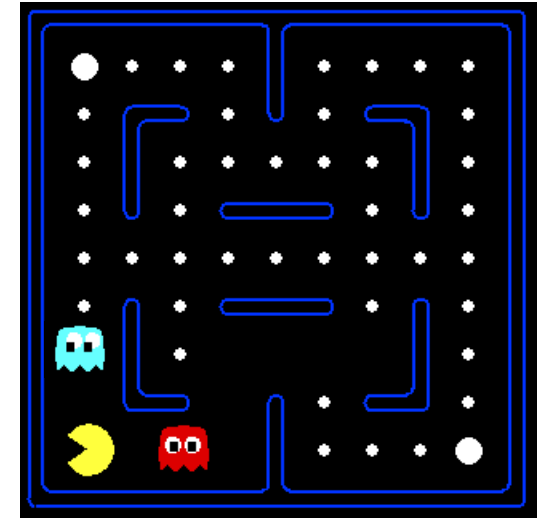
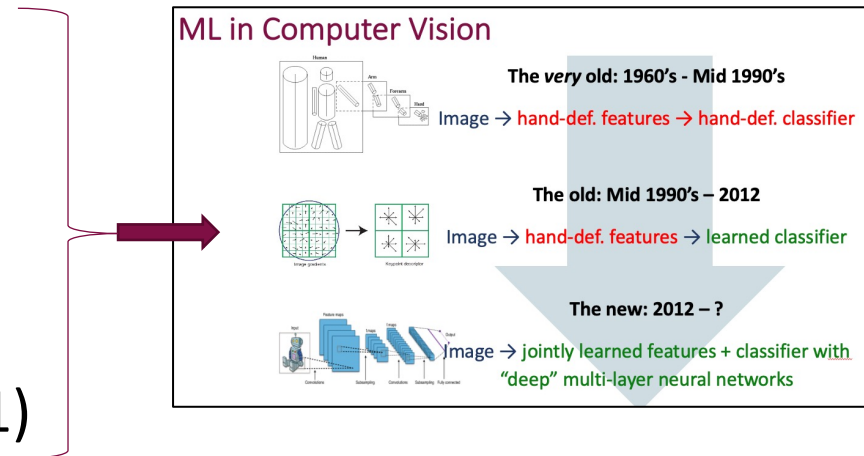


Q-Learning

- In many real situations, we cannot possibly learn about every single state+action!
 - Too many state-action pairs to visit them all in training
 - Too many state-action pairs to hold the Q-tables in memory
- Instead, we want to generalize:
 - Learn about some small number of training Q-states from experience
 - Generalize that experience to new, similar Q-states
 - This is a fundamental idea in machine learning, and we see it over and over again

Feature-Based Representations

- Solution: describe a state using a vector of features
 - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
 - Example features:
 - Distance to closest ghost
 - Distance to closest dot
 - Number of ghosts
 - $1 / (\text{dist to dot})^2$
 - Is Pacman in a tunnel? (0/1)
 - etc.
 - Can also describe a q-state (s, a) with features
 - e.g. action moves closer to food

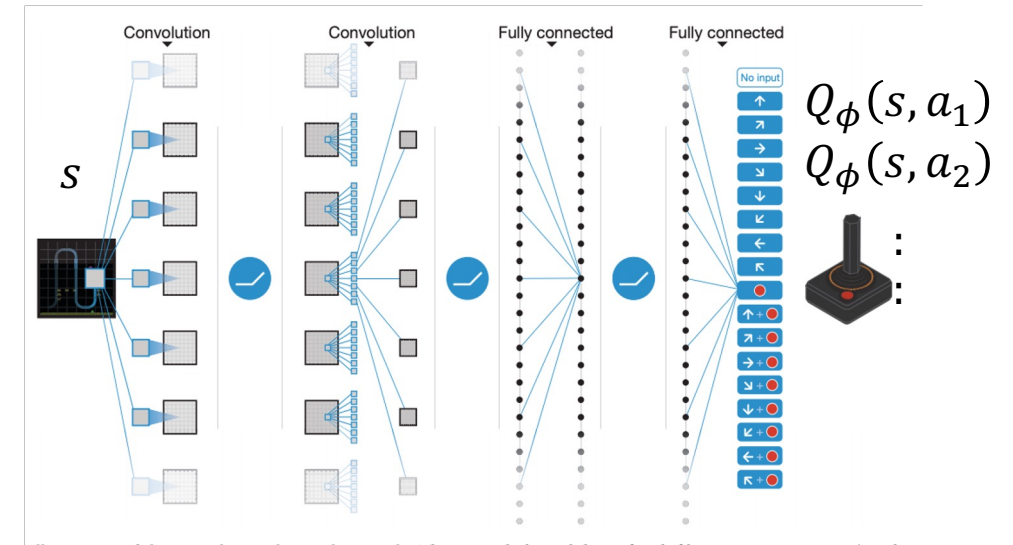


As we now do in computer vision/NLP, can we avoid engineering these features?

A Neural Network to Predict Q from “Raw” State Input

Predict Q-values with a deep neural network

- **Input:** the state, e.g. an image
- **Output:** Q-values of various actions
- **Learning:**



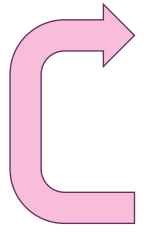
gradient descent* with the squared Bellman error loss:

$$\left(\left(R + \gamma \max_{a'} Q_\phi(s', a') \right) - Q_\phi(s, a) \right)^2$$

$= y_i$

As always, the policy action is the one with the highest predicted Q-value

Deep Q-Learning v1

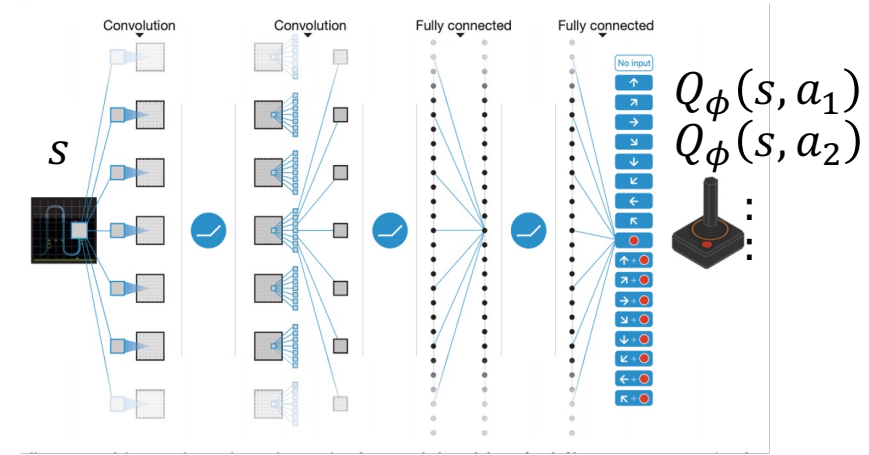


1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$

2. $\mathbf{y}_i = r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}')$

3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

$$= \frac{d}{d\phi} (Q_\phi - y_i)^2$$



Note: we pretend that y_i is a constant while computing the gradient, to resemble regression

Incremental update step \rightarrow gradient descent* on the squared Bellman error loss!

Closely connected to the tabular Q learning update. Hint: if you replace the neural network with a Q table, its parameters ϕ are just Q value entries?

- 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:

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

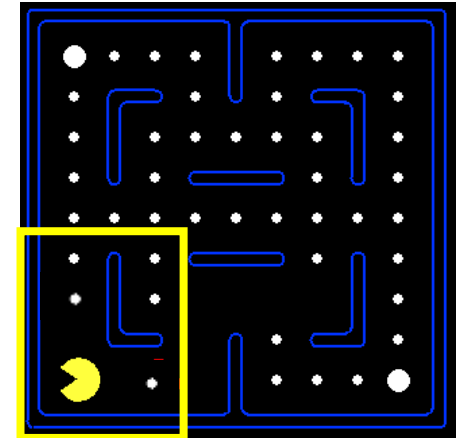
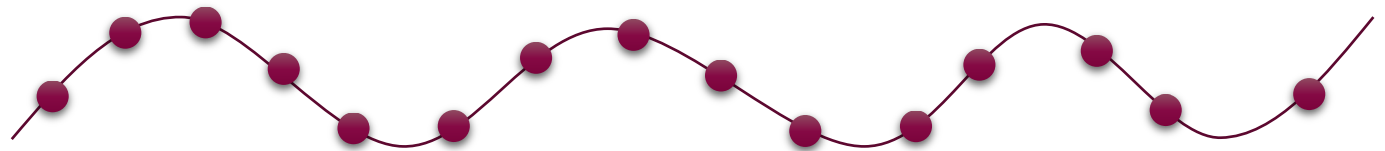
Problems with Deep Q-Learning v1

- ↪
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}')] \right)$

Problems:

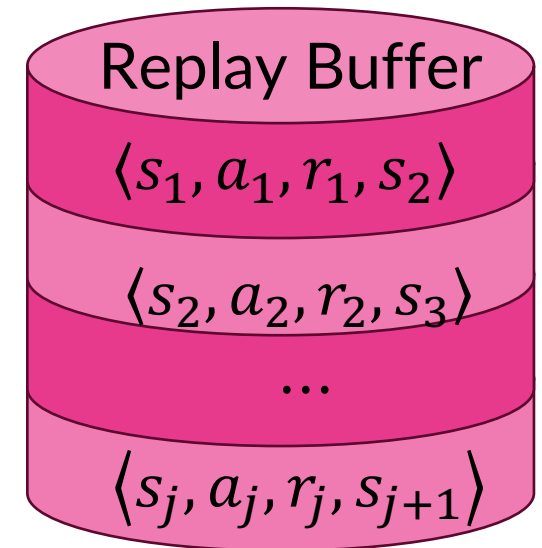
1. sequential states are strongly correlated (not i.i.d.) ↪

So consecutive Q updates drive the network to overfit to recently encountered states and forget previous experiences



Addressing Correlations: Experience Replay

- Q-Learning is “off-policy”: we don’t say anything about the specific actions that need to be executed, and we don’t need the transitions to be in sequence.
- Maintain a “replay buffer” of previous experiences
- Perform Q-updates based on a sample from the replay buffer
- Advantages:
 - Breaks correlations between consecutive samples
 - Each experience step may influence multiple gradient updates



FIFO or Priority Queue

Deep Q Learning v2 (with replay buffer \mathcal{D})

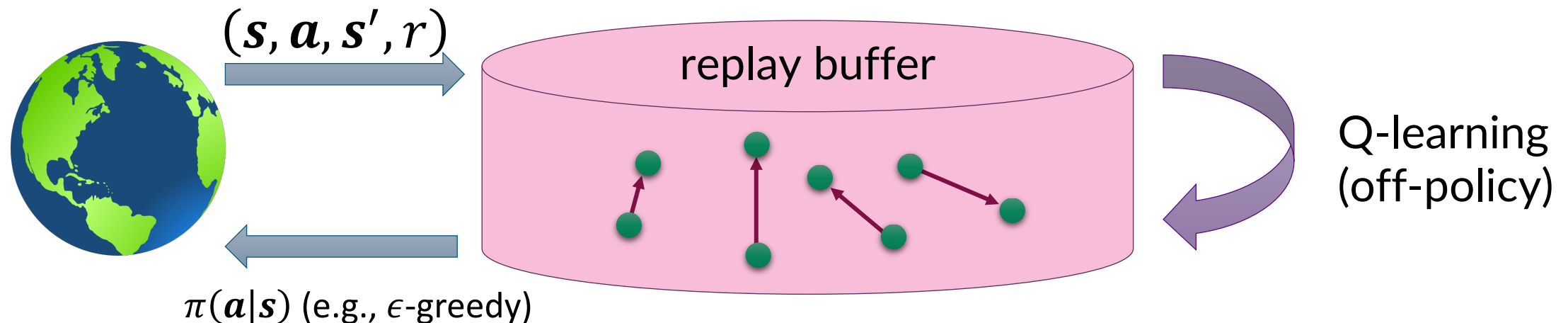
Deep Q Learning v1

1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)] \right)$

Deep Q Learning v2

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{D}
2. Loop K times, do:
 3. sample a batch of $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$'s from \mathcal{D}
 4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)] \right)$

$K = 1$ is common, though larger K may sometimes be more efficient

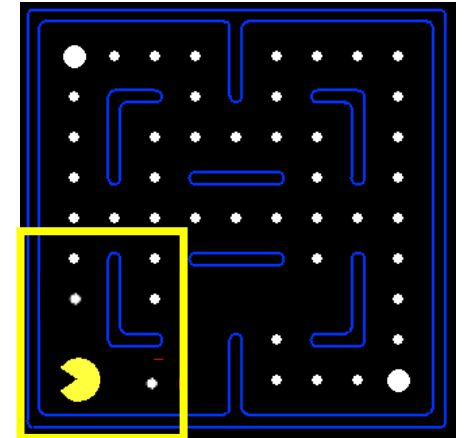


Problems with Deep Q-Learning v1

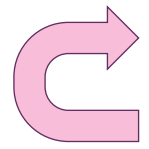
- ↪
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}')] \right)$

Problems:

1. sequential states are strongly correlated (not i.i.d.)
2. Target value is always changing!



Problem: Moving Target for Q-regression



1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$

$$2. \phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \underbrace{[r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)]}_{\text{no gradient through target value}} \right)$$

no gradient through target value

Problem: Instability (e.g., rapid changes) in $Q(\cdot)$ can cause it to diverge

- Q-learning is *not* gradient descent on any fixed objective!

Solution: use two nets to provide stability

- The Q-network is updated regularly
- The target network is an older version of the Q-network, updated occasionally

$$\left(\underbrace{Q_\phi(s, a)}_{\text{computed via Q-network}} - \left(r_i + \gamma \max_{a'} \underbrace{Q_{\phi'}(s', a')}_{\text{computed via target network}} \right) \right)^2$$

Deep Q Learning v3

Deep Q Learning v2

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{D}
2. Loop K times, do:
 3. sample a batch of $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$'s from \mathcal{D}
 4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

Deep Q Learning v3

1. save target network parameters: $\phi' \leftarrow \phi$
 2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{D}
 3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{D}
 4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r_i + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$
- targets don't change in inner loop!
- supervised regression

This is the “classic” deep Q Learning algorithm from 2015!*

*(usually $K=1$)

Imitation Learning Through Behavior Cloning

Solving sequential decision making problems with supervised learning!



Imitation of Televised Models by Infants

Andrew N. Meltzoff
University of Washington

Supervised learning of Action Policies?

- Given the current “state” s , make a decision $\hat{y} = \max_y \pi_\theta(a|s)$.
 - Supervision => labels for “good” decisions that maximize future rewards.
 - So, we’d like to have some dataset of (state s , good decision a) pairs. Then we could try running supervised learning just as always.

Behavior Cloning (BC)



expert

observed states

s_1, s_2, \dots, s_H

a_1, a_2, \dots, a_H

actions

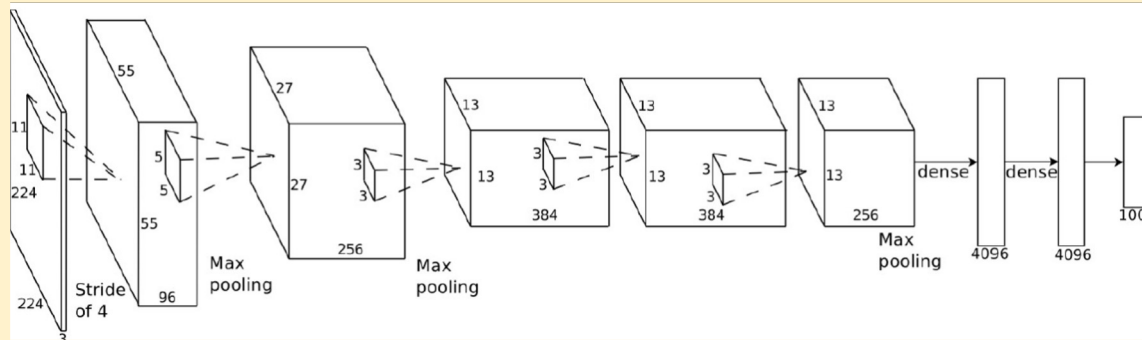
training
data

supervised
learning

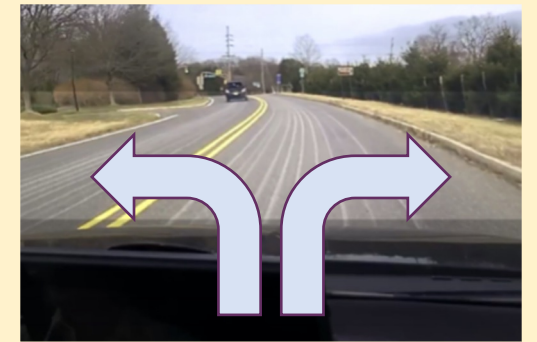
policy
 $\pi_\theta(a_t | s_t)$



observed state s_t



convolutional network



action a_t

Behavior Cloning Objective Function

Supervised maximum-likelihood objective to train a function that maps from expert sensory inputs to expert actions.

$$Loss = -\frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \log \pi_{\theta}(a_{i,t} | s_{i,t}) \right)$$

Demonstration data Expert actions

Could minimize by following the gradient:

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

trajectories time

Likelihood gradient:
“Change the policy to make these actions more likely”.

Does this work?

Clean Restroom (teleop)



10x speed

Mobile Aloha, Stanford

Wipe Wine
(autonomous)



Real-time

Use Cabinets
(autonomous)
(lift a 3 lbs pot)



Real-time

Typically requires a large number of demonstrations (a few 100s) to learn well.
Mobile Aloha, Stanford

...

Exhibit A: Zero-Shot Policies From Web Human Videos



Could we learn general skill policies
without specific reference to any one robot, scene, objects?



Shi et al 2024, (under review)





“Off-The-Shelf” Imitation From Web Human Videos Alone



Step 3: Deploying Human Arm Policies on Robot Arms*

Fang et al, AnyGrasp, T-RO 2022

Bahl et al, VRB, CVPR 2023

Skill	Slide Opening	Slide Closing	Vertical Hinge Opening	Vertical Hinge Closing
Object	Drawer	Drawer	Cupboard	Cupboard
Rollout				

Shi et al 2024, (under review)

“Off-The-Shelf” Imitation From Web Human Videos Alone



Step 3: Deploy Human Arm Policies on Robot Arms*

Skill library

Slide Opening

Slide Closing

Vertical Hinge
Opening

Vertical Hinge
Closing

Skill

Pouring

Pouring

Stirring

Stirring

Object

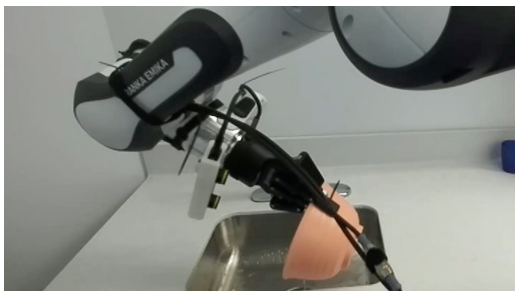
Water Into Sink

Pasta Into Pot

Food in Pot

Pasta In Water

Rollout


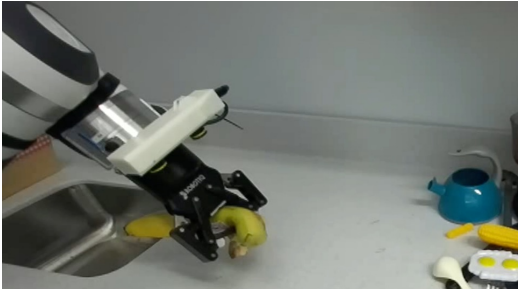




Shi et al 2024, (under-review)



“Off-The-Shelf” Imitation From Web Human Videos Alone

Step 3: Deploy Human Arm Policies on Robot Arms*

Skill library	Slide Opening	Slide Closing	Vertical Hinge Opening	Vertical Hinge Closing
	Pouring	Stirring		
Skill	Picking	Picking	Placing	Placing
Object	Can	Banana	Cup	Pasta Into Drawer
Rollout				



“Off-The-Shelf” Imitation From Web Human Videos Alone

Step 3: Retargeting Human Arm Policies to Robot Arms*

Skill library

Slide Opening

Slide Closing

Vertical Hinge
Opening

Vertical Hinge
Closing

Pouring

Stirring

Picking

Placing

Skill

Cutting

Cutting

Pouring

Cutting

Object

Tofu

Banana

Salt Into Pan
(Unseen)

Cake
(Unseen)

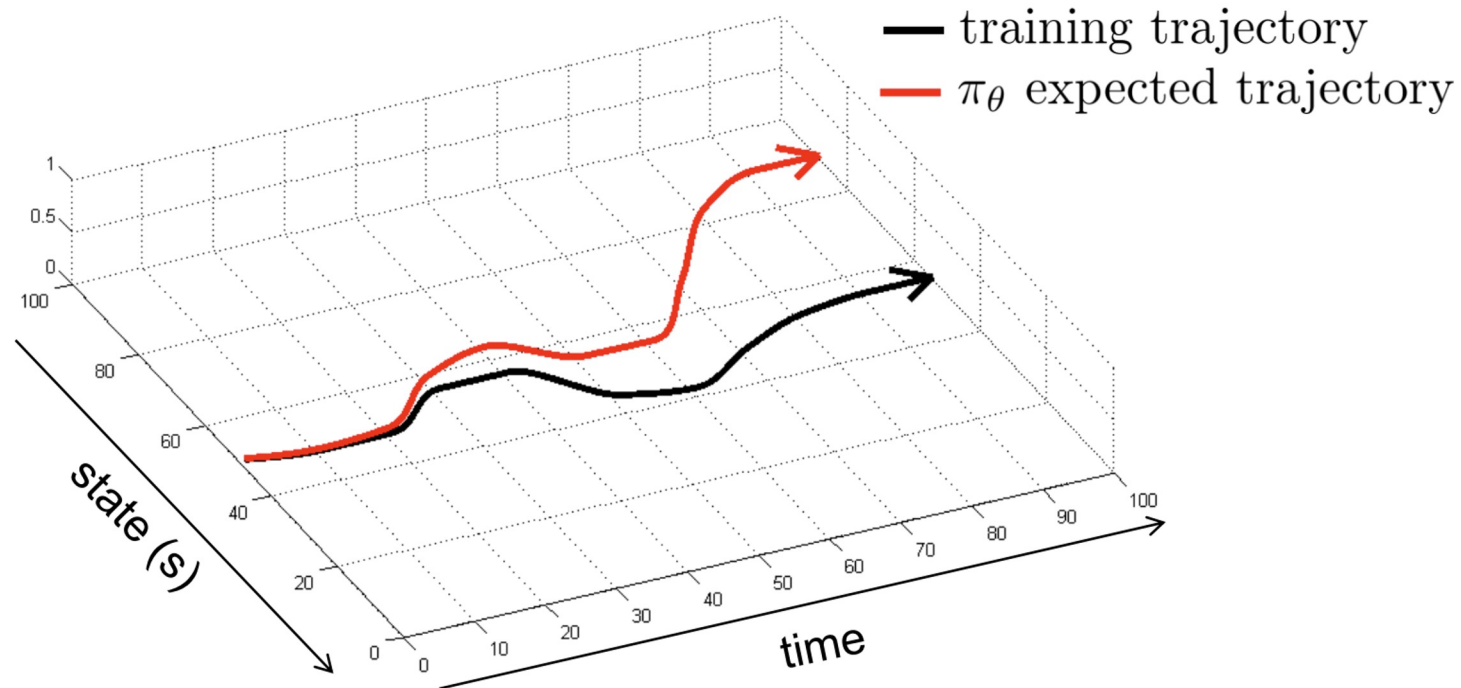
Rollout



Shi et al 2024, (under review)

Key Issue with BC: Distributional Shift

The policy is trained on *demonstration data* that is different from the data it encounters in the world.




The cloned policy is imperfect; this leads to “compounding” errors, and the agent soon encounters unfamiliar states, leading to failure.

Note how these errors arise from ignoring the the *sequential, interconnected* nature of the task. Past decisions influence future states!

Active Behavior Cloning: DAGGER

A general trick for handling distributional shift: requery expert on new states encountered by the initial cloned policy upon execution, then retrain.

- 
1. Train $\pi_{\theta}(a_t|s_t)$ from expert data $\mathcal{D} = \{s_1, a_1, \dots, s_N, a_N\}$
 2. Run $\pi_{\theta}(a_t|s_t)$ to get dataset $\mathcal{D}_{\pi} = \{s_1^{new}, \dots, s_M^{new}\}$
 3. Ask expert to label each state in \mathcal{D}_{π} with actions a_t^{new}
 4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_{\pi}$

Assumes it is okay to keep asking the expert all through the training process.

“Queryable experts”. Might not always be practical.

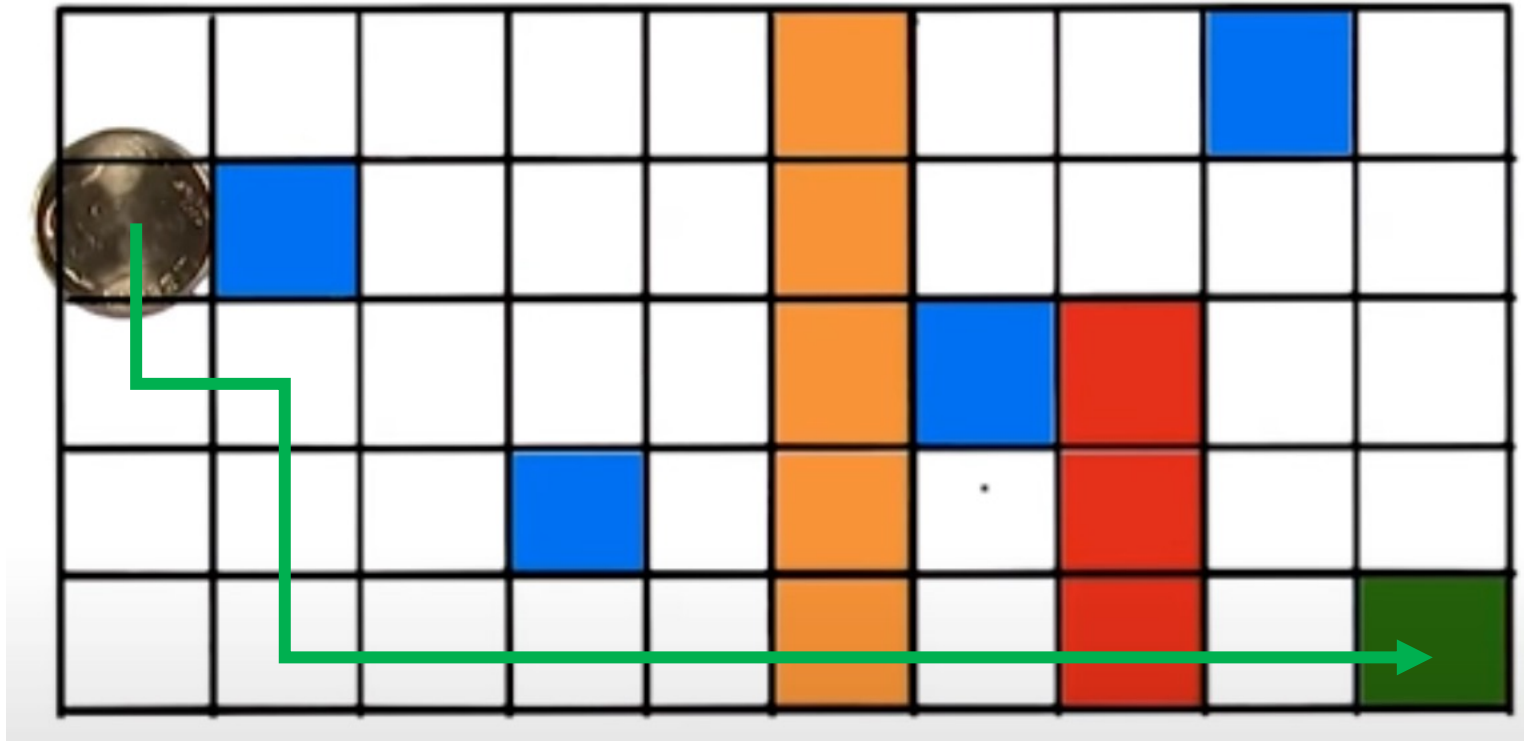
Aside: Distribution Shift More Broadly

- When supervised ML systems are deployed, it is common for the distribution to shift.
 - E.g. when a new spam classifier is deployed on gmail, spammers might notice that their old spamming techniques are not working, and innovate to break the new spam classifier.
- One strategy to fix this is continuous data aggregation, like in DAGGER.
 - E.g., Allow users to mark new emails that slip through the filter as spam. Add these to the training data, and retrain the spam classifier from time to time.

Lesson: ML systems *are* often deployed in sequential decision making settings without realizing it: later inputs may be influenced in some complex way by older decisions of the ML system. Warrants caution!

Other Ways to Do Imitation

- BC might not generalize beyond demonstrations. Instead learn explicitly about the “reward” function that the demonstrator is trying to maximize?
 - This is called “inverse reinforcement learning”



Would you conclude that this agent likes / dislikes:

- Blue squares?
- White squares?
- Orange squares?
- Red squares?
- Green square?

Knowing the *reward* could inform more generalizable imitation, e.g. starting from a different location than expert

BC Operates Per-Timestep, Not Aware of Future Impacts

- Suppose you try to imitate driving. The imitator is not perfect, and you either:
 - Are slower by 5 mph than the expert behavior on a highway, or
 - Are off by 5 mph as you start your car in your garage (e.g. moving forward at 4 mph, instead of backing out at 1 mph).
 - **BC objective might value both errors similarly, but one is much worse!**

BC objective, by simply mimicking the immediate expert actions, is not aware of any future impacts of an agent's actions.

Compare to RL which tries to explicitly optimize $\sum_t r_t$.

Imitation vs RL

- Imitation is often *very* useful. In most cases where you have access to clean expert demonstrations, you should aim to use it through some kind of imitation. But there are limitations.
- Compared to RL, BC usually takes the short-term myopic view:
 - The BC loss is only per-timestep deviations from the expert actions.
 - It does not account for the impacts of current actions on the future.
- More broadly, imitation is limited to mimicking experts and cannot discover new solutions. What about solving new problems, like controlling a new robot, or beating the world's best Go player? RL is your best bet.
- There are also ways to naturally combine imitation and RL (beyond the scope of this class).

Policy Gradients

RL That Looks *A Little Bit* Like Behavior Cloning



Recall: Behavioral Cloning for Imitation Learning

The BC gradient w.r.t. policy parameters θ looked like:

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

Demonstration data

Expert
actions



“Policy Gradient” Methods

Update policy parameters with the gradients of the expected utility in an episode by following policy π_θ

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} + \alpha \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_\theta} \left[\sum_{t'=t} r_{t'} \right]$$

π_θ induces a trajectory distribution, which induces a reward distribution.

It turns out that the gradient $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_\theta} [\sum_t r_t]$ works out to:

$$\frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\boldsymbol{\theta}} \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \sum_{t'=t}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

Note: we will focus for now on “finite-horizon” settings, i.e., T is finite.

Note: we are ignoring discount factors for now, all formulae will easily generalize



Compare to the BC Gradient

BC Gradient: Demonstration data Expert actions

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

RL “Policy Gradient”:

~~Demonstration data~~ ~~Expert actions~~

$$\frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \left[\sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}) \right] \right)$$

trajectories

time

Likelihood gradient: “Change policy to make these actions more likely”.

Critic: “how good was this trajectory?”
--- credit assignment

Recall: we start out with no data at all ... so where does this data come from?

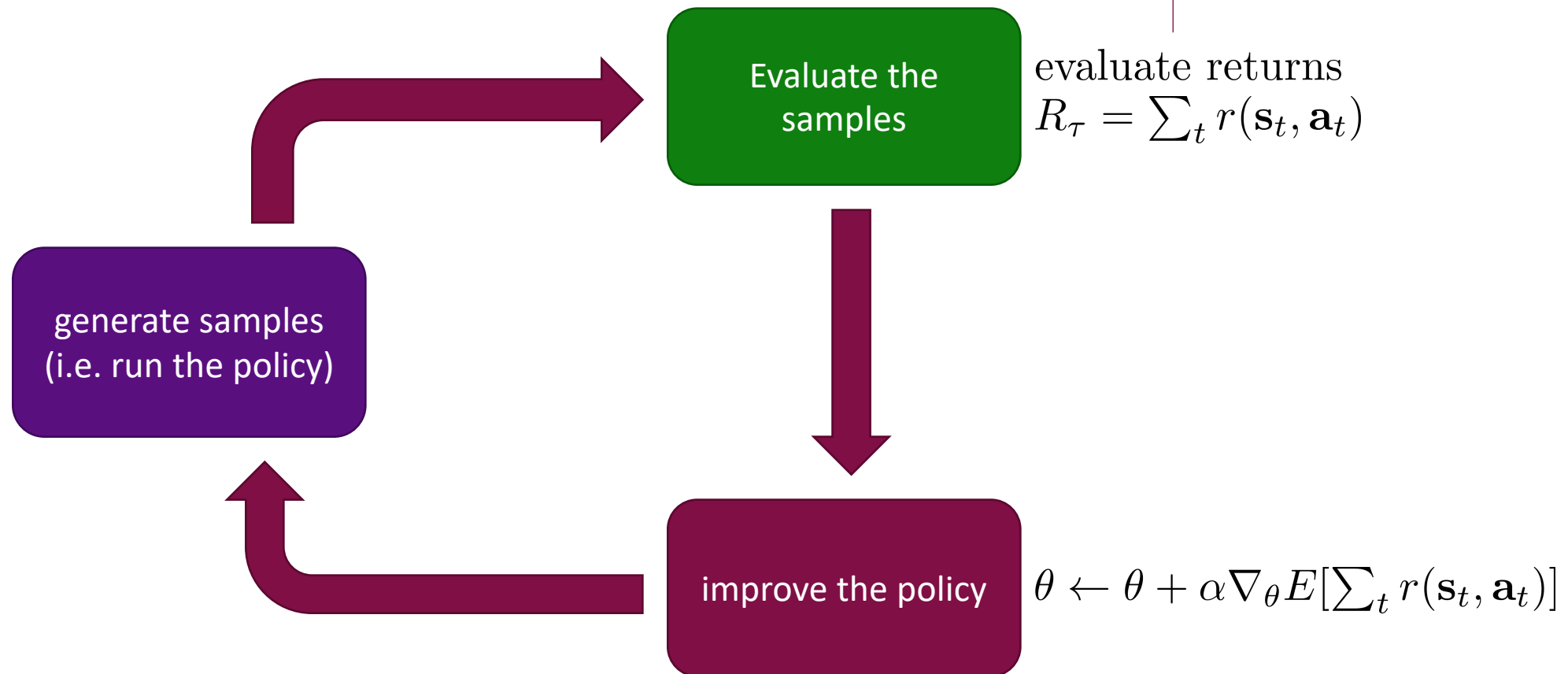
Ans: We generate our own data during learning ... this is trial-and-error learning!



The basic policy gradients algorithm: REINFORCE

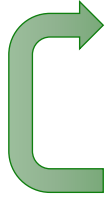
REINFORCE algorithm:

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



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

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!

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





Causal policy gradient

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

Causality: policy at time t' cannot affect reward at time t when $t < t'$

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

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

“On-Policy” Learning

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \left(\sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right)$$

- The policy gradient increases the likelihood of those past actions that yielded good eventual utility **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.

Whither Exploration?

- **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
$$r_t \leftarrow r_t + \lambda H(\pi_\theta(a_t|s_t))$$
which incentivizes more uncertain policies, inducing more exploration.
 $\lambda \rightarrow 0$ during training.

“Policy Gradient” with Discount Factor γ

With discount factor set to 1, the policy gradient we have seen is:

$$\frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \left[\sum_{t'=t}^T r(s_{i,t}, a_{i,t}) \right] \right)$$

With non-trivial discount factors, the policy gradient simply changes to:

$$\frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \left[\sum_{t'=t}^T \gamma^{t'-t} r(s_{i,t}, a_{i,t}) \right] \right)$$



Policy gradient with automatic differentiation

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \left(\sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right)$$

How can we compute policy gradients with automatic differentiation?

We need a graph such that its gradient is the policy gradient!

Just implement “pseudo-loss” as a weighted maximum likelihood:

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

 cross entropy (discrete) or squared error (Gaussian)

Policy gradient with automatic differentiation

Pseudocode example (with discrete actions):

Maximum likelihood as in behavior cloning:

```
# Given:
# actions - (N*T) x Da tensor of actions
# states - (N*T) x Ds tensor of states
# 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)
loss = reduce_mean(negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

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)
```

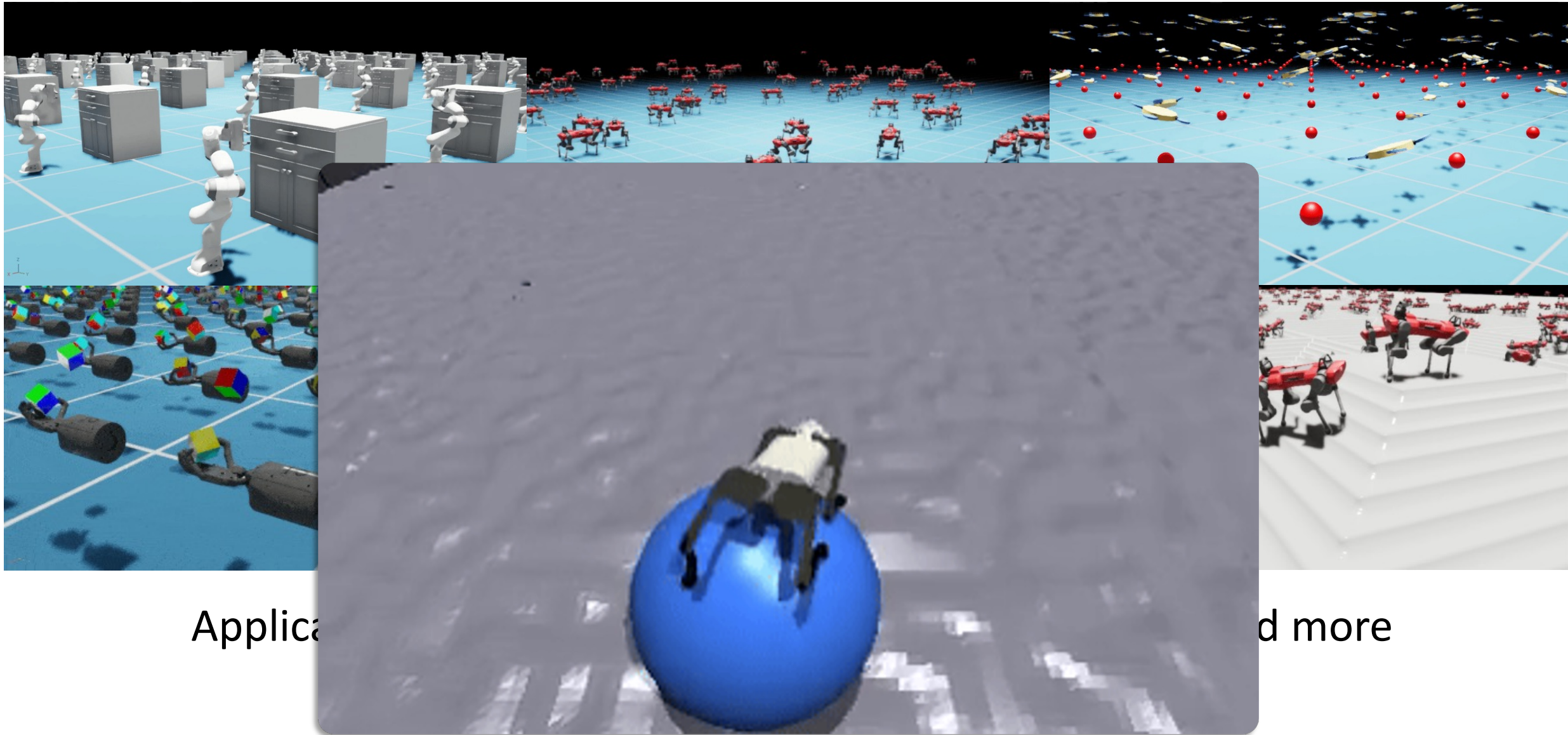
$$\tilde{J}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$$

q_values

Policy gradient in practice

- Remember that the “policy gradient” of expected utility has high variance.
 - Expected utility is estimated by sampling a small number of trajectories from the policy.
 - This isn’t the same as supervised learning!
 - Gradients are often very noisy!
- Consider using much larger batches to reduce the variance
- Tweaking learning rates is very hard
 - Adaptive step size rules like ADAM can be OK-ish
 - We’ll learn about policy gradient-specific learning rate adjustment methods later!
- Popular policy gradient approaches today: PPO, TRPO ...
- RL implementation details can be hard to get right. Good to start with **popular repositories: OpenAI stable-baselines, CleanRL etc.**

Scaling Simulated Experiences



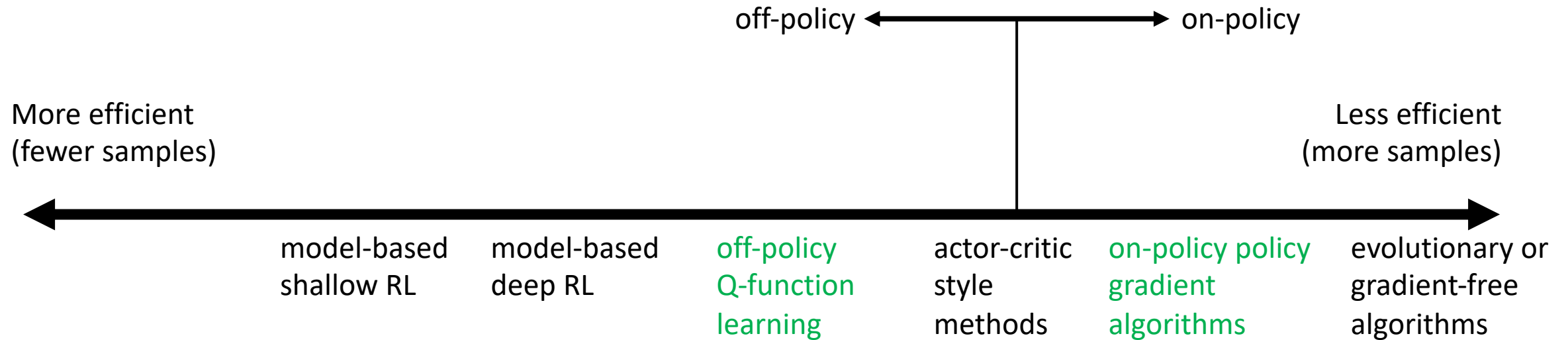
Applica

d more



1X speed

Many More Kinds of RL Algorithms



But policy gradients are among the most stable approaches that work most broadly, and take limited wall clock time even though many samples.