Introduction to Deep RL, Part 1

Joshua Achiam

OpenAI

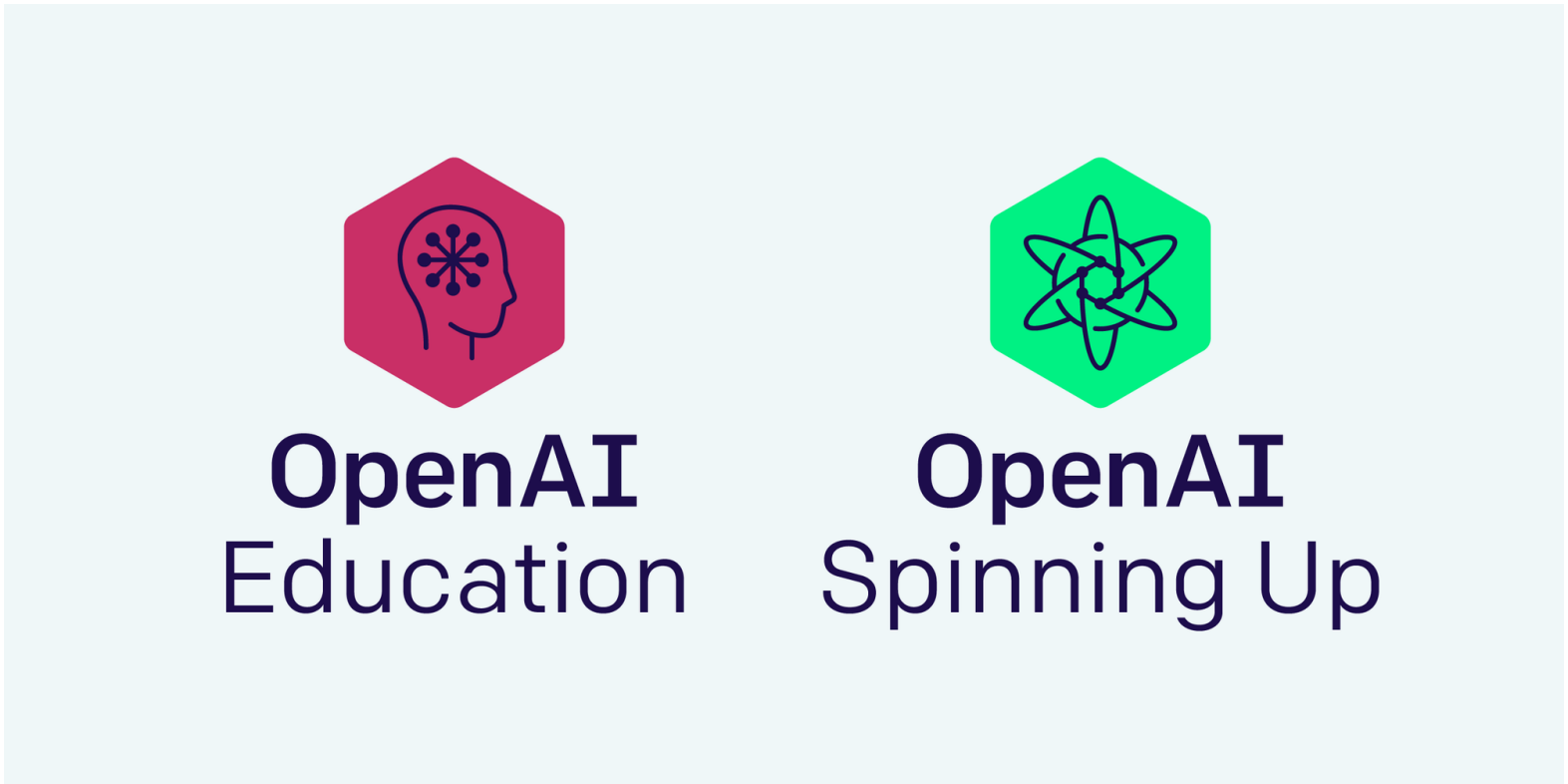
February 2, 2019

1.0: What is Spinning Up?

Education at OpenAI

Mandate from OpenAI Charter:

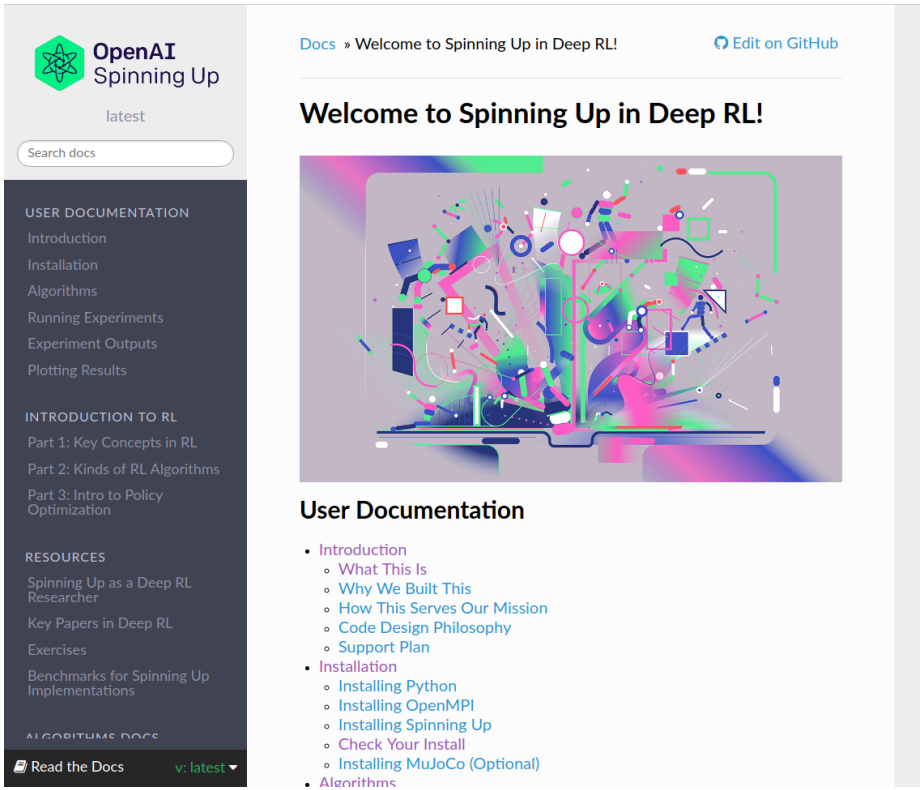
- "We seek to create a global community working together to address AGIs global challenges."
- "We are committed to providing public goods that help society navigate the path to AGI."



https://blog.openai.com/openai-charter/

Spinning Up in Deep RL

Deep RL will be a key technology in AGI: therefore, it is important to educate the public.



- A short intro to RL
- An essay about becoming an RL researcher
- A curated list of important papers
- A code repo of key algorithms (VPG, TRPO, PPO, DDPG, TD3, SAC)
- Warm-up coding exercises

https://spinningup.openai.com/en/latest/

Spinning Up Workshops

Hypothesis:

- Working with people directly will help them learn and build skills faster

Educational objectives:

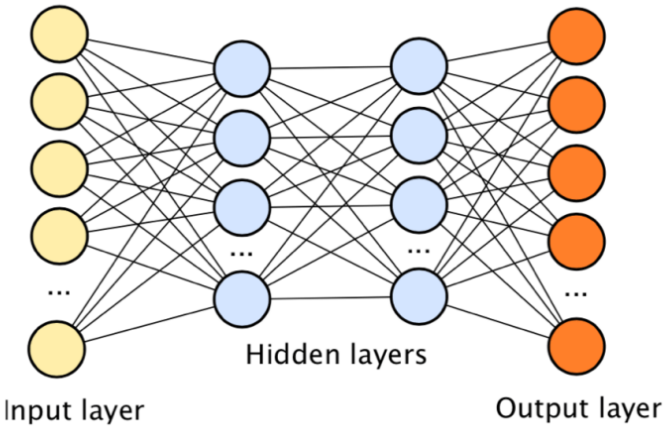
- Teach current capabilities and limitations of deep RL
- Raise awareness of work that needs doing in the field
- Participants run and tinker with deep RL algorithms for the first time, and feel confident that they can keep doing it

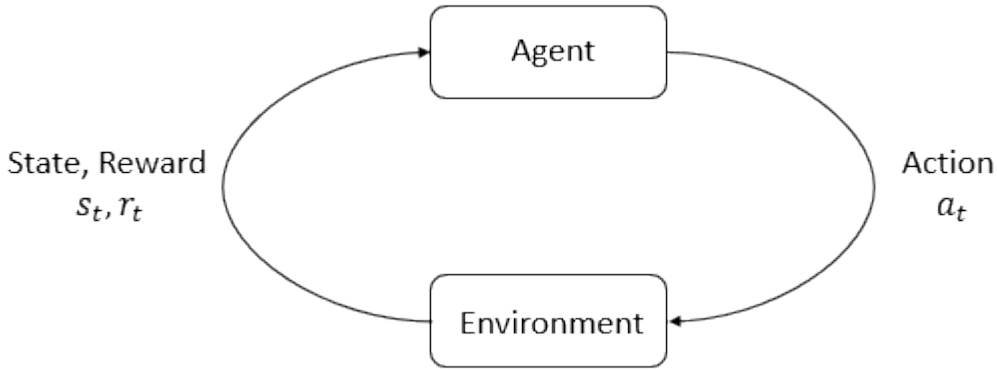
1.1: What is deep RL, and why do we need it?

What is Deep RL?

Deep RL is the combination of **reinforcement learning (RL)** with **deep learning**

- **Reinforcement learning** is about solving problems by **trial and error**
- **Deep learning** is about using **deep neural networks** to solve problems
- $\Longrightarrow$ **Deep reinforcement learning** trains deep neural networks with trial and error





Deep neural network[1] and RL interaction loop

[1] Deep neural network image from Gary Marcus

When would you want to use RL?

RL is useful when

- you have a sequential decision-making problem
- you do **not** know the optimal behavior already[1]
- but you can still evaluate whether behaviors are good or bad

That is,

RL is useful when evaluating behaviors is easier than generating them

[1] Critical difference from supervised learning!

When would you want to use deep learning?

Deep learning is useful when

- you want to approximate a function
- function requires "intelligence"
- inputs and/or outputs are high-dimensional
- lots of data is available

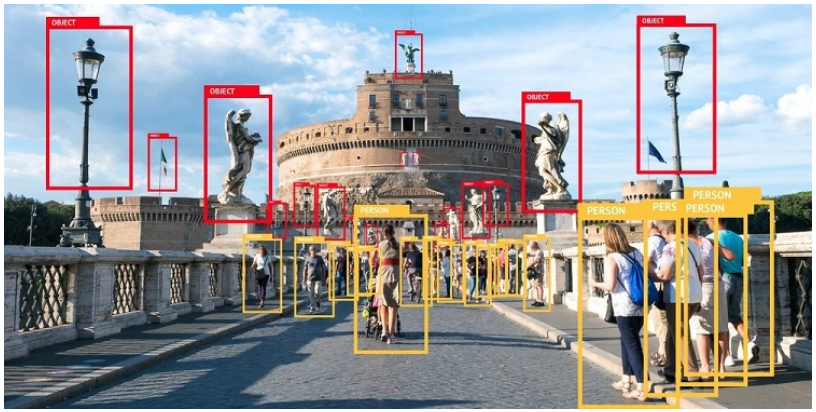


Figure: Object detection[1]

This includes **tons** of problems!

- image recognition / facial classification
- sentiment analysis
- neural machine translation
- automatic speech recognition
- etc.

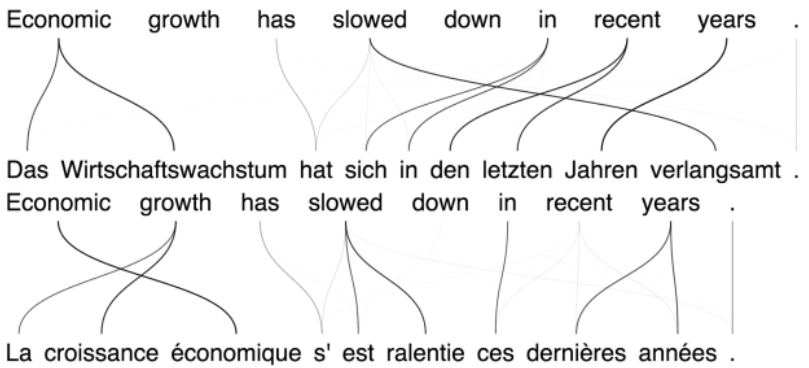


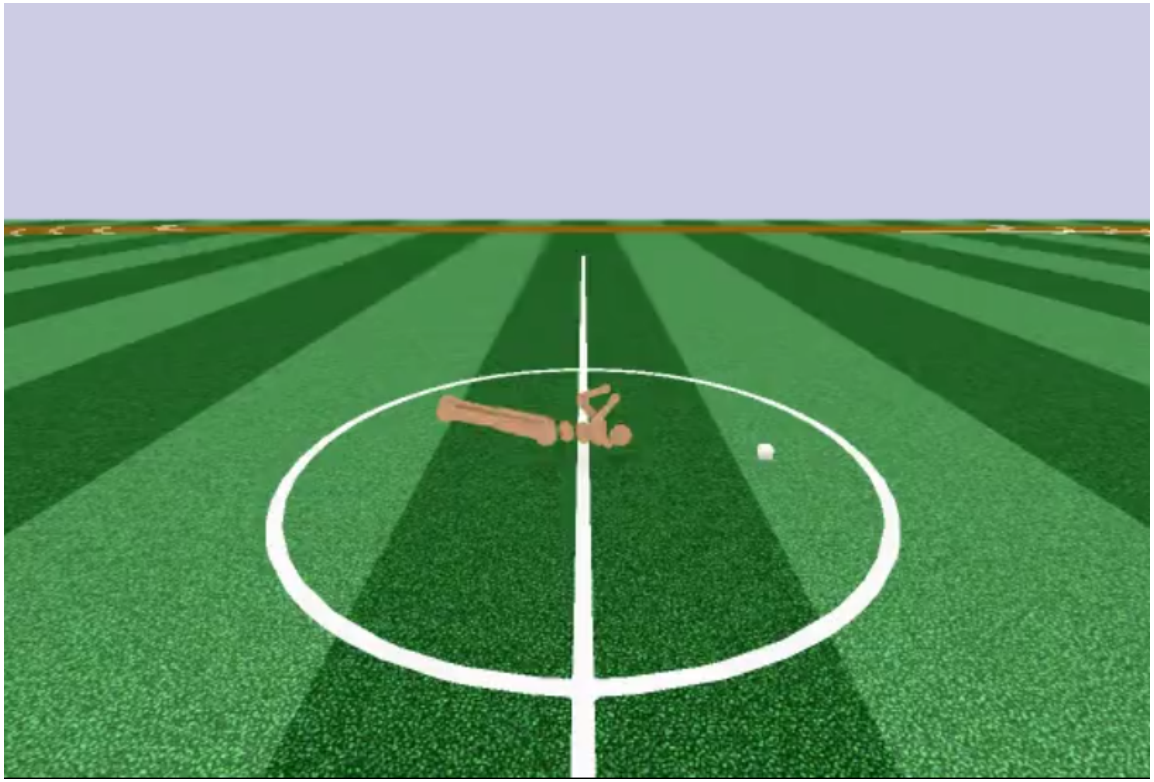
Figure: Machine translation[2]

[1] Intel Software, "A Closer Look at Object Detection, Recognition and Tracking"
[2] Nvidia Developer Blog, "Introduction to Neural Machine Translation with GPUs"

When would you want to use deep RL?

Deep RL can...

- Play video games from raw pixels
- Control robots in simulation and in the real world
- Play Go, Dota, and Starcraft at superhuman levels



Interlude: Recap of deep learning patterns

Deep Learning: High-level

Usual paradigm:

- we want to find a **model** that gives target **outputs** for particular **inputs**,
- we represent the model as a **function of parameters**,

$$f_\theta(x) = W_2\sigma\left(W_1 x + b_1\right) + b_2, \qquad \theta = \{W_1, W_2, b_1, b_2\}$$

- we can evaluate the model performance with a **differentiable loss function** that depends on **a set of data**,

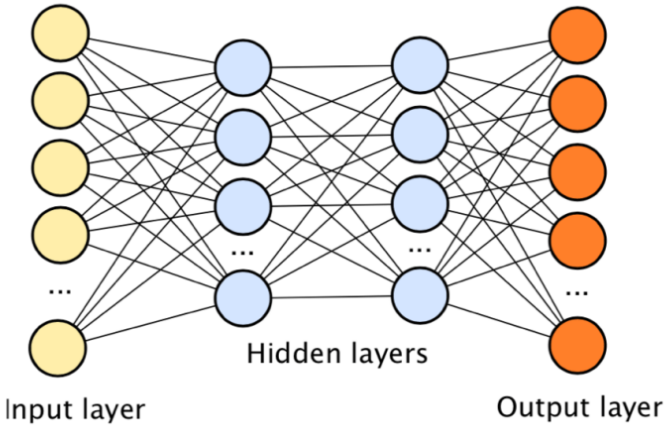
$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} L(x, y, f_\theta(x))$$

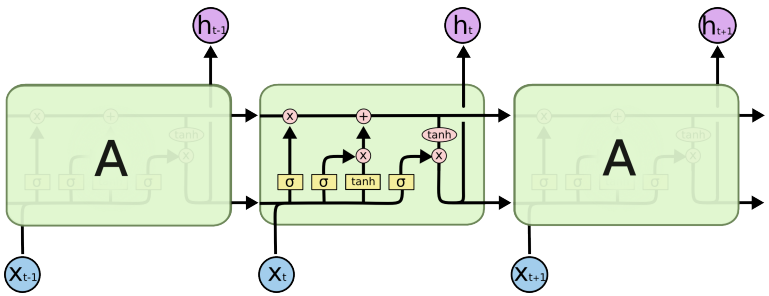
- and we find the optimal model by **gradient descent on the loss**.

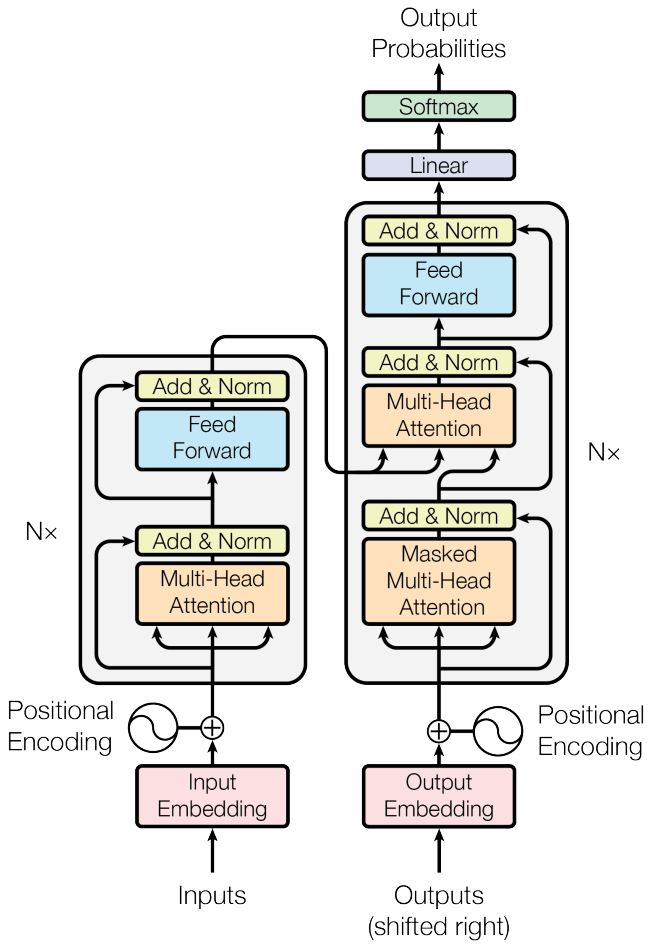
$$\theta \leftarrow \theta - \alpha\nabla_\theta\mathcal{L}(\theta)$$

Deep Learning: What makes it deep?

- "Deep" refers to using **function composition** as the building block for models
- Deep models have many **layers**: output of one layer is input to next







[1] MLP credit: Gary Marcus, LSTM credit: Chris Olah, Transformer credit: Vaswani et al 2017

Deep Learning: What else?

- Regularizers make optimization problems better-behaved:

$$\mathcal{L}(\theta) \rightarrow \mathcal{L}(\theta) + \lambda\Omega(\theta)$$

- Normalization makes optimization easier:

$$a \rightarrow \frac{g}{\sigma}(a - \mu) + b$$

- Adaptive optimizers (eg Adam) makes optimization faster:

$$m \leftarrow \beta_1 m + (1 - \beta_1)g$$
$$v \leftarrow \beta_2 v + (1 - \beta_2)g^2$$
$$\theta \leftarrow \theta - \alpha \frac{m}{\sqrt{v} + \epsilon}$$

- Reparameterization trick comes in handy sometimes:

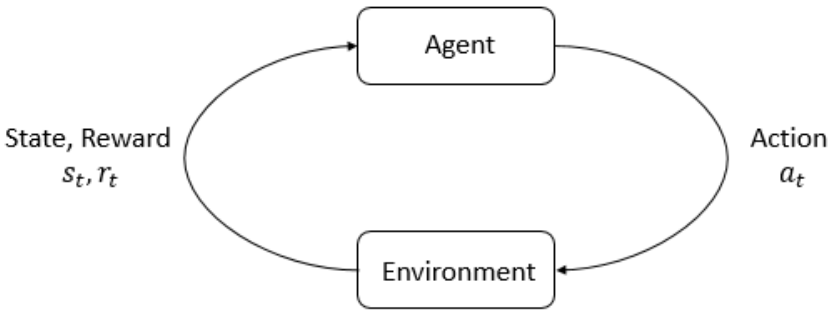
$$\underset{x \sim p_\theta}{\mathrm{E}}[F(x)] = \underset{z \sim \mathcal{N}}{\mathrm{E}}[F(\xi(\theta, z))]$$

- For links to detailed resources, see the Spinning Up essay!

1.2: How do we formulate RL problems?

What is RL?

- An *agent* interacts with an *environment*.




```
obs = env.reset()
done = False
while not(done):
  act = agent.get_action(obs)
  next_obs, reward, done, info = env.step(act)
  obs = next_obs
```

- The goal of the agent is to maximize cumulative reward (called *return*).
- The agent figures out how to attain its goal by trial and error.
- Reinforcement learning (RL) is a field of study for algorithms that do that.

Key Concepts in RL

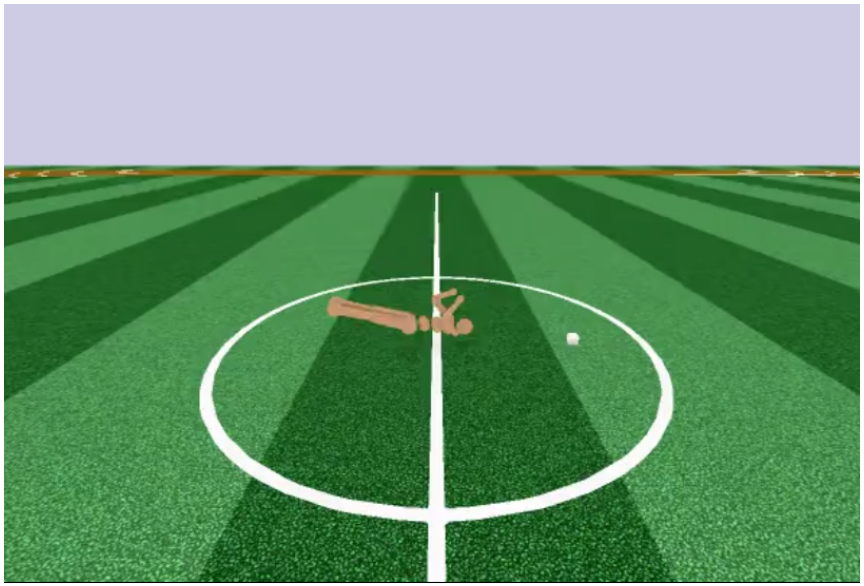
Before we can talk about algorithms, we have to talk about:

- Observation and action spaces
- Policies
- Trajectories
- Reward and return
- The RL optimization problem
- Value and Action-Value Functions

Observation and action spaces

- A **state** is a complete-information description of the world
- An **observation** is what the agent sees about the current state of the world
- An environment is **fully observed** if the agent observes the whole state, otherwise **partially observed**
- States, observations, and **actions** may be **continuous** or **discrete**

Partially observed, continuous observation, discrete action



Fully observed, continuous observation, continous action

Policies

A **policy** π is a rule for selecting actions. It can be either

- **stochastic**, which means that it gives a probability distribution over actions, and actions are selected randomly based on that distribution ($a_t \sim \pi(\cdot|s_t)$),
- or **deterministic**, which means that π directly maps to an action ($a_t = \pi(s_t)$).

Policies

A **policy** π is a rule for selecting actions. It can be either

- **stochastic**, which means that it gives a probability distribution over actions, and actions are selected randomly based on that distribution ($a_t \sim \pi(\cdot|s_t)$),
- or **deterministic**, which means that π directly maps to an action ($a_t = \pi(s_t)$).

Examples of policies:

- Stochastic policy over discrete actions:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
logits = tf.layers.dense(net, units=num_actions, activation=None)
actions = tf.squeeze(tf.multinomial(logits=logits,num_samples=1), axis=1)
```

- Deterministic policy for a vector-valued continuous action:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
actions = tf.layers.dense(net, units=act_dim, activation=None)
```

Trajectories

- A **trajectory** τ is a sequence of states and actions in an environment:

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

  - The initial state s_0 is sampled from a *start state distribution* μ:

$$s_0 \sim \mu(\cdot).$$

  - State transitions depend only on the most recent state and action. They could be deterministic:

$$s_{t+1} = f(s_t, a_t),$$

    or stochastic:

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

- A trajectory is sometimes also called an **episode** or **rollout**.

Reward and Return

The **reward** function of an environment measures how good state-action pairs are:

$$r_t = R(s_t, a_t).$$

- Example: if you want a robot to run forwards but use minimal energy, $R(s, a) = v - \alpha \|a\|_2^2$.

Reward and Return

The **reward** function of an environment measures how good state-action pairs are:

$$r_t = R(s_t, a_t).$$

- Example: if you want a robot to run forwards but use minimal energy, $R(s, a) = v - \alpha \|a\|_2^2$.

The **return** of a trajectory is a measure of cumulative reward along it. There are two main ways to compute return:

- Finite horizon undiscounted sum of rewards::

$$R(\tau) = \sum_{t=0}^{T} r_t$$

- Infinite horizon discounted sum of rewards:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

where $\gamma \in (0, 1)$. This makes rewards less valuable if they are further in the future. (Why would we ever want this? Think about cash: it's valuable to have it sooner rather than later!)

Reward-to-Go

A closely-related quantity is **reward-to-go**, which is "return starting from a state or timestep":

$$R_t = \sum_{t'=t}^{T} r_{t'}$$

or

$$R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}$$

In RL we want a policy which maximizes expected return. Thus the performance measure is

$$J(\pi) = \underset{\tau \sim \pi}{\mathrm{E}} \left[R(\tau) \right],$$

and the **optimal policy** π^* is:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

Note that by $\tau \sim \pi$, we mean

$$s_0 \sim \mu(\cdot), \qquad a_t \sim \pi(\cdot|s_t), \qquad s_{t+1} \sim P(\cdot|s_t, a_t).$$

Value Functions, Action-Value Functions, and Advantage Functions

Value and action-value functions tell you the expected return after a state or state-action pair.

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathrm{E}} [R(\tau) | s_0 = s] \qquad \text{Start in } s \text{ and then sample from } \pi$$

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{\mathrm{E}} [R(\tau) | s_0 = s, a_0 = a] \qquad \text{Start in } s, \text{ take action } a, \text{ then sample from } \pi$$

$$V^{*}(s) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}} [R(\tau) | s_0 = s] \qquad \text{Start in } s \text{ and then act optimally}$$

$$Q^{*}(s,a) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}} [R(\tau) | s_0 = s, a_0 = a] \qquad \text{Start in } s, \text{ take action } a, \text{ then act optimally}$$

Value and action-value functions are connected:

$$V^{\pi}(s) = \underset{a \sim \pi}{\mathrm{E}} [Q^{\pi}(s,a)]$$

The advantage function for a policy tells you how much better or worse one action is than average:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Bellman Equations

The value functions satisfy recursive **Bellman equations**:

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[r(s,a) + \gamma V^{\pi}(s') \right]$$

$$V^{*}(s) = \max_{a} \mathop{\mathrm{E}}_{s'\sim P} \left[r(s,a) + \gamma V^{*}(s') \right]$$

$$Q^{\pi}(s,a) = \mathop{\mathrm{E}}_{s'\sim P} \left[r(s,a) + \gamma \mathop{\mathrm{E}}_{a'\sim\pi} \left[Q^{\pi}(s',a') \right] \right]$$

$$Q^{*}(s,a) = \mathop{\mathrm{E}}_{s'\sim P} \left[r(s,a) + \gamma \max_{a'} Q^{*}(s',a') \right]$$

Basic idea: **The value of your starting point is the reward you expect to get from being there, plus the (discounted) value of wherever you land next.**

Put another way: the cash you make in the rest of your life is the cash you make today plus all the cash you make for the rest of your life starting tomorrow

The optimal Q function, Q^*, is especially important because it gives us a policy. In any state s, the optimal action is

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

We can measure how good a Q^*-approximator, Q_θ, is by measuring its **mean-squared Bellman error**:

$$\ell(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(s,a,s',r)\in\mathcal{D}} \left(Q_\theta(s, a) - \left(r + \gamma \max_{a'} Q_\theta(s', a') \right) \right)^2.$$

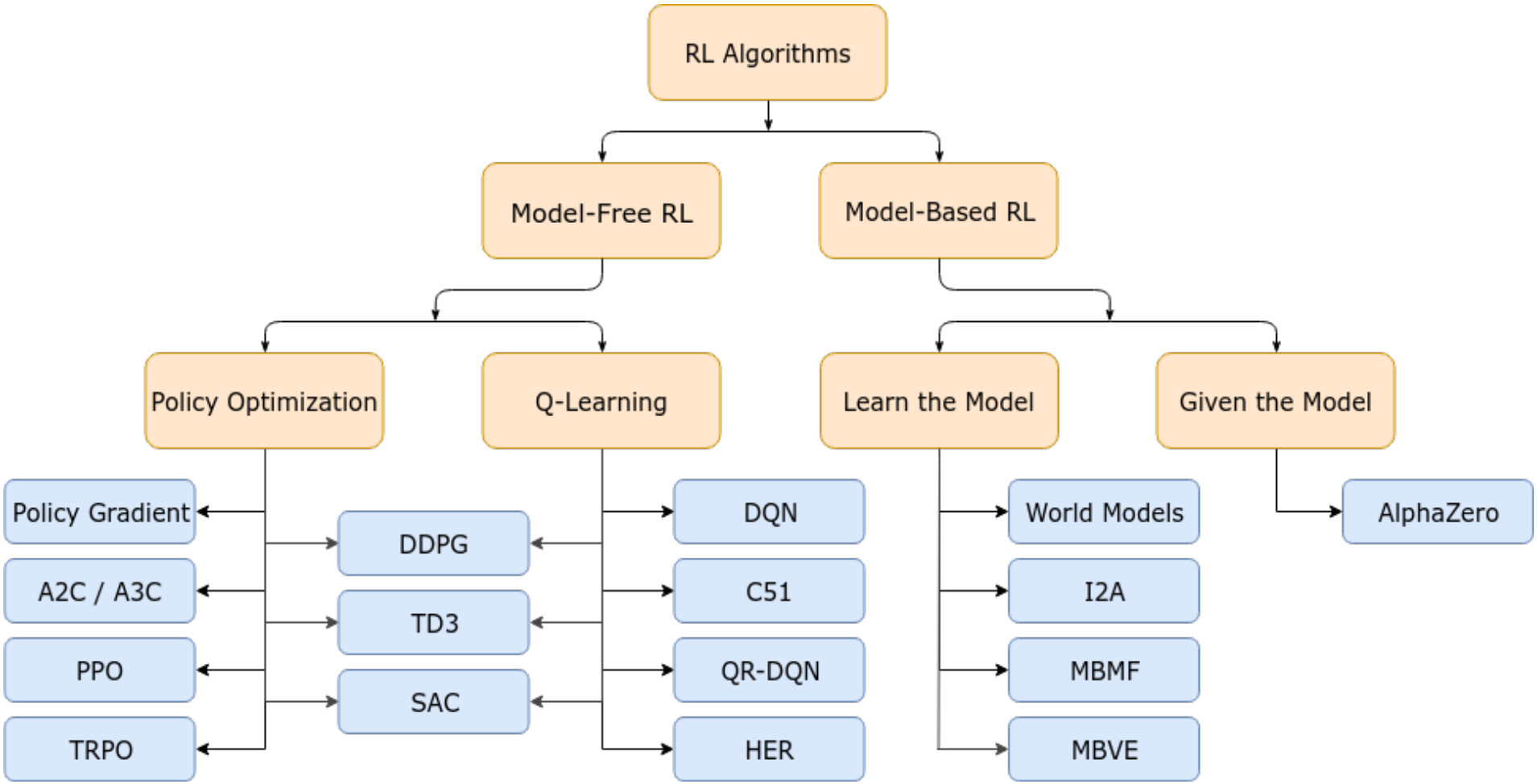
This (roughly) says how well it satisfies the Bellman equation

$$Q^*(s, a) = \mathop{\mathrm{E}}_{s'\sim P} \left[r(s, a) + \gamma \max_{a'} Q^*(s', a') \right]$$

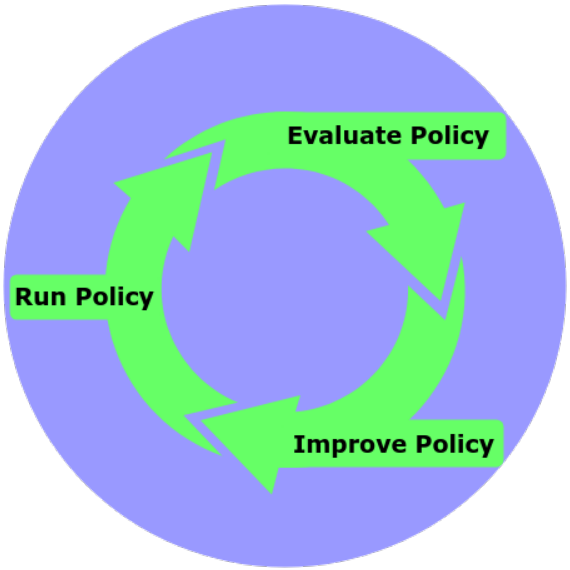
1.3: What kinds of RL algorithms are there?

Deep RL Algorithms

There are many different kinds of RL algorithms! This is a non-exhaustive taxonomy (with specific algorithms in blue):



Alphabet soup aside—what are they doing?



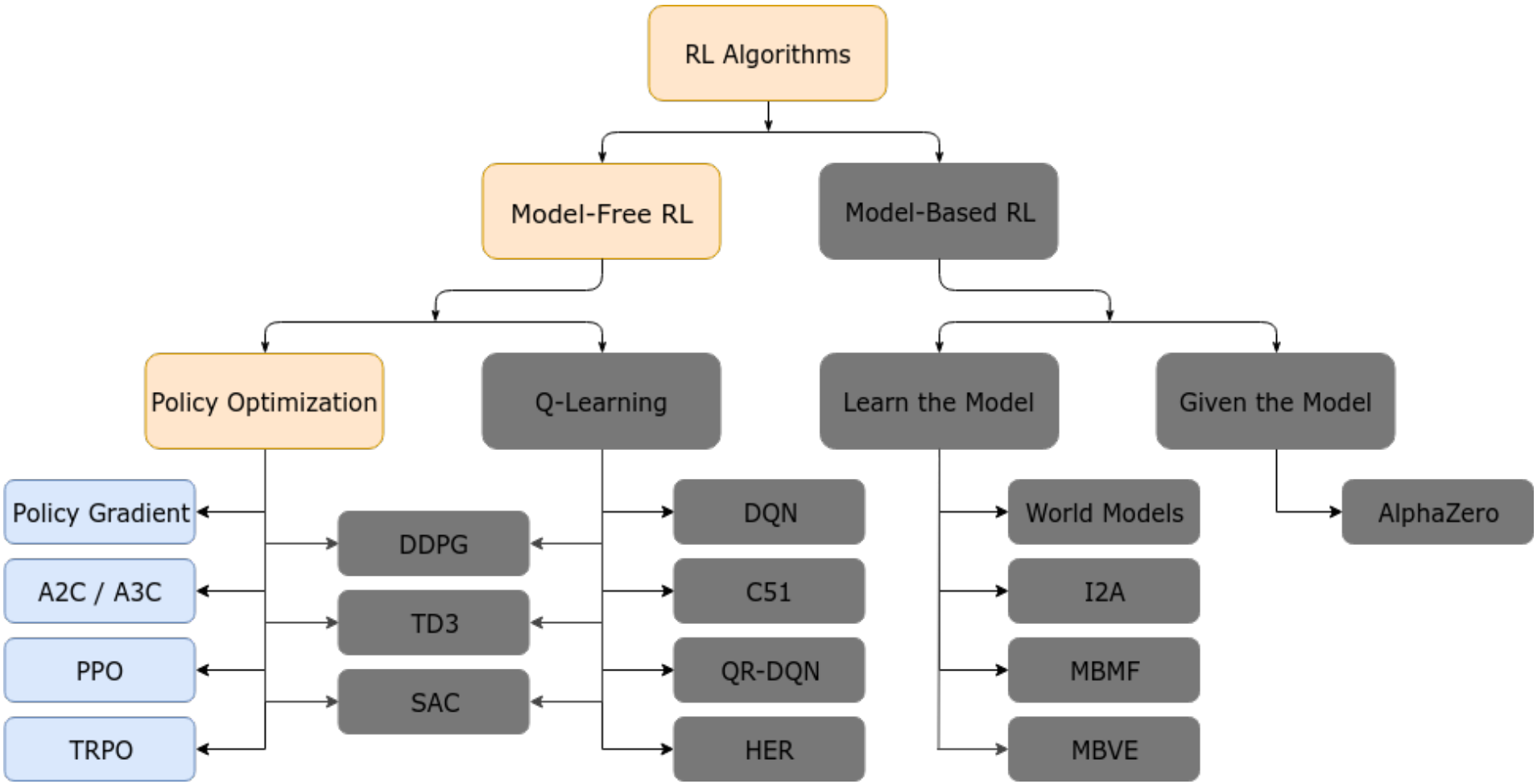
Key steps in all RL algorithms:

- Run policy: actually act in env
- Evaluate policy: estimate V^{π} or Q^{*}
- Improve policy: do something which lets you pick better actions

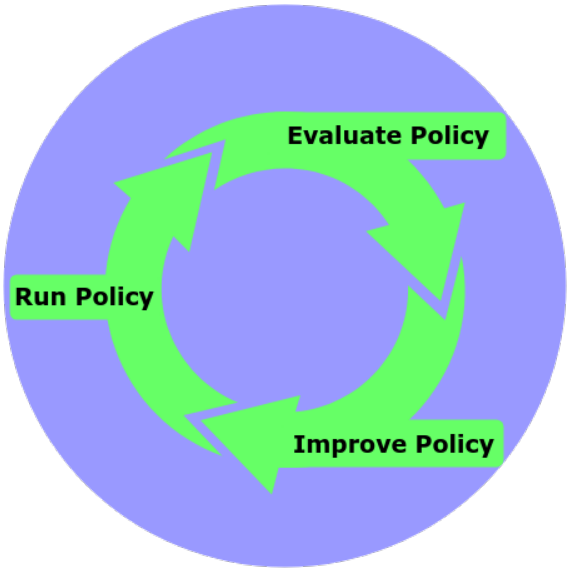
Major design decisions:

- Use a model of the env or not (can slot into any of those three steps)
- Optimize stochastic policy directly or learn Q^{*} as main controller

Model-Free RL: Policy Optimization



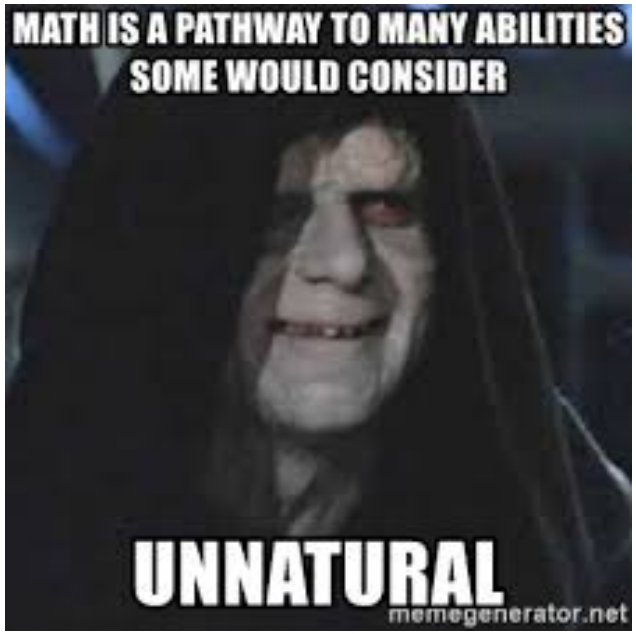
Model-Free RL: Policy Optimization



Policy Optimization

- Run policy: Collect trajectories $\tau \sim \pi_\theta$
- Evaluate policy: Estimate V^{π_θ}, A^{π_θ} using current trajectories (**on-policy**)
- Improve policy: Increase likelihood of actions with high advantage

Warning: Math



Wait, but why?

Deep RL is not mature enough to be a black box yet: you need to know some of this stuff to work with it successfully.

Mathematical Foundations of Policy Optimization

In policy optimization, we train a **stochastic policy** π_θ, usually in an **on-policy way**, to **maximize performance** $J(\pi_\theta)$.

What we'll cover:

- Direct gradient ascent on $J(\pi_\theta)$ (VPG)
- Various formulations of $\nabla_\theta J(\pi_\theta)$
- Some material about natural policy gradients

The Policy Gradient

Goal: derive an expression for $\nabla_\theta J(\pi_\theta)$ which we can compute with a sample estimate, as the basis for a direct gradient ascent algorithm

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

The Policy Gradient

Goal: derive an expression for $\nabla_\theta J(\pi_\theta)$ which we can compute with a sample estimate, as the basis for a direct gradient ascent algorithm

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

Well what happens if we just...

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \underset{\tau \sim \pi_\theta}{\mathrm{E}} [R(\tau)]$$

Problem: parameters are in distribution!

The Policy Gradient: First Steps

Solution: expand expectation into integral, use log-derivative trick

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathrm{E}}[R(\tau)] \\
&= \nabla_\theta \int d\tau P(\tau|\pi_\theta) R(\tau) \\
&= \int d\tau \nabla_\theta P(\tau|\pi_\theta) R(\tau) \\
&= \int d\tau P(\tau|\pi_\theta) \nabla_\theta \log P(\tau|\pi_\theta) R(\tau) \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}[\nabla_\theta \log P(\tau|\pi_\theta) R(\tau)]
\end{aligned}$$

But are we done yet? No! Still need to compute $\nabla_\theta \log P(\tau|\pi_\theta)$

The Policy Gradient: Gradient of Trajectory Distribution

What is $P(\tau|\pi_\theta)$?

$$P(\tau|\pi_\theta) = \mu(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Thus:

$$\begin{aligned}
\nabla_\theta \log P(\tau|\pi_\theta) &= \nabla_\theta \log \left(\mu(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)\right) \\
&= \nabla_\theta \left(\log \mu(s_0) + \sum_{t=0}^{T} (\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t))\right) \\
&= \nabla_\theta \log \mu(s_0) + \sum_{t=0}^{T} (\nabla_\theta \log P(s_{t+1}|s_t, a_t) + \nabla_\theta \log \pi_\theta(a_t|s_t)) \\
&= \cancel{\nabla_\theta \log \mu(s_0)} + \sum_{t=0}^{T} \left(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t)\right)
\end{aligned}$$

$$\therefore \nabla_\theta \log P(\tau|\pi_\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

The Policy Gradient: So Far

Putting it all together so far:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

So we could estimate with:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

But not good enough! Variance will be high.

The Policy Gradient: Variance Reduction

Insight: future actions and past rewards should be uncorrelated. That is:

$$\text{for } t > t', \quad \mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) r_{t'}\right] = 0$$

Thus:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=0}^{T} r_{t'}\right] \\
&= \sum_{t=0}^{T}\sum_{t'=0}^{T} \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) r_{t'}\right] \\
&= \sum_{t=0}^{T}\sum_{t'=t}^{T} \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) r_{t'}\right] \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'}\right]
\end{aligned}$$

The Policy Gradient: Alternate Forms

What we have currently: "Reward-to-Go" policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T} r_{t'}\right]$$

Observe: expectation can be broken up, letting us transform "Reward-to-Go" into Q^π:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T} r_{t'}\right] \\
&= \sum_{t=0}^{T}\mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T} r_{t'}\right] \\
&= \sum_{t=0}^{T}\mathop{\mathrm{E}}_{\tau_{0:t}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\mathop{\mathrm{E}}_{\tau_{(t+1):T}\sim\pi_\theta}\left[\sum_{t'=t}^{T} r_{t'}\right]\right] \\
&= \sum_{t=0}^{T}\mathop{\mathrm{E}}_{\tau_{0:t}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t,a_t)\right] \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t,a_t)\right]
\end{aligned}$$

The Policy Gradient: Baselines

What is a **baseline**? A function $b(s_t)$ with

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(Q^{\pi_\theta}(s_t, a_t) - b(s_t) \right) \right]$$

Claim: this works for any b! Proof:

$$\begin{aligned}
\mathop{\mathrm{E}}_{a_t \sim \pi_\theta(\cdot|s_t)} [\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] &= \mathop{\mathrm{E}}_{a_t \sim \pi_\theta(\cdot|s_t)} [\nabla_\theta \log \pi_\theta(a_t|s_t)] \, b(s_t) \\
&= \left(\int da \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \right) b(s_t) \\
&= \left(\int da \nabla_\theta \pi_\theta(a_t|s_t) \right) b(s_t) \\
&= \left(\nabla_\theta \int da \pi_\theta(a_t|s_t) \right) b(s_t) \\
&= \cancel{(\nabla_\theta 1)} b(s_t) \\
&= 0
\end{aligned}$$

The Policy Gradient: Advantage Form

If we choose $b = V^{\pi}$, we get the **advantage form** of the policy gradient:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)\right)\right] \\ &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t,a_t)\right]\end{aligned}$$

Why do we want this? Better signal in sample estimate: removes "stuff that would have happened anyway" from Q^{π}

The Policy Gradient: Review

What we've shown so far:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T} r_{t'}\right] \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)Q^{\pi_\theta}(s_t,a_t)\right] \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)A^{\pi_\theta}(s_t,a_t)\right]
\end{aligned}$$

Which form do we use? Almost always the last one.

The Policy Gradient: Key Concepts

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]$$

- Pushes up the probabilities of "good" actions and pushes down probabilities of "bad" actions
- To estimate, **must sample on-policy**

The Policy Gradient: Policy Evaluation

The policy gradient expression gives us the **policy improvement** step, but we need to compute advantages: so how do we do **policy evaluation**?

Answer: first learn V^π approximator by regression:

$$\min_\phi \frac{1}{N} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \left(V_\phi(s_t) - \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \right)^2$$

Then use it to estimate A^π, usually with **generalized advantage estimation**

The Policy Gradient: Policy Evaluation

The policy gradient expression gives us the **policy improvement** step, but we need to compute advantages: so how do we do **policy evaluation**?

Answer: first learn V^π approximator by regression:

$$\min_\phi \frac{1}{N} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \left(V_\phi(s_t) - \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \right)^2$$

Then use it to estimate A^π, usually with **generalized advantage estimation**

Wait, did you pull a fast one? Discount factor is in now? **Policy gradient implementations usually use discounted value functions, even though they (ostensibly) optimize the undiscounted objective.**

The Policy Gradient: Generalized Advantage Estimation

N-Step Advantage Estimates:

$$\hat{A}_t^{\pi(n)} = \underbrace{\sum_{t'=t}^{n} \gamma^{t'-t} r_{t'} + \gamma^{n+1} V_\phi(s_{t+n+1})}_{\approx Q^\pi} - \underbrace{V_\phi(s_t)}_{\approx V^\pi}$$

Choice of n is a bias-variance trade-off:

$n = 0$	High bias, low variance
$n = \infty$	Low bias, high variance

Generalized Advantage Estimates[2]:

$$\begin{aligned}\hat{A}_t^{\pi,\lambda} &= (1-\lambda)\sum_{n=0}^{\infty} \lambda^n \hat{A}_t^{\pi(n)} \\ &= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \left(r_{t'} + \gamma V_\phi(s_{t'+1}) - V_\phi(s_{t'})\right)\end{aligned}$$

(Those dreaded words: proof left as exercise for the reader)

[2] Schulman et al, 2017: "High-Dimensional Continuous Control Using Generalized Advantage Estimation"

Vanilla Policy Gradient: Pseudocode

Algorithm 1 Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
9: **end for**

Trust Region Policy Optimization: Pseudocode

Algorithm 2 Trust Region Policy Optimization

Input: initial policy parameters θ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Form sample estimates for

- policy gradient $\hat{g}_k$ (using advantage estimates)
- and KL-divergence Hessian-vector product function $f(v) = \hat{H}_k v$

Use CG with n_{cg} iterations to obtain $x_k \approx \hat{H}_k^{-1} \hat{g}_k$

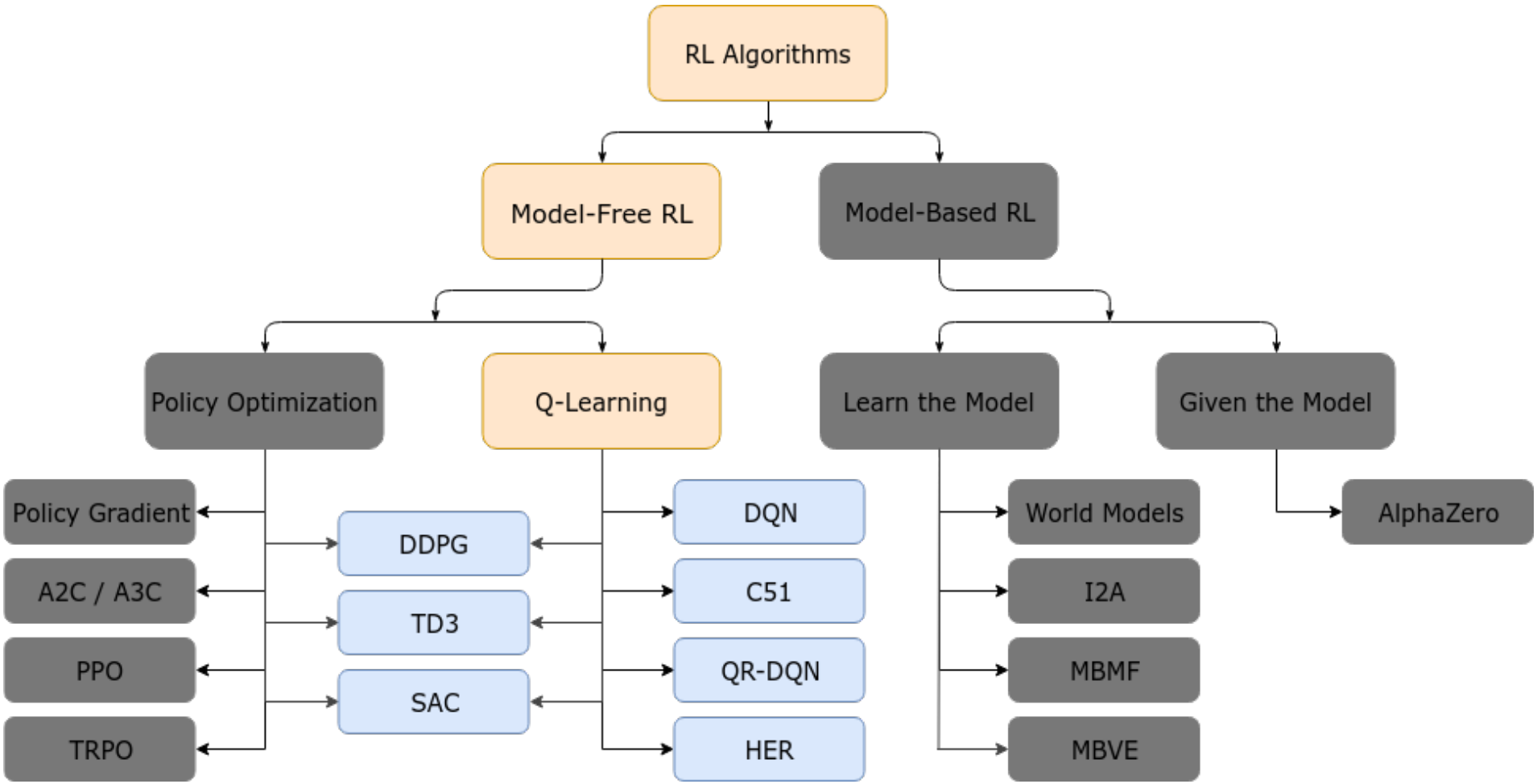
Estimate proposed step $\Delta_k \approx \sqrt{\frac{2\delta}{x_k^T \hat{H}_k x_k}} x_k$

Perform backtracking line search with exponential decay to obtain final update

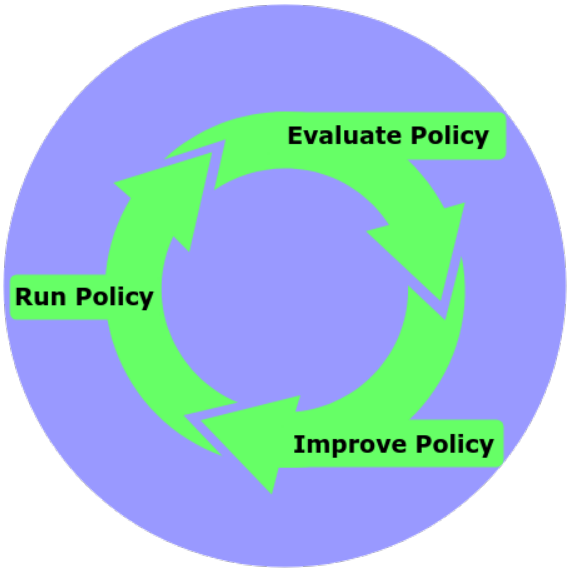
$$\theta_{k+1} = \theta_k + \alpha^j \Delta_k$$

end for

Model-Free RL: Q-Learning



Model-Free RL: Q-Learning



Q-Learning

- Run policy: Step in env with action from Q_θ, store to replay buffer
- Evaluate policy: Update Q_θ to minimize Bellman error, using all previous data (**off-policy**)
- Improve policy: $a^* = \arg\max_a Q_\theta(s, a)$

Q-Learning Updates by Bootstrapping

- Collect experience in the environment using a policy which trades off between acting randomly and acting according to current Q_θ
- Interleave data collection with updates to Q_θ to minimize Bellman error by bootstrapping:

$$\min_\theta \sum_{(s,a,s',r)\in\mathcal{D}} \left(Q_\theta(s,a) - y(s',r)\right)^2$$

where

$$y(s',r) = r + \gamma \max_{a'} Q_\theta(s',a'),$$

and gradients don't propagate through y.

Getting Q-Learning to Work (DQN)

Experience replay:

- Data distribution changes over time: as your Q function gets better and you *exploit* this, you visit different (s, a, s', r) transitions than you did earlier
- Stabilize learning by keeping old transitions in a replay buffer, and taking minibatch gradient descent on mix of old and new transitions

Getting Q-Learning to Work (DQN)

Experience replay:

- Data distribution changes over time: as your Q function gets better and you *exploit* this, you visit different (s, a, s', r) transitions than you did earlier
- Stabilize learning by keeping old transitions in a replay buffer, and taking minibatch gradient descent on mix of old and new transitions

Target networks:

- Bootstrapping with function approximators is unstable!
- It's *like* regression, but it's not: targets depend on parameters θ—so an update to Q changes the target!

$$y(s', r) = r + \gamma \max_{a'} Q_\theta(s', a').$$

- Stabilize it by *holding the target fixed* for a while: keep a separate target network, $Q_{\theta_{targ}}$, and every k steps update $\theta_{targ} \leftarrow \theta$
- If unstable, why do it? Because it works well! (Plus theory, later)

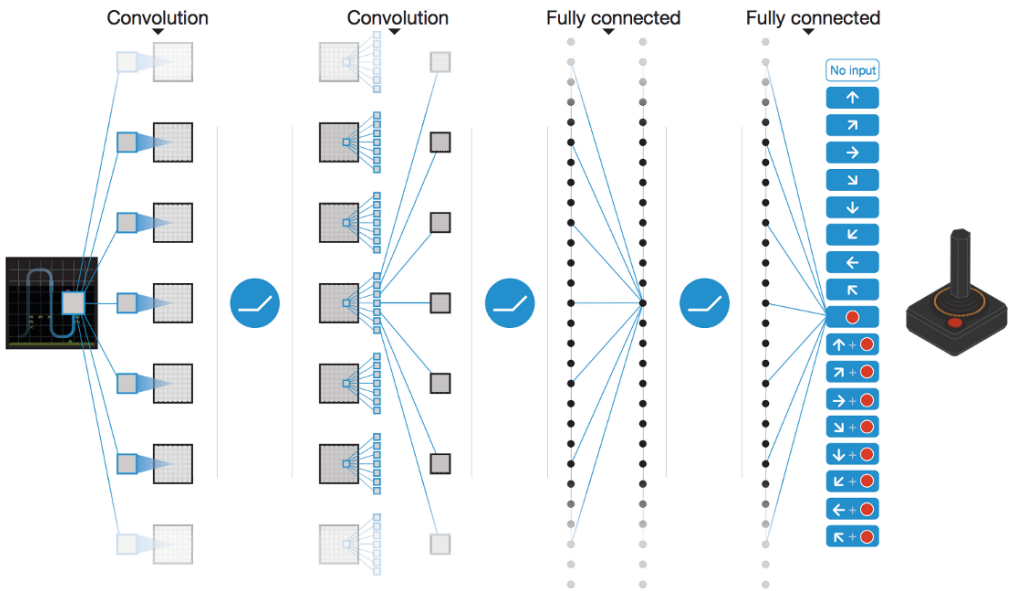
In DQN, Action Space Matters

What we've described so far requires the computation of

$$\max_a Q_\theta(s, a),$$

both for selecting actions and calculating update targets.

Problem: for continuous action spaces, this is nontrivially hard! DQN therefore only applies for **discrete actions**, because we can output one Q-value per action from the network.



[1]Image credit: Mnih et al, 2015

Algorithm 3 Deep Q-Learning

Randomly generate Q-function parameters θ
Set target Q-network parameters $\theta_{targ} \leftarrow \theta$
Make empty replay buffer $\mathcal{D}$
Receive observation s_0 from environment
for $t = 0, 1, 2, \ldots$ **do**
  With probability ϵ, select random action a_t; otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
  Step environment to get s_{t+1}, r_t and end-of-episode signal d_t
  Linearly decay ϵ until it reaches final value ϵ_f
  Store $(s_t, a_t, r_t, s_{t+1}, d_t) \rightarrow \mathcal{D}$
  Sample mini-batch of transitions $B = \{(s, a, r, s', d)_i\}$ from $\mathcal{D}$
  For each transition in B, compute

$$y = \begin{cases} r & \text{transition is terminal } (d = \text{True}) \\ r + \gamma \max_{a'} Q_{\theta_{targ}}(s', a') & \text{otherwise} \end{cases}$$

  Update Q by gradient descent on regression loss:

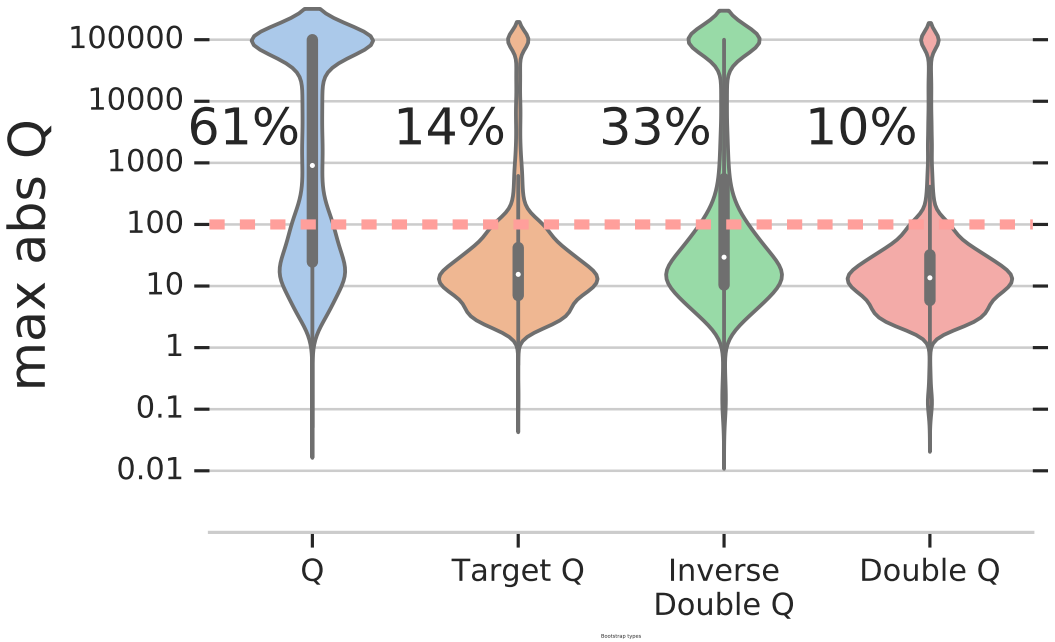
$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{(s,a,y) \in B} (Q_\theta(s, a) - y)^2$$

  if $t \% t_{update} = 0$ **then**
    Set $\theta_{targ} \leftarrow \theta$
  end if
end for

Caveat Emptor: DQN Can Easily Break

Even with many additional tricks, DQN-style algorithms still occasionally diverge (Q-values go to unrealistically large or negative values and control performance tanks). Why?

To answer that question, we have to dive into some math.



[1]van Hesselt et al, 2018: "Deep Reinforcement Learning and the Deadly Triad"

Operator view of Bellman equation: define **optimal Bellman operator** $\mathcal{T}^* : \mathcal{Q} \to \mathcal{Q}$ to have values

$$[\mathcal{T}^* Q](s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right].$$

Optimal Q-function is fixed-point of $\mathcal{T}^*$:

$$Q^* = \mathcal{T}^* Q^*$$

$\mathcal{T}^*$ has a special property: it is a **contraction** on $\mathcal{Q}$ in the sup norm (max absolute value over all states and actions).

Foundations of Q-Learning: Contraction Maps

A function $f : X \to X$ on a normed vector space $(X, \|\cdot\|)$ is said to be a **contraction** with modulus β if

$$\forall x, y \in X, \quad \|f(x) - f(y)\| \leq \beta \|x - y\|.$$

Why do we care about contractions? Because they have **unique fixed-points** and the fixed points can be obtained by repeated application of the operator!

To see this, consider the sequence $\{x_0, x_1, ...\}$ with $x_{i+1} = f(x_i)$. Observe that

$$\begin{aligned} \|x_{i+1} - x_i\| &= \|f(x_i) - f(x_{i-1})\| \\ &\leq \beta \|x_i - x_{i-1}\| \\ &\leq \beta^i \|x_1 - x_0\| \end{aligned}$$

The distance between iterates shrinks by a factor of β each iteration!

Foundations of Q-Learning: Value Iteration

$\mathcal{T}^*$ is a contraction on $\mathcal{Q}$ with modulus γ, so we should be able to obtain Q^* in the limit of the sequence:

$$Q_{i+1} = \mathcal{T}^* Q_i.$$

This approach is **value iteration**, a classic RL algorithm.

Problems: When the state/action space is large and we use function approximation,

- We can't compute all of $\mathcal{T}^* Q_i$
- $\mathcal{T}^* Q_i$ may not be representable in our function class $\mathcal{Q}_\theta$

Foundations of Q-Learning: Approximating Value Iteration

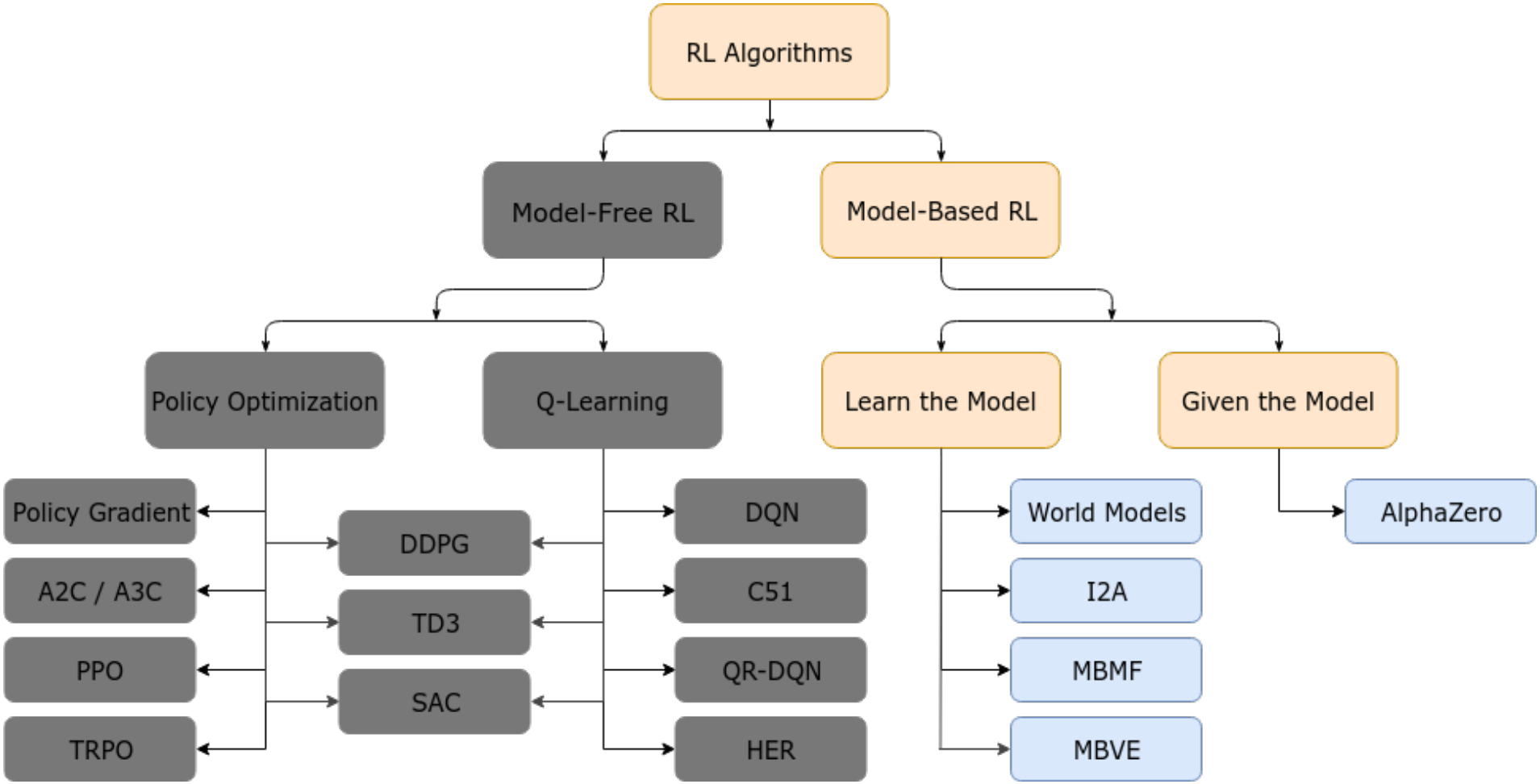
So what we do is push Q_θ towards $\mathcal{T}^* Q_\theta$ for the state-action pairs where we can estimate it:

$$\theta \leftarrow \theta + \alpha \underbrace{\left(\hat{\mathcal{T}}^* Q_\theta(s,a) - Q_\theta(s,a)\right)}_{r+\gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)} \nabla_\theta Q_\theta(s,a)$$

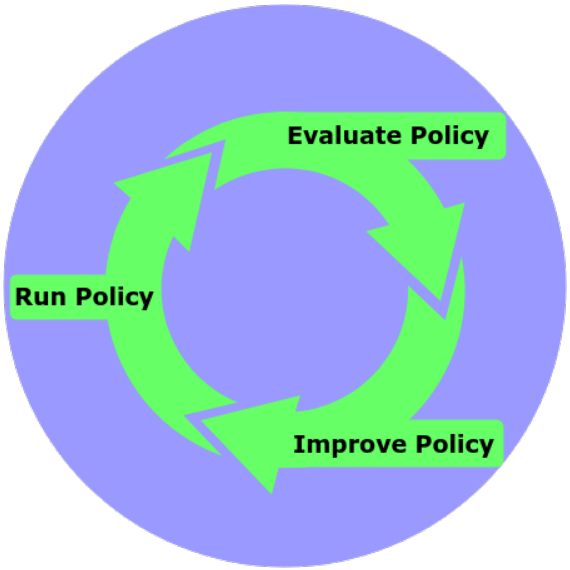
which is deep Q-learning. This may or may not preserve the contraction: when it doesn't, divergence happens!

Important to realize: **this means deep Q-learning isn't minimizing a real loss function**.

Model-Based RL



Model-Based RL



Model-Based RL

- Can make use of an environment model (a simulator) anywhere in the loop to improve!
- Downside: model not usually available
- Models can be very hard to learn, so model-based is not yet as reliable as model-free