

DeepMind

Introduction to Reinforcement Learning

Feryal Behbahani & Matt Hoffman

Special thanks to:

Bobak Shahriari & Nando de Freitas

MLSS 2020



Motivation

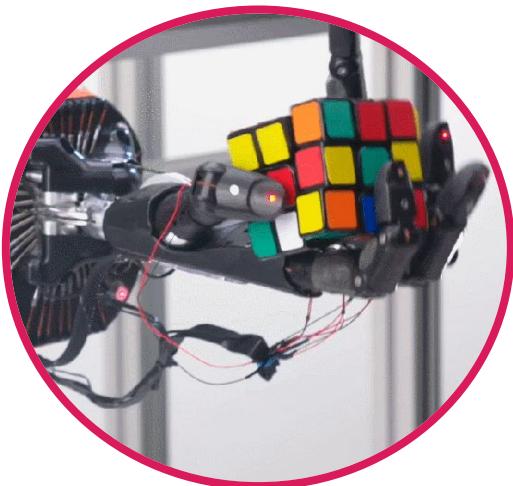
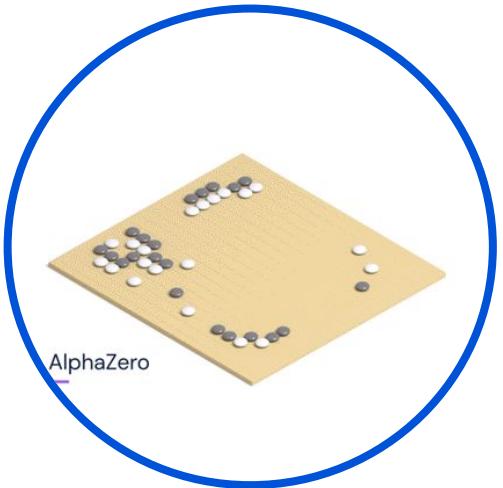
- How do we learn to solve problems?
 - Trial and error
- Reinforcement learning (RL) is a general framework to express how this process is performed.
 - Introduced to explain how animals learn, but has had big impact on neuroscience too



CRAIG SWANSON © WWW.PERSPICUITY.COM



Successes of RL in machine learning



Overview

Fundamental Concepts

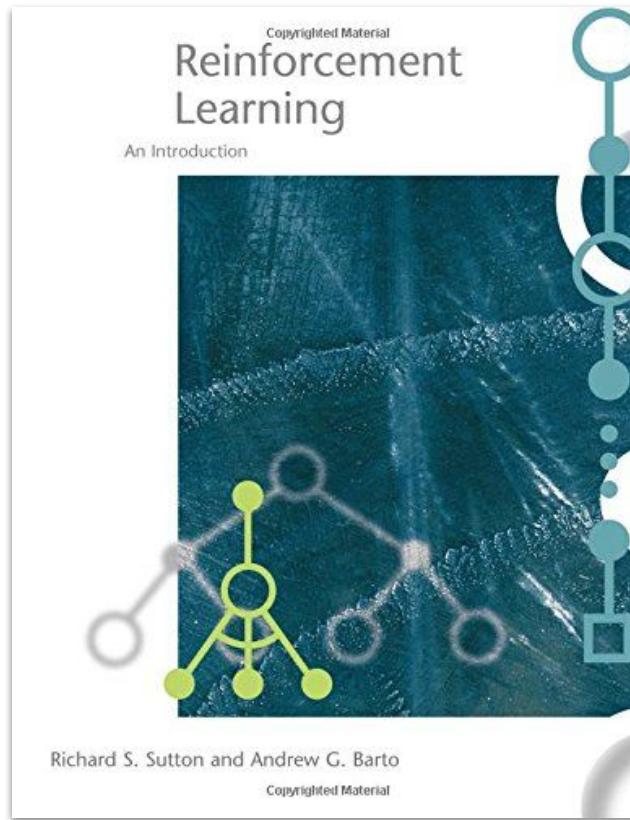
Value-based methods

Policy-based methods

Challenges & opportunities



Reinforcement Learning: An Introduction



Overview

Fundamental Concepts

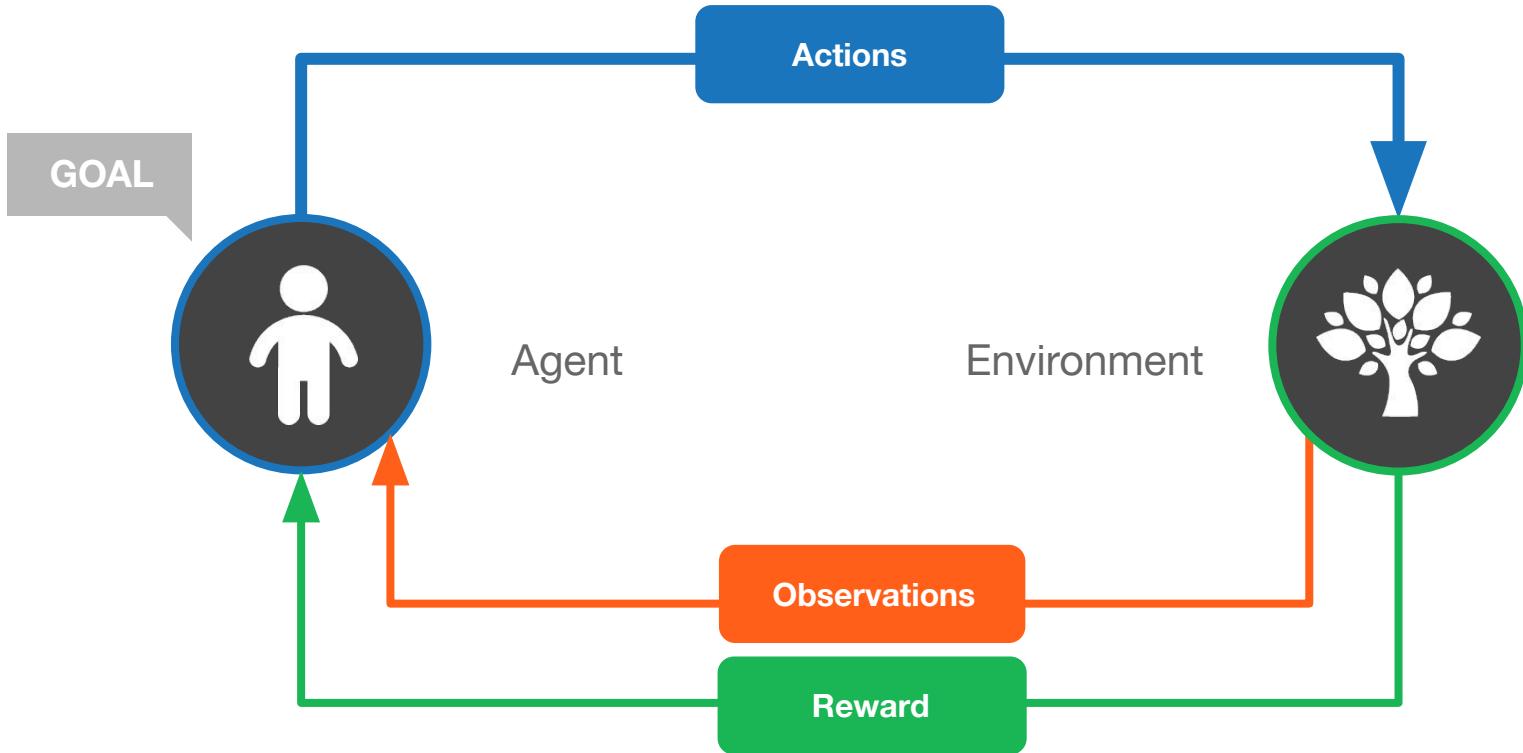
Value-based methods

Policy-based methods

Challenges & opportunities



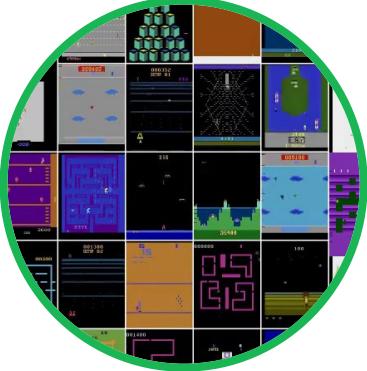
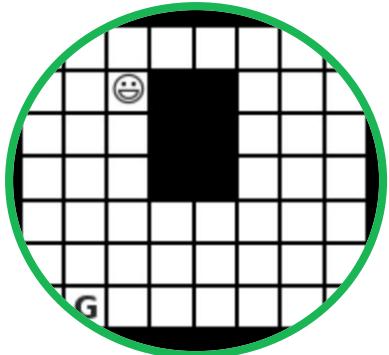
What is reinforcement learning?



Learning to achieve goals via trial and error by interacting with an environment



What is an environment?



What is an environment?

Environment receives **actions** and in turn provides **observations** and **rewards** to the agent.

It can be stochastic, complex and not fully observable to the agent.



Learning to achieve **goals** via **trial and error** by interacting with an **environment**

What is an agent?

GOAL



Agent learns a policy mapping
observations to **actions**.

The agent's **goal** is to maximize its
cumulative **reward**.

Learning to achieve **goals** via **trial and error** by interacting with an **environment**



Where do rewards come from?

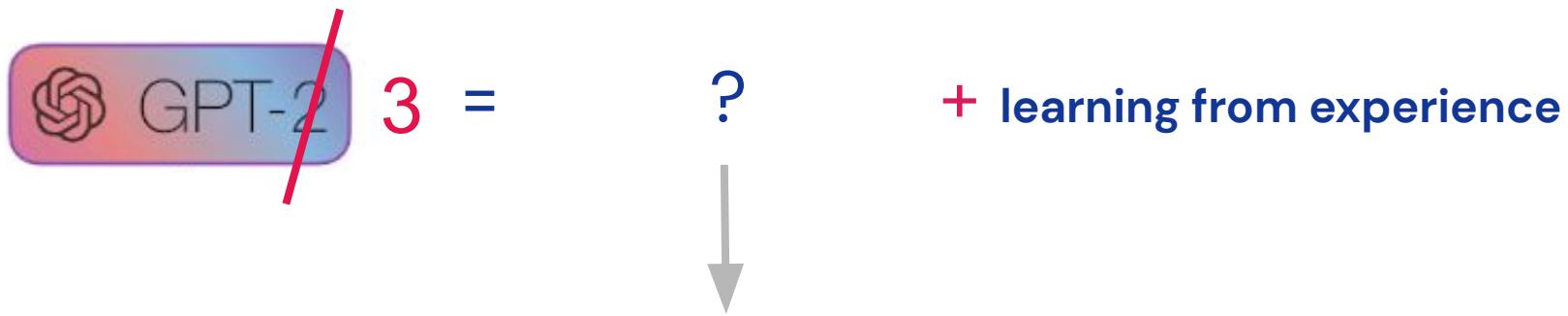
Reward is typically a scalar feedback signal.

- **Extrinsic**
 - Environmental
 - Typically hard-coded by the environment designer
- **Intrinsic**
 - Curiosity/Surprise/Novelty
 - Learning progress
 - Empowerment
 - Explanation
 - Compression



Why is RL hard?

RL = exploration to gather experience + learning from experience



In supervised learning, data is often gathered and filtered by humans, which alleviates the **exploration problem**.

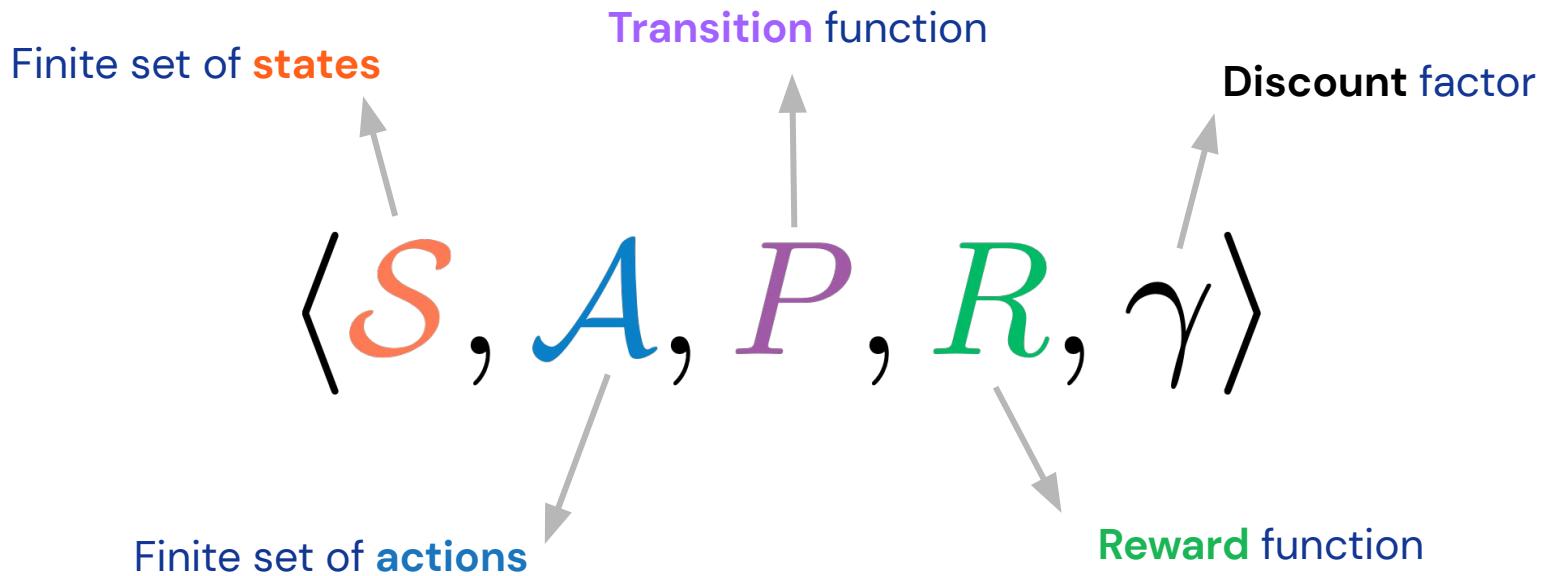


Why is RL hard?

- Sequential decision making
- Exploration might not be trivial
- The environment is often unknown
- Rewards are unknown, delayed and possibly sparse
- Non-stationarity
- High-dimensional continuous states & actions



Markov Decision Process (MDP)



We often solve the **RL problem** by modelling it as an **MDP**.



Markov Decision Process (MDP)

$$\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$$

Markov property: the next state depends only on the previous state, and not all of the states that came before it.

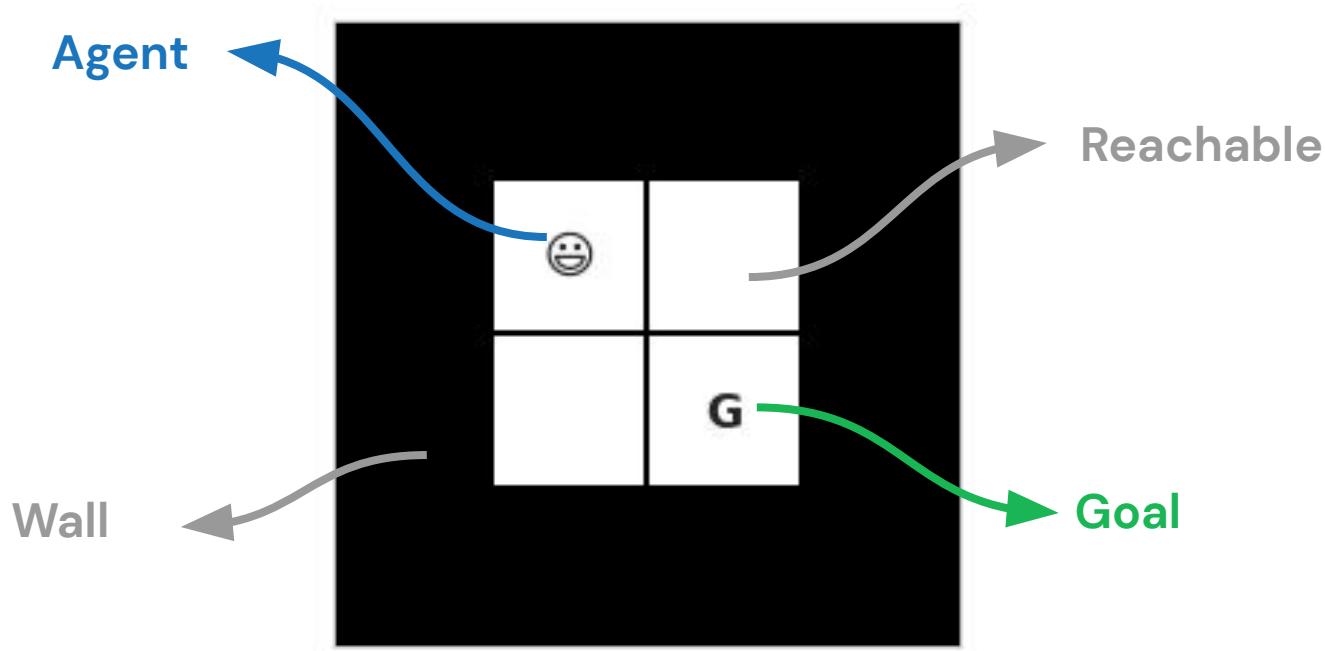
$$P(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0) = P(s_{t+1} | s_t, a_t)$$

Future Present Past Future Present



Grid World Environment

Size of the world: [2 X 2]



State space \mathcal{S}

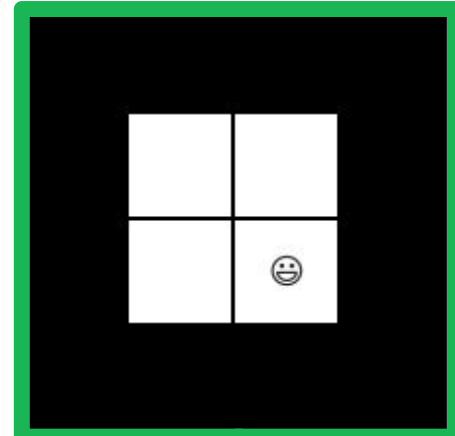
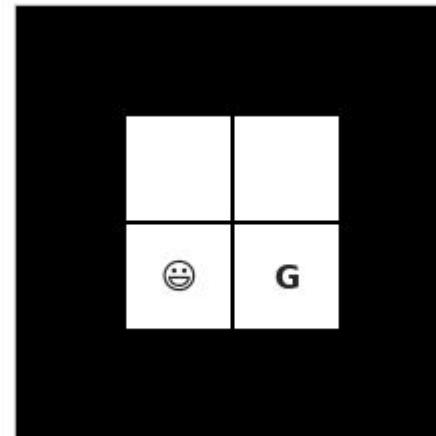
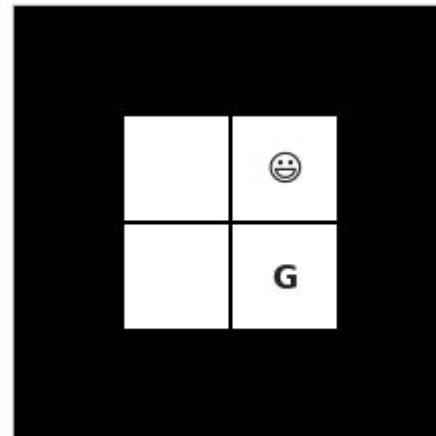
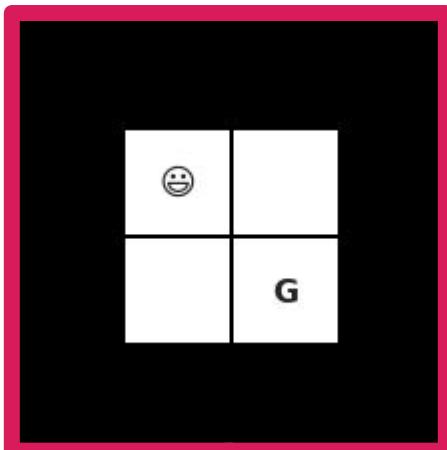
There are 4 unique states in this environment.

s_0

s_1

s_2

s_3

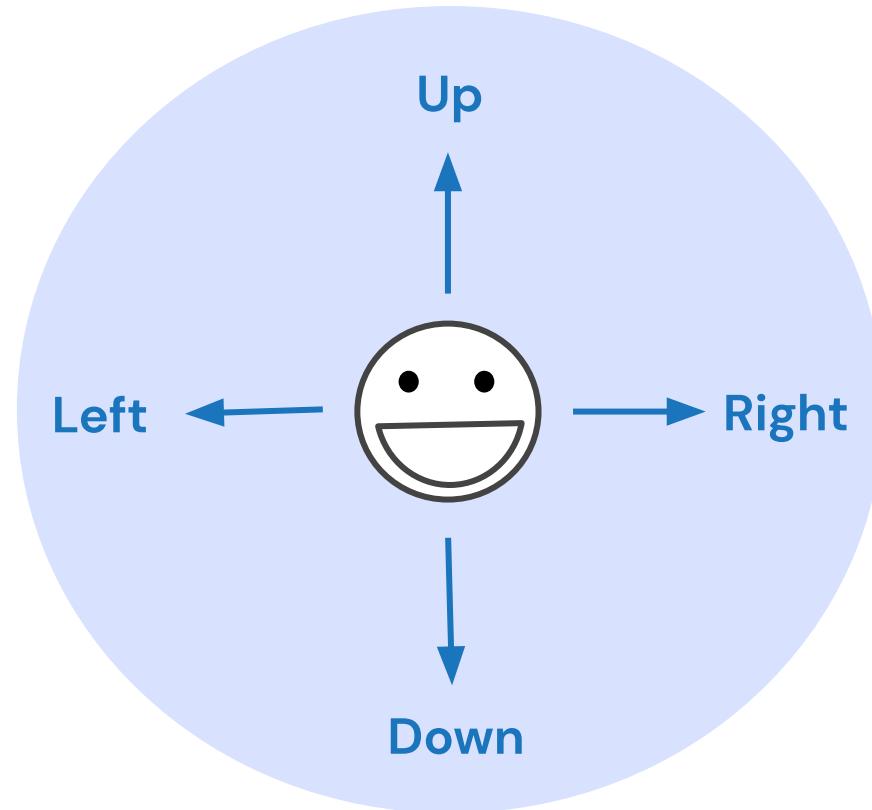
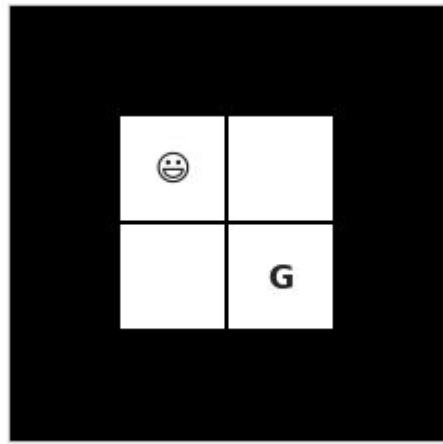


Start State

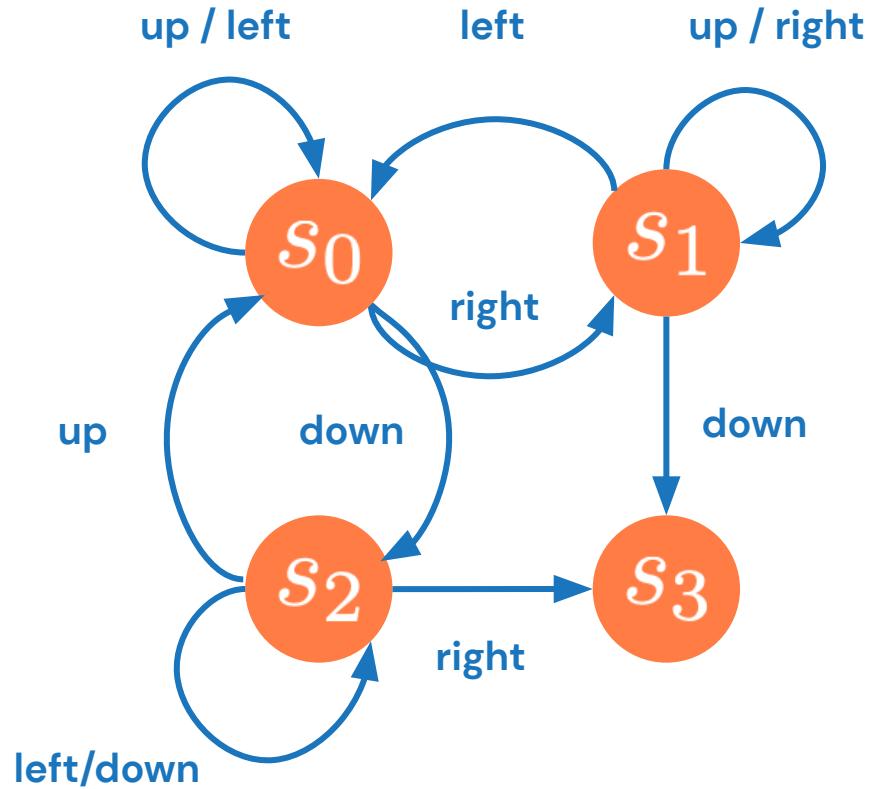
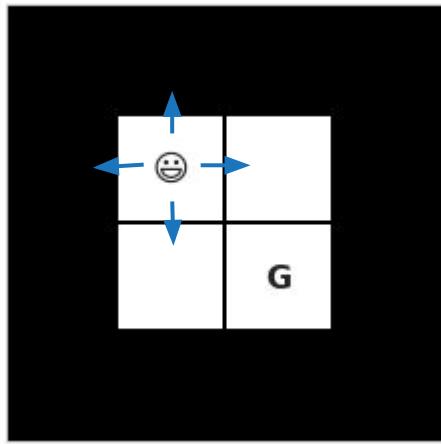
Terminal State



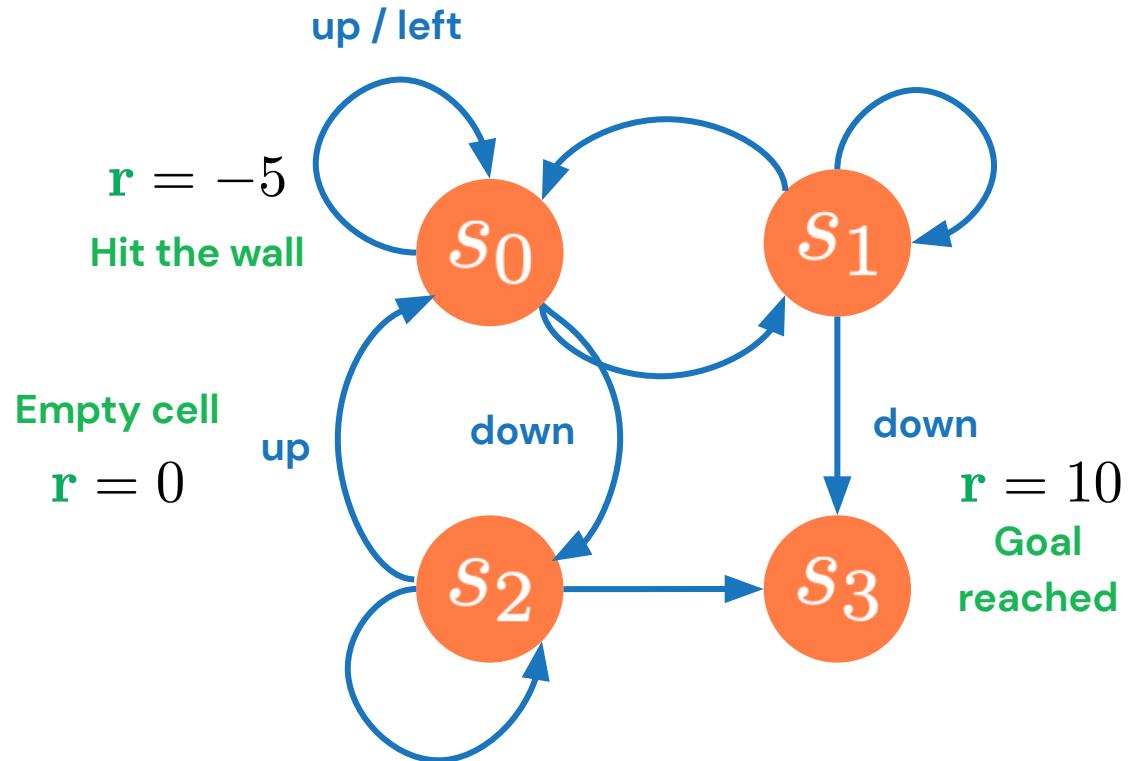
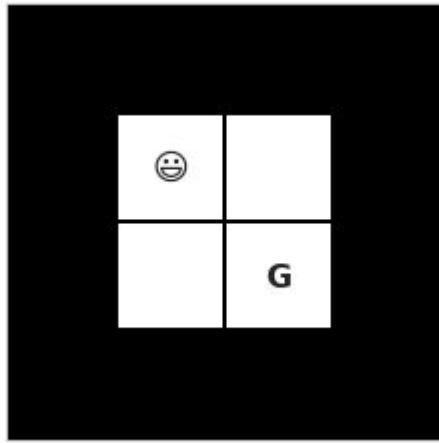
Action Space \mathcal{A}



Transition model P



Rewards R



Agent's Goal

The agent's **goal** is to maximize the **discounted sum of future rewards**.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$


Return

Discount γ specifies how much **future rewards** are worth compared to **immediate reward**.



Policy

Policy maps **states** to **actions**.

$$a_t \sim \pi(a_t | s_t)$$

Typically in RL we are searching for the **optimal policy**, which will maximize the **discounted sum of future rewards**.



State Value Function

State value function maps **states** to **expected rewards**.

$$V^\pi(s_t) = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t]$$

It is equal to **expected total reward** for an agent starting from that **state** and following its **policy**. It specifies how **good** it is to be in a given **state**.



State-Action Value Function

State-action value function maps an **action** in a given **state** to **expected rewards**.

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t, a_t]$$

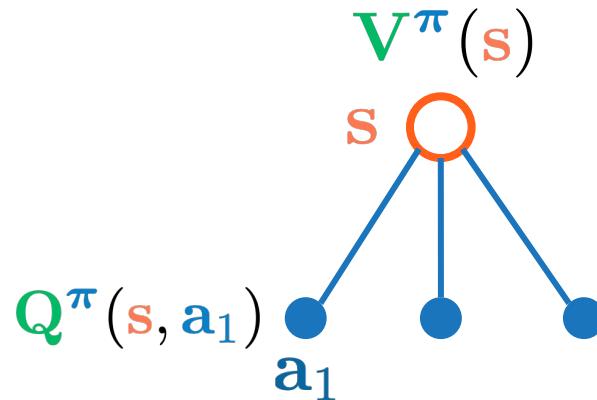
It is equal to **expected total discounted reward** for an agent starting from state **s** and performing action **a** and following its **policy**.



State-Action Value vs State Value

The relationship between V^π and Q^π

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s, a)$$



Overview

Fundamental Concepts

Value-based methods

Policy-based methods

Challenges & opportunities



What are we going to learn?

1. How to **compute** the value function?
2. How to **use it to optimise** the policy?
3. How to **learn** it, if we don't have the model of the environment?



Bellman Expectation Equation



Return at time t

$$V^\pi(s_t) = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t]$$

Return at time t+1

$$V^\pi(s_t) = \mathbb{E}_\pi [r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots) | s_t]$$

$$V^\pi(s_t) = \mathbb{E}_\pi [r_t + \gamma V(s_{t+1}) | s_t]$$

The **state-action value function** can be similarly decomposed to:

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi \mathbb{E}_{s_{t+1}} [r_t + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t, a_t]$$

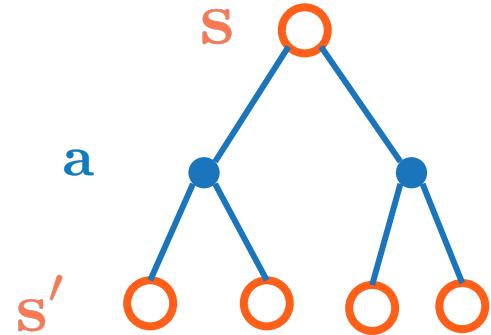


Bellman's principle of optimality

The **value** of a **state** under an **optimal policy** is equal to the **expected return** from the **best action** from that **state**.

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \max_{a'} Q^*(s', a')$$



Optimal policy

An **optimal policy** can be found by maximising over the **optimal state-action value function**. If we know the $\mathbf{Q}^*(\mathbf{s}, \mathbf{a})$ we immediately have the optimal policy $\pi^*(\mathbf{s})$.

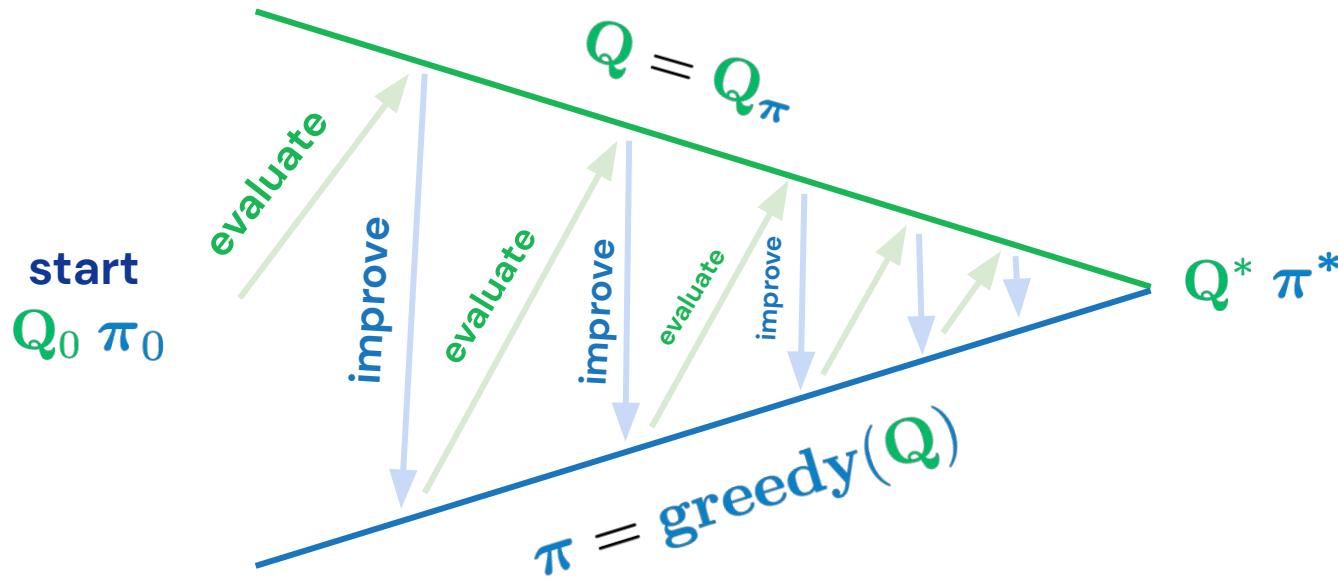
$$\pi^*(\mathbf{s}) = \max_{\mathbf{a}} \mathbf{Q}^*(\mathbf{s}, \mathbf{a})$$

We say that the **optimal policy** is **greedy** with respect to the **optimal value function**.



How to find the optimal policy? (Policy Iteration)

We can start from a random policy and random value function and find the optimal policy and value function through the following iterative process:



Why does this work? (Policy Iteration Theorem)

Given the **true value function** $Q_\pi(s, a)$ for any **policy** π for $s \in \mathcal{S}$ and $a \in \mathcal{A}$ it can always be **greedified** to obtain a **better policy**:

$$\pi' = \arg \max_a Q^\pi(s, a)$$

Where **better** means:

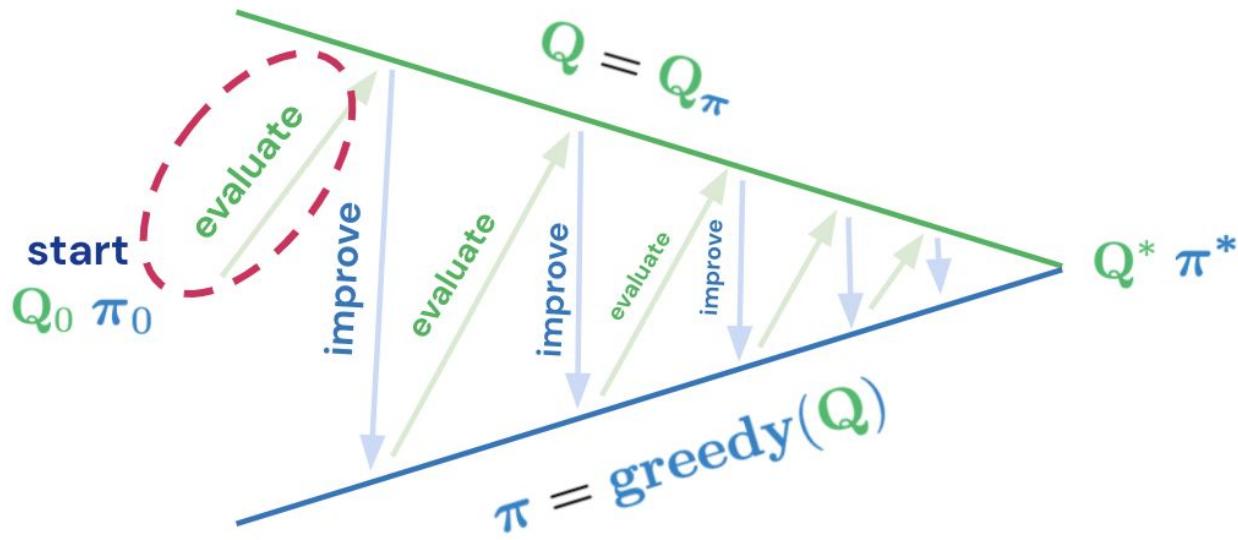
$$Q^{\pi'}(s, a) \geq Q^\pi(s, a)$$

Equality happens when we find the **optimal value function** and **policy**.



Iterative policy evaluation

During policy evaluation we are making the value function consistent with the current policy, also through an **iterative process**.



Iterative policy evaluation

Evaluate a given policy π iteratively to find its **true value function**:

$$Q_0 \rightarrow Q_1 \rightarrow \dots \rightarrow Q^\pi$$

At each iteration k :

For all states $s \in \mathcal{S}$ and $a \in \mathcal{A}$

Update $Q_k(s, a)$ from $Q_{k-1}(s', a')$



Model-based policy evaluation: Dynamic Programming

Dynamic programming finds the **optimal policy** given a perfect **model of the environment**.

DP algorithms turn Bellman equations into update rules and iterate until convergence:

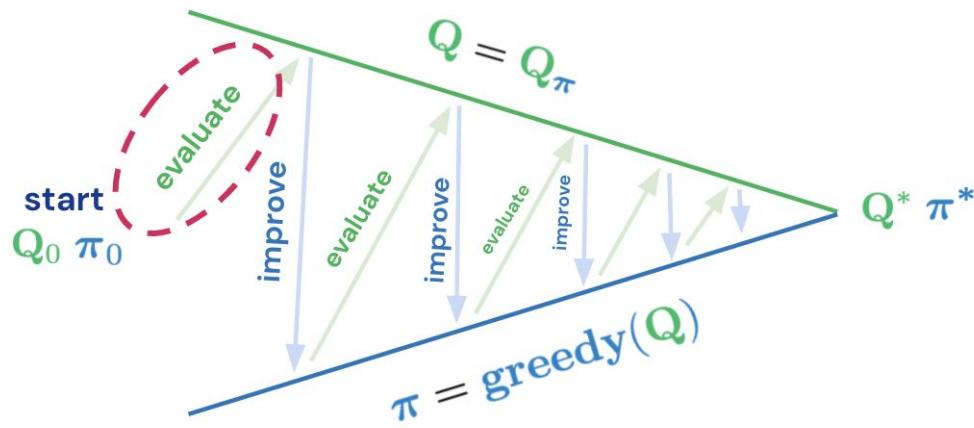
$$\mathbf{Q}_k(\mathbf{s}, \mathbf{a}) \leftarrow \mathbf{R}(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathbf{P}(\mathbf{s}' | \mathbf{s}, \mathbf{a}) \sum_{\mathbf{a}' \in \mathcal{A}} \pi_{k-1}(\mathbf{a}' | \mathbf{s}') \mathbf{Q}_{k-1}(\mathbf{s}', \mathbf{a}')$$



Policy Iteration

We can start from a random policy and random value function and find the optimal policy and value function through the following iterative process:

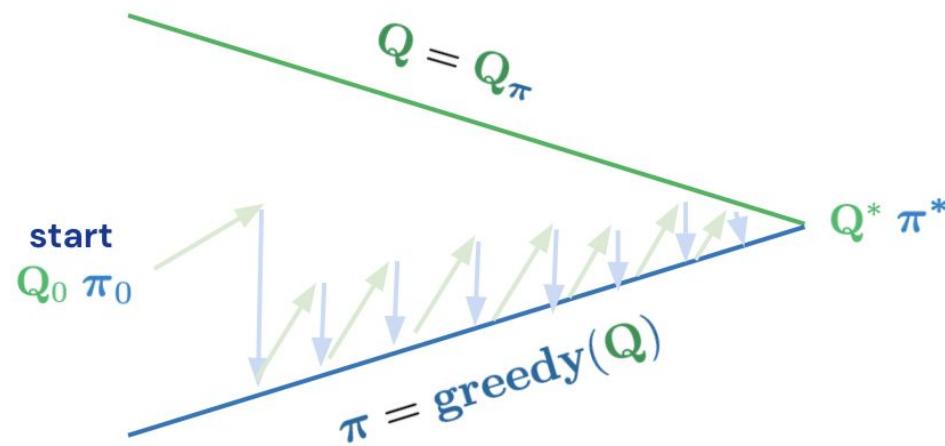
$$Q_k(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \sum_{a' \in \mathcal{A}} \pi_{k-1}(a'|s') Q_{k-1}(s', a')$$



Value Iteration

We can iteratively improve our estimation of the value function by directly using the **Bellman optimality equation**.

$$Q_k(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \max_{a'} Q_{k-1}(s', a')$$



Model-free policy evaluation

If we **do not have access to the true model of the environment**, we can instead learn directly from episodes of experience.

- Monte Carlo Methods
- Temporal Difference Learning



Monte Carlo Policy Evaluation

The goal is to update the value function towards the **actual return**.
We have to wait until the end of the episode to start learning.

$$Q_k(s_t, a_t) \leftarrow Q_{k-1}(s_t, a_t) + \alpha (G_t - Q_{k-1}(s_t, a_t))$$

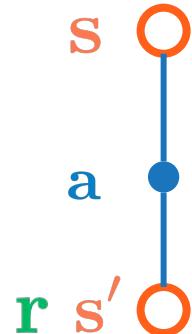
 TD Target

Reminder $G_t = r_t + \gamma r_{t+1} + \dots + \gamma^{T-1} r_T$



Temporal Difference Learning

Instead of waiting until the end of the episode, TD can learn from incomplete episodes via **bootstrapping**, updating a guess from a guess.



$$Q_k(s, a) \leftarrow Q_{k-1}(s, a) + \alpha \underbrace{(r + Q_{k-1}(s', \pi(s')) - Q_{k-1}(s, a))}_{\text{TD Target}}$$



TD vs MC

- TD learns **after each step** while MC has to wait for the **end of episode** when **return** is known.
- TD can learn in **non-episodic** environments while MC cannot.

Bias/Variance Tradeoff

- **Return** is an **unbiased estimation** of the value function while TD target is a **biased estimation** (a guess from a guess).
- TD target is much **lower variance** than the **return**
 - **Return** depends on sequences of random **actions**, **transitions** and **rewards**.
 - TD target depends on **one random action**.



Model-free learning

When we don't have access to the model of environment we learned how to do **model-free evaluation**, now we focus on **model-free learning** where we want to learn to optimise the **value function** of an unknown **MDP**.

On-policy learning “Learning on the job”

Off-policy learning “Look over someone's shoulder”



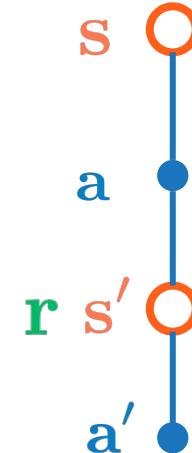
Exploration

- **ϵ -Greedy** is a simple **policy improvement** method that allows for continual exploration.
- **Exploit:** with probability $1-\epsilon$ we select a **greedy action** wrt the **value function**.
- **Explore:** with probability ϵ a **random action** is uniformly selected instead.



SARSA: On-policy TD Control

- SARSA learns the **policy** by executing it in the environment to gather experiences.
- We often use **ϵ -Greedy policy** to gather experiences and each update needs access to $(\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}', \mathbf{a}')$.

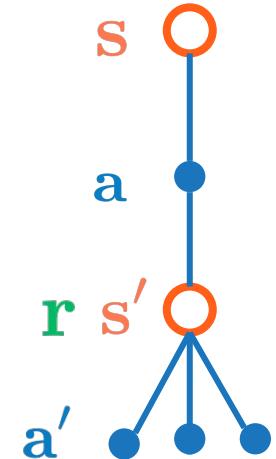


$$\mathbf{Q}(\mathbf{s}, \mathbf{a}) \leftarrow \mathbf{Q}(\mathbf{s}, \mathbf{a}) + \alpha(\mathbf{r} + \gamma \mathbf{Q}(\mathbf{s}', \mathbf{a}') - \mathbf{Q}(\mathbf{s}, \mathbf{a}))$$



Q Learning: Off-policy TD Control

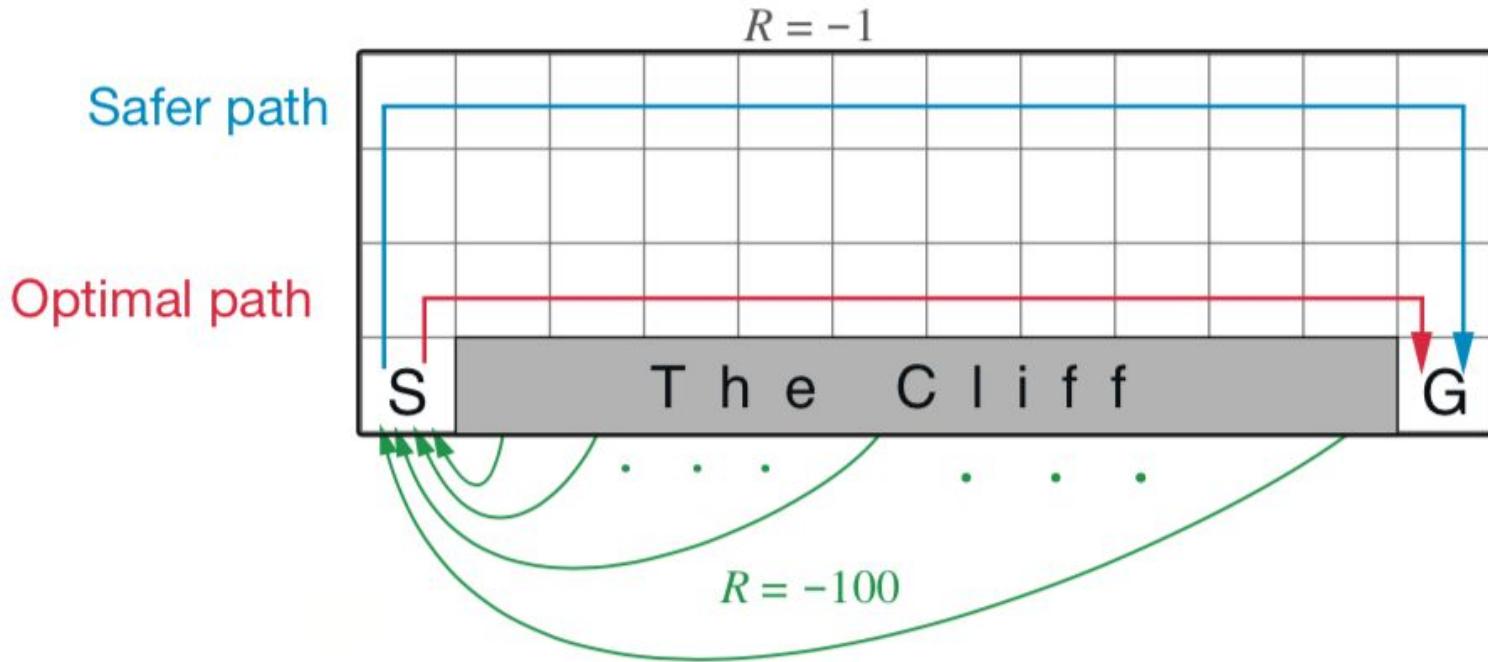
- Q-learning learns about the **value** of its **greedy policy** from experiences gathered by a **different policy** (e.g. ϵ -Greedy).
- We use **max** of the Q-value for a' instead of executing the policy for an extra step.



$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

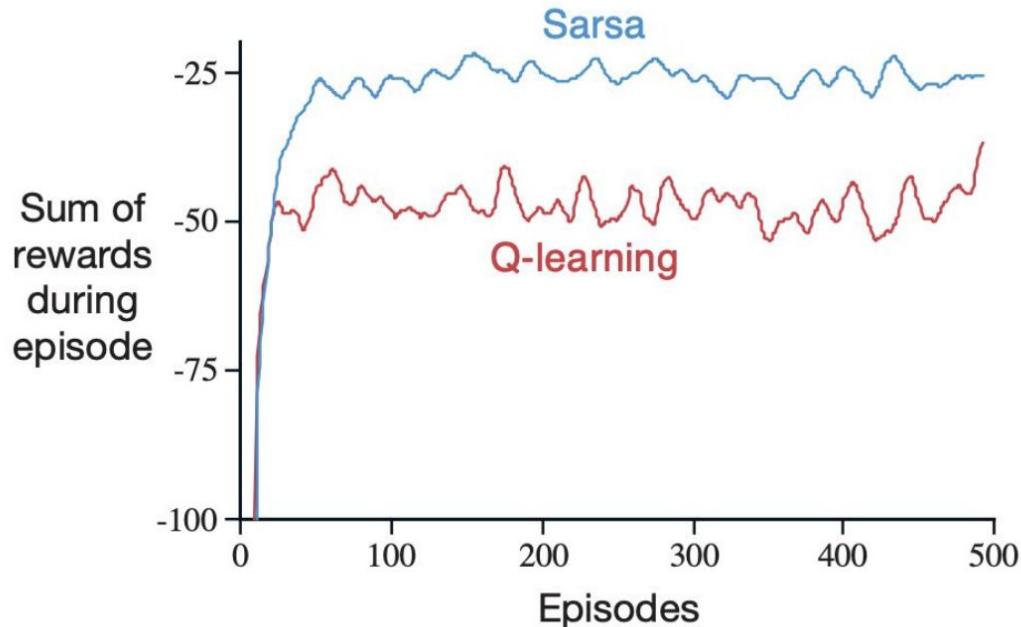
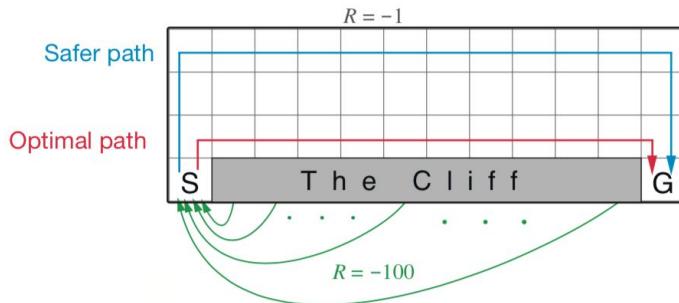


SARSA vs Q-Learning: The Cliff-walking Example

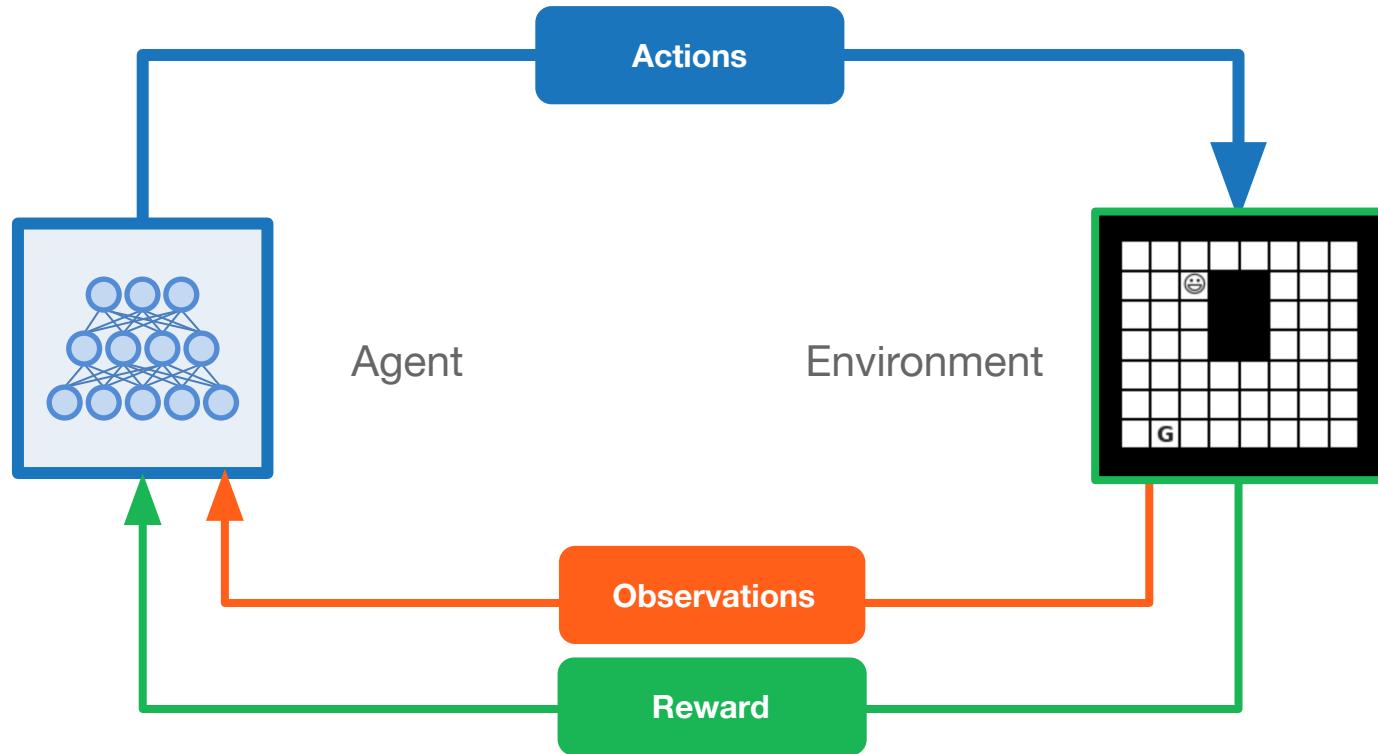


SARSA vs Q-Learning: The Cliff-walking Example

- Q-learning learns the **optimal path** while its online performance is worse than SARSA.
- SARSA learns the **safer path**.

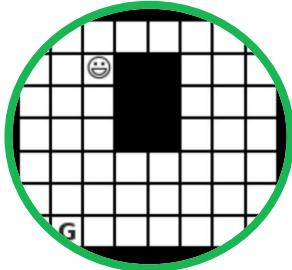


Function approximation



Why function approximation?

- Up to now we used lookup tables to represent the **value function**, for example each $s \in \mathcal{S}$ and $a \in \mathcal{A}$ has an entry in the **Q-table** $Q(s, a)$
- In **large MDPs** with large **action** and **state** space this becomes infeasible due to **memory requirements** and **inefficiency of learning** the **value** of each **state** in isolation.



Why function approximation?

- With **function approximation**, we can tackle these **large MDPs**, by **generalizing from seen states to unseen states**.
- We can use the **TD target** we learned about earlier as a **loss function** to optimize using gradient descent.
- Our goal is to find the parameter vector **w** minimising the **mean-squared error** between the **approximate value function** $\hat{Q}_w(s, a)$ and the **true value function**.



The deadly triad

Bad news: there is possibility of **divergence** when we combine these three powerful ingredients:

Function approximation

Significantly generalising from experiences

+ Bootstrapping

Learning value estimates from other value estimates (e.g. in TD and DP)

+ Off-policy learning

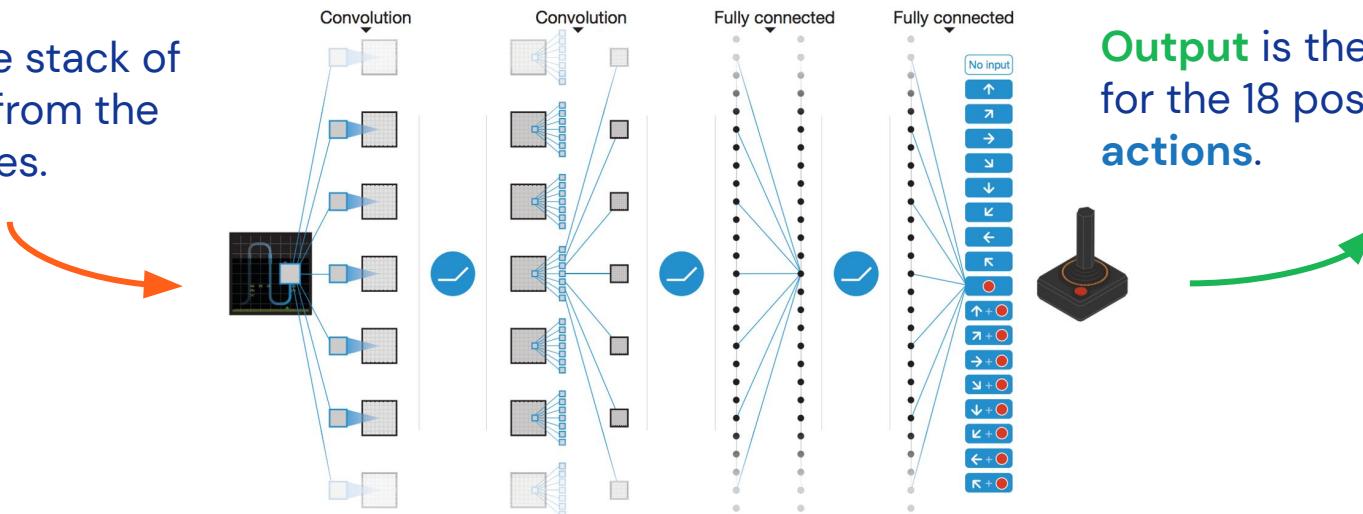
Learning from experiences collected by a different behaviour policy than the target policy we are learning about (e.g. Q-learning).



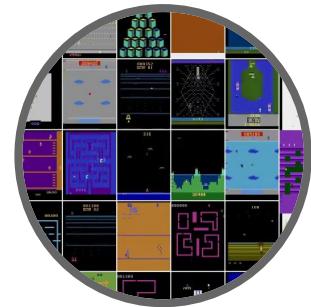
Deep Q-Networks (DQN)

End-to-end learning of the **value function** $Q(s, a)$ from pixels.

Input is the stack of raw pixels from the last 4 frames.

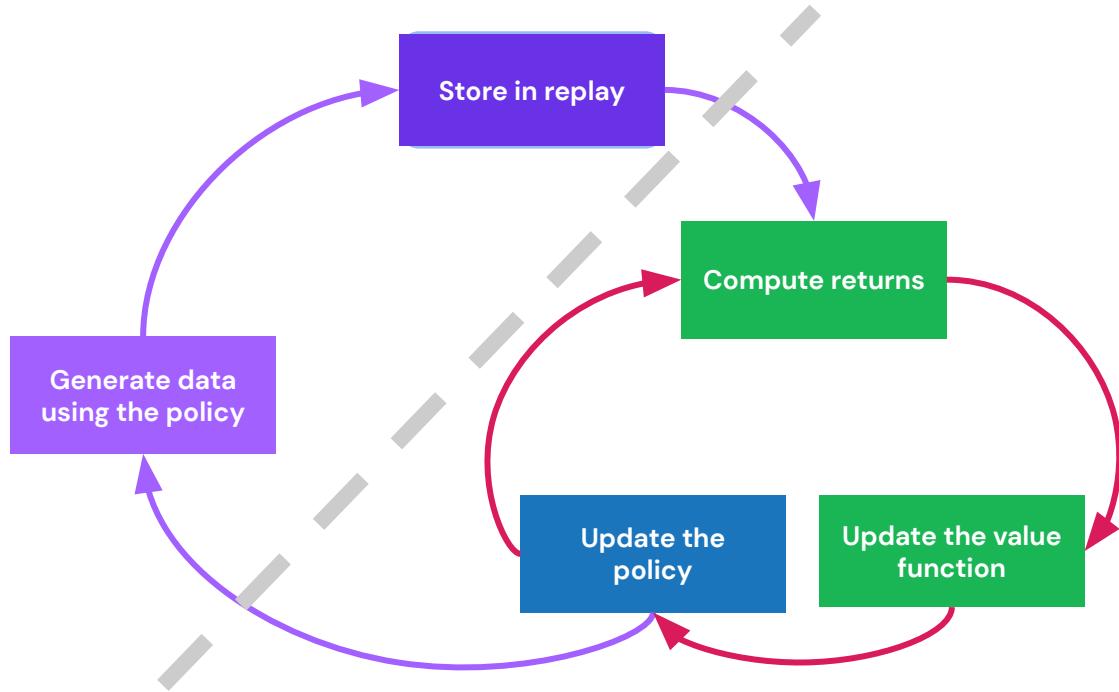


(Mnih et al., 2013)



Replay

Off-policy methods can also make use of **replay** (storage of data)



In some sense there are
two loops:

- Data Generation
and
- Parameter optimization

Can we split these?



The DQN Loss

With DQN we can't even apply the Q-function update in closed form

$$L(\mathbf{W}) = \mathbb{E}_{\mathbf{s}, \mathbf{a}, \mathbf{s}'} \left[\underbrace{\left(\mathbf{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}') + \gamma \max_{\mathbf{a}'} \mathbf{Q}_{\mathbf{W}_{\text{old}}}(\mathbf{s}', \mathbf{a}') - \mathbf{Q}_{\mathbf{W}}(\mathbf{s}, \mathbf{a}) \right)^2}_{\text{optimized value function}} \right]$$

old/updated value function

so instead we minimize the loss between

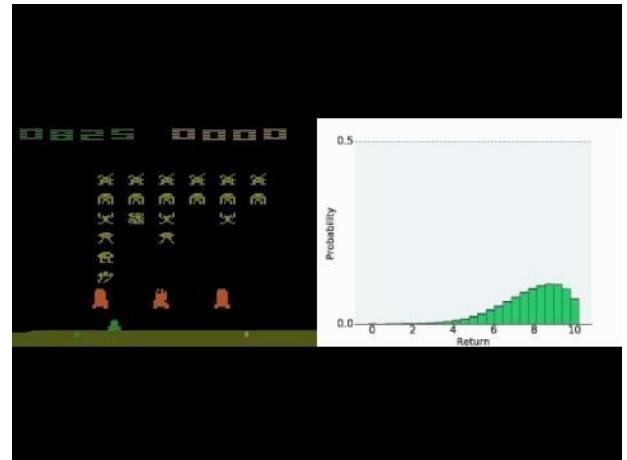
- an **Bellman-updated** value function and
- the **current** value



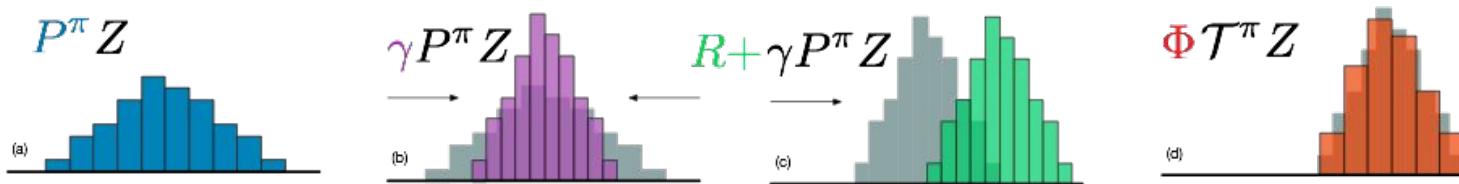
Distributional RL

Distributional RL goes further and learns a **distribution over returns!**

$$Q_{\mathbf{W}}(\mathbf{s}, \mathbf{a}) = \mathbb{E} \underbrace{\mathbf{Z}_{\mathbf{W}}(\mathbf{s}, \mathbf{a})}_{\text{distribution over value}}$$



The **Bellman update** and **loss** now involve these distributions

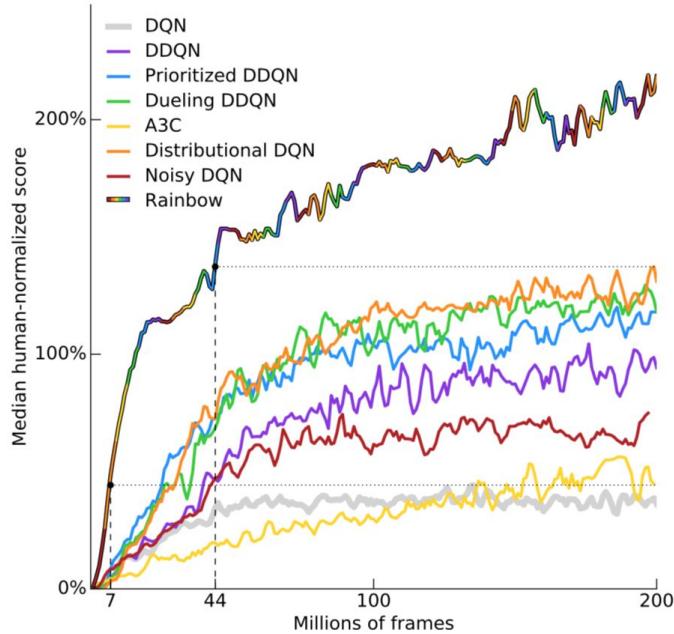


(Bellemare et al., 2017)

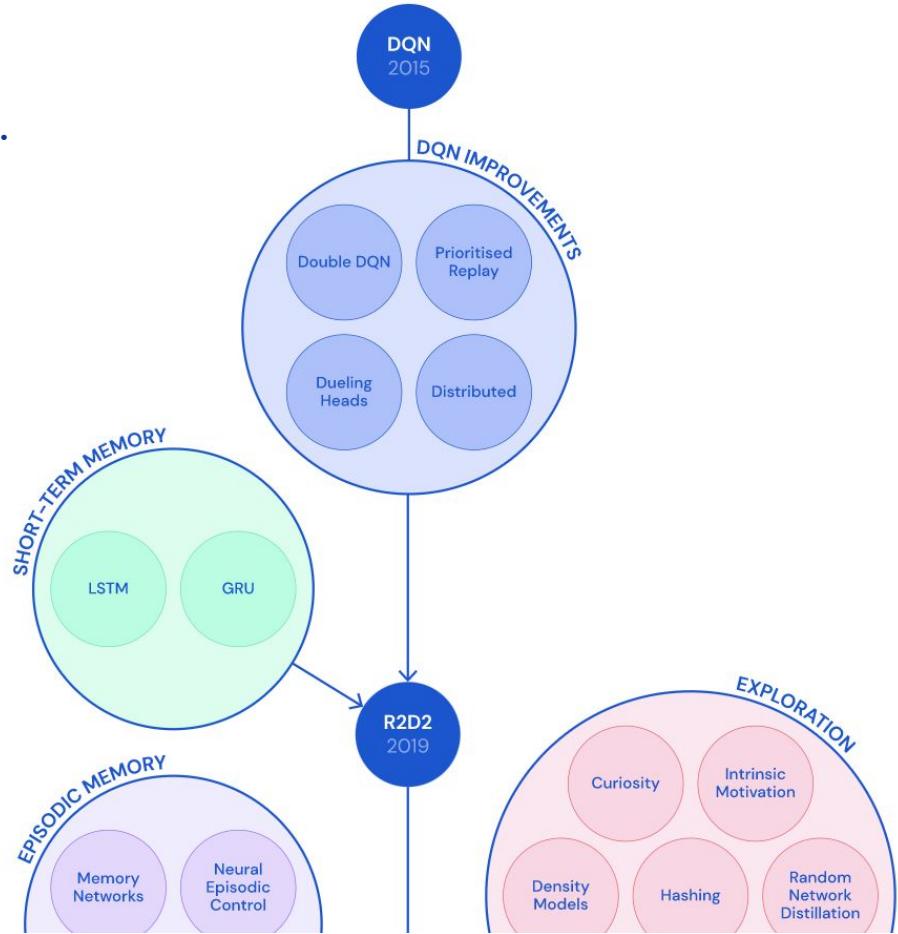


Rainbow

Combines a collection of algorithmic improvements and large-scale training.

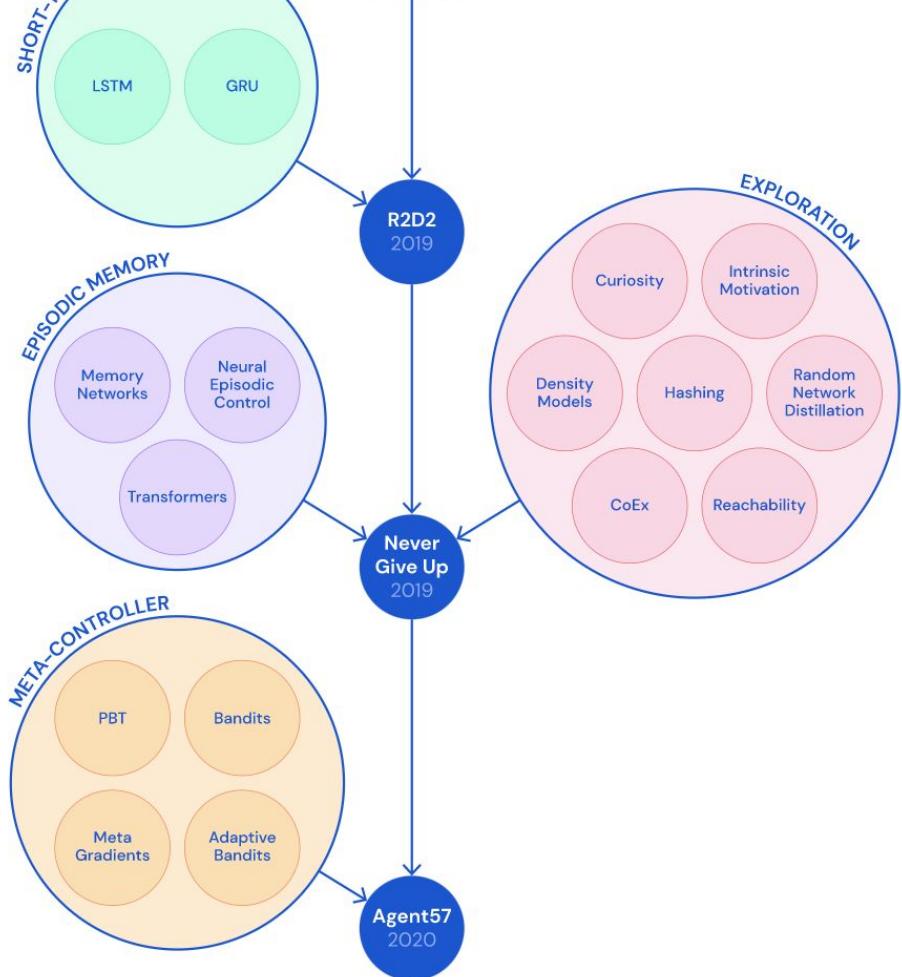


(Hessel et al., 2018)



Agent-57

- Leverages an efficient exploration strategy which is novelty seeking
- Bandit meta-controller that adapts the long-term vs short-term behaviour of the agent.



(Puigdomènech Badia, et al., 2020)

Overview

Fundamental Concepts

Value-based methods

Policy-based methods

Challenges & opportunities



What are we going to learn?

1. What is a **parameterized policy**
2. How to derive the **policy gradient**
3. How this relates to **value functions**
4. How minor variations lead to **vastly different algorithms**



What is policy optimisation?

We explicitly represent the **policy** with its own parameters independent of any **value function**

$$a_t \sim \pi_\theta(a_t | s_t) \quad \text{vs} \quad a_t = \arg \max_a Q(s_t, a) = \pi(s)$$

We update the **policy parameters** using **gradient ascent** to maximise expected future reward.



Why policy optimisation?

- Learning the **policy** can sometimes be simpler than learning the **value function**
- Particularly better suited in **continuous control** and **large action spaces**.
- In **some** problems the optimal policy is **stochastic**
- But stochasticity can make **exploration** easier



Directly parameterizing the policy

Future rewards
exactly as seen
previously!

We'll now turn to **parameterized policies** given by θ

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

and we will write the **expected return** as a function of these parameters

$$\begin{aligned} J(\theta) &= \mathbb{E}_{s_{0:T+1}, a_{0:T}} \left[\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right] \\ &= \sum_{s_{0:T+1}, a_{0:T}} \left[\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right] \Pr(s_{0:T+1}, a_{0:T}) \end{aligned}$$

Summing over all possible trajectories

Multipled by their probability

In what follows we'll show how to **compute the gradient** of this



The log-derivative trick

The gradient of our **expected return** moves inside and only applies to the distribution over trajectories

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \sum_{s_{0:T+1}, a_{0:T}} \left(\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right) \nabla_{\theta} \Pr(s_{0:T+1}, a_{0:T} | \theta) \\ &= \sum_{s_{0:T+1}, a_{0:T}} \left(\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right) \underbrace{\left(\nabla_{\theta} \log \Pr(s_{0:T+1}, a_{0:T} | \theta) \right)}_{\text{The "log-derivative trick"}} \Pr(s_{0:T+1}, a_{0:T} | \theta)\end{aligned}$$

$$\nabla f(x) = \nabla \log f(x) f(x)$$

which can be again re-written as an expectation over **trajectories**

$$= \mathbb{E}_{s_{0:T+1}, a_{0:T}} \left[\left(\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right) \left(\nabla_{\theta} \log \Pr(s_{0:T+1}, a_{0:T} | \theta) \right) \right]$$

Generally useful!



Gradient of a trajectory's log-probability

Recall from the **Markov property** our probability factorizes

$$\Pr(\mathbf{s}_{0:T+1}, \mathbf{a}_{0:T} | \boldsymbol{\theta}) = \Pr(\mathbf{s}_0) \prod_{t=0}^T \pi_{\boldsymbol{\theta}}(\mathbf{a}_t | \mathbf{s}_t) \mathbf{P}(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

so the gradient of its log simplifies to

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \log \Pr(\mathbf{s}_{0:T+1}, \mathbf{a}_{0:T} | \boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left[\log \Pr(\mathbf{s}_0) + \sum_{t=0}^T \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t | \mathbf{s}_t) + \sum_{t=0}^T \log \mathbf{P}(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t) \right] \\ &= \sum_{t=0}^T \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t | \mathbf{s}_t) \end{aligned}$$

independent of θ



The policy gradient (putting it all together)

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \mathbb{E} \left[\left(\sum_{t=0}^T \gamma^t R(s_t, a_t, s_{t+1}) \right) \left(\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right) \right] \\ &= \mathbb{E} \left[\sum_{t=0}^T \left(\sum_{n=0}^T \gamma^t R(s_n, a_n, s_{n+1}) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]\end{aligned}$$

← Distribute the policy

$$= \mathbb{E} \left[\sum_{t=0}^T \left(\overbrace{\sum_{n=0}^{t-1} \gamma^t R(s_n, a_n, s_{n+1})}^{\text{before } t} + \overbrace{\sum_{n=t}^T \gamma^t R(s_n, a_n, s_{n+1})}^{\text{after } t} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

$$= \mathbb{E} \left[\sum_{t=0}^T \left(\underbrace{\sum_{n=t}^T \gamma^t R(s_n, a_n, s_{n+1})}_{\text{future rewards } G_t} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

rewards at time $n < t$ are independent of actions a_t and the gradient becomes zero.

future rewards G_t

direction to move to make action more likely

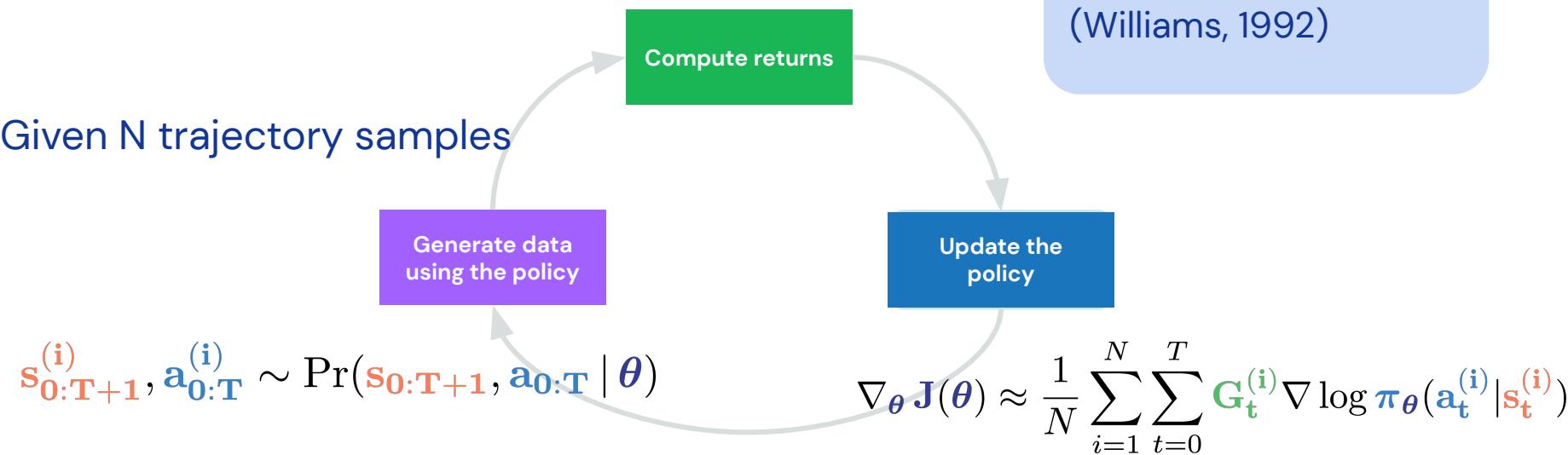


“Vanilla” policy gradients (REINFORCE)

$$G_t^{(i)} = \sum_{n=t}^T R(s_n^{(i)}, a_n^{(i)}, s_{n+1}^{(i)})$$

The basis of the
REINFORCE algorithm
(Williams, 1992)

Given N trajectory samples



$$s_{0:T+1}^{(i)}, a_{0:T}^{(i)} \sim \Pr(s_{0:T+1}, a_{0:T} | \theta)$$

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^T G_t^{(i)} \nabla \log \pi_{\theta}(a_t^{(i)} | s_t^{(i)})$$



Connection with Value-based methods

Recall the **Q-function**, evaluated at some **initial state** and **action**,

$$Q^{\pi_\theta}(s_0, a_0) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0, a_0 \right]$$

We can replace reward trajectories with this function in our gradient, i.e.

$$\nabla_\theta J(\theta) = \mathbb{E} \left[\sum_{t=0}^T \gamma^t Q^{\pi_\theta}(s_t, a_t) \nabla \log \pi_\theta(a_t | s_t) \right]$$

discounting makes
dependence on **initial**
state more important

future rewards



The Policy Gradient Theorem

The **policy gradient theorem** (Sutton et al., 2000) directly relates the **policy gradient** and **Q**

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{d^{\pi_{\theta}}(s)\pi_{\theta}(a|s)} [\overbrace{Q^{\pi_{\theta}}(s, a)}^{\leftarrow} \nabla \log \pi_{\theta}(a|s)]$$

where $d^{\pi_{\theta}}(s)$ is the **stationary distribution** under the given policy*

Policy search methods often differ primarily in how
future rewards are estimated.

*or the discounted probability
of visiting at any time t.



Advantage Actor Critic (A2C) and Baselines

The policy gradient can have *high variance*. Can we reduce it?

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{d^{\pi}(\mathbf{s}) \pi(\mathbf{a}|\mathbf{s})} [(\underbrace{Q^{\pi_{\theta}}(\mathbf{s}, \mathbf{a}) - V^{\pi_{\theta}}(\mathbf{s})}_{A^{\pi_{\theta}}(\mathbf{s}, \mathbf{a})}) \nabla \log \pi_{\theta}(\mathbf{a}|\mathbf{s})]$$

Baseline! Expectation zero
because independent of \mathbf{a}

The **advantage** measures how much **better** \mathbf{a} is than following the policy $\pi_{\theta}(\mathbf{s})$

- A2C:
- Estimate Q using returns $G_t^{(i)}$
 - Estimate V with a neural network
- (Mnih, et al., 2015)



Deterministic Policy Gradients (DPG)

Usually **optimal policies** are deterministic $\mathbf{a} = \pi_\theta(\mathbf{s})$

- Q-learning can do this with value-functions.
- Can we do so for policy search?

The **deterministic policy gradient theorem** (Silver et al., 2014) does exactly this:

$$\nabla_\theta J(\theta) = \mathbb{E}_{d^\pi(\mathbf{s})} \left[\underbrace{\nabla_a Q^{\pi_\theta}(\mathbf{s}, \mathbf{a})|_{\mathbf{a}=\pi_\theta(\mathbf{s})}}_{\text{direction to move } \mathbf{a} \text{ to make the value higher}} \underbrace{\nabla_\theta \pi_\theta(\mathbf{s})}_{\text{how moving } \theta \text{ influences the direction of } \mathbf{a}} \right]$$

Used with deep networks this is Deep DPG (**DDPG**)
(Lillicrap et al., 2018)



Actor Critic Methods in General

We have *sneakily* introduced **Actor-Critic** methods that:

- Learn a **value function** (V or Q) or **advantage**
- Use this **value function** to learn the **policy**

For example **DDPG** explicitly splits these into two steps:

- Update a parameterized $Q_w(s, a)$ using **TD-learning**
- Update a parameterized $\pi_\theta(s)$ using Q



What about the gradient step?

Sometimes **just following** the gradient step can be unstable

- Many algorithms **penalize** the gradient step
- Often based on the **entropy or KL** between the new and old policy

$$D_{KL}(\pi_{\theta'} \parallel \pi_{\theta})$$

This leads to novel algorithms:

- KL + REINFORCE = **TRPO** or **PPO**
- Entropy + A2C = **ACER** or **IMPALA**
- KL + Actor-Critic = **MPO**

(Schulman et al., 2015; 2017)

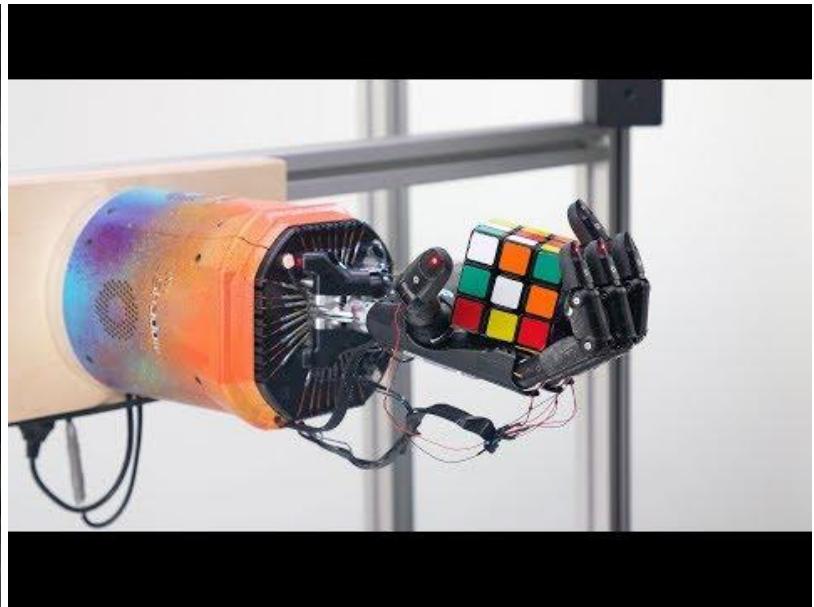
(Wang et al., 2016)

(Espeholt et al., 2018)

(Abdolmaleki et al., 2018)



On-policy methods like PPO shine in simulation

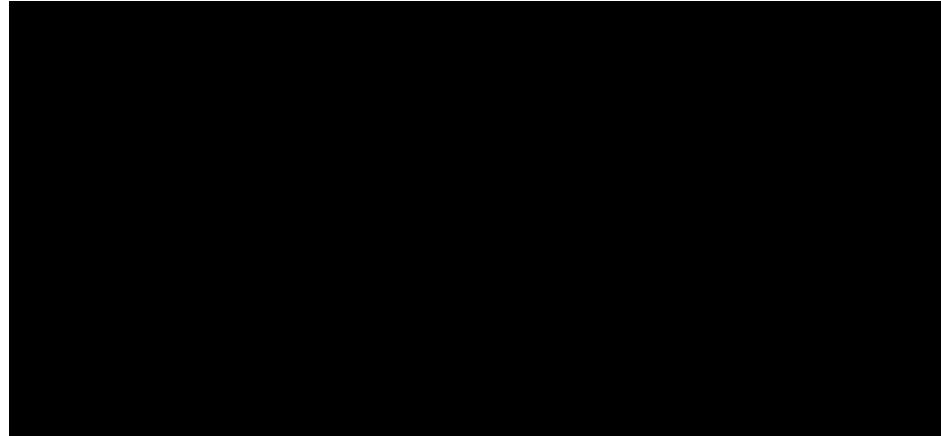
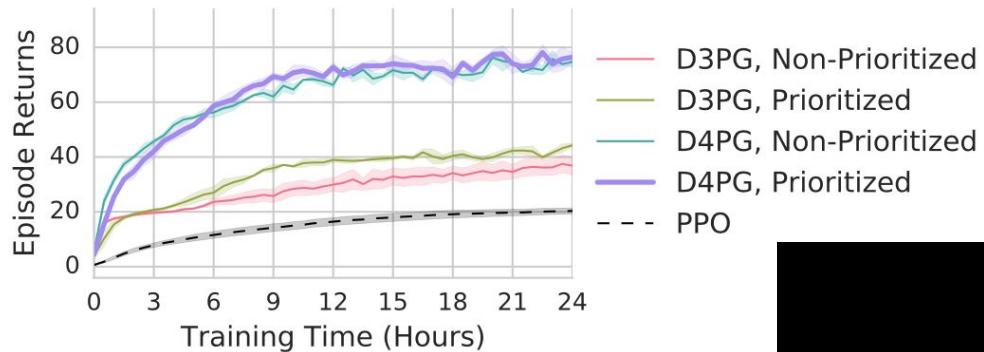


(OpenAI, et al., 2019)



Distributed Distributional DDPG (D4PG)

Adds **Distributed** and **Distributional** to DDPG.

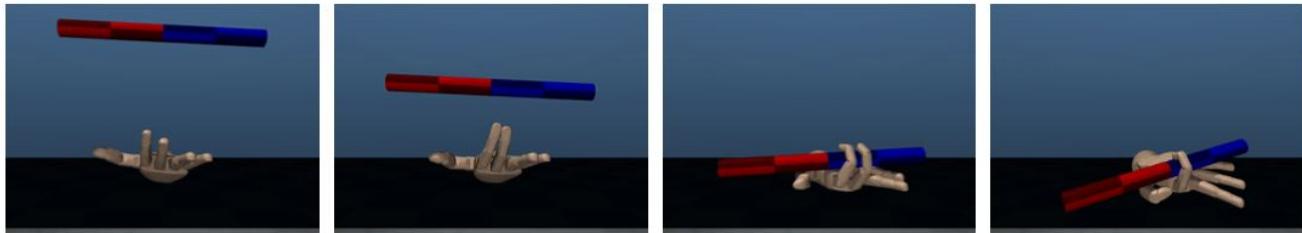


(Barth-Maron et al., 2018)

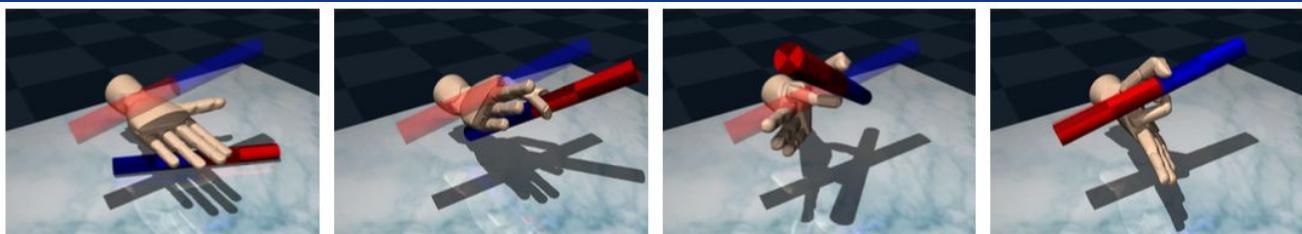


Manipulation results

Catch:



Match moving target:



Pickup and Orient:



Overview

Fundamental Concepts

Value-based methods

Policy-based methods

Challenges & opportunities



Plethora of RL methods

Value-based
vs
policy-based

Model-based
vs
Model-free

on-policy
vs
off-policy

Offline
vs
Online

Episodic
vs
Lifelong

...



Real world Reinforcement Learning

Challenges

- Large action and state spaces
- Usually interaction with the environment is **expensive** e.g. robotics
- Reward function is unspecified, risk sensitive or multi-objective
- Large and unknown delays in actuators, sensors and rewards
- Requirements for fast inference
- Exploration
- Generalization



Real world Reinforcement Learning

Opportunities

- Offline-RL (e.g. D4RL, RL Unplugged)
- Automatic dataset construction / improve exploration
- RL Frameworks (e.g. Acme, Dopamine, RLLib, SEED)
- Representation learning (e.g. object based, contrastive losses)
- Temporal abstractions (e.g. options)
- Memory (e.g. transformers, LSTMs, NTM)

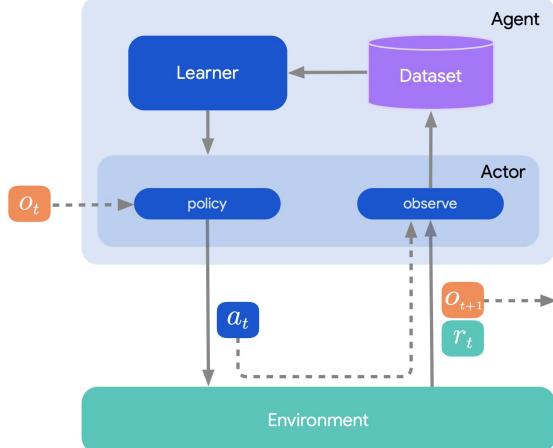


Distributed RL

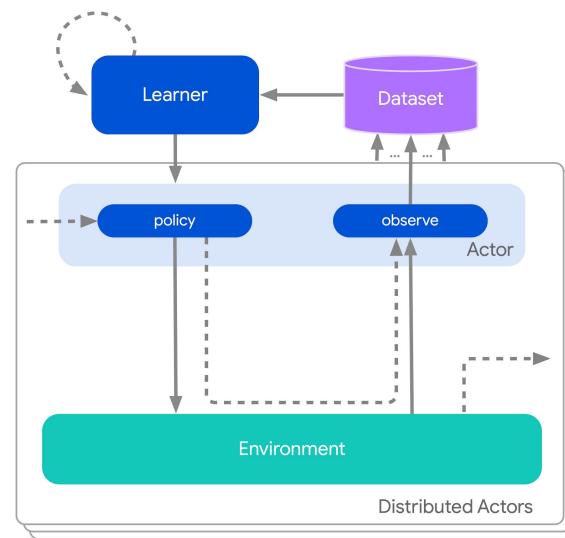
There are often **two loops** in RL: data collection and optimization

- We can exploit this by splitting the loops and running in parallel!

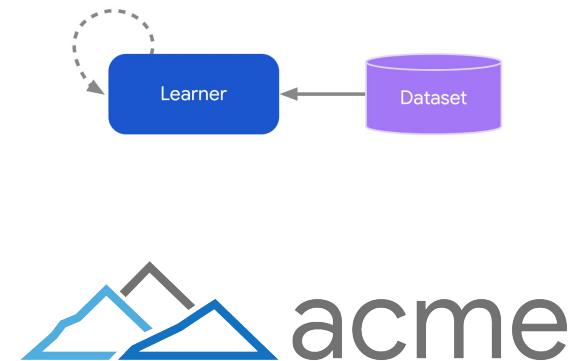
Single-process



Distributed



Batch



Want to learn more?

Books

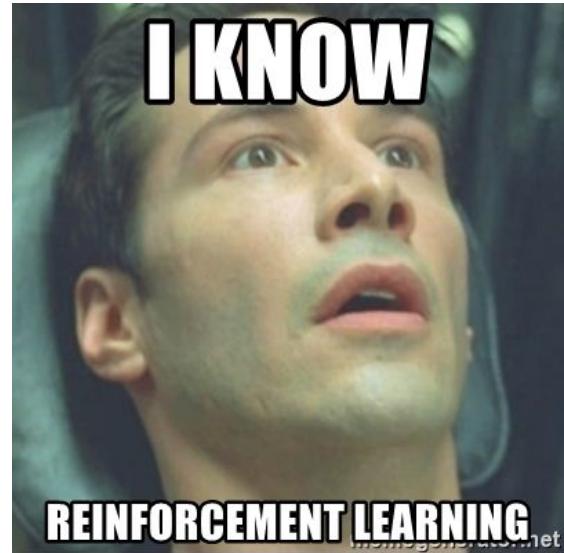
- [Reinforcement Learning: an Introduction by Sutton & Barto](#)
- [Algorithms for Reinforcement Learning by Csaba Szepesvari](#)

Lectures and course

- [RL Course by David Silver](#)
- [Reinforcement Learning Course | UCL & DeepMind](#)
- [Emma Brunskill Stanford RL Course](#)
- [RL Course on Coursera by Martha White & Adam White](#)

More practical

- [Spinning Up in Deep RL by Josh Achiam](#)
- [Acme white paper & Colab tutorial](#)



DeepMind

Thank you

Special thanks to



Bobak Shahriari



Nando de Freitas



[@mw_hoffman](https://twitter.com/mw_hoffman)
[@feryalmp](https://twitter.com/feryalmp)



References

- Mnih et al., 2013 [Playing Atari with Deep Reinforcement Learning](#)
- Bellemare, Dabney, Munos, 2017 [A Distributional Perspective on Reinforcement Learning](#)
- van Hasselt et al., 2018 [Deep Reinforcement Learning and the Deadly Triad](#)
- Hessel et al., 2017 [Rainbow: Combining Improvements in Deep Reinforcement Learning](#)
- Puigdomènech Badia et al., 2020 [Agent57: Outperforming the Atari Human Benchmark](#)
- Sutton et al., 2000 [Policy Gradient Methods for Reinforcement Learning with Function Approximation](#)
- Lillicrap et al., 2015 [Continuous control with deep reinforcement learning](#)
- Barth-Maron et al., 2018 [Distributed Distributional Deterministic Policy Gradients](#)
- Mnih et al., 2016 [Asynchronous Methods for Deep Reinforcement Learning](#)
- OpenAI et al., 2019 [Solving Rubik's Cube with a Robot Hand](#)
- Schulman et al., 2015 [Trust Region Policy Optimization](#)
- Schulman et al., 2017 [Proximal Policy Optimization Algorithms](#)
- Espeholt et al., 2018 [IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures](#)
- Abdolmaleki et al., 2018 [Maximum a Posteriori Policy Optimisation](#)
- Wang et al., 2016 [Sample Efficient Actor-Critic with Experience Replay](#)
- Silver et al., 2017 [A general reinforcement learning algorithm that masters chess, shogi and Go through self-play](#)
- Fu et al., 2020 [D4RL: Datasets for Deep Data-Driven Reinforcement Learning](#)
- Gulcehre et al., 2020 [RL Unplugged: Benchmarks for Offline Reinforcement Learning](#)

