

# *Deep Learning*

*A course about theory & practice*



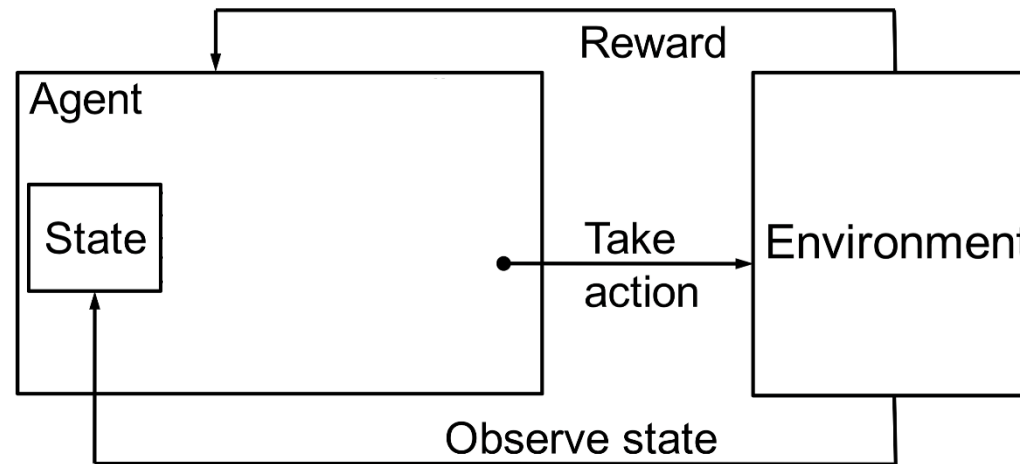
## **Deep Reinforcement Learning**

Marco Piastra

# *Basics (Intuition)*

# Deep Reinforcement Learning (DRL)

- Reinforcement Learning



# Q-Learning

## ▪ Q-Learning Algorithm

Initialize  $\hat{Q}(s, a)$  at random, put the agent in a random state  $s$

Repeat:

- 1) Select the action  $\operatorname{argmax}_a \hat{Q}(s, a)$  with probability  $(1 - \epsilon)$  otherwise, select  $a$  at random
- 2) The agent is now in state  $s'$  and has received the reward  $r$
- 3) Update  $\hat{Q}(s, a)$  by

$$\Delta \hat{Q}(s, a) = \alpha [r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a)]$$

### **Critical aspects:**

*Tabular representation of  $\hat{Q}(s, a)$*

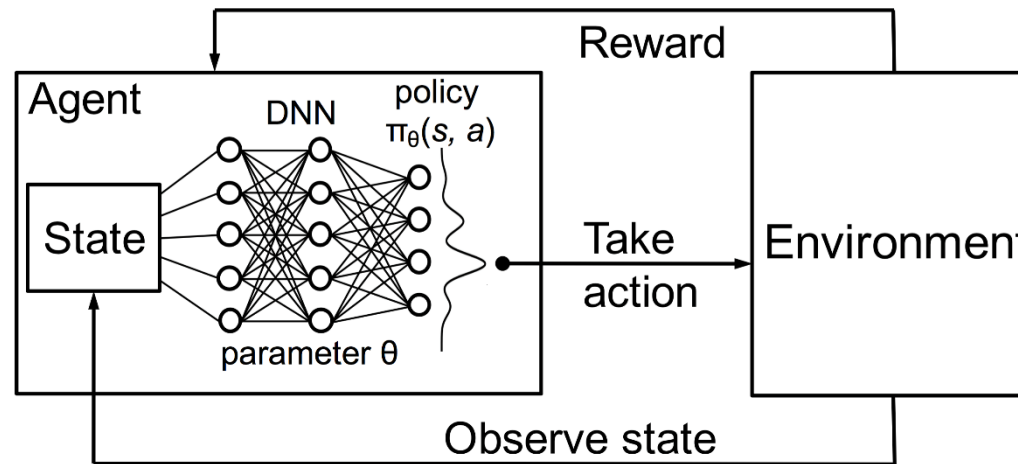
*Suitable for discrete action spaces only*

*Very slow learner: each pair  $(s, a)$  is a distinct entry, to be learned directly or by using a linear function approximator*

# Deep Reinforcement Learning (DRL)

## ■ Deep Reinforcement Learning

Using a deep neural network for the  $\hat{Q}(s, a)$  approximator



*The optimal policy is learnt incrementally by using a deep neural network*

# Deep Reinforcement Learning

## ▪ Deep Q-Learning Algorithm (intuitive)

Initialize  $\hat{Q}(s, a; \theta)$ , put the agent in a random state  $s$

Repeat:

- 1) Select the action  $\operatorname{argmax}_a \hat{Q}(s, a; \theta)$  with probability  $(1 - \varepsilon)$  otherwise, select  $a$  at random
- 2) The agent is now in state  $s'$  and has received the reward  $r$
- 3) Define:

$$y = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \operatorname{argmax}_{a'} \hat{Q}(s', a'; \theta) & \text{otherwise} \end{cases}$$

- 4) Perform gradient descent over  $(y - \hat{Q}(s, a; \theta))^2$

### **Fundamental Idea:**

Use a deep neural network to learn the approximator  $\hat{Q}(s, a; \theta)$  from the examples collected while **exploring – exploiting**

Replace the update step with DNN training

# Deep Reinforcement Learning

## ▪ Deep Q-Learning Algorithm (intuitive)

Initialize  $\hat{Q}(s, a; \theta)$ , put the agent in a random state  $s$

Repeat:

- 1) Select the action  $\operatorname{argmax}_a \hat{Q}(s, a; \theta)$  with probability  $(1 - \varepsilon)$  otherwise, select  $a$  at random
- 2) The agent is now in state  $s'$  and has received the reward  $r$
- 3) Define:

$$y = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \operatorname{argmax}_{a'} \hat{Q}(s', a'; \theta) & \text{otherwise} \end{cases}$$

- 4) Perform gradient descent over  $(y - \hat{Q}(s, a; \theta))^2$

### **CAREFUL**

Training  $\hat{Q}(s, a; \theta)$  may be non-trivial...

# *DQN Algorithm*

# Deep Q-Learning

## ■ Playing Atari with Deep Reinforcement Learning

[2013, V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, <http://arxiv.org/abs/1312.5602>, see also <http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html>]

A software system only

Runs on virtually any Linux-based system, it contains optional provisions for GPU

*It's open source*

<https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner>

Sophisticated machine-learning techniques

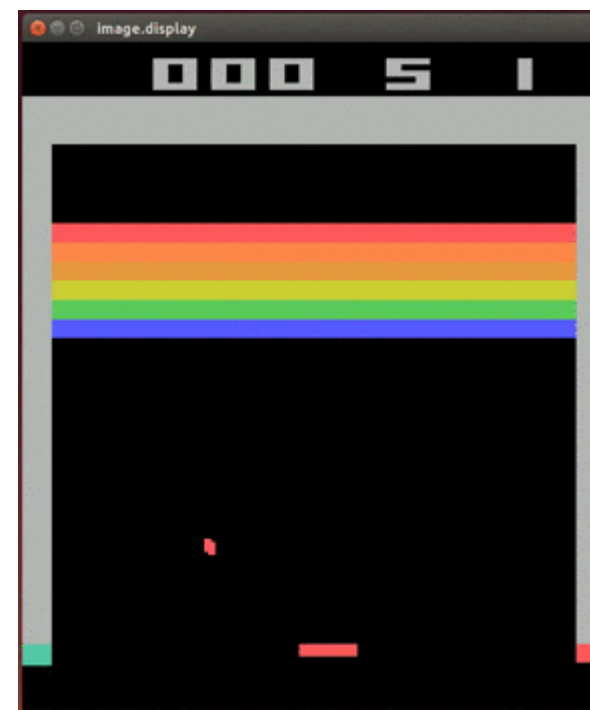
Uses deep reinforcement learning  
in combination with convolutional neural networks (CNN)

Same configuration, multiple games

Same configuration applied to arcade games  
It learned to play 7 (2013) or 49 (2015) different games

*It is autonomous*

*It learns by itself*, it receives no human expertise as input  
In many cases, it outperforms human players



(from GitHub)

# Deep Q-Learning

- **DQN Algorithm** [<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>]

---

## Algorithm 1 Deep Q-learning with Experience Replay

---

Initialize replay memory  $\mathcal{D}$  to capacity  $N$

Initialize action-value function  $Q$  with random weights

**for** episode = 1,  $M$  **do**

    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  select a random action  $a_t$

        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$

        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3

**end for**

**end for**

*states are images, which require some preprocessing*

*(see next slide)*

# Deep Q-Learning

## Loss function

$$L(\theta^{(t)}) = \mathbb{E}_{s,a} \left[ \left( y^{(t)} - Q(s, a; \theta^{(t)}) \right)^2 \right]$$
$$= \mathbb{E}_{s,a,s'} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta^{(t-1)}) - Q(s, a; \theta^{(t)}) \right)^2 \right]$$

$\theta^{(t)}$  Online network, updated at every single step  
 $\theta^{(t-1)}$  Offline network, updated only every  $C$  steps

## Gradient

$$\nabla_{\theta} L(\theta^{(t)}) = \mathbb{E}_{s,a,s'} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta^{(t-1)}) - Q(s, a; \theta^{(t)}) \right) \nabla_{\theta} Q(s, a; \theta^{(t)}) \right]$$

- It is computed at each iteration (see algorithm)
- It compares the last (actual) step (variable  $y$  in the algorithm) ... with the value given by  $Q$
- The expectation is approximated by the average on the minibatch

# *Reinforcement Learning Reformulation*

# Reinforcement Learning Reformulation

## Trajectory

$$\tau := \langle (s_t, a_t) \rangle_{t=0}^T$$

i.e., a sequence of *states* and *actions*.

It can be either finite or infinite, depending on  $T$

## Reward

Reward function, *general formulation*:

$$r_t := r(s_t, a_t, s_{t+1})$$

Depending on the application, it could be *simplified* as:

$$r_t := r(s_t, a_t), \quad r_t := r(s_t)$$

## Return

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

*we will use these forms from now on, for conciseness*

It is discounted when  $\gamma < 1$  or undiscounted, when  $\gamma = 1$  (*when trajectories are finite*)

# Reinforcement Learning Reformulation

## Policy

$$\pi(a \mid s) := P(A_t = a \mid S_t = s)$$

A policy may be *deterministic*:

$$\pi(s) := a$$

## Value Function (of a policy)

$$V^\pi(s) := \mathbb{E}[r(S_t) + \gamma r(S_{t+1}) + \gamma^2 r(S_{t+2}) + \dots \mid \pi, S_t = s]$$

## Action-Value function (of a policy)

$$\begin{aligned} Q^\pi(s, a) &:= \sum_{S_{t+1}} P(S_{t+1} \mid s, a) \cdot V^\pi(S_{t+1}) \\ &= \mathbb{E}[r(S_t) + \gamma r(S_{t+1}) + \gamma^2 r(S_{t+2}) + \dots \mid \pi, S_t = s, a_t = a] \end{aligned}$$

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

# Reinforcement Learning Reformulation

**Value Function** (of a policy)

$$V^\pi(s) := \mathbb{E}_{\tau \sim \pi} [R(\tau) \mid s_0 = s]$$

**Action-Value function** (of a policy)

$$Q^\pi(s, a) := \mathbb{E}_{\tau \sim \pi} [R(\tau) \mid s_0 = s, a_0 = a]$$

**Optimal Value Function**

$$V^*(s) := \max_{\pi} \mathbb{E}_{\tau \sim \pi} [R(\tau) \mid s_0 = s]$$

**Optimal Action-Value Function**

$$Q^*(s, a) := \max_{\pi} \mathbb{E}_{\tau \sim \pi} [R(\tau) \mid s_0 = s, a_0 = a]$$

# Reinforcement Learning Reformulation

- **Connecting Value and Action-Value Functions**

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

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

- **Optimal Policy (*Deterministic*)**

$$a^*(s) = \operatorname{argmax}_a [Q^*(s, a)]$$

- **Advantage Function**

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

It tells how advantageous (or disadvantageous) is a specific *action* compared to what is prescribed by the *policy*

# *Policy Gradient*

# Reinforcement Learning Reformulation (2)

## Probability of a trajectory

$$P(\tau|\pi) := P(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \pi(a_t|s_t)$$

*probability of initial states*      *transition probability (i.e. the 'model')*

## Expected return of a policy

$$J(\pi) := \mathbb{E}_{\tau \sim \pi} [R(\tau)] = \int_{\tau \sim \pi} P(\tau|\pi) R(\tau)$$

where  $\tau \sim \pi$  is the space of all the trajectories distributed as  $\pi(a_t|s_t)$

## Central Reinforcement Learning Problem

$$\pi^* := \operatorname{argmax}_{\pi} J(\pi)$$

i.e. finding the policy with the highest expected return

# Policy Gradient

## Parametric Policy

A generic policy that depends on parameters  $\theta$

$$\pi_{\theta}$$

For instance, in the **DQN Algorithm**, the **Action-Value Function** is approximator is a Deep Neural Network

$$\hat{Q}(s, a; \theta)$$

## Policy Gradient Ascent

At each iteration, improve parameters using *expected returns* as the loss function:

$$\theta^{(k+1)} = \theta^{(k)} + \eta \nabla_{\theta} J(\pi_{\theta} |_{\theta^{(k)}})$$

*easier said than done ...*

# Policy Gradient

## 1) Probability of a trajectory, given a parametric policy

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

## 2) Log-Derivative

*(Let's take another direction)*

By the chain rule, this derivative is:

$$\nabla_\theta \log P(\tau|\pi_\theta) = \frac{1}{P(\tau|\pi_\theta)} \nabla_\theta P(\tau|\pi_\theta)$$

It follows:

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta) \nabla_\theta \log P(\tau|\pi_\theta)$$

*Remember this 'mathematical trick':  
it will be played again...*

# Policy Gradient

## 3) Log-Probability of a trajectory

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

*probability of a trajectory*

$$\log P(\tau|\pi_\theta) := \log P(s_0) + \sum_{t=0}^{T-1} [\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)]$$

*these terms do NOT depend on  $\theta$*

## 4) Gradient of the Log-Probability

$$\nabla_\theta \log P(\tau|\pi_\theta) := \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

*This is the purpose of the 'mathematical trick':  
discounting the transition probability (a.k.a. "model")  $P$*

# Policy Gradient

## ▪ Basic Policy Gradient

$$J(\pi) = \int_{\tau \sim \pi} P(\tau|\pi) R(\tau)$$

*expected return*

$$\nabla_{\theta} J(\pi_{\theta}) = \int_{\tau \sim \pi_{\theta}} \nabla_{\theta} P(\tau|\pi_{\theta}) R(\tau)$$

*this term does NOT depend on  $\theta$*

$$= \int_{\tau \sim \pi_{\theta}} P(\tau|\pi_{\theta}) \nabla_{\theta} \log P(\tau|\pi_{\theta}) R(\tau)$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta}} [\nabla_{\theta} \log P(\tau|\pi_{\theta}) R(\tau)]$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) R(\tau) \right]$$

*This last term is an expectation: it can be estimated from a sample mean*

# Policy Gradient

## ▪ Basic Policy Gradient

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau) \right]$$

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

Estimated gradient (mean)

Dataset: a sample of actual trajectories

# Policy Gradient

## ■ Basic Policy Gradient

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau) \right]$$

*An entire trajectory? Even in the past?*

Better switch to this:

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right]$$

*Reward from  $t$  onward  
(‘reward-to-go’)*

# Simple Policy Gradient

## ▪ Pseudo-Algorithm

Initialize the weights  $\theta$  of a DNN  $\hat{Q}(s, a; \theta)$  at random

**Repeat:**

1) For  $M$  episodes

Start in initial state  $s_0$

For  $t$  from 0 to  $T$

play by  $a_t \sim \pi_\theta(a|s_t)$

Collect the episode trajectory  $\tau = \langle (s_t, a_t) \rangle_{t=0}^T$  and store it in  $\mathcal{D}$

2) Sample a random minibatch  $\mathcal{B} = \{\tau_i\}$  from  $\mathcal{D}$

$$\Delta\theta = \eta \frac{1}{|\mathcal{B}|} \sum_{\tau \in \mathcal{B}} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

How can we 'sample a policy' in practice?

replay buffer

# Sampling a Policy

## Problem

Sampling actions from a stochastic policy

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

Intended meaning:

$$\pi_\theta(a_t|s_t) \propto \hat{Q}(a_t, s_t; \theta)$$

*the probability of each action should be proportional to the expected return*

## Discrete Action Space

Consider  $\hat{Q}(a_t, s_t; \theta)$  as the **logit** of a softmax

$$\pi_\theta(a_t|s_t) := \frac{\exp(\hat{Q}(a_t, s_t; \theta))}{\sum_{a \in \mathcal{A}(s_t)} \exp(\hat{Q}(a, s_t; \theta))}$$

and sample accordingly

*All possible actions in state  $s_t$*

# *Actor-Critic*

# Actor-Critic

## An Aside: Expected Grad-Log Probability (EGLP lemma)

**EGLP Lemma.** Suppose that  $P_\theta$  is a parameterized probability distribution over a random variable,  $x$ . Then:

$$\mathbb{E}_{x \sim P_\theta} [\nabla_\theta \log P_\theta(x)] = 0.$$

### Proof

Recall that all probability distributions are *normalized*:

$$\int_x P_\theta(x) = 1.$$

Take the gradient of both sides of the normalization condition:

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0.$$

Use the log derivative trick to get:

$$\begin{aligned} 0 &= \nabla_\theta \int_x P_\theta(x) \\ &= \int_x \nabla_\theta P_\theta(x) \\ &= \int_x P_\theta(x) \nabla_\theta \log P_\theta(x) \\ \therefore 0 &= \mathbb{E}_{x \sim P_\theta} [\nabla_\theta \log P_\theta(x)]. \end{aligned}$$

[image from: [https://spinningup.openai.com/en/latest/spinningup/rl\\_intro3.html](https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html)]

# Actor-Critic

## Policy Gradient

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right]$$

Due to the EGLP lemma:

$$\mathbb{E}_{a_t \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) b(s_t)] = 0$$

for any function  $b(s_t)$  that depends on  $s_t$  only (i.e.,  $b(s_t)$  is constant w.r.t. to  $a_t$ )

## Policy Gradient with Baseline

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( \left( \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right) - \overset{\text{baseline}}{b(s_t)} \right) \right]$$

We can subtract term-wise any function  $b(s_t)$  without altering the expectation

# Actor-Critic

## Actor-Critic

(typical formulation)

*state-value function as baseline*

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( \left( \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right) - V^{\pi}(s_t) \right) \right]$$

Note that:

$$\begin{aligned} \left( \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right) - V^{\pi}(s_t) &= (r(s_t, a_t) + V^{\pi}(s_{t+1})) - V^{\pi}(s_t) \\ &= Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \\ &= A^{\pi}(s_t, a_t) \end{aligned}$$

it's the **advantage function**

# Actor-Critic

## Actor-Critic

(typical formulation)

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{\pi}(s_t, a_t) \right]$$

'Actor' 'Critic'

In practice,  $V^{\pi}(s_t)$  is estimated via  $\hat{V}(s; \phi)$   
namely, another DNN with specific parameters  $\phi$

$$\hat{A}(s_t, a_t) := \left( r(s_t, a_t) + \hat{V}(s_{t+1}; \phi) \right) - \hat{V}(s_t; \phi)$$

*What are the advantages? "It reduces variance"*

Intuitively  $\hat{Q}(s, a; \theta)$  depends also on how the action space is explored  
whereas  $\hat{V}(s_t; \phi)$  depends only on actual rewards  $r(s_t, a_t)$

# Actor-Critic

## ▪ Pseudo-Algorithm

Initialize the weights  $\theta, \phi$  of two DNNs  $\pi_\theta(a|s), \hat{V}(s; \phi)$  at random

**Repeat:**

1) For  $M$  episodes

Start in initial state  $s_0$

For  $t$  from 0 to  $T$

play by  $a_t \sim \pi_\theta(a|s_t)$

Collect all episode **trajectories**  $\tau_r := \langle (s_t, a_t, r_t, s_{t+1}) \rangle_{t=0}^T$  and store them in  $\mathcal{D}$  replay buffer

2) For a random minibatch  $\mathcal{B} = \{(s_i, a_i, r_i, s_{i+1})\}$  from  $\mathcal{D}$

Evaluate

$$\hat{A}(s_i, a_i) = (r_i + \hat{V}(s_{i+1}; \phi)) - \hat{V}(s_i; \phi)$$

Update weights

$$\Delta\phi = -\eta_\phi \nabla_\phi \left( \hat{A}(s_i, a_i) \right)^2 \quad \text{gradient descent: } \hat{V}(s; \phi) \text{ should converge to } V(s)$$

$$\Delta\theta = \eta_\theta \nabla_\theta J(\pi_\theta) = \eta_\theta \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}(s_i, a_i) \quad \text{gradient ascent: increase } J(\pi_\theta) \text{ as much as possible}$$

# Actor-Critic

- **Pseudo-Algorithm** (update step in more detail)

2) For a random minibatch  $\mathcal{B} = \{(s_i, a_i, r_i, s_{i+1})\}$  from  $\mathcal{D}$

Evaluate

$$\hat{h} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{1}{T} \sum_{t=0}^T \nabla_{\phi} \left( \hat{V}(s_t; \phi) - \hat{R}_t \right)^2$$

and:

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

$$\hat{R}_t = \left( r_t + \hat{V}(s_{t+1}; \phi) \right)$$

$$\hat{A}_t = \left( r_t + \hat{V}(s_{t+1}; \phi) \right) - \hat{V}(s_t; \phi)$$

Update

$$\phi^{(t+1)} = \phi^{(t)} - \eta_{\phi} \hat{h}$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta_{\theta} \hat{g}$$

*NOTE: standard mini-batch gradient optimization can be replaced by Adam or other optimizers*

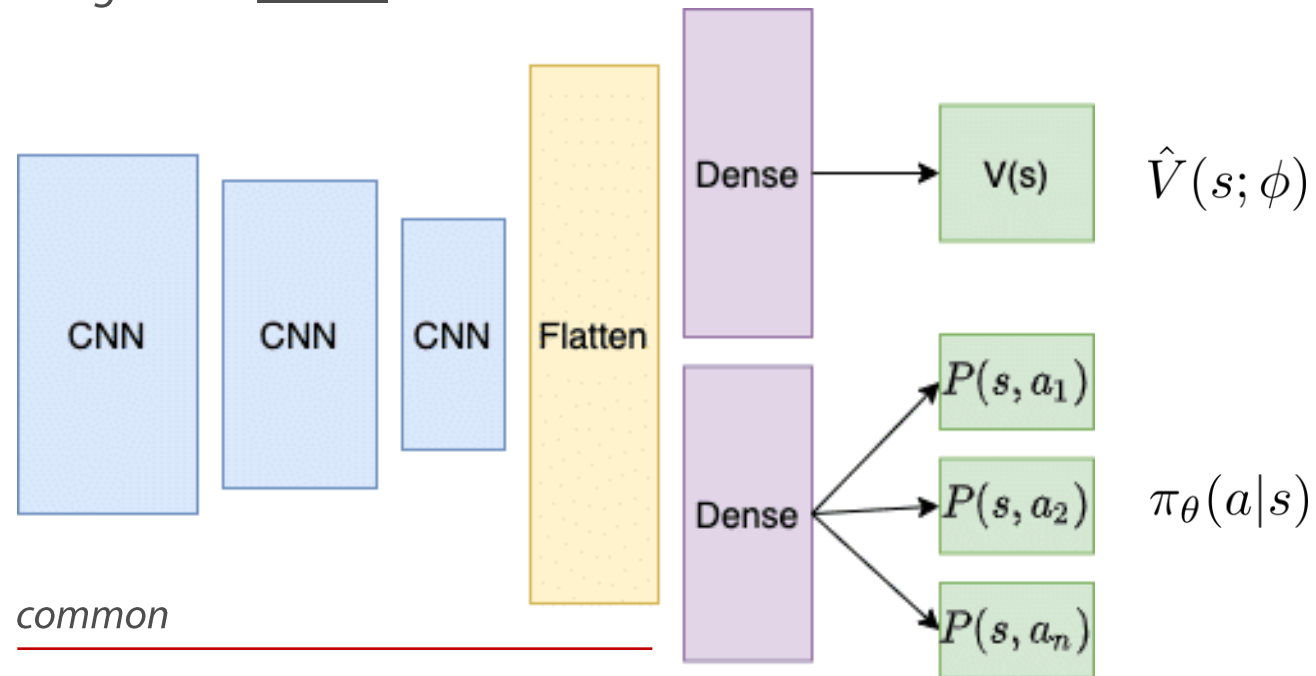
# Actor-Critic

## ▪ Network Architecture (typical)

A bifurcated structure which includes:

- A common part
- A V-head
- A  $\pi$ -head

*It follows that part of the weights are shared*



# *Proximal Policy Optimization (PPO)*

# Improving Training Stability

In the Actor-Critic approach, the *policy update* step is critical:

$$\Delta\theta = \eta_{\theta} \nabla_{\theta} J(\pi_{\theta}) = \eta_{\theta} \nabla_{\theta} \log \pi_{\theta}(a_i | s_i) \hat{A}(s_i, a_i)$$

Unlike supervised learning, deep reinforcement learning is **a very delicate process**: even small steps  $\Delta\theta$  may induce *large changes in policy*, which may compromise the entire optimization

The objective of Proximal Policy Optimization (PPO) is ensuring that steps  $\Delta\theta$  **are kept within a viable range**, determined via *hyperparameters*

# Improving Training Stability

The *policy update* step rewritten:

$$\theta^{(t+1)} = \theta^{(t)} + \eta_{\theta} \nabla_{\theta} J(\pi_{\theta})$$

where:

$$\nabla_{\theta} J(\pi_{\theta}) = \nabla_{\theta} \log \pi_{\theta}(a_i | s_i) \hat{A}(s_i, a_i)$$

Introduce the *surrogate advantage* term,  
a measure of how a new policy  $\theta$  performs relative to a current policy  $\theta^{(t)}$  :

$$\mathcal{L}(\theta, \theta^{(t)}) = \mathbb{E}_{s, a \sim \pi_{\theta^{(t)}}} \left[ \frac{\pi_{\theta}(a | s)}{\pi_{\theta^{(t)}}(a | s)} \hat{A}^{\pi_{\theta^{(t)}}}(s, a) \right]$$

It can be proven that:

$$\nabla_{\theta} J(\pi_{\theta}) \Big|_{\theta^{(t)}} = \nabla_{\theta} \mathcal{L}(\theta, \theta^{(t)})$$

# PPO-clip

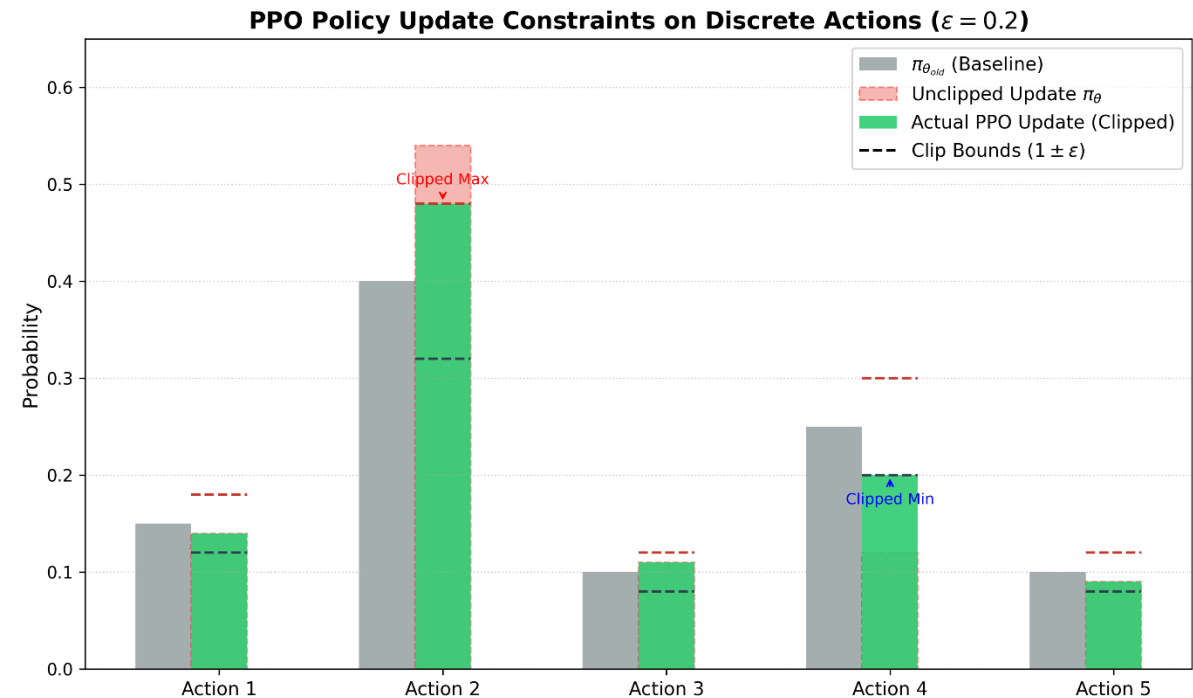
In PPO clip, the term  $J(\pi_\theta)$  is replaced by the clipped surrogate advantage term:

$$L(\theta, \theta^{(t)}) = \mathbb{E}_{s, a \sim \pi_{\theta^{(t)}}} \left[ \min \left( \frac{\pi_\theta(a|s)}{\pi_{\theta^{(t)}}(a|s)} \hat{A}^{\pi_{\theta^{(t)}}}(s, a), \text{clip} \left( \epsilon, \hat{A}^{\pi_{\theta^{(t)}}}(s, a) \right) \right) \right]$$

where:

$$\text{clip}(\epsilon, \hat{A}) := \begin{cases} 1 + \epsilon & \text{if } A \geq 0 \\ 1 - \epsilon & \text{otherwise} \end{cases}$$

and  $\epsilon$  is a suitable hyperparameter value



# *Group Relative Policy Optimization (GRPO)*

# Reinforcement Learning for LLMs

## Basic Reinforcement Learning: Step-by-Step Trajectory

In an environment like *a robot navigating a maze* or *an agent playing chess*, the process is **sequential**:

- **State** ( $s_i$ ): The current board position
- **Action** ( $a_i$ ): Moving a single piece
- **Next State** ( $s_{i+1}$ ): The new board position (after the opponent's move)

## Large Language Models (LLMs): Single-Turn Trajectory

In the framework used for LLMs, the entire text generation is treated as a **single complete trajectory**:

- **State** ( $s$ ): The input prompt (e.g., "*Write a Python function to sort a list*")
- **Action** ( $a$ ): The **entire completed response** generated by the LLM (*the full block of code*)
- **Next State** ( $s_T$ ): Terminal (*the episode is over*)

Consequently, the cumulative reward  $R(a)$  for the entire episode can be directly computed

In the context of LLMs,  $R(a)$  is typically provided by an automated verifier—such as a *Python interpreter* for code or a *regex grader* for math—or a reward model trained on human preference data

# Replacing Critic with a Group Verifier

## Basic Actor-Critic in RL

In a step-by-step trajectory:

$$\hat{A}(s_i, a_i) = \left( r_i + \hat{V}(s_{i+1}; \phi) \right) - \hat{V}(s_i; \phi)$$

Estimating the *advantage function* requires a separate network  $\hat{V}(s_i; \phi)$

## Single-Turn Trajectory (LLM)

The *advantage function estimation* becomes

$$\hat{A}(s, a) = R(a) - \hat{V}(s)$$

↙ which equals  $R(a)$

In other words, the Critic is no longer necessary to estimate the cumulative reward

## Replacing Critic with Average Group Reward

With GRPO, the LLM generates *multiple candidate responses* for a single prompt, allowing the average cumulative reward of the group  $G$  to replace the Critic baseline

$$\hat{A}(s, a^{(i)}) = R(a^{(i)}) - \frac{1}{G} \sum_{j=1}^G R(a^{(j)}) \quad \mathcal{G} = \{a^{(1)}, \dots, a^{(G)}\} \quad \text{Group of generated answers (at moderate temperature)}$$