Deep Reinforcement Learning for Control & Robotics

How it Works, Where it Works, and Where it Doesn’t (yet!)

Jonathan Scholz

DeepMind

Goals of This Tutorial

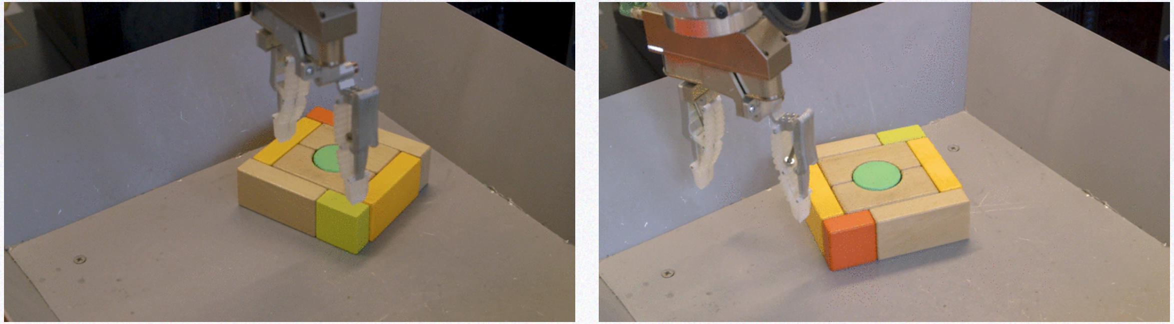
1. Intuitive understanding of how RL algorithms work
2. Survey of Policy Gradient Methods
3. How can you apply this to your robotics problems?

Outline

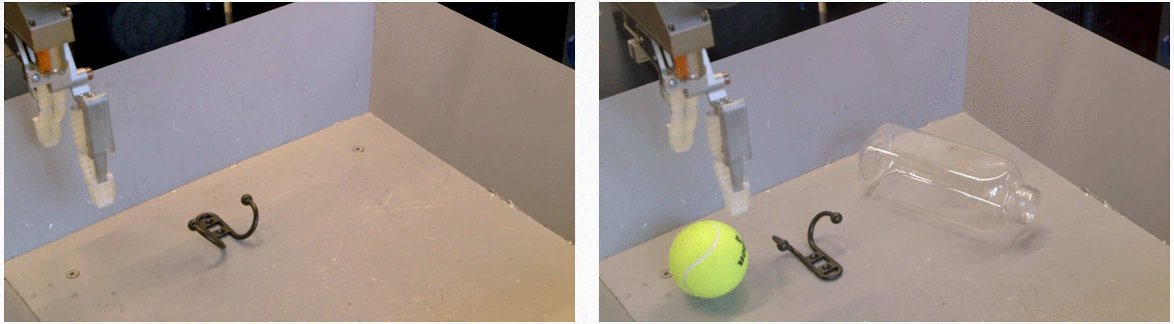
- Motivational videos
- Part 1: Q-Learning Walkthrough
- Part 2: Policy-Gradient Survey
  - Vanilla Policy-Gradient Methods
  - Value-Gradient Methods
- Open Challenges

RL Success Stories — Grasp (QT-Opt)

Singulation



Learned reactive grasp behaviors



video source

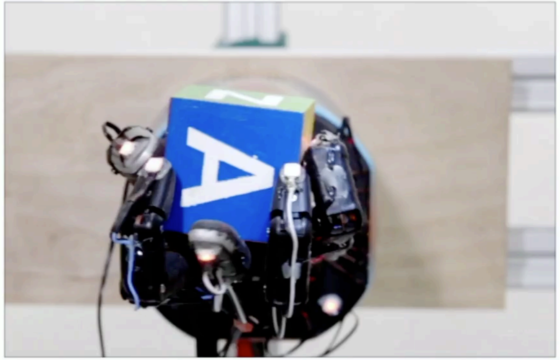
RL Success Stories — Locomotion (ANYmal)

Recovery Behaviors

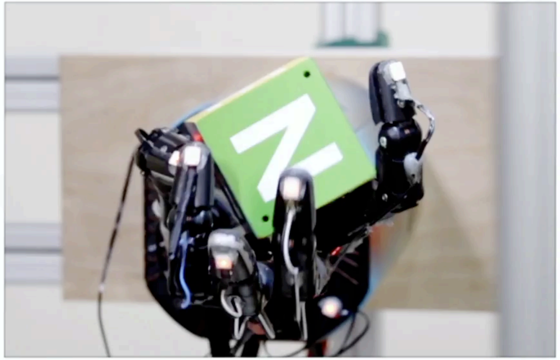
Stairs

video source

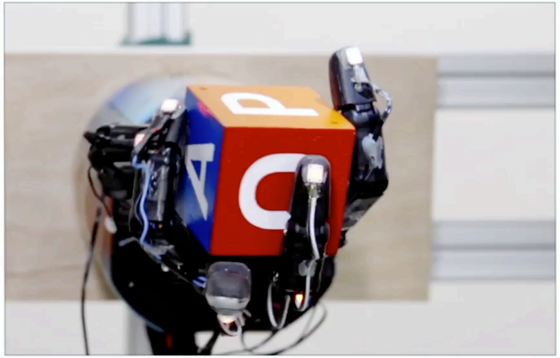
RL Success Stories — Manipulation (OpenAI)



FINGER PIVOTING



SLIDING



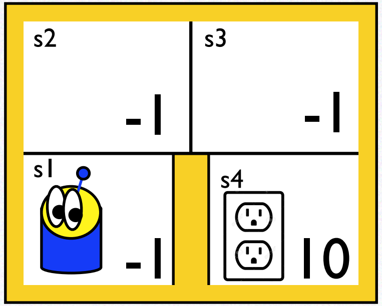
FINGER GAITING

video source

Outline

- Motivational videos
- Part 1: Q-Learning Walkthrough
- Part 2: Policy-Gradient Survey
  - Vanilla Policy-Gradient Methods
  - Value-Gradient Methods
- Open Challenges

Markov Decision Process



$$MDP = \{S, A, T, R, (\gamma)\}$$

$$S = \{s_1, s_2, s_3, s_4\}$$

$$A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$$

$$T = P(s'|s, a)$$

$$R = \begin{cases} 10, & s = s_4 \\ -1, & \text{otherwise} \end{cases}$$

Problem:
sometimes we
can't do this

Bellman

$$V(s) = r(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')$$
$$= \max_a Q(s,a)$$

Value-Iteration

```
while $\forall s \in S$ :    $|V_k(s) - V_{k+1}(s)| > \epsilon$ do
    $V_{k+1}(s) \leftarrow \max_a [r(s,a) + \gamma \sum_{s'} P(s'|s,a)V_k(s')]$
end while
```

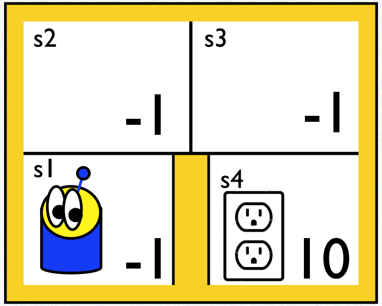
From Q-function to Q-Learning

➡ Key question: How to remove dependency on model?

$$
\begin{aligned}
Q(s,a) &= R(s,a) + \gamma \max_{a'} \sum_{s'} P(s'|s,a) Q(s',a') && \text{by definition} \\
&\approx R(s,a) + \gamma \max_{a'} Q(s',a'), \quad s' \sim P(s'|s,a) && \textbf{sample approximation} \\
&\approx (1-\alpha) Q(s,a) + \alpha \left(R(s,a) + \gamma \max_{a'} Q(s',a') \right) && \text{smoothing} \\
&\approx Q(s,a) - \alpha Q(s,a) + \alpha R(s,a) + \alpha\gamma \max_{a'} Q(s',a') && \\
&\approx Q(s,a) + \alpha \left(R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) && \text{canonical form} \\
&\approx Q(s,a) + \alpha(\delta_{TD}) && \text{TD error}
\end{aligned}
$$

DeepMind

Q-Learning

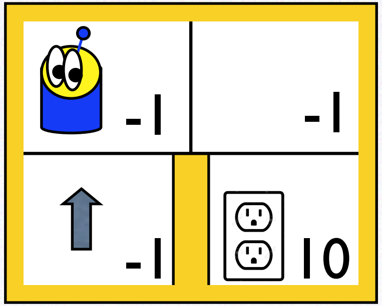


$\alpha = .7$

	↑	↓	←	→
S_1	0	0	0	0
S_2	0	0	0	0
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Q-Learning

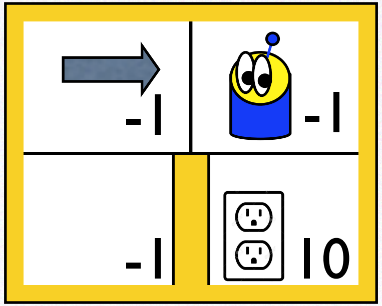


$$Q^{est}(S_1, \uparrow) =$$
$$.7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	0
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Q-Learning



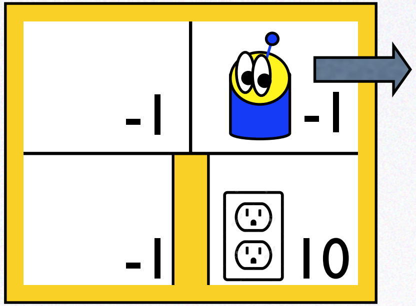
$Q^{est}(S_2, \rightarrow) =$

$.7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Q-Learning

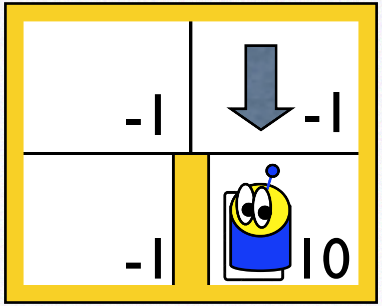


$$Q^{est}(S_3, \rightarrow) =$$
$$.7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	0	0	-.7
S_4	0	0	0	0

Q-Table

Q-Learning

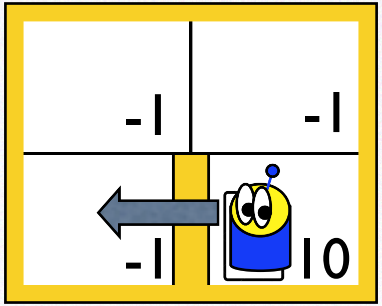


$Q^{est}(S_3, \downarrow) =$
$.7(-1 + .9 \max (0, 0, 0, 0)) + .3 \times 0$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	-.7	0	-.7
S_4	0	0	0	0

Q-Table

Q-Learning

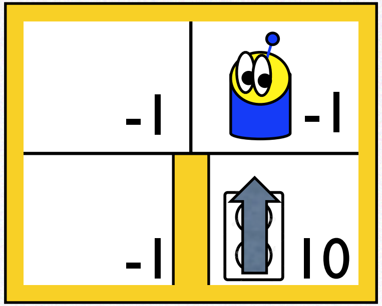


$Q^{est}(S_4, \leftarrow) =$
$.7(10 + .9 \max(0, 0, 0, 0)) + .3 \times 0$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	-.7	0	-.7
S_4	0	0	7	0

Q-Table

Q-Learning

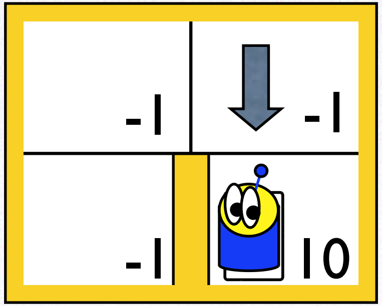


$$Q^{est}(S_4, \uparrow) =$$
$$.7(10 + .9 \max(0, -.7, 0, -.7)) + .3 \times 0$$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	-.7	0	-.7
S_4	7	0	7	0

Q-Table

Q-Learning



$Q^{est}(S_3, \downarrow) =$
$.7(-1 + .9 \max(7,0,7,0)) + .3 \times -.7$

	↑	↓	←	→
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	3.5	0	-.7
S_4	7	0	7	0

Q-Table

Pros and Cons of Tabular Q-Learning

Converges… eventually

Pros

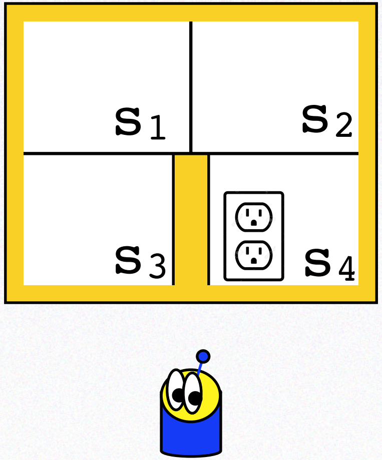
- Optimality guarantees
- Monotonic policy improvement*
- Does not require knowing a transition model

Cons

- Scales horribly … Curse of dimensionality
- Only works for discrete state-action spaces

*if doing full policy-evaluation before updating

Linear Function Approximation



One-hot encoding of states and actions

s_1 = [1, 0, 0, 0]
s_2 = [0, 1, 0, 0]
…

↑ = [1, 0, 0, 0]
↓ = [0, 1, 0, 0]
…

Linear Function Approximation

Represent Q as a *linear function of features*

$$Q(s,a) = \begin{pmatrix} \theta_{s_1} \\ \theta_{s_2} \\ \dots \\ \theta_{a_1} \\ \theta_{a_2} \\ \dots \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 1 \\ \dots \end{pmatrix}$$

Non-Linear Function Approximation

Represent Q as a *non-linear function* of features, *e.g.:*

$$Q(s,a) = \theta_{A_1} \max(0, \theta_{A_2} s + \theta_{A_3} a + \theta_B)$$

Neural Networks

Represent Q as a non-*linear function of features, eg*

$$Q(s,a) = \theta_{A_1}\mathrm{ReLu}(\theta_{A_2}s + \theta_{A_3}a + \theta_B)$$

$$\mathrm{ReLu}(x) = \max(0,x)$$

Deep Neural Networks

Compose Nonlinear Functions

$$Q(s, a) = \theta_{A_1} \mathrm{ReLu}(\theta_{B_1} + \theta_{A_2} \mathrm{ReLu}(\theta_{B_2} + \theta_{A_3} s + \theta_{A_4} a)$$

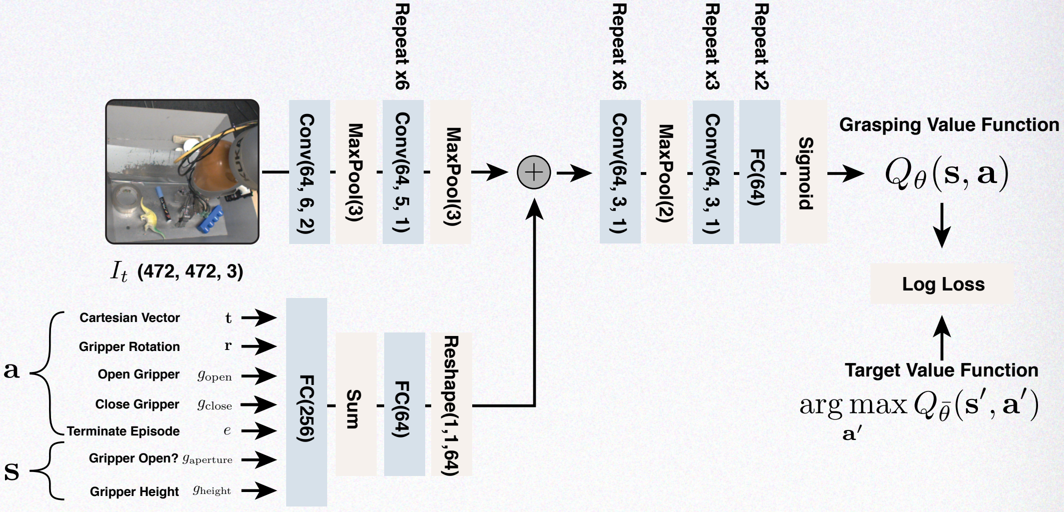
↑

Deep-RL!

Example Modern Deep RL Architecture

Key point: the RL algorithm doesn’t care about the *parameterization*

- ➡ Sees same 1-2 quantities:
  1. Action (log) probabilities
  2. Action-value estimate
- ➡ Nice property: RL losses can be used to drive DL representation learning



Q-Network from QT-Opt, Kalashnikov 2018

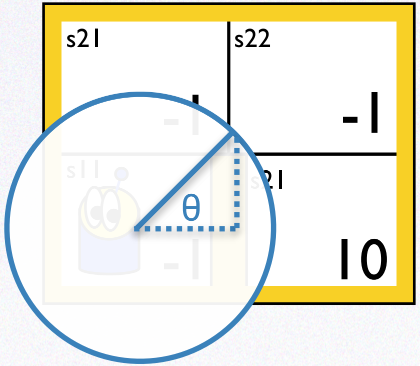
Q-Learning — Take aways

- Directly learns empirical return (cost-to-go)
    - Q absorbs all future outcomes in a single statistic
- Generic, but very sample-inefficient
- Only has global optimum guarantees in tabular setting
- Key to scaling = function approximation (rest of this talk)

Outline

- Motivational videos
- Part 1: Q-Learning Walkthrough
- Part 2: Policy-Gradient Survey
    - Vanilla Policy-Gradient Methods
    - Value-Gradient Methods
- Open Challenges

Motivation



(silly) Example Continuous Action Space

$$s' = s + \begin{bmatrix} \lfloor sin(\theta) \rceil \\ \lfloor cos(\theta) \rceil \end{bmatrix}$$

$a = \theta \in \mathbb{R}^1$

	0.0	0.01	0.02	…
s_{11}				
s_{12}				
s_{22}				
s_{21}				

Q… Table?

One Solution

Parameterize the policy explicitly!

can make this a
policy parameter too

E.g. a Gaussian Policy for continuous actions

$$\pi_\theta(s,a) = N(\phi(s)^T\theta, \sigma^2)$$

Can do with discrete actions too (SoftMax)

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^T\theta}$$

Some basis for state
(and actions), e.g. RBF

New problem: How to optimize the parameters of our policy?

Policy Optimization Problem Statement

- ***J***: an objective function measuring policy performance

$$J(\theta) = V^{\pi_\theta}(s_0)$$

- **Gradient of *J* w.r.t. *θ***: the direction to change each policy parameter to increase (or decrease) our objective

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- **Key question for this talk**: How to estimate this gradient efficiently?
  - ➡ **Simpler question**: how to estimate the gradient of the expectation of a function of a random-variable?

$$\boxed{\begin{array}{c} \nabla_\theta \mathbb{E}_{\pi_\theta}\left[V^{\pi_\theta}(s_0)\right] \\ \nabla_\theta \mathbb{E}_{p(z;\theta)}\left[f(z)\right] \end{array}}$$

Simplest Approach — Finite Differences

For each dimension *i* in *[1, n]*:

- ➡ estimate i^{th} partial-derivative by perturbing i^{th} component of θ by a small amount

$$J(\theta) = \mathbb{E}_{p(z;\theta)}\left[f(z)\right]$$

Requires *n* evaluations of *J* to compute gradient for policy with *n* parameters

- ➡ Each evaluation of *J* may involve numerous executions/simulations to approximate the expectation
- ➡ Inefficient, but simple and works for any policy, even if non-differentiable

$$\frac{\partial J(\theta)}{\theta_i} \approx \frac{J(\theta + \epsilon u_i) - J(\theta)}{\epsilon}$$

u_i is a vector with *1* in i^{th} component and *0* elsewhere

Detour: Score-Function Estimators

a.k.a. the log-derivative trick
a.k.a. likelihood-ratio

- Want to estimate $\mathbb{E}_{p(z;\theta)}[f(z)] \quad x \sim p(z;\theta)$
- Require $\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)]$ for optimization
- Useful identity: $\nabla_\theta \log p(z;\theta) = \dfrac{\nabla_\theta p(\mathbf{z};\theta)}{p(\mathbf{z};\theta)}$

Detour: Score-Function Estimators

a.k.a. the log-derivative trick
a.k.a. likelihood-ratio

$$\frac{\nabla_\theta p(\mathbf{s};\theta)}{p(\mathbf{s};\theta)}$$

$$\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)] = \int \nabla_\theta p(z;\theta) f(z) dz$$

← Exchange the derivative and the integral

$$= \int \frac{p(z;\theta)}{p(z;\theta)} \nabla_\theta p(z;\theta) f(z) dz$$

← Probabilistic identity trick

$$= \int p(z;\theta) \nabla_\theta \log p(z;\theta) f(z) dz$$

← Log-derivative trick

$$= \mathbb{E}_{p(z;\theta)}[f(z) \nabla_\theta \log p(z;\theta)]$$

← Back to expectation

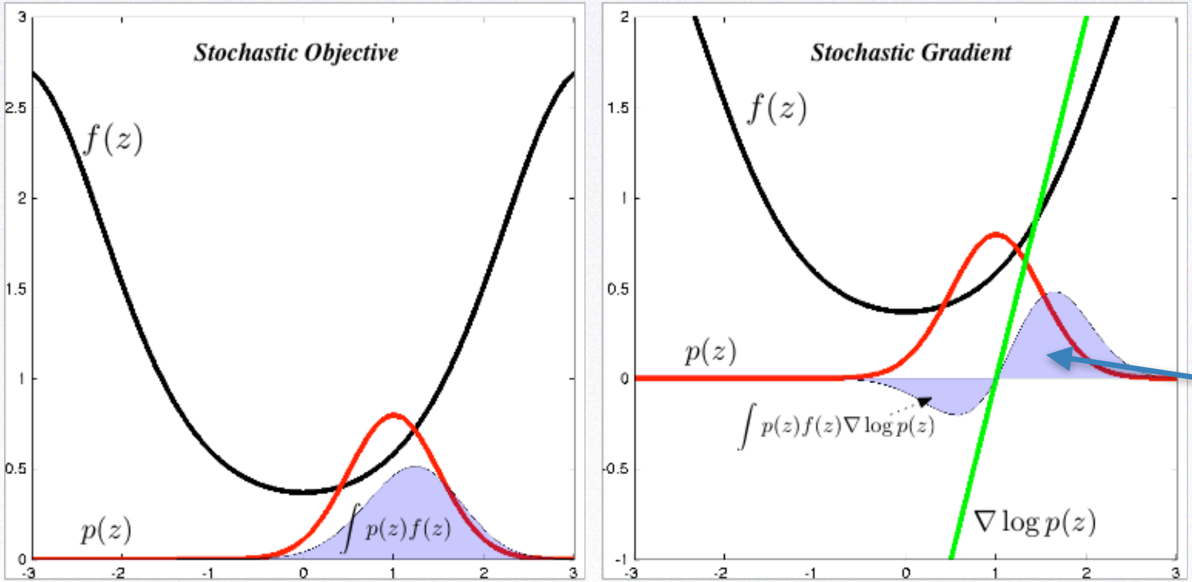
$$\approx \frac{1}{S} \sum_{s=1}^{S} f(z^{(s)}) \nabla_\theta \log p(z^{(s)};\theta) \quad z^{(s)} \sim p(z)$$

← Sample approximation

DeepMind

Score-Function Estimators

a.k.a. the log-derivative trick



This quantity is what we'll approximate with samples

courtesy Shakir M

Generalizing to *Control*

The random variable is now the action *a*

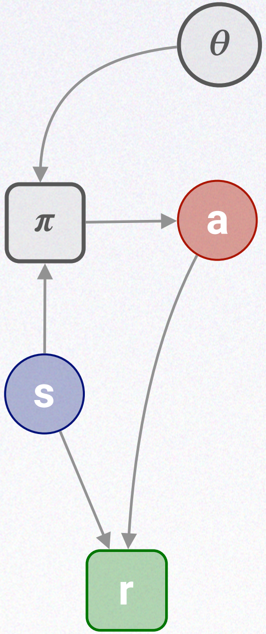
All *a* are conditionally independent given the state *s*, and parameterized by the *policy*

$$p(z;\theta) \rightarrow p(a_t|s_t;\theta) = \pi_\theta(a_t|s_t)$$

The “function” is now the *Return*

$$f(z) \rightarrow \sum_t r(s_t, a_t)$$

Vanilla Policy Gradient — Single time-step “Bandit”



$$\nabla_\theta J(\theta) = \nabla_\theta \underline{\int p(s)} \int \pi_\theta(a|s) r(s,a) \text{ da } \underline{\text{ds}}$$

$$= \int p(s) \int \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) r(s,a) \text{ da ds}$$

$$\approx \frac{1}{S} \sum_{s=1}^{S} \nabla_\theta \log \pi_\theta(a^{(i)}|s^{(i)}) r(s^{(i)}, a^{(i)})$$

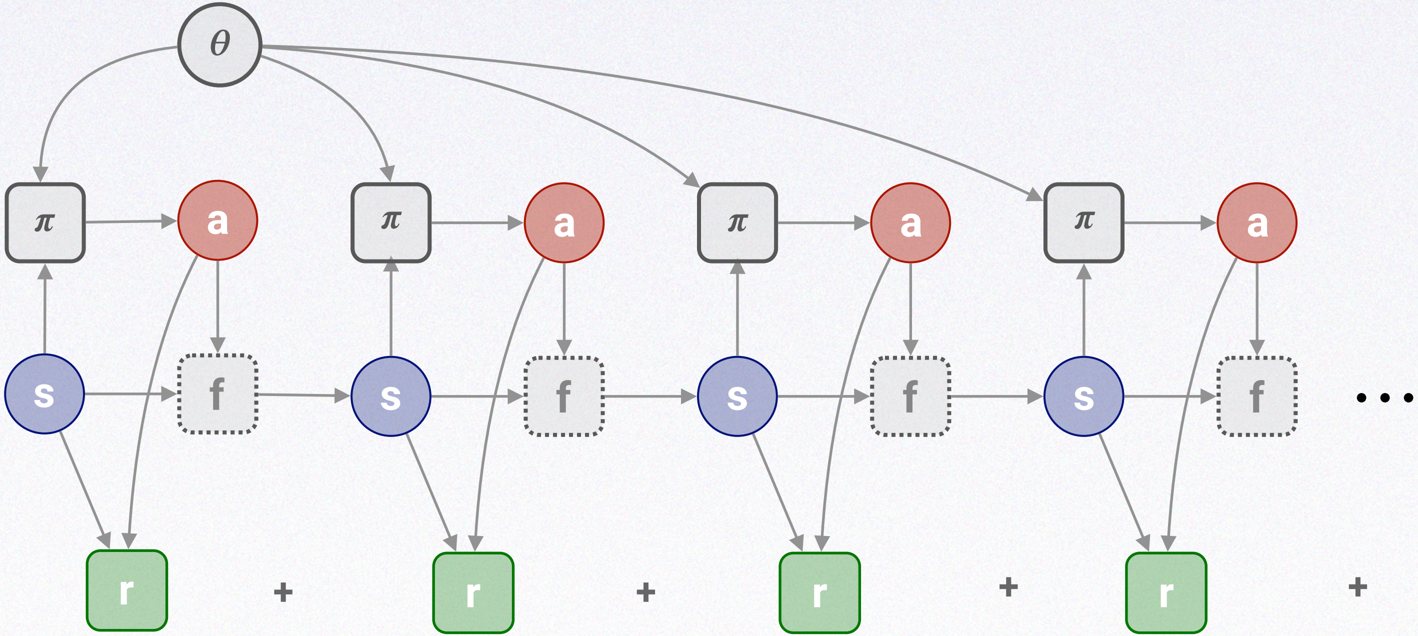
$$\text{where } s^{(i)} \sim p(s), a^{(i)} \sim \pi_\theta(\cdot|s^{(i)})$$

Introduced 1 new quantity: the start-state distribution *p(s)*

Figure credit: N. Heess

DeepMind

Generalizing to *Trajectories*

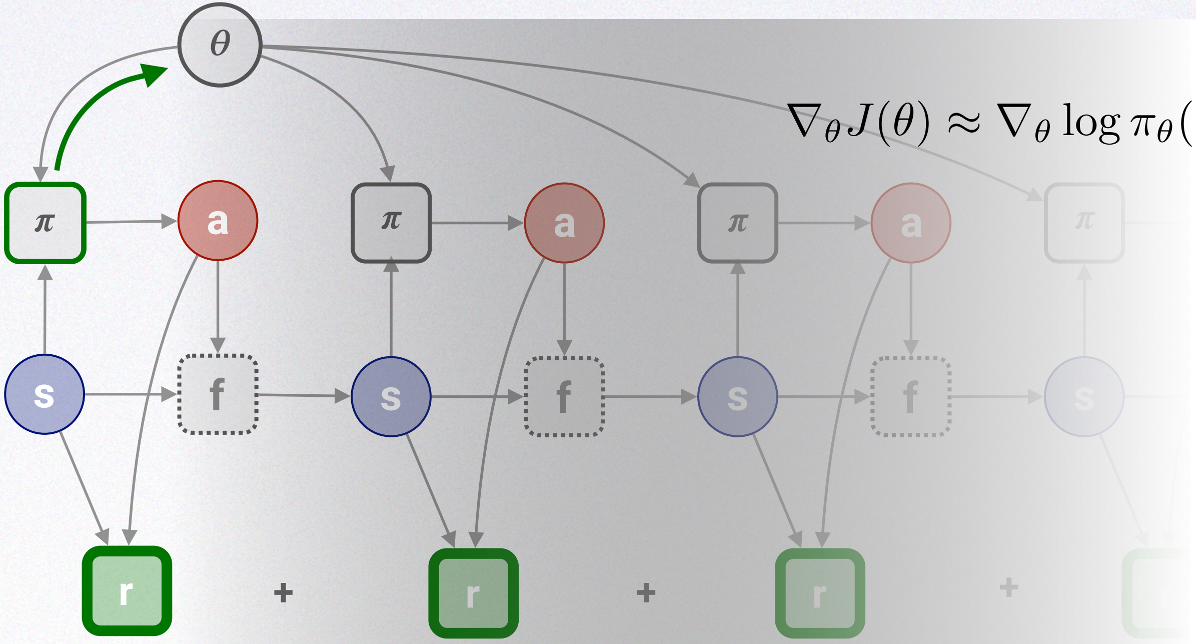


$$p(\tau) = p(s_0)\pi(a_0|s_0)p(s_1|s_0,a_0)\pi(a_1|s_1)p(s_2|s_1,a_1)\dots$$

$$J(\theta) = \mathbb{E}_{p(\tau)}\left[\sum_t r(s_t,a_t)\right]$$

Figure credit: N. Heess

Policy Gradient — Trajectories

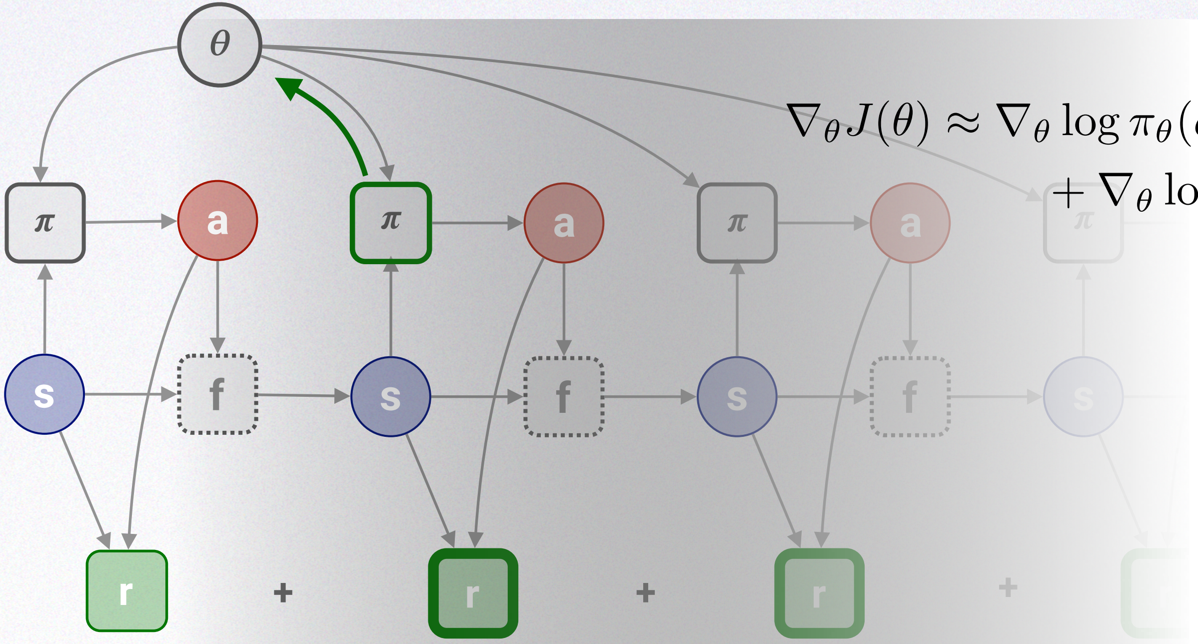


$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_0|s_0) R_0$$

$$R_0 = r_0 + r_1 + r_2 + r_3 + \ \ldots$$

Figure credit: N. Heess

Policy Gradient — Trajectories



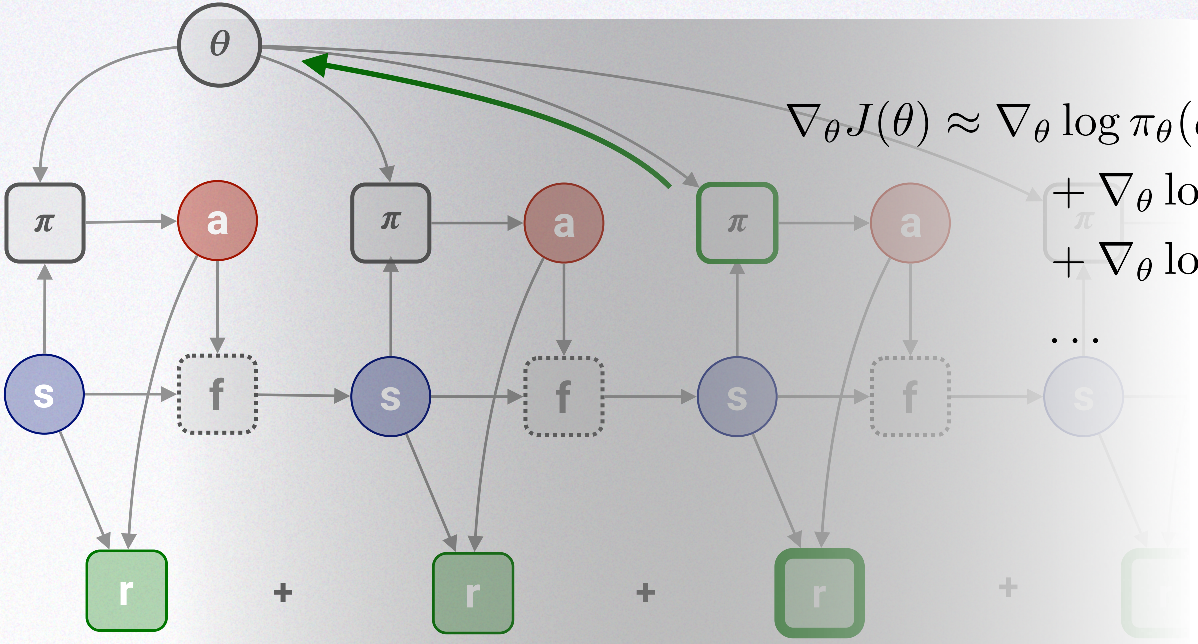
$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_0|s_0) R_0 + \nabla_\theta \log \pi_\theta(a_1|s_1) R_1$$

$$R_1 = r_1 + r_2 + r_3 + \ \ldots$$

Figure credit: N. Heess

DeepMind

Policy Gradient — Trajectories



$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_0|s_0)R_0$$
$$+ \nabla_\theta \log \pi_\theta(a_1|s_1)R_1$$
$$+ \nabla_\theta \log \pi_\theta(a_2|s_2)R_2$$
$$\ldots$$

$$R_2 = r_2 + r_3 + \ \ldots$$

Figure credit: N. Heess

DeepMind

The Policy Gradient Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\underline{Q^\pi(s,a)}\right]$$

The “return” under π. Doesn’t stipulate how this is estimated

The *Reinforce* Algorithm

```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \underline{R_t}$
        end for
    end for
    return $\theta$
end function
```

Problems with Vanilla Policy Gradient?

What if all our rewards are 1000-1001, vs, say 0-1?

All our gradient estimates are 1000x!

Could change our learning rates...

Would still have variance due to states

Detour Cont'd: Adding a Baseline

a.k.a. *control variate*

$$\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)] = \mathbb{E}_{p(z;\theta)}[(f(z) - b)\nabla_\theta \log p(z;\theta)]$$

Can be arbitrary
Won't affect expectation if not function of θ

But, why?

- ➡ To make variance as low possible
- ➡ Natural candidate:

$$\boxed{b = \mathbb{E}_{p(z;\theta)}[f(z)]}$$

Why?

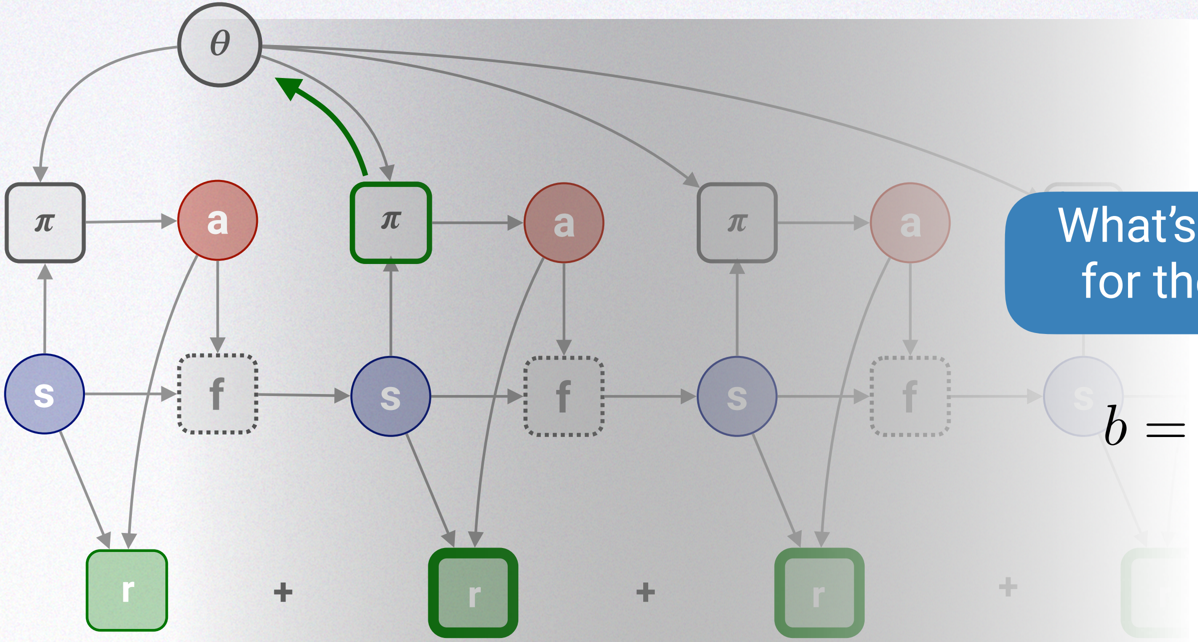
$$= \mathbb{E}_{p(z;\theta)}[f(z)\nabla_\theta \log p(z;\theta)] - b\int p(z;\theta)\nabla_\theta \log p(z;\theta)dz$$

$$= \mathbb{E}_{p(z;\theta)}[f(z)\nabla_\theta \log p(z;\theta)] - b\int \nabla_\theta p(z;\theta)dz$$

$$= \mathbb{E}_{p(z;\theta)}[f(z)\nabla_\theta \log p(z;\theta)] - b\underbrace{\nabla_\theta \int p(z;\theta)dz}_{\nabla_\theta \text{const} = 0}$$

$$= \mathbb{E}_{p(z;\theta)}[f(z)\nabla_\theta \log p(z;\theta)]$$

Policy Gradient — Variance Reduction



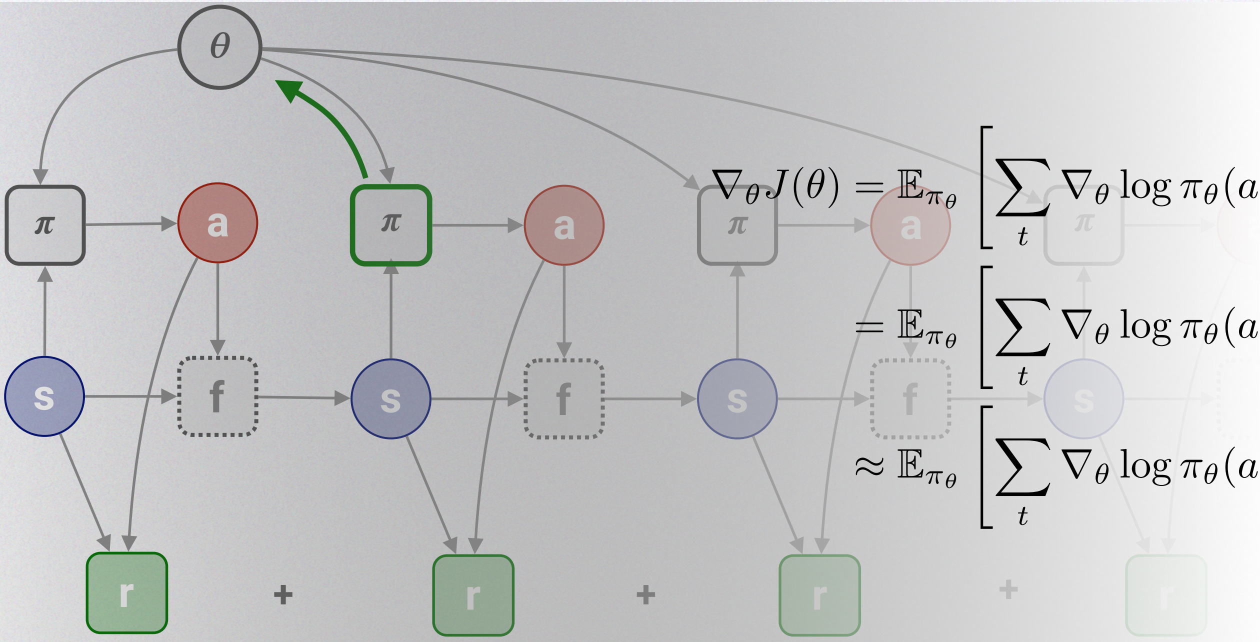
$$R_1 = r_1 + r_2 + r_3 + \ \dots$$

What’s a good candidate for the trajectory case?

$$b = \mathbb{E}_{\pi(a|s;\theta)}[V(s)]$$

Figure credit: N. Heess

Policy Gradient — Variance Reduction



$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)[R_t - V(s_t)]\right]\\ &= \mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)[Q^\pi(s_t,a_t) - V^\pi(s_t)]\right]\\ &\approx \mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)[\hat{Q}(s_t,a_t) - \hat{V}(s_t)]\right]\end{aligned}$$

Figure credit: N. Heess

Return Surrogates

$$\mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)[\hat{Q}(s_t,a_t) - \hat{V}(s_t)]\right]$$

- Value-baseline removes variance in policy gradient across states, by “absorbing” stochasticity in the dynamics (and policy) into a separate expectation
- But what if the reward itself is stochastic?
    - We have an estimator for exactly this statistic: Q!
- The PG theorem actually gives a sound basis for using $\hat{Q}$ instead of the empirical return
- Subject to some technical conditions on compatibility between the policy and critic, but we usually don’t worry about this in DL setting.

Policy Gradient — Menu of Algorithms

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\right]$$

Various estimators for Q^π

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) R\right]$$

unbiased, high var.

REINFORCE

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \hat{Q}(s,a)\right]$$

biased, low var.

Q Actor-Critic

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) (Q^\pi(s,a) - \hat{V}(s))\right]$$

Advantage Actor-Critic

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) (r_t + \ldots + \gamma^k r_{t+k} + \gamma^{k+1} \hat{V}(s_{t+k}) - V(s_t))\right]$$

K-step Truncated Advantage

Policy-Gradient Recap

Intuition: a Monte-Carlo estimator that uses samples of the total return as weights to “reinforce” good action gradients

- The likelihood-ratio trick unpacks to $\nabla_\theta \log p(z;\theta) = \dfrac{\nabla_\theta p(\mathbf{z};\theta)}{p(\mathbf{z};\theta)}$
- Has an intuitive interpretation:
  - Scales gradient inversely proportional to the action probability, **to compensate for the policy’s preference for this action**

Q: What would happen if we simply scaled by $\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)\right]$ instead?

(Forget our derivation for a moment)

A: Would have stronger gradients for actions we tried a lot
➡ Would reinforce arbitrary initialization!

Policy Gradient — Take aways

- Foundational of most modern RL algorithms
- Pros:
    - ➡ Minimal assumptions: only (log) policy has to be differentiable; the rest is samples
    - ➡ Supports both discrete and continuous states and actions
    - ➡ Well studied, many tricks to reduce variance, e.g. value-functions
- Cons:
    - ➡ Still not very efficient, e.g. for robotics
    - ➡ Only defined for on-policy case; each data-point used *once*
    - ➡ Sensitive to hyper parameters

Outline

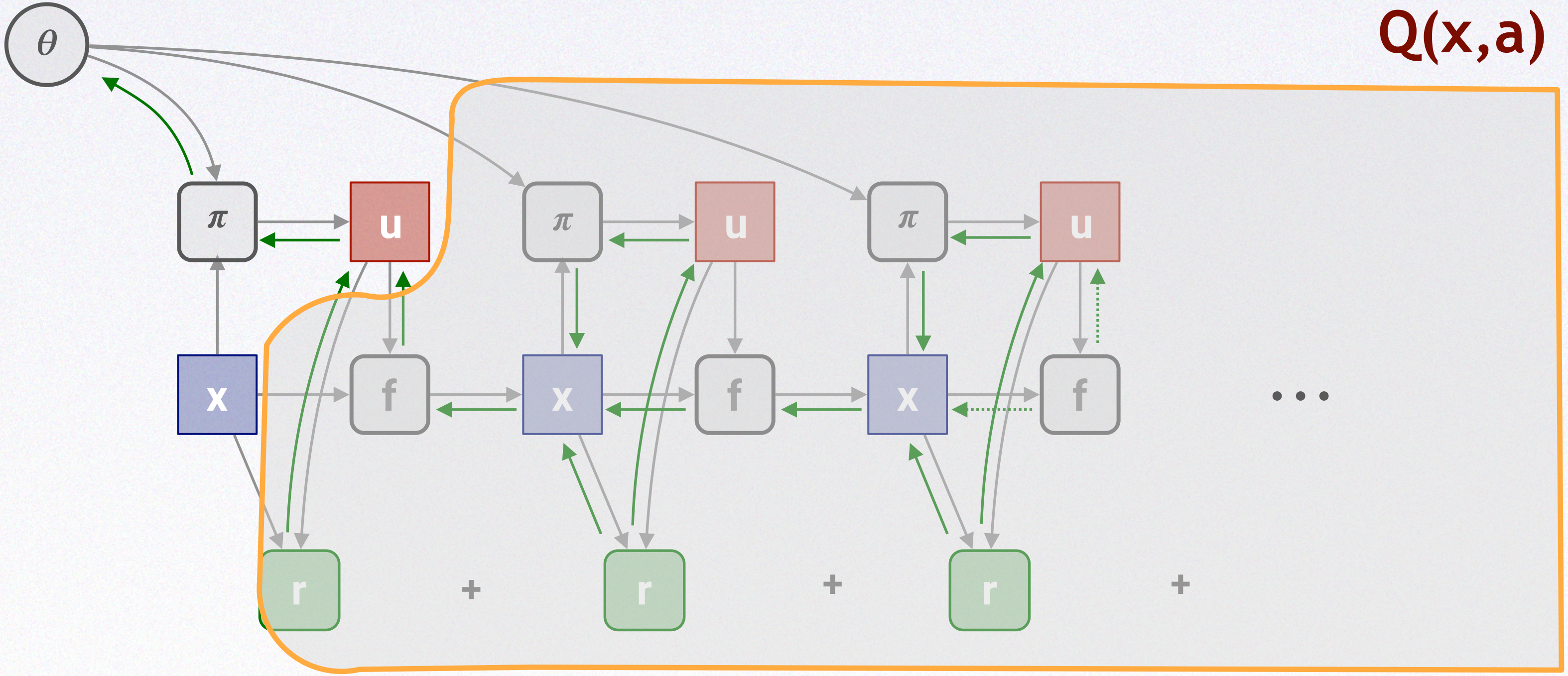
- Motivational videos
- Part 1: Q-Learning Walkthrough
- Part 2: Policy-Gradient Survey
    - Vanilla Policy-Gradient Methods
    - Value-Gradient Methods
- Open Challenges

Value Gradients — Intuition

- Alternative way to get a policy gradient that **directly asks the critic for the ascent direction** in action-space, rather than monte-carlo estimating by sampling it
- Has some trade-offs vs. Vanilla PG, but on net is more applicable to robotics*

*Opinion of the author :)

Q: The Truncated Trajectory Gradient

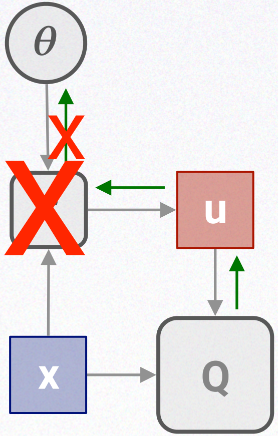


Gradients provide a lot of information, especially in high-dimensional spaces!
Can we exploit gradients more directly for policy search?

Slide credit: N. Heess

Handling Stochasticity

- How to back-propogate through a stochastic policy (or critic, or model)? (Can’t back-propogate through an RNG)



Detour: Pathwise Derivative Estimators

a.k.a. the reparameterization trick

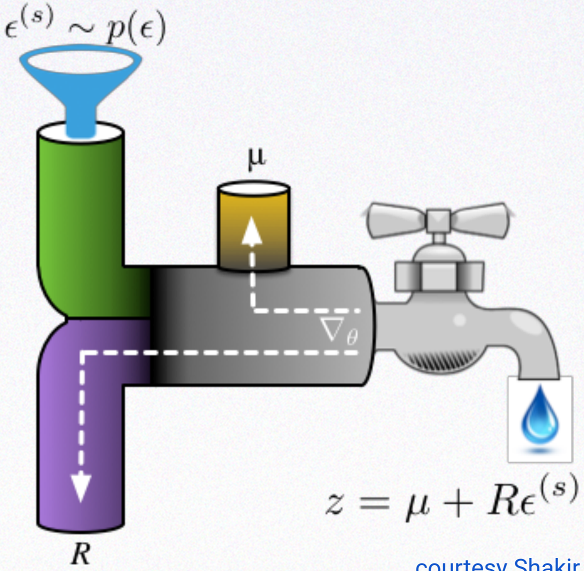
Key idea: replace a random variable with a *deterministic* transformation of a simpler random variable

Gaussian Example

$$N(\mu, RR^T) = \mu + R\epsilon, \quad \epsilon \sim N(0,1)$$

Implies legal change of variables

$$z \sim p(z;\theta) = g(\theta, \epsilon), \epsilon \sim N(0,1)$$



courtesy Shakir M

Possible with many common probability distributions

Detour: Pathwise Derivative Estimators

a.k.a. the reparameterization trick

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)] &= \nabla_\theta \int p(z;\theta) f(z) dz \\
&= \nabla_\theta \int p(\epsilon) f(g(\theta,\epsilon)) d\epsilon && \longleftarrow \text{Change of variables} \\
&= \nabla_\theta \mathbb{E}_{p(\epsilon)}[f(g(\theta,\epsilon))] \\
&= \mathbb{E}_{p(\epsilon)}[\nabla_\theta f(g(\theta,\epsilon))] && \longleftarrow \text{Push gradient through expectation (unrelated to } \varepsilon) \\
&= \mathbb{E}_{p(\epsilon)}[\nabla_z f(g(\theta,\epsilon)) \nabla_\theta g(\theta,\epsilon)] && \longleftarrow \text{Chain rule!}
\end{aligned}
$$

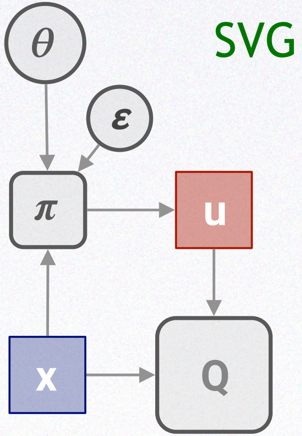
Stochastic Value Gradients (SVG)

Recall Vanilla Policy-Gradient

$$\mathbb{E}_{s\sim\rho^{\pi_\theta},a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\right]$$

Stochastic Value-Gradient

$$\mathbb{E}_{s\sim\rho^{\pi_\theta},p(\epsilon)}\left[\nabla_\theta \pi_\theta(s,\epsilon)\nabla_a Q(s,a)|_{a=\pi_\theta(s,\epsilon)}\right]$$



- ➡ Compared to VPG, replaces expectation over actions w/ expectation over noise source
- ➡ Derivative of all model components now inside the expectation

Figure credit: N. Heess

DeepMind

Deterministic Policy Gradient (DPG)

Recall Vanilla Policy-Gradient

$$\mathbb{E}_{s\sim\rho^{\pi_\theta},a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\right]$$

Deterministic Policy-Gradient

$$\mathbb{E}_{s\sim\rho^{\pi_\theta}}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q(s,a)|_{a=\pi_\theta(s)}\right]$$

➡ Limiting case of SVG as noise -> 0

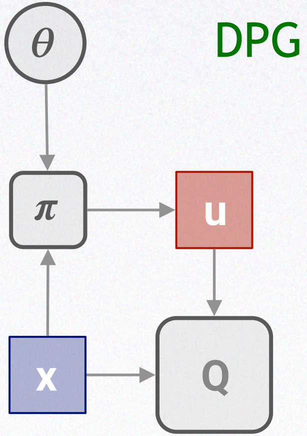


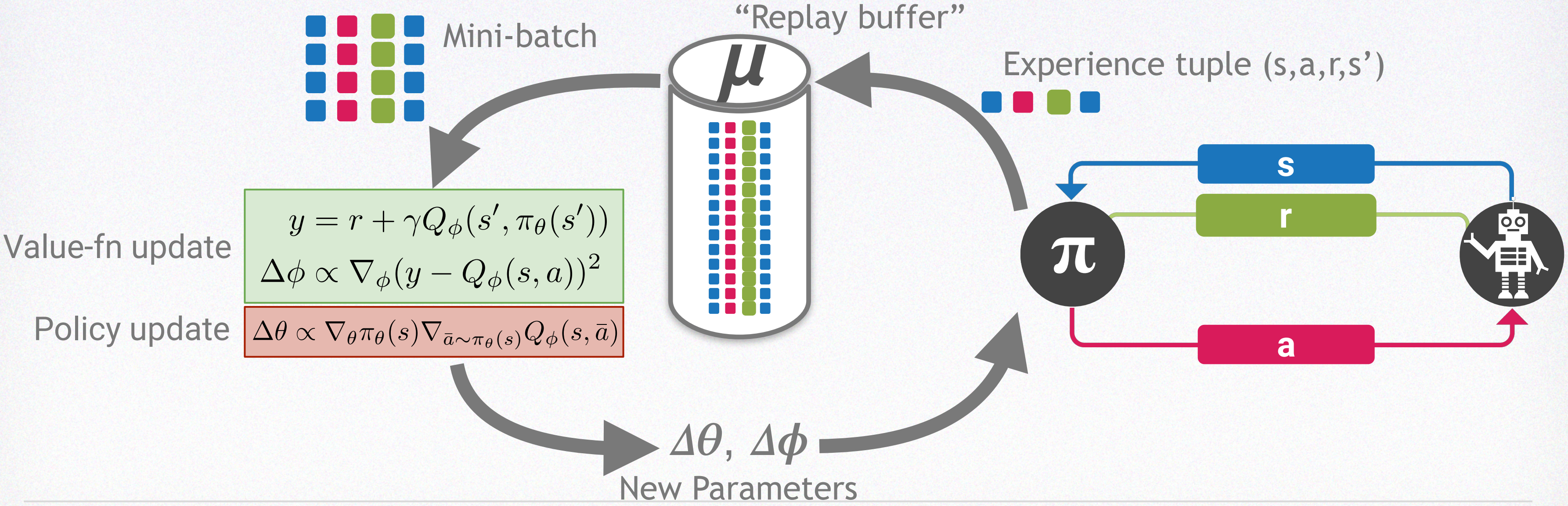
Figure credit: N. Heess

DeepMind

Off-policy learning & experience replay

Key idea: train policy π using data from a different behavior policy μ (e.g. π_{old}, human, …)

Experience replay: a database of experience tuples / trajectories



$$y = r + \gamma Q_\phi(s', \pi_\theta(s'))$$

$$\Delta\phi \propto \nabla_\phi (y - Q_\phi(s,a))^2$$

$$\Delta\theta \propto \nabla_\theta \pi_\theta(s) \nabla_{\bar{a}\sim\pi_\theta(s)} Q_\phi(s, \bar{a})$$

Slide credit: N. Heess

Value gradients in practice: SVG & DPG

Master algorithm:

```
initialize $\pi, \pi^{target}, Q, Q^{target}$
for i=1 ... n do
    Collect data with behavior policy $\pi^b$
    Add trajectory data to replay $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
    Sample minibatch $\mathcal{B}$ of samples $s_t, a_t, r_t, s_{t+1}[\ldots]$
    Compute $Q$ update using $\mathcal{B}$, $\pi^{target}$, and $Q^{target}$
    Compute $\pi$ update using $\mathcal{B}$ and $Q$
    if $\mathrm{mod}(i, M) = 0$ then
        $\pi \leftarrow \pi^{target}$
        $Q \leftarrow Q^{target}$
    end if
end for
```

Key ingredients:

1. Arbitrary behavior policy
2. Off-policy learning of Q^{π}
3. Off-policy updates of π
4. Experience replay
5. Target networks for stability
    - i.e. an old version of our network parameters that we update periodically

Slide credit: N. Heess

Value gradients in practice: SVG & DPG

Master algorithm:

```
initialize $\pi, \pi^{target}, Q, Q^{target}$
for i=1 ... n do
    Collect data with behavior policy $\pi^b$
    Add trajectory data to replay $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
    Sample minibatch $\mathcal{B}$ of samples $s_t, a_t, r_t, s_{t+1}[\ldots]$
    Compute $Q$ update using $\mathcal{B}$, $\pi^{target}$, and $Q^{target}$
    Compute $\pi$ update using $\mathcal{B}$ and $Q$
    if $\mathrm{mod}(i, M) = 0$ then
        $\pi \leftarrow \pi^{target}$
        $Q \leftarrow Q^{target}$
    end if
end for
```

Can act with arbitrary policy to collect data. E.g. for DPG

$$\pi^b(s) = \pi(s) + \epsilon$$

where $\epsilon \sim N(0, \sigma^2)$

DeepMind

Slide credit: N. Heess

Value gradients in practice: SVG & DPG

Master algorithm:

```
initialize $\pi, \pi^{target}, Q, Q^{target}$
for i=1 ... n do
    Collect data with behavior policy $\pi^b$
    Add trajectory data to replay $s_0, a_0, r_0, s_1, a_1, r_1, \dots$
    Sample minibatch $\mathcal{B}$ of samples $s_t, a_t, r_t, s_{t+1}[\dots]$
    Compute $Q$ update using $\mathcal{B}$, $\pi^{target}$, and $Q^{target}$
    Compute $\pi$ update using $\mathcal{B}$ and $Q$
    if mod$(i, M) = 0$ then
        $\pi \leftarrow \pi^{target}$
        $Q \leftarrow Q^{target}$
    end if
end for
```

Off policy Q-learning

Core insight:

$$Q^{\pi} = r(s,a) + \mathbb{E}\left[V^{\pi}(s'|s,a)\right]$$

is true for *any* tuple *(s, a, r, s')*!

Update for Q:

$$y = r(s,a) + Q^{\text{target}}(s, \pi^{\text{target}}(s))$$

$$\Delta\phi \propto \nabla_{\phi}(y - Q_{\phi}(s,a))^2$$

DeepMind

Slide credit: N. Heess

Value gradients in practice: SVG & DPG

Master algorithm:

```
initialize $\pi, \pi^{target}, Q, Q^{target}$
for i=1 . . . n do
    Collect data with behavior policy $\pi^b$
    Add trajectory data to replay $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
    Sample minibatch $\mathcal{B}$ of samples $s_t, a_t, r_t, s_{t+1}[\ldots]$
    Compute $Q$ update using $\mathcal{B}$, $\pi^{target}$, and $Q^{target}$
    Compute $\pi$ update using $\mathcal{B}$ and $Q$
    if $\mathrm{mod}(i, M) = 0$ then
        $\pi \leftarrow \pi^{target}$
        $Q \leftarrow Q^{target}$
    end if
end for
```

Policy update

DPG:

$$\Delta\theta \propto \nabla_\theta \pi_\theta(s) \nabla_{\bar{a} \sim \pi_\theta(s)} Q_\phi(s, \bar{a})$$

SVG:

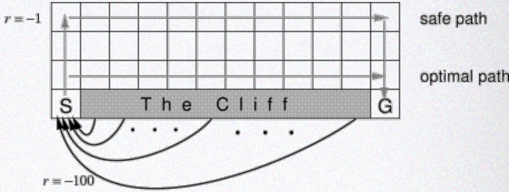
$$\Delta\theta \propto \mathbb{E}_{p(\epsilon)} \left[\nabla_\theta \pi_\theta(s, \epsilon) \nabla_{\bar{a} \sim \pi_\theta(s, \epsilon)} Q_\phi(s, \bar{a}) \right]$$

Slide credit: N. Heess

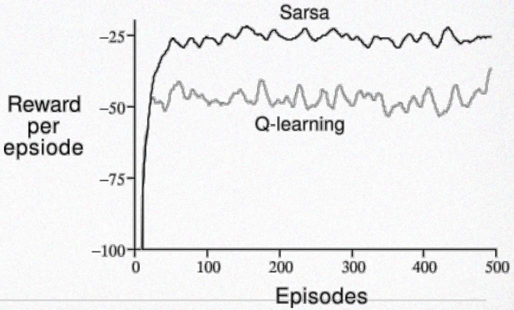
Off-Policy Methods — Textbook Version

$$Q(s,a) \leftarrow Q(\underset{\uparrow}{s},\underset{\uparrow}{a}) + \alpha\left(\underset{\uparrow}{R(s,a)} + \gamma Q(\underset{\uparrow}{s'},\underset{\uparrow}{a'}) - Q(s,a)\right)$$

state **a**ction + **r**eward + (next) **s**tate (next) **a**ction = **SARSA**



- Otherwise same as Q-learning, but “on-policy”
    - Use *a’* instead of $\max_a$ when computing target
- Less greedy, so addresses problem of locally high-reward/risk states (e.g. cliff task)
- Otherwise, Q-learning and SARSA both looking at essentially the same data… right?

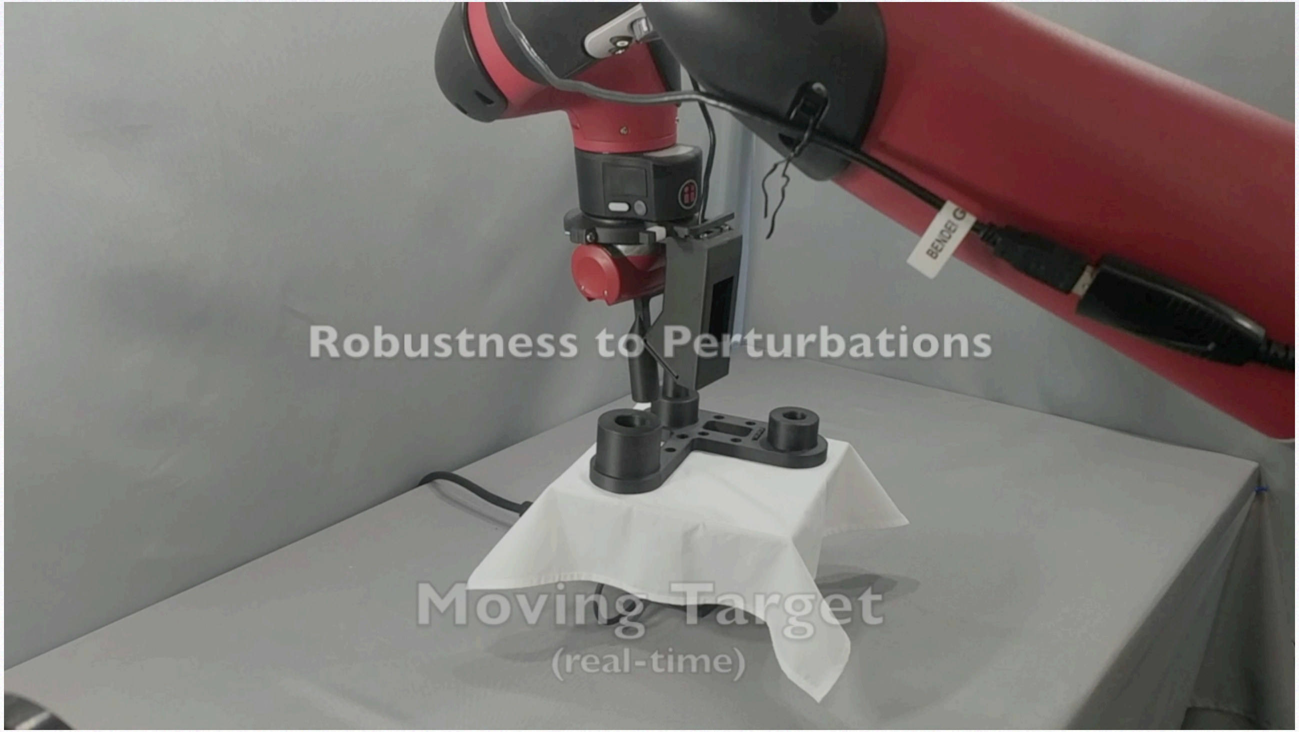


Off-Policy Methods — Closer Look

- Distinction is **fundamental**
  - Some algorithms, *e.g.* VPG, only make sense on-policy
- Distinction is **practical**
  - Many algorithms, *e.g.* IMPALA are slightly off-policy due to delays
  - Off-policy data can come from *any* policy, e.g. people

Off-Policy RL Success Story

Imitation + RL — DPG from Demonstrations (Vecerik 2018)



video source

DeepMind

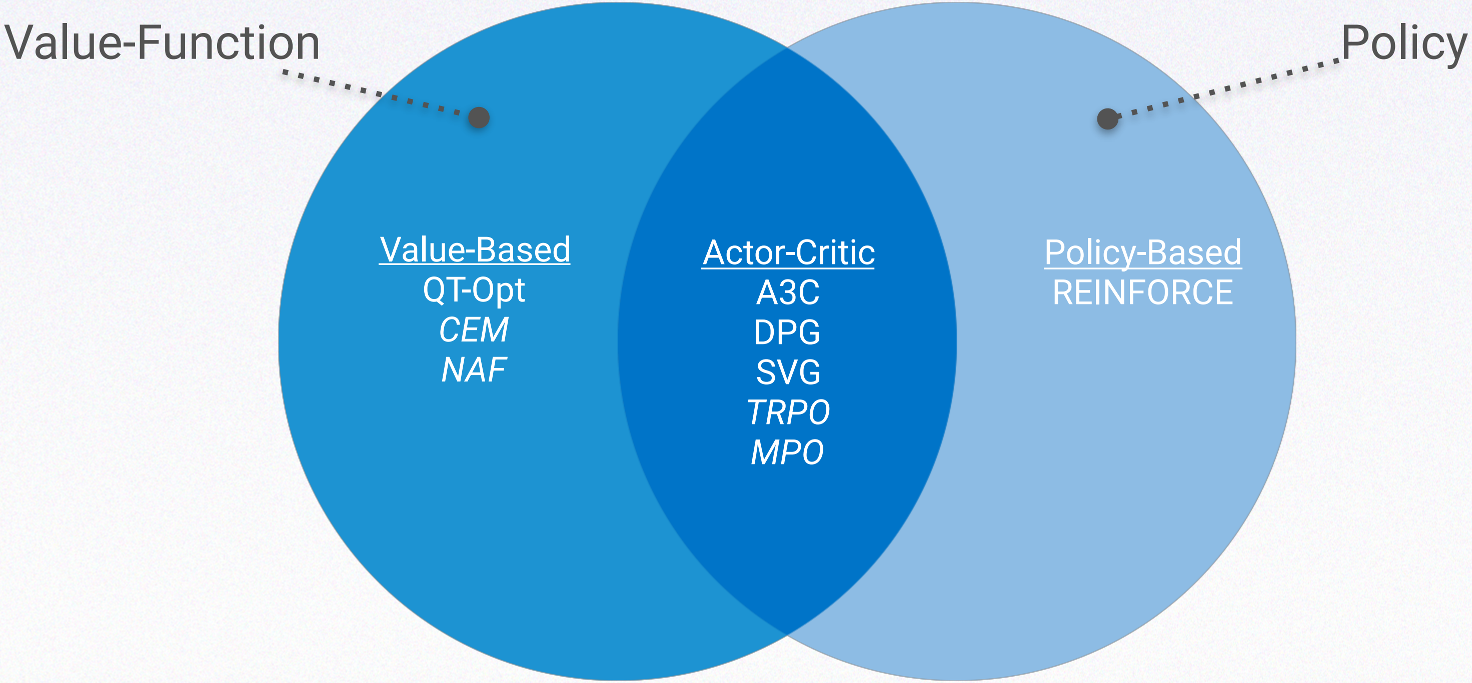
A few other tricks to get this working

- Add both successful and unsuccessful expert demonstrations to seed replay memory
- Auxiliary loss for classifying demonstrator actions
- Need to learn a reward function from pixels
- Need a safety - compliance module
- Tuning all hyper-parameters
- Distributional Q-function

Value Gradient — Take aways

- Policy Gradients purely by back-propagation
- Pros:
  - ➡ Same general setting as VPG
  - ➡ Can be trained on-policy or off-policy
  - ➡ Can use stochastic or deterministic policies
  - ➡ More efficient; lets you re-use data
- Cons:
  - ➡ Generally less stable than VPG methods

Value-Based and Policy-Based Methods



Value-Function

Policy

Value-Based
QT-Opt
CEM
NAF

Actor-Critic
A3C
DPG
SVG
TRPO
MPO

Policy-Based
REINFORCE

Off-Policy and On-Policy Methods

Estimates value
of greedy policy
given data from
other policies

Off-Policy
DPG
SVG
QT-Opt
MPO
*IMPALA**

Estimates value
of current policy

On-Policy
REINFORCE
A3C
TRPO

Policy Gradient — Summary

3 ways to compute policy gradients

Finite-Difference

- Use if **policy and critic are non-differentiable**
- **Most expensive** - requires expectation for each partial derivative, scales linearly with # policy params

Vanila PG

- Use if your policy is differentiable but critic / return are not
- Much **cheaper and lower variance than FD**; pulls gradient computation inside expectation analytically

Value-Gradients

- Use if your **policy and value function are both differentiable**
- **Lowest variance**; expectation only over states and possibly a noise-generator
- Caveat: Value networks not trained to have good gradients, so can see unstable “delusion” behavior.

Outline

- Motivational videos
- Part 1: Q-Learning Walkthrough
- Part 2: Policy-Gradient Survey
  - Vanilla Policy-Gradient Methods
  - Value-Gradient Methods
- Open Challenges

Open Challenges

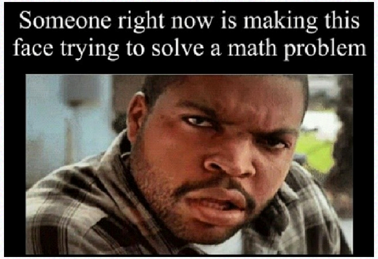
Hyperparam Sensitivity

Sample Efficiency

Off-policy Learning

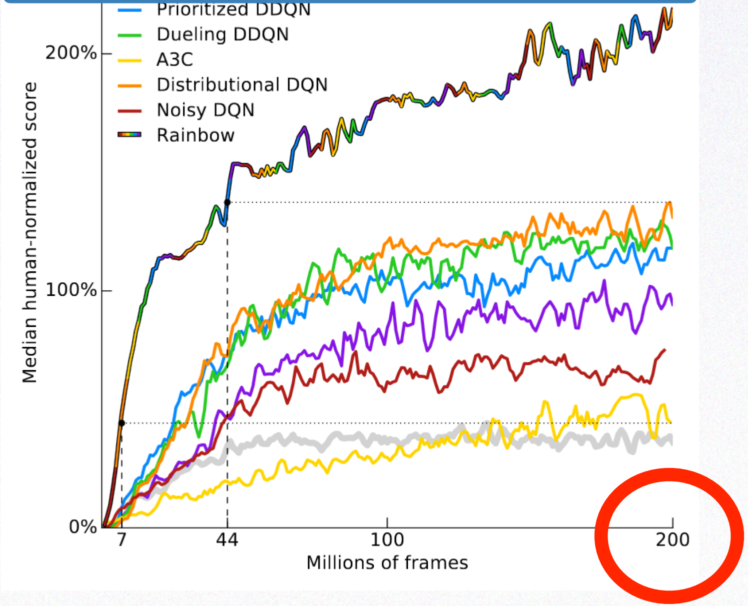
Imitation Learning

Model-based RL

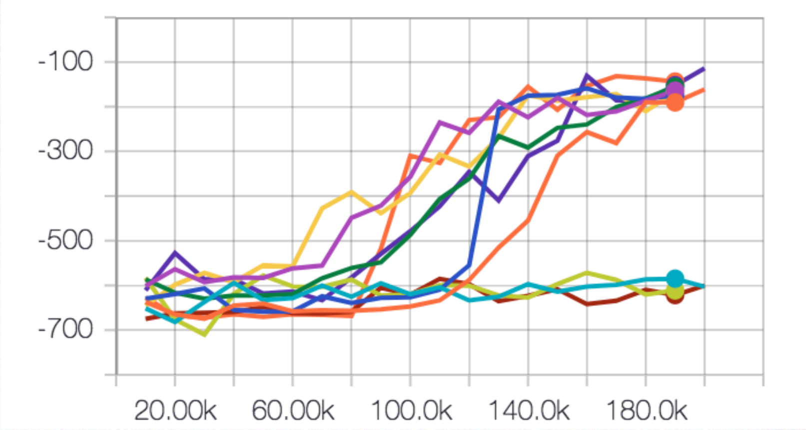


The “Ugly” — Learning Curves

What we say our learning curves look like



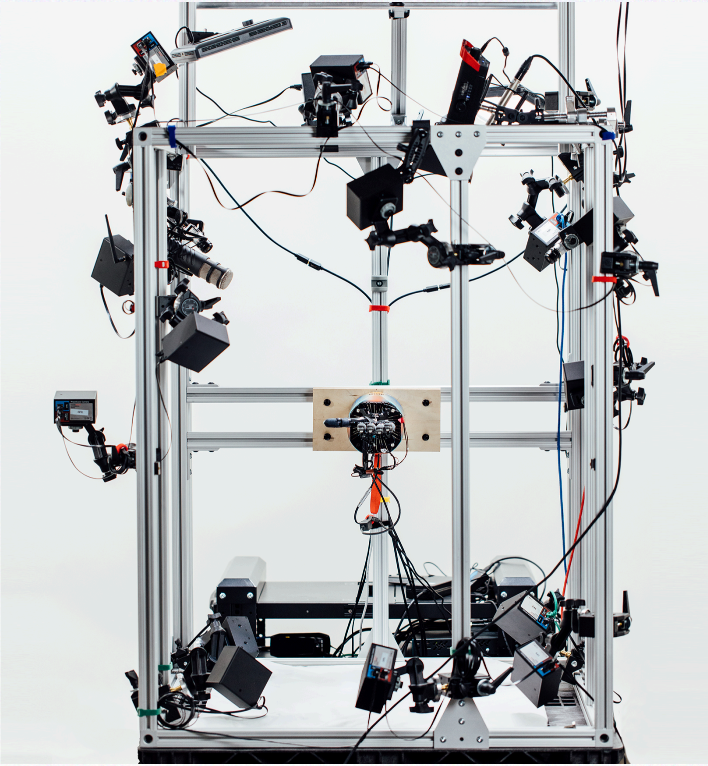
What they actually look like



- High Variance
- Failures due to random seed
- Imagine what actual hyper-params do

The “Ugly” — Instrumentation for Real Experiments

Can train “end-to-end” … if you provide object features as observations or rewards



Where Doesn’t RL Work Yet?

In short.. everywhere

➡ Combinatorially complex tasks, e.g. assembly
  - Motion planning still dominates, even in sim

➡ Long-horizon tasks that can’t be simulated well

➡ Anywhere data is expensive (wouldn’t use DPG to learn Atlas backflip)

➡ Anything on industrial arms (too stiff to explore safely)

Conclusion

If you’re looking for an RL algorithm to apply for your tasks I’d suggest:

- ➡ A2C, if you’re in sim and want to implement it yourself
- ➡ TRPO if you’re in sim and just want to get going with RL without fiddling with hypers
- ➡ DPG/SVG (or RS0) if you’re on a real robot and willing to put time in to tune. Warning: still a bit of an art requiring both DL and RL intuition
- ➡ MPO, if you want to be up with the recent trends

Important Topics Not Covered

- Asynchronous Methods (A3C, IMPALA, QT-Opt, …)
- Trust-Region Methods (TRPO, PPO, Natural Gradient, …)
- Model-Based Methods (iLQG, MPC, SVG(k), GPS, …)
- EM-based Methods (PoWER, REPS, **MPO**, …)
- Off-Policy Corrections (ACER, Retrace, V-trace [in IMPALA], …)
- Trajectory-Based Representations (DMPs, Splines, …)

THANK YOU