Deep Reinforcement Learning

Lecture 2: Policy Gradient Methods

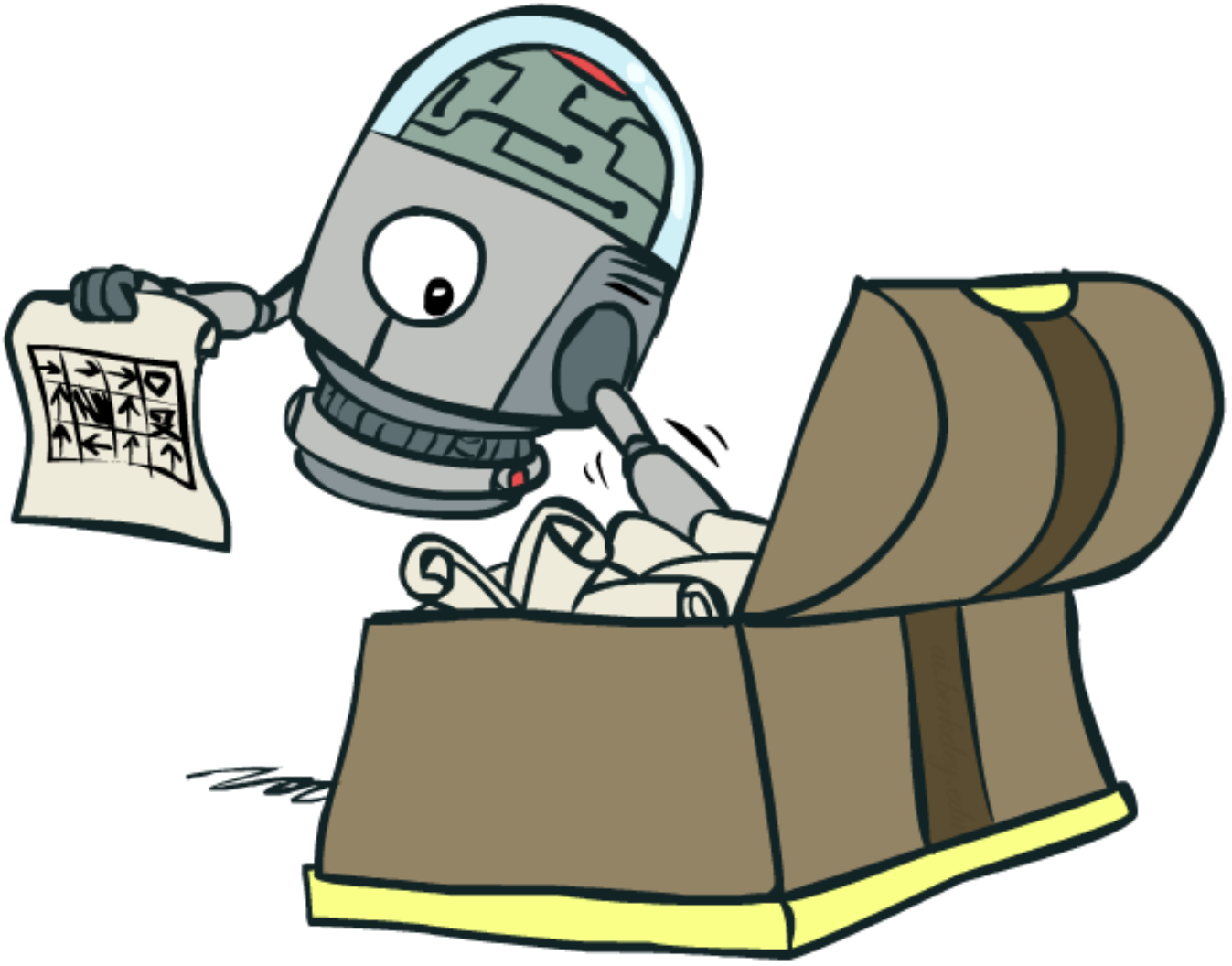
Nicole Orzan
n.orzan@rug.nl

university of groningen

Outline

- Value-based versus Policy-based Reinforcement Learning
- The Policy Gradient Theorem
- Monte-Carlo Policy Gradient
- Actor-Critic Algorithms
- Constrained Policy Gradients

Policy Gradient Methods



Policy Gradient Methods

Last week we saw how we can use a function approximator to learn an approximation of the value functions:

$$V(s;\theta) \approx V^{\pi}(s)$$

$$Q(s,a;\theta) \approx Q^{\pi}(s,a)$$

But we can also parameterize (and learn) the **policy** directly.

To do so, we define our parametric policy as:

$$\pi(a|s;\theta) = P[a|s;\theta]$$

where $\theta \in \mathbf{\Theta}$, and $\mathbf{\Theta} \in \mathbb{R}^n$ is the space of real-valued vectors of policy parameters, in which θ lives.

We will optimize θ by means of gradient ascent.

Policy Gradient Methods

Policy optimization methods have several advantages over the value-based methods that we have discussed last week.

Some of these advantages are:

- Since the probabilities change smoothly, there are stronger theoretical convergence guarantees (even with non-linear function approximation)
- Do not involve maximization over values in order to compute greedy policies: allow to learn stochastic policies and consider continuous action spaces

Policy Gradient Methods

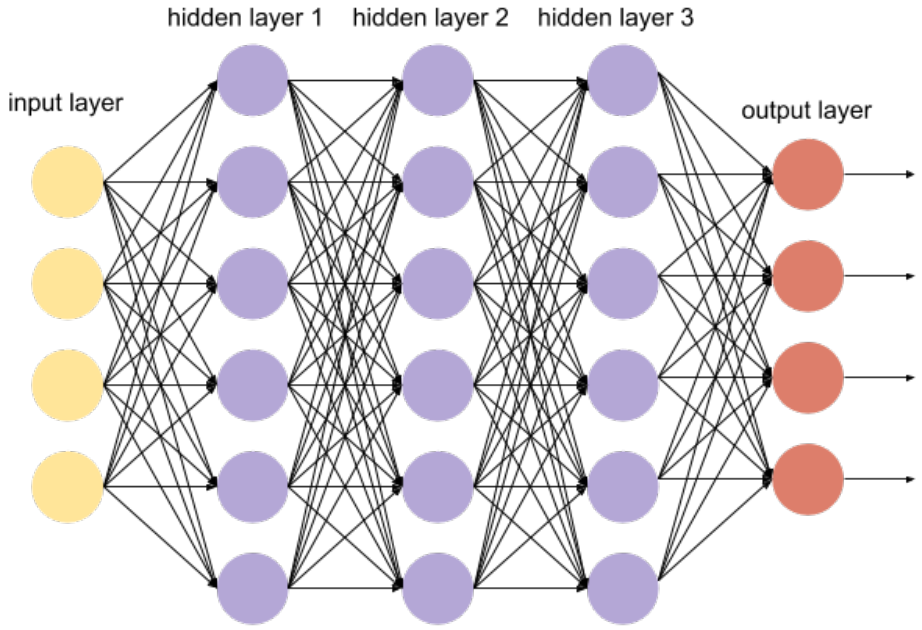
However, they also present some important drawbacks:

- Do not produce a value function as output
- Can converge to local rather than global optima
- Sample inefficient and with high variance (slow convergence)

Policy Gradient Methods

Our main **goal** is to learn the best θ for our policy $\pi(a|s;\theta)$.

If the policy is represented by a neural network f, θ denotes the parameters of the policy, i.e. the weights of the model:



Policy Gradient Methods

We need to consider how to **design** the neural architecture.

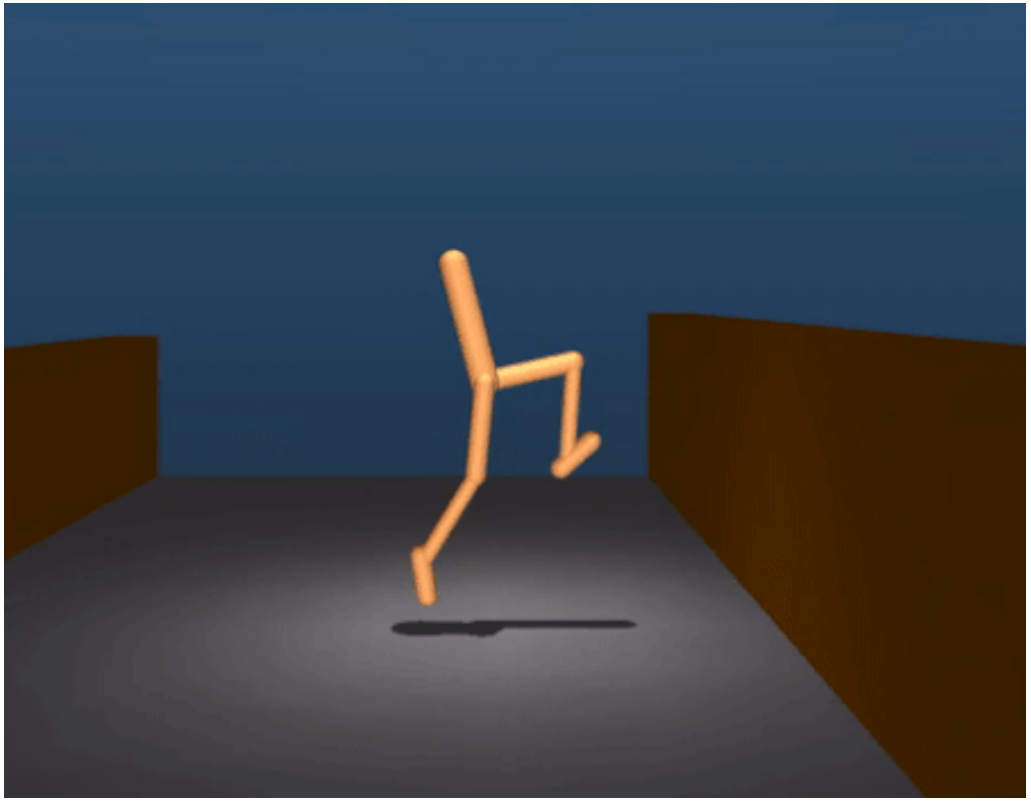
What is the shape of the output layer?

- For discrete action spaces, we have one output neuron for every possible action. As our goal is that of estimating a probability distribution over the action space we use the popular softmax activation function:
  $softmax(\mathbf{z}) = \frac{e^{\mathbf{z}}}{\sum_{j=1}^{K} e^{z_j}}$

Policy Gradient Methods

- On the other hand, for continuous actions spaces, the policy can be a multivariate gaussian distribution where the means of the distribution are given by the Neural Network f:

$$\pi(\mathbf{a}_t|\mathbf{s}_t;\theta_t) = \mathbb{N}(f(\mathbf{s}_t), \mathbf{\Sigma}_t)$$



Policy Gradient Methods

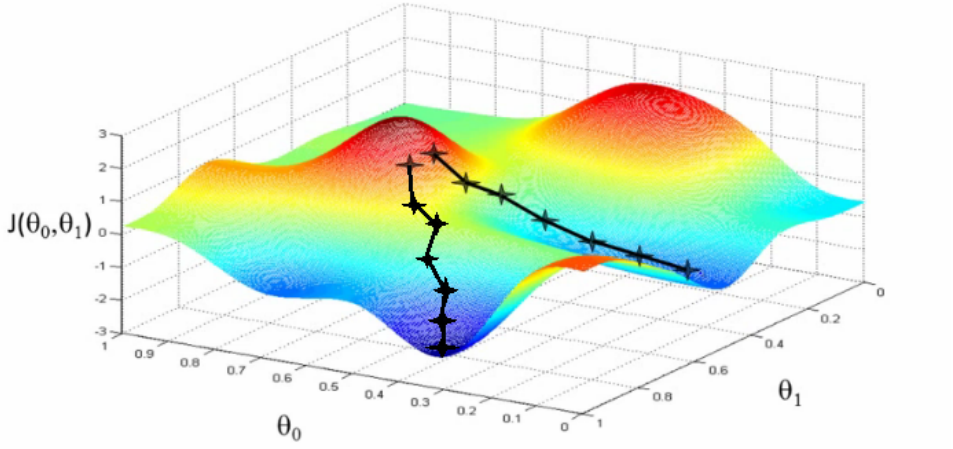
The underlying **idea** of policy gradient methods is to learn θ by updating the weights thanks to the gradient of some performance measure $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

where α is the step-size parameter and

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

is the column vector of partial derivatives.



Finite Differences

We can approximate the gradient using the finite differences method.

The partial derivative on axis k can be approximated by perturbing θ by an amount ϵ on its $k-$th dimension ($0 < k < n$).

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon e_k) - J(\theta)}{\epsilon}$$

Where e_k is the unit vector with 1 on the $k-$th dimension, and 0 everywhere else.

The policy gradient can be estimated by computing the partial derivative in each dimension.

Analytical Derivation of the Policy Gradient

We want to learn how to modify this parameter θ in time in order to learn the optimal policy $\pi^\star$ and solve the problem of control.

We will derive the **Policy Gradient Theorem**, which gives us an analytical expression for the gradient of the performance function.

We assume that our policy $\pi(a|s;\theta)$ is differentiable, and that we know its gradient $\nabla_\theta \pi(a|s;\theta)$.

The Policy Gradient Theorem

Who is $J(\theta)$?

The Policy Gradient Theorem

Who is $J(\theta)$?

Our **objective** is:

$$\theta^{\star} = \mathbf{argmax}_{\theta \in \mathbf{\Theta}} \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

Where $p_{\theta}(\tau)$ is the probability of encountering a trajectory τ, it depends on the policy π, which corresponds to the **parameters** of the neural network:

$$p_{\theta}(\tau) = p_{\theta}(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t;\theta) p(s_{t+1}|s_t, a_t)$$

The Policy Gradient Theorem

Who is $J(\theta)$?

Our **objective** is:

$$\theta^\star = \mathbf{argmax}_{\theta \in \mathbf{\Theta}} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

Where $p_\theta(\tau)$ is the probability of encountering a trajectory τ, it depends on the policy π, which corresponds to the **parameters** of the neural network:

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t;\theta) p(s_{t+1}|s_t, a_t)$$

NOTE: Here we work in episodic environments (without γ) in order to make the calculations easier, but the same results are true for infinite summations with the addition of the γ term.

The Policy Gradient Theorem

Given:

$$\theta^\star = \mathbf{argmax}_{\theta \in \mathbf{\Theta}} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

We define $J(\theta)$ as:

$$J(\theta) := \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

It follows that our goal is to find:

$$\theta^\star \in \mathbf{argmax}_{\theta \in \mathbf{\Theta}} J(\theta).$$

The Policy Gradient Theorem

$$\theta^{\star} \in \mathbf{argmax}_{\theta \in \mathbf{\Theta}} J(\theta)$$

This definition has the appealing property of inducing a natural ordering over policies since: $\pi(a|s, \theta) \geq \pi(a|s, \theta')$ when $J(\theta) \geq J(\theta')$.

Here we can find a **parallelism** with value functions:

$$\pi \geq \pi' \text{ if } V^{\pi}(s) \geq V^{\pi'}(s) \; \forall \; s \in S$$

Where the optimal value function is defined as:

$$V^{\star}(s) = \mathbf{max}_{\pi} V^{\pi}(s)$$

with optimal policy:

$$\pi^{\star} = \mathbf{argmax}_{\pi} V^{\pi}(s) \; \forall s \in S$$

The Policy Gradient Theorem

We can write:

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)] \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)}[G(\tau)] \\
&= \int p_\theta(\tau) G(\tau) d\tau
\end{aligned}
$$

Where we have $G(\tau) = \sum_t r(s_t, a_t)$.

We want to update our policy weights using:

$$
\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)
$$

Therefore want to find a way to estimate the gradient $\nabla_\theta J(\theta)$ that does **not depend** on $p(s_1)$ and $p(s_t + 1 | s_t, a_t)$: we work in a model-free setting.

The Policy Gradient Theorem

We want to find an analytical expression for the gradient of the performance measure:

$$\nabla_\theta J(\theta) = \nabla_\theta \Big[\int p_\theta(\tau) G(\tau) d\tau \Big] = \int \nabla_\theta p_\theta(\tau) G(\tau) d\tau$$

The Policy Gradient Theorem

We want to find an analytical expression for the gradient of the performance measure:

$$\nabla_\theta J(\theta) = \nabla_\theta \Big[\int p_\theta(\tau) G(\tau) d\tau \Big] = \int \nabla_\theta p_\theta(\tau) G(\tau) d\tau$$

We can write down a useful identity:

$$\begin{aligned} \nabla_\theta p_\theta(\tau) &= p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} \\ &= p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \end{aligned}$$

The Policy Gradient Theorem

We want to find an analytical expression for the gradient of the performance measure:

$$\nabla_\theta J(\theta) = \nabla_\theta \Big[\int p_\theta(\tau) G(\tau) d\tau \Big] = \int \nabla_\theta p_\theta(\tau) G(\tau) d\tau$$

We can write down a useful identity:

$$\begin{aligned} \nabla_\theta p_\theta(\tau) &= p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} \\ &= p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \end{aligned}$$

Plugging the trick into the equation we get:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) G(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) G(\tau)] \end{aligned}$$

The Policy Gradient Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) G(\tau)]$$

Remembering that:

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t;\theta) p(s_{t+1}|s_t, a_t)$$

The Policy Gradient Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) G(\tau)]$$

Remembering that:

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t;\theta) p(s_{t+1}|s_t, a_t)$$

We can rewrite:

$$\begin{aligned}\nabla_\theta \log p_\theta(\tau) &= \nabla_\theta \left[\log p(\mathbf{s_1}) + \sum_{t=1}^{T} \log \pi(\mathbf{a_t}|\mathbf{s_t};\theta) + \log p(\mathbf{s_{t+1}}|\mathbf{s_t}, \mathbf{a_t}) \right] \\ &= \nabla_\theta \sum_{t=1}^{T} \log \pi(a_t|s_t;\theta).\end{aligned}$$

The Policy Gradient Theorem

Plugging this term into $\nabla_\theta J(\theta)$ we get that, for any differentiable policy $\pi(a|s;\theta)$:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) G(\tau)] \\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta)\right)\left(\sum_{t=1}^{T} r(s_t,a_t)\right)\right].\end{aligned}$$

Which is called the **Policy Gradient Theorem**.

The Policy Gradient Theorem

If we take a closer look at the gradient of the performance measure:

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

Remembering that:

$$\nabla_\theta \log \pi(a_t|s_t;\theta) = \frac{\nabla_\theta \pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta)}$$

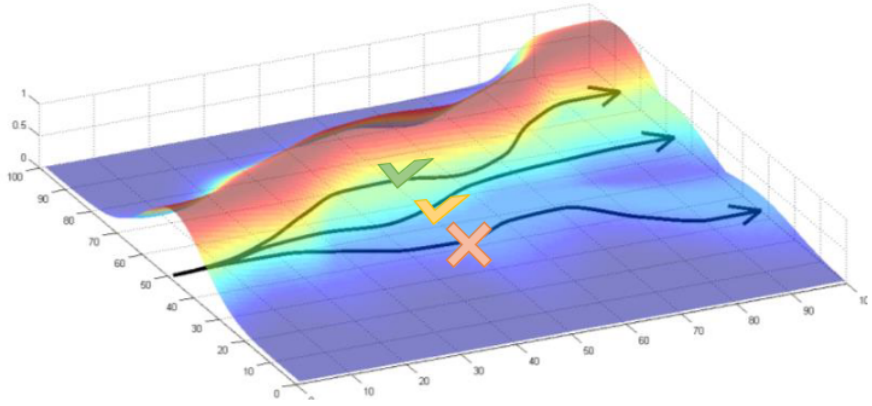
- The update is directly proportional to the return of the trajectory
- The update is inversely proportional to the probability of taking an action in a state

The Policy Gradient Theorem

If we take a closer look at the gradient of the performance measure:

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

- It rises the log probabilities of the good trajectories
- It lowers the log probabilities of the bad trajectories



Break

The REINFORCE Algorithm

We just found the following Policy Gradient result:

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

The gradient of the objective J can be approximated with:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a_t | s_t; \theta) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right).$$

This is convenient as we can now use it to perform an update on the weights of the network:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_\mathbf{t}).$$

The REINFORCE Algorithm

Algorithm 1 REINFORCE Algorithm

Input: differentiable policy $\pi(a|s, \boldsymbol{\theta})$; step size sequence $\alpha_k > 0$; batch size N
Initialize $\boldsymbol{\theta}$ arbitrarily
for k in N_{EPISODES}:
    Generate N trajectories to fill dataset D_k
    $\widehat{\nabla}_\theta J(\boldsymbol{\theta}_k, D_k) \leftarrow \frac{1}{N} \sum_N \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a|s, \boldsymbol{\theta}) \right) \left(\sum_{t=1}^{T} r_t \right)$
    $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k \widehat{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k, D_k)$
return $\boldsymbol{\theta}$

- One of the most famous policy gradient algorithm, first introduced by Williams in 1992 is the REINFORCE algoritm.
- It considers the finite-horizon setting.
- The REINFORCE algorithm uses an estimator which is the simplest Monte-Carlo Policy Gradient update computed on batches of episodes.

The REINFORCE Algorithm

While theoretically grounded, the REINFORCE algorithm is known to be **slow** and suffer from **high variance**.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right)$$

The REINFORCE Algorithm

While theoretically grounded, the REINFORCE algorithm is known to be **slow** and suffer from **high variance**.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right)$$

How can we reduce the variance? Using causality:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \left(\sum_{t'=1}^{T} r(s_{t'}, a_{t'}) \right)$$

The REINFORCE Algorithm

While theoretically grounded, the REINFORCE algorithm is known to be **slow** and suffer from **high variance**.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right)$$

How can we reduce the variance? Using causality:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \left(\sum_{t'=1}^{T} r(s_{t'}, a_{t'}) \right)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \right)$$

Where the last term is considers the so-calles rewards **to-go**.

The REINFORCE Algorithm

There are other **practical limitations**, that make the REINFORCE algorithm struggling in finding $\pi^\star$:

- For infinite sample sizes, the policy gradient will always converge to the (local) optima. However, for small sample sizes, the policy gradient will be very sensitive, and the high-variance problem will make it very hard to use in practice.
- Furthermore, what happens if good samples have reward $r_t = 0$? In these cases, the policy will not be updated!

Both these problems would benefit of using a **baseline**.

REINFORCE with Baseline

The simplest solution to **reduce variance** and to avoid the **null update** when $r_t = 0$, is to include a comparison of the returns $G(\tau)$ to a baseline b.

This results in:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) \right) \big(G(\tau) - b \big)$$

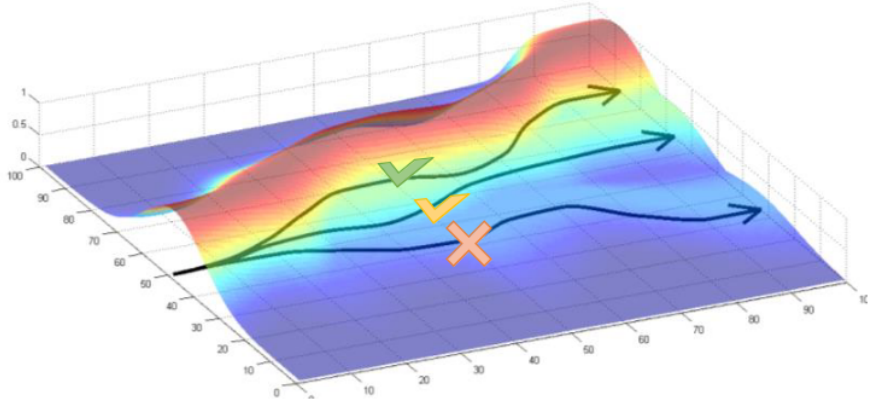
Where the simplest baseline is:

$$b = \frac{1}{N} \sum_{i=1}^{N} G(\tau) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} r(s_t, a_t)$$

REINFORCE with Baseline

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) \right) \big(G(\tau) - b \big)$$

In this way we are aligning the gradient update with our intuition: we are **increasing** the log-probabilities of the actions that bring rewards **better** than average, and **decreasing** the ones that bring rewars **worse** than average (even if, for example, all the rewards that we get from the environments are positive).



REINFORCE with Baseline

We could also define our baseline as approximation of the state value function $V(s)$:

$$V(s) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{k=1}^{T} r_{t+k} \middle| s_t = s \right]$$

- Using the value function as a baseline allows to explicit the dependence on the **state** and rescale with respect to its value.
- If all actions have high (low) values we need a higher (lower) baseline to differentiate the higher valued actions from the less highly valued ones

REINFORCE with Baseline

By adding a constant to the gradient:

- The variance of the gradient is riduced
- The expected value of the gradient will not change.

Let's show it:

$$\begin{aligned}\mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) b] &= \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) b \; d\tau \\ &= b\int \nabla_\theta p_\theta(\tau) d\tau \\ &= b\nabla_\theta \int p_\theta(\tau) d\tau \\ &= b\nabla_\theta 1 = 0.\end{aligned}$$

REINFORCE with Baseline

Also note that the average of rewards is **not** the optimal baseline.

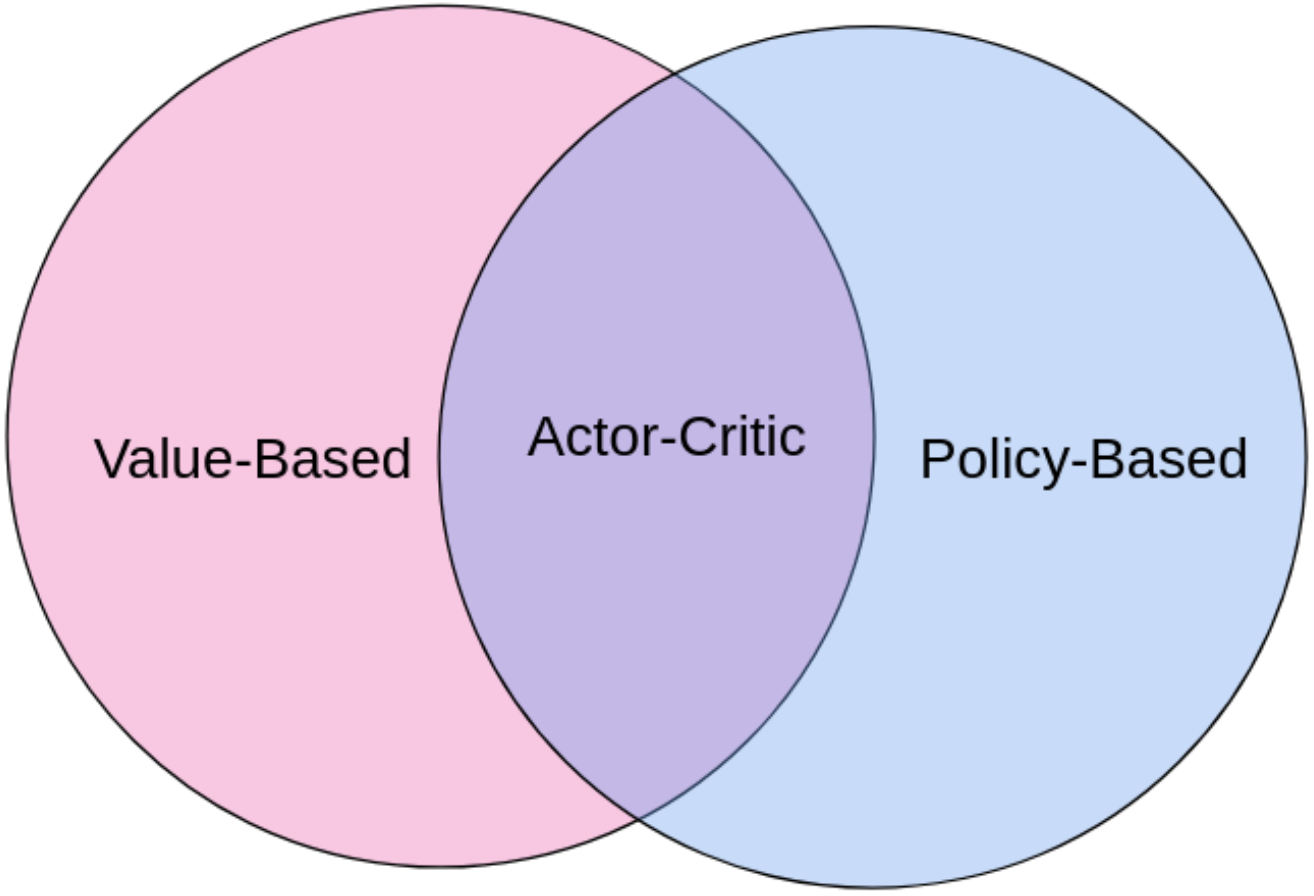
We could derive the optimal baseline by writing down the variance of $\mathbb{E}_{\tau \sim p_\theta(\tau)}\Big[\nabla_\theta \log p_\theta(\tau)\big(G(\tau) - b\big)\Big]$ and solving for b.

This results in:

$$
\begin{aligned}
b &= \frac{\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2 G(\tau)]}{\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2]} \\
&= \frac{\mathbb{E}[(\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta))^2 G(\tau)]}{\mathbb{E}[(\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta))^2]}
\end{aligned}
$$

Actor-Critic Methods

Actor-Critic methods merge the strengths of both Value-Based and Policy-Based methods.



Actor-Critic Methods

The idea is to use the common value functions used last week to reduce the variance of the policy gradient estimates. These value functions are called the **critic** of the algorithm.

We can use as critic the state value function $V(s)$, the state-action value function $Q(s,a)$ or the advantage function $A(s,a)$.

The gradient of the objetive becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t \nabla_\theta \log p_\theta(\tau) Q(\tau) \right]$$

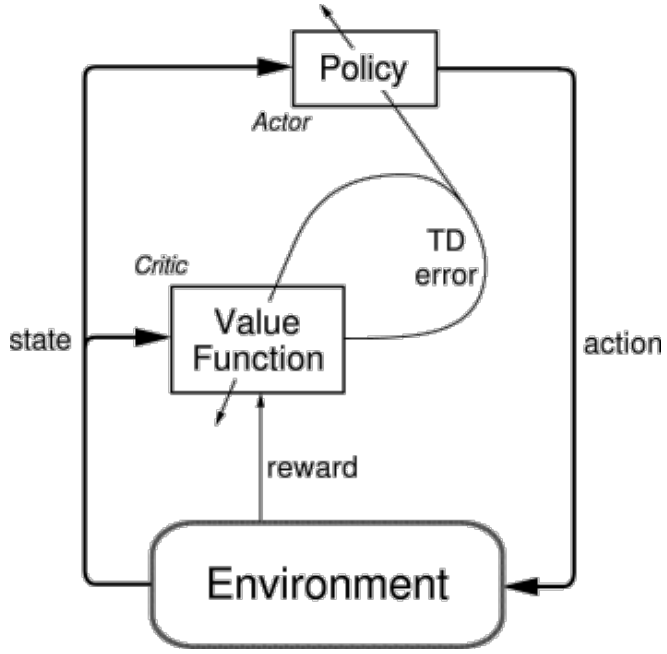
The bias introduced through bootstrapping and reliance on the state representation reduces variance and accelerates learning.

Actor-Critic Methods

Usually the critic:

- Is estimated using a function approximator (with parameters ω)
- Is learned through **bootstrapping**

The value function can be learned by adding Temporal-Difference (TD) learning into the policy gradient loop:



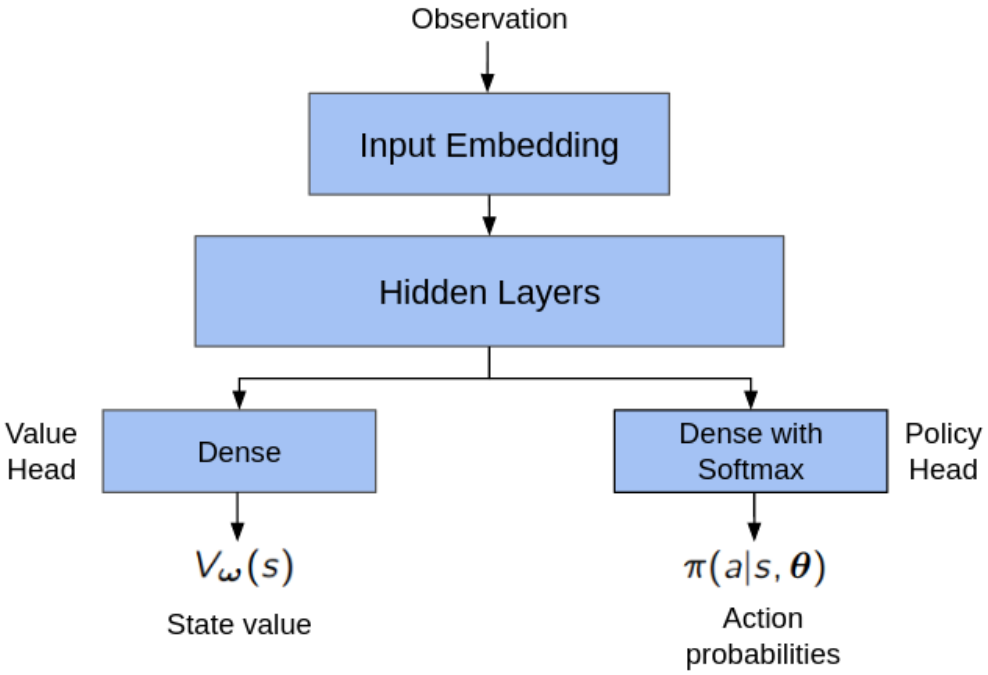
Actor-Critic Methods

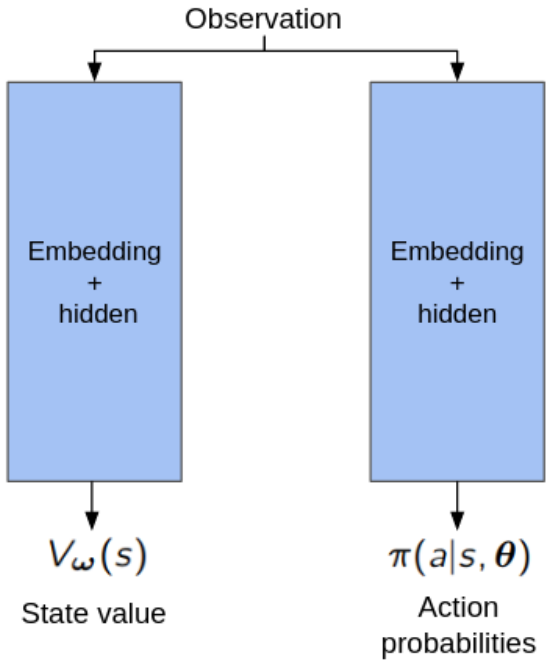
Algorithm 2 Actor−Critic Algorithm

Input: differentiable policy $\pi(a|s, \boldsymbol{\theta})$, critic $Q_{\boldsymbol{\omega}}(s, a)$;
step size sequences $\alpha_\theta > 0$ and $\alpha_\omega > 0$
Initialize $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ arbitrarily
for e in N_{EPISODES}:
    Initialize s_0
    until s_t not TERMINAL:
        take action $a_t \sim \pi(\cdot|s_t, \boldsymbol{\theta}_t)$, get r_t, s_{t+1}
        $\delta \leftarrow r_t + \gamma Q_{\boldsymbol{\omega}}(s_{t+1}, a_{t+1}) - Q_{\boldsymbol{\omega}}(s_t, a_t)$
        $\boldsymbol{\omega}_{t+1} \leftarrow \boldsymbol{\omega}_t + \alpha_{\omega_t} \delta \nabla_\omega Q_\omega(s, a)$
        $\widehat{\nabla}_\theta J(\boldsymbol{\theta}_t) \leftarrow \nabla_\theta \log \pi(a_t|s_t, \theta_t)\big(Q_\omega(s_t, a_t)\big)$
        $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha_{\theta_t} \widehat{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$
return $\boldsymbol{\theta}$, $\boldsymbol{\omega}$

Actor-Critic Methods

Different network structures:

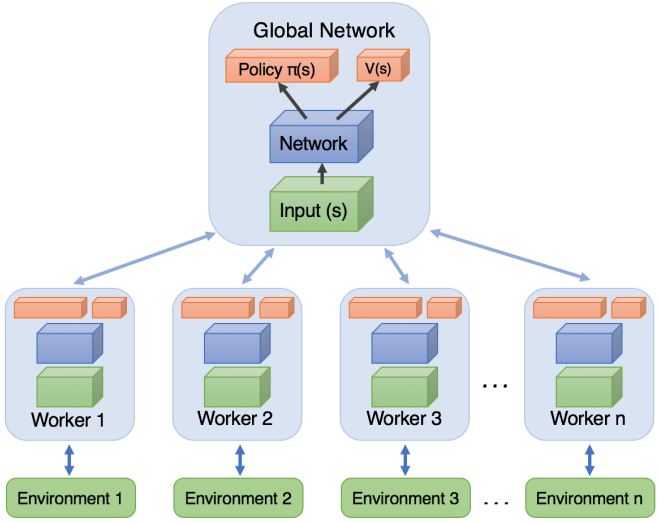




Asynchronous Advantage Actor Critic (A3C)

Proposed by Mnih et al., 2016, as a model where the policy and value function networks share the first layers of weights.

This algorithm is designed to efficiently train **large models** (deep neural networks for polices and critics).



A synchronous variant of A3C exists as well: the Advantage Actor Critic (A2C), comprehensive of a single learner.

Deterministic Policies

Recall last week's objective function:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

- Experience Replay Buffer: D
- Mini-Batch Training: $\mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)}$
- A Target Network: θ^-
- Reward Clipping

Deterministic Policies

Recall last week's objective function:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

- Experience Replay Buffer: D
- Mini-Batch Training: $\mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)}$
- A Target Network: θ^-
- Reward Clipping

DQN works with high-dimensional observation spaces, but handles only discrete and low-dimensional action spaces

Deterministic Policies

When we use DQN, we work with **deterministic** policies:

$$\pi(s;\theta) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s,a;\theta) \text{ for all } s \in \mathcal{S}$$

In **continuous** action spaces, greedy maximization becomes problematic!

- We would need to perform an optimization process at every time step of the learning process.
- In practical terms, the optimization would be too slow to be performed, especially when in presence of function approximators and non-trivial action spaces.

Deterministic Policies

Solution: move the policy in the direction of the gradient of Q, rather than globally maximising Q.

$$\nabla_\theta J(\theta) := \mathbb{E}[\nabla_\theta Q(s, a; \theta) \nabla_\theta \pi(s; \theta)]$$

This function has been proved to be the policy gradient (Deterministic Policy Gradient Theorem), so it can be used to update the policy's weights:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta)$$

Algorithms that are based on this idea are:

- Deterministic Policy Gradient (DPG)
- Deep Deterministic Policy Gradient (DDPG)

Automatic Policy Differentiation

How can we use tools such as PyTorch in order to compute the gradients of the performance function in an efficient way?

This is **not trivial** as our friendly objective sometimes can be very expensive to estimate:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(\mathbf{a_t}|\mathbf{s_t};\theta) \right) \left(\sum_t r(s_t, a_t) \right)$$

We will create a loss function $\mathcal{L}(\theta)$, which is **not** the performance function, but whose gradient is the gradient of the (estimated) performance function. So:

$$\mathcal{L}(\theta) \neq J(\theta)$$
$$\nabla_\theta \mathcal{L}(\theta) = \widehat{\nabla_\theta J}(\theta)$$

In this way we will be able to use the backpropagation algorithm, computing the derivative of the loss function, and backpropagate it throught the Neural Network.

Automatic Policy Differentiation

A common shape for the estimator of the performance function is the following:

$$\widehat{\nabla_\theta J}(\theta) = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi(a_t|s_t;\theta)\hat{A}_t]$$

Where $\hat{\mathbb{E}}_t$ is the estimate of the expectation and $\hat{A}_t$ is an estimate of the advantage function, computed at time step t. Those are computed by averaging over a finite batch of samples of trajectories.

The loss function employed is:

$$\mathcal{L}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t;\theta)\hat{A}_t]$$

Which is **not** the Reinforcement Learning objective! However, the differentiation of this loss function will provide us with and estimator of $\nabla_\theta J(\theta)$.

Automatic Policy Differentiation

Example for a one-step trajecory and discrete action policy in PyTorch:

$$
\begin{aligned}
&\text{probs} = \text{policynet}(\text{state}) \\
&\text{m} = \text{Categorical}(\text{probs}) \\
&\text{action} = \text{m.sample}() \\
&\text{nextstate}, \text{reward} = \text{env.step}(\text{action}) \\
&\text{loss} = -\text{m.logprob}(\text{action}) * \text{reward} \\
&\text{loss.backward}()
\end{aligned}
$$

Constrained Policy Gradient

The policy gradient update has the problem that some parameters will affect the change in probabilities more than others.

- The policy gradient direction computed (or estimated) under policy π_θ, due to its local (first−order) nature, does not take into account the fact that the updated policy parameters $\theta' = \theta + \alpha \nabla J(\theta)$ will induce a possibly very different distribution over states and actions.

- This distributional mismatch issue, peculiar of RL, can be a source of performance oscillations and unexpected, potentially unsafe behavior.

A solution to this could be to re-scale the process, so that the updates are of equal size in the **policy space**, and not in the **parameters space**.

Constrained Policy Gradient

Ideally we want our policy not to change too much during the update process. We need to use a function that allows us to compute the distance among distributions:

$$D(\pi_\theta, \pi_{\theta'}) \leq \epsilon.$$

Two popular algorithms that are based on this idea are:

- Trust Region Policy Optimization (TRPO)
- Proximal Policy Optimization (PPO)

Trust Region Policy Optimization (TRPO)

This algorithm has been proposed by Schulman et al. in 2015.

In the TRPO algorithm, the objective of the optimization process is slightly different than the one we used until now:

$$\mathcal{L}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{old})} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r_t(\theta)\hat{A}_t]$$

subject to:

$$\hat{\mathbb{E}}_t [KL(\pi(\cdot|s_t;\theta)||\pi(\cdot|s_t;\theta_{old}))] \leq \delta.$$

Where KL is the Kullback-Leibler divergence (or relative entropy) and is a measure of difference between two probability distributions. A simple way enforce the KL constraint is to use lagrange multipliers.

Proximal Policy Optimization (PPO)

The Proximal Policy Optimization (PPO) algorithm is an improvement to TRPO, proposed by Schulman et al. in 2017.

Here, the hard KL constraint is removed in favour of a penalization over too large policy updates.

The main objective is the following one:

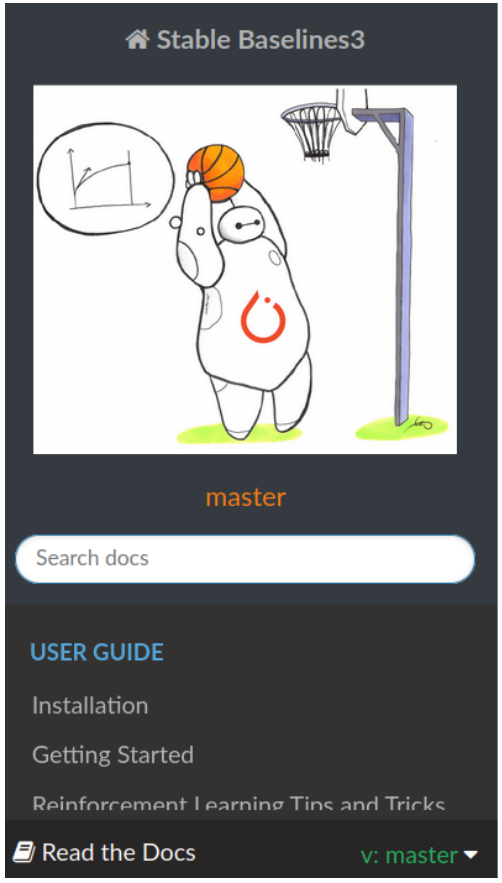
$$\mathcal{L}_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\mathbf{min}(r_t(\theta)\hat{A}_t, \mathbf{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

The final objective contains as well an entropy regularization term, and the squared error loss $\mathcal{L}_t^{VF}(\theta) = (V_\theta(s_t) - V_t^{targ})^2$:

$$\mathcal{L}_t(\theta) = [\mathcal{L}_t^{CLIP}(\theta) - c_1\mathcal{L}_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

Where c_1 and c_2 are hyperparameters.

Where to find these algorithms?



Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations Edit on GitHub

Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations

Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines.

Github repository: https://github.com/DLR-RM/stable-baselines3

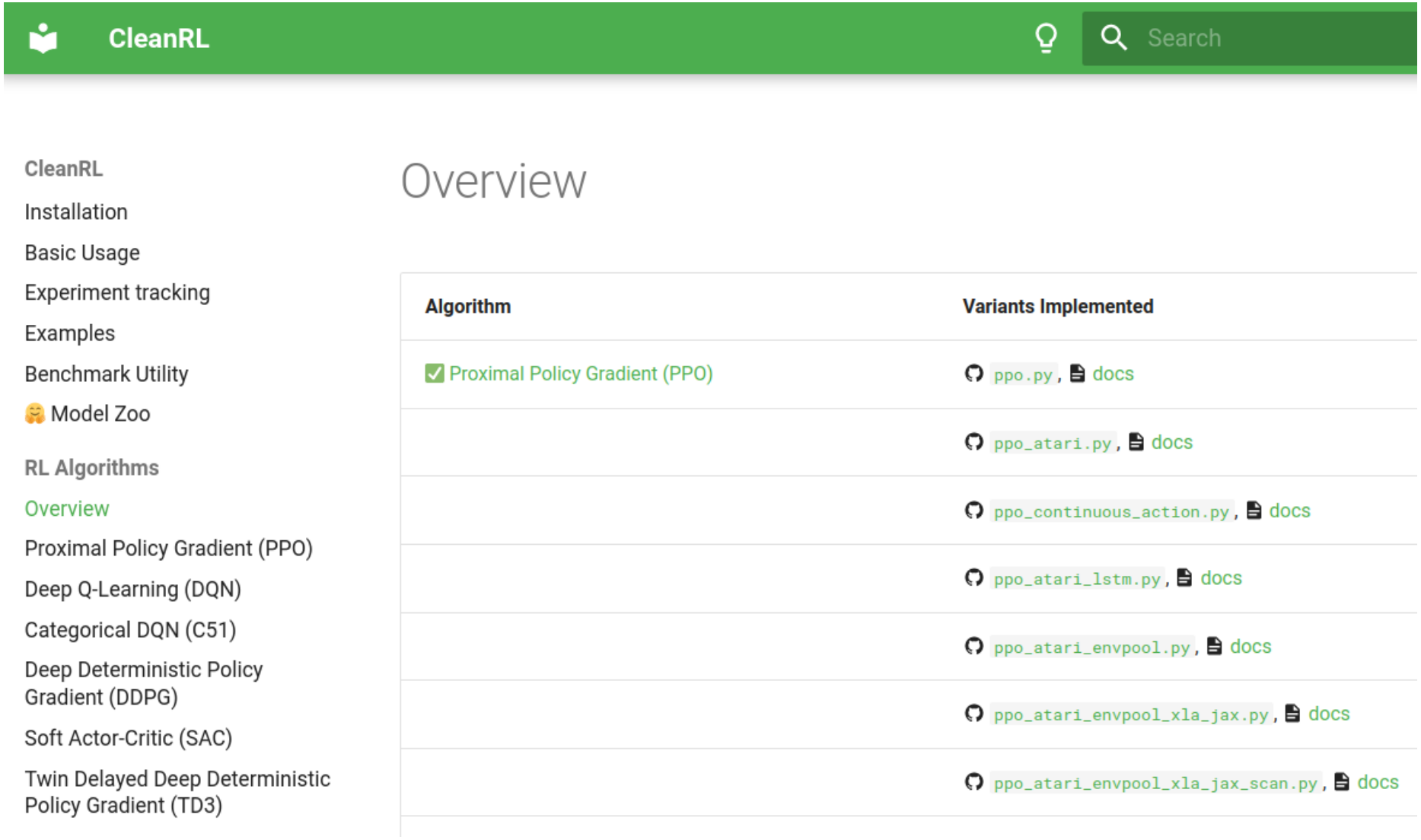
Paper: https://jmlr.org/papers/volume22/20-1364/20-1364.pdf

RL Baselines3 Zoo (training framework for SB3): https://github.com/DLR-RM/rl-baselines3-zoo

RL Baselines3 Zoo provides a collection of pre-trained agents, scripts for training, evaluating agents, tuning hyperparameters, plotting results and recording videos.

SB3 Contrib (experimental RL code, latest algorithms): https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

Where to find these algorithms?



Recap

Today we saw:

- Value-based versus Policy-based Reinforcement Learning
- Analytical derivation and The Policy Gradient Theorem
- Monte-Carlo Policy Gradient (REINFORCE)
- Causality, baselines and critics to reduce variance
- Application to deterministic policies
- Constrained Policy Gradient, TRPO and PPO

References

- Sutton and Barto "Reinforcement Learning: an introduction"; Chapter 13, "Policy Gradient Methods"
- Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 8, 229–256 (1992). https://doi.org/10.1007/BF00992696
- John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, Philipp Moritz: Trust Region Policy Optimization. ICML 2015: 1889-1897
- Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu: Asynchronous Methods for Deep Reinforcement Learning. ICML 2016: 1928-1937
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov: Proximal Policy Optimization Algorithms. CoRR abs/1707.06347 (2017)

References

- Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra: Continuous control with deep reinforcement learning. ICLR 2016

- David Silver lecture on policy gradient: https://www.youtube.com/watch?v=KHZVXao4qXs

- Berkeley Deep RL course: https://rail.eecs.berkeley.edu/deeprlcourse/

See you next week

Additional Material

Importance Sampling

Let's imagine we have samples from θ' and we want to estimate the gradient for our parameter verctor θ. We can use the following mathematical trick:

$$
\begin{aligned}
\mathbb{E}_{x \sim p(x)}[f(x)] &= \int f(x)p(x)dx \\
&= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\
&= \mathbb{E}_{x \sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]
\end{aligned}
$$

Importance Sampling

Let's imagine we have samples from θ' and we want to estimate the gradient for our parameter verctor θ. We can use the following mathematical trick:

$$\begin{aligned}\mathbb{E}_{x\sim p(x)}[f(x)] &= \int f(x)p(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

Using this trick, the gradient of the performance function becomes:

$$\begin{aligned}J(\theta) &= \mathbb{E}_{\tau\sim p_\theta(\tau)}[G(\tau)] \\ &= \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}G(\tau)\right].\end{aligned}$$

Importance Sampling

From which we get the following off-policy objective:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\nabla_\theta \frac{p_\theta(\tau)}{p_{\theta'}(\tau)} G(\tau) \right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} \nabla_\theta \log p_\theta(\tau) G(\tau) \right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\left(\prod_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} \right) \nabla_\theta \log p_\theta(\tau) G(\tau) \right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\left(\prod_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} \right) \left(\sum_{t=1}^{T} \nabla_\theta \log \pi(\mathbf{a_t}|\mathbf{s_t};\theta) \right) G(\tau) \right].
\end{aligned}
$$

Where we are optimizing the action policy $\pi(a_t|s_t;\theta)$ while we are sampling our trajectories using the policy $\pi(a_t|s_t;\theta')$s