# Reinforcement Learning Cheat Sheet

## Markov Decision Process

A tuple $(T, S, (A_s, p(\cdot|s,a), r_t(s,a), 1 \leqslant t \leqslant T, s \in S, a \in A_s))$ where:

finite set of states : $s \in S$
finite set of actions : $a \in A$
state transition probabilities :
$p(s'|s,a) = Pr\{S_{t+1} = s' | S_t = s, A_t = a\}$
reward for a given state-action pair :
$r_t(s,a) \in \mathbb{R}$

We seek to maximize :
if T is finite : expected total reward

$$R = \mathbb{E}\left[\sum_{t=1}^{T} r_t(s_t, a_t)\right]$$

if T is infinite : discounted reward (typically $\lambda = 0,9$)

$$R = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=1}^{T} \lambda^{t-1} r_t(s_t, a_t)\right]$$

We can then define the expected return at time t :

$$R_t = \mathbb{E}\left[\sum_{t'=t}^{T} r_{t'}(s_{t'}, a_{t'})\right]$$

or

$$R_t = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t'=t}^{T} \lambda^{t'-1} r_{t'}(s_{t'}, a_{t'})\right]$$

## Value Function

Value function describes *how good* it is to be in a specific state $s$ under a certain policy $\pi$. For MDP:

$$V^\pi(s) = \mathbb{E}\left[R^\pi | s_1^\pi = s\right]$$

Informally, it is the expected return when starting from $s$ and following $\pi$

### Optimal

$$V^*(s) = \sup_\pi V_\pi(s)$$

## Action-Value (Q) Function

We can also denote the expected reward for state, action pairs.

$$Q^\pi(s,a) = \mathbb{E}\left[R^\pi | s_1^\pi = s, a_1^\pi = a\right]$$

Complexity of evaluating the Q-function with finite T : $A^T S^{T+1}$, i.e. the number of possible trajectories

### Optimal

The optimal value-action function:

$$Q^*(s,a) = \sup_\pi Q^\pi(s,a)$$

Using this new notation we can redefine $V^*$, using $S^*(s,a)$ :

$$V^*(s) = \max_{a \in A_s} Q^*(s,a)$$

Intuitively, the above equation express the fact that the value of a state under the optimal policy **must be equal** to the expected return from the best action from that state.

## Bellman's Equations

### Bellman's Optimality Equation

$$V^*(s) = \max_{a \in A_s} \underbrace{\left[r(s,a) + \lambda \sum_j p(j|s,a) V^*(j)\right]}_{Q(s,a)}$$

### Bellman's Expectation Equation

$$V^\pi(s) = \mathbb{E}_\pi\left[r^\pi(s) + \lambda \sum_j \pi(j|s) V^\pi(j)\right]$$

## Contraction Mapping

This part is a preliminary to Dynamic Programming

### Definition

Let $(X,d)$ be a metric space and $f : X \to X$. We say that $f$ is a *contraction* if there is a real number $k \in [0,1[$ such that

$$d(f(x), f(y)) \leq kd(x,y)$$

for all $x$ and $y$ in $X$, where the term $k$ is called a *Lipschitz coefficent* for $f$.

### Contraction Mapping theorem

Let $(X,d)$ be a complete metric space and let $f : X \to X$ be a contraction. Then there is one and only one fixed point $x^*$ such that

$$f(x^*) = x^*$$

Moreover, if $x$ is any point in $X$ and $f^n(x)$ is inductively defined by $f^2(x) = f(f(x))$, $f^3(x) = f(f^2(x))$, $\ldots$, $f^n(x) = f(f^{n1}(x))$, then $f^n(x) \to x^*$ as $n \to \infty$. This theorem guarantees a unique optimal solution for the dynamic programming algorithms detailed below.

## Dynamic Programming

When we know the MDP, we can use Bellman's equation to find the optimal policy by just *planning* (no learning)

### Value Iteration

```
Initialize $V_0(s) \in \mathbb{R}$, e.g $V_0(s) = 0$
while $\|V_n - V_{n-1}\| \geqslant \epsilon$ do
    foreach $s \in S$ do
        $V_{n+1}(s) = \max_a \left[r(s,a) + \lambda \sum_j p(j|s,a) V_n(j)\right]$
    end
end
output: Probably Approximately Correct (PAC) optimal
        value function $V \approx V^*$
```

**Algorithm 1:** Value Iteration

We can then derive a PAC optimal policy $\pi \approx \pi^*$ such that

$$\pi(s) \in \underset{a}{\operatorname{argmax}}[r(s,a) + \lambda \sum_j p(j|s,a) V_n(j)]$$

VI converges almost surely to an $\epsilon$-optimal policy with complexity $\Theta(S^2 A)$ for each iteration

### Policy Iteration

Needs more steps but computes directly the optimal policy and is quicker than Value Iteration most of the time.

### Howard's Policy Iteration

```
Initialize $V(s) \in \mathbb{R}$, e.g $V(s) = 0$, $\pi(s) \in A_s$ for all $s \in S$
while $\pi_n \neq \pi_{n-1}$ do
    1. Policy Evaluation
    foreach $s \in S$ do
        $V_n^\pi(s) = r^\pi(s) + \lambda \sum_j \pi(j|s) V_n^\pi(j)$
    end
    2. Policy Improvement
    foreach $s \in S$ do
        $\pi_{n+1}(s) = \underset{a}{\operatorname{argmax}} \left[r(s,a) + \lambda \sum_j p(j|s,a) V_n^\pi(j)\right]$
    end
end
output: PAC optimal value function and policy $V \approx V^*$
        and $\pi \approx \pi^*$
```

**Algorithm 2:** Howard's Policy Iteration

### Simplex Policy Iteration

```
Initialize $V(s) \in \mathbb{R}$, e.g $V(s) = 0$, $\pi(s) \in A_s$ for all $s \in S$
while $\pi_n \neq \pi_{n-1}$ do
    1. Policy Evaluation
    foreach $s \in S$ do
        $V(s) = \max_{a \in A_s} \left[r(s,a) + \lambda \sum_j p(j|s,a) V_n^\pi(j)\right]$
        Choose $s_0 \in \underset{s \in S}{\operatorname{argmax}}(V(s) - V_n^\pi(s))$
    end
    2. Policy Improvement
    foreach $s \in S \backslash s_0$ do
        $\pi_{n+1}(s) = \pi_n(s)$
    end
    $\pi_{n+1}(s_0) = \underset{a \in A_{s_0}}{\operatorname{argmax}} \left[r(s_0,a) + \lambda \sum_j p(j|s_0,a) V_n^\pi(j)\right]$
end
output: PAC optimal value function and policy $V \approx V^*$
        and $\pi \approx \pi^*$
```

**Algorithm 3:** Simplex Policy Iteration

Both of the above-mentioned versions converge with complexity $\Theta(S^\omega) + \Theta(S^2 A)$ per iteration. Howard's and Simplex PI require respectively $\mathcal{O}(\frac{A}{1-\lambda} \log \frac{1}{1-\lambda})$ and $\mathcal{O}(\frac{AS}{1-\lambda} \log \frac{1}{1-\lambda})$ iterations to converge (HPI is better with large state-spaces)

# A few definitions

## Model-free vs Model-based

In Model-based methods (e.g. Dynamic Programming), we have complete knowledge of the model of the environment, contrary to Model-free methods (e.g. Monte Carlo).

| | **Model-based** | **Model-free** |
|---|---|---|
| Transitions/Rewards are | **known** | **unknown** |
| Can predict without action | **yes** | **no** |
| Type of strategy | **Planning** | **Learning** |

## On-policy vs Off-policy

In On-policy learning (e.g. SARSA) the agent learns the value of the policy being carried out by the agent. In Off-policy learning (e.g. Q-Learning), the learning process is independent of the agent's actions, which are sampled from a **behavior policy**.

## Exploration vs Exploitation

An agent is explorating when it is trying to get new information that can possibly lead to higher rewards, and it is exploiting when it is maximizing its reward based on the information it already has.

## Prediction vs Control

In prediction tasks, given a policy, we want to compute how well it performs, while control tasks seek to find the optimal policy.

# Monte Carlo Methods

In Monte Carlo (MC) methods, we evaluate policies through sampling : the value being considered as the mean return. MC is Model-free, and on-policy. MC learns from complete episodes (no bootstrapping), and consequently all episodes must terminate.

To implement it for control tasks, we use a MC Policy Evaluation, and add a Policy Improvement (e.g. $\epsilon$-greedy)

```
Initialize $Q(s,a) = 0$, $\pi(s) \in A_s$ for all $s \in S, a \in A_s$
foreach episode $1 \leqslant i \leqslant n$ do
    1. Monte-Carlo Evaluation
    Sample a full trajectory
      $\tau_i = ((s_{1,i}, a_{1,i}, r_{1,i}), ..., (s_{T_i,i}, a_{T_i,i}, r_{T_i,i}))$ from $\pi$
    $R = 0$
    foreach $t$ in $T_i, T_{i-1}, ..., 1$ do
        $R = r_{t,i} + \lambda R$
        if $s_{t,i}$ is not visited later then
            $Q^{(i)}(s_{t,i}, a_{t,i}) =$
              $Q^{(i-1)}(s_{t,i}, a_{t,i}) + \frac{1}{i}\left(R - Q^{(i-1)}(s_{t,i}, a_{t,i})\right)$
        end
    end
    2. $\epsilon$-greedy Policy Improvement
    foreach $s$ in the episode do
        $\pi(s) = \begin{cases} \operatorname{argmax}_a Q(s,a) & \text{w.p. } 1-\epsilon \\ \text{uniform}(A_s) & \text{w.p. } \epsilon \end{cases}$
    end
end
```

**Algorithm 4:** First-visit Monte Carlo Control

For non-stationary problems (i.e. transitions and rewards **slowly** change over time), the Monte Carlo estimate for, e.g, $Q$ is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R - Q(s_t, a_t)\right]$$

Where $\alpha$ is the learning rate, quantifying how much we want to forget about past experiences.

# Stochastic Approximation

This part is a preliminary to Temporal Difference.

## Robbins-Monro

To find the root $x^* \in \mathbb{R}$ of a continuous function $h$ when we observe $Y \,/\, \mathbb{E}[Y(x)] = h(x)$

```
Initialize $x^*_{(0)} \in \mathbb{R}$
for $k \geqslant 0$ do
    $x^*_{(k+1)} = x^*_{(k)} - \alpha_k Y(x^*_{(k)})$
end
```

**Algorithm 5:** Robbins-Monro (RM)

($h$ must verify $\exists \beta > 0 / \forall x \in \mathbb{R}^d, h(x)(x - x^*) \geqslant \beta \|x - x^*\|^2$)

## SA for extrema localisation

If we use $x(t_k) = x^*_{(k)}$ with $t_k = \sum_{i=1}^{k-1} \alpha_i$ (i.e. $\alpha_k = t_k - t_{k-1}$), we can use the RM algorithm :

$$\frac{x^*_{(k+1)} - x^*_{(k)}}{\alpha_k} = \underbrace{\frac{x(t_{k+1}) - x(t_k)}{t_{k+1} - t_k}}_{\to \dot{x}(t_k) \text{ as } k \to \infty} = Y(x(t_k))$$

Under certain assumptions, we have that

$$\lim_{k \to \infty} x^{(k)} = x^*$$

where $x^*$ is the only globally stable point of $\dot{x} = h(x)$.
By adding 2 more assumptions, we can derive an asynchronous version (only a subset of coordinates of $x^{(k)}$ are updated at each iteration), which can be used for Temporal Difference Learning.

# Temporal Difference

Temporal Difference (TD) methods learn directly from raw experience without a model of the environment's dynamics. TD substitutes the expected discounted reward $R_t$ from the episode with an estimation:

**MC Update** $$V(s_t) \leftarrow V(s_t) + \alpha_t \left[\underbrace{R_t}_{target} - V(s_t)\right]$$

**TD Update** $$V(s_t) \leftarrow V(s_t) + \alpha_t \left[\underbrace{r_t + \lambda V(s_{t+1})}_{target} - V(s_t)\right]$$

TD methods are Model-free, with bootstrapping (i.e. only the current estimate is used as target).
TD presents the advantage of being adapted to continuing environments where some episodes never terminate, thanks to **on-line learning**, i.e. update after every step. Q-Learning and SARSA are the most common TD methods.

# SARSA

SARSA (State-Action-Reward-State-Action) applies directly the above-mentioned principle.

```
Initialize $Q(s,a) = 0$, $\pi(s) \in A_s$ for all $s \in S, a \in A_s$
foreach episode $1 \leqslant i \leqslant n$ do
    foreach step $t$ until end do
        Choose $a_t$ $\epsilon$-greedily
        Observe $r_t, s_{t+1}, a_{t+1}$
        $Q^{(t+1)}(s_t, a_t) = Q^{(t)}(s_t, a_t) +$
          $\alpha\left[r_t + \lambda Q^{(t)}(s_{t+1}, a_{t+1}) - Q^{(t)}(s_t, a_t)\right]$
    end
end
```

**Algorithm 6:** SARSA

## Q-Learning

In Q-Learning, the target is different since it uses the optimal action at state $s_{t+1}$

```
Initialize $Q(s,a) = 0$, $\pi(s) \in A_s$ for all $s \in S, a \in A_s$
foreach episode $1 \leqslant i \leqslant n$ do
    foreach step $t$ until end do
        Choose $a_t$ $\epsilon$-greedily
        Observe $r_t, s_{t+1}$
        $Q^{(t+1)}(s_t, a_t) = Q^{(t)}(s_t, a_t) +$
          $\alpha\left[r_t + \lambda \max_{b \in A} Q^{(t)}(s_{t+1}, b) - Q^{(t)}(s_t, a_t)\right]$
    end
end
```

**Algorithm 7:** Q-Learning

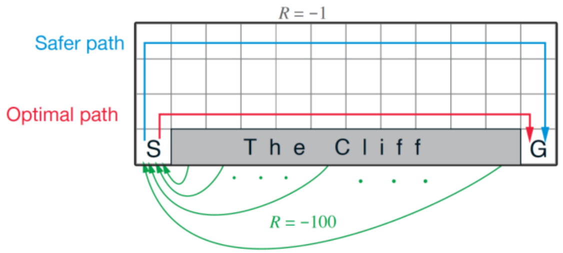
## Comparison of SARSA and Q-Learning

SARSA is on-policy while Q-Learning is off-policy.

Indeed, Q-Learning learns the true Q-function of the optimal policy, not the Q-function of the behavior policy.

Meanwhile SARSA learns the Q-function of the behavior policy including the $\epsilon$-greedy exploration phase.

This implies that SARSA is "safer" since it takes into account the random exploration of the agent, which may be better if the agent is used while learning in some cases:



SARSA tends to follow a safer path

# Policy Gradients

## Introduction

The objective is to maximize the value function from the first state. To do so we parameterize the policy such that $\pi_\theta(s,a)$ is the probability of taking action $a$ in state $s$ and is defined with respect to $\theta \in \mathbb{R}^n$.
The parameterization can be done by any kind of estimator but we typically use a neural network.
If we note a trajectory as $\tau = (s_1, a_1, ..., s_T, a_T)$ with $s_1$ sampled from a distribution $p$, we seek to maximize

$$J(\theta) = \mathbb{E}_{s_1 \sim p}\left[V^{\pi_\theta}(s_1)\right]$$

with

$$V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} r(s_t, a_t) \mid s_1 = s\right]$$

To maximize $J$ and find the optimal policy $\pi_{\theta*}(s,a)$, we apply gradient descent on theta, i.e. we iteratively update theta :

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$$

## The Policy Gradient Theorem

The PG Theorem gives an expression of the gradient of $J$.
Since $J(\theta)$ can be equivalently written as

$$J(\theta) = \sum_\tau \pi_\theta(\tau) R(\tau)$$

By using the log derivative trick we get

$$\nabla J(\theta) = \sum_\tau \pi_\theta(\tau) \times (\nabla \log \pi_\theta(\tau) R(\tau))$$

$$\Leftrightarrow \nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(\tau) R(\tau)\right]$$

and finally

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(s_t, a_t)\right)\left(\sum_{t=1}^{T} r(s_t, a_t)\right)\right]$$

## REINFORCE

REINFORCE is the direct application of the PG Theorem with Monte-Carlo philosophy. It only works with MC Rollouts, i.e. full trajectories, and thus requires finite episodes.

```
Initialize $\theta^{(0)} \in \mathbb{R}^n$
**foreach** *episode* $1 \leqslant k \leqslant n$ **do**
    Generate $\tau_k = (s_{1,k}, a_{1,k}, r_{1,k}, ..., s_{T_k,k}, a_{T_k,k}, r_{T_k,k})$
    $\theta^{(k+1)} =$
      $\theta^{(k)} + \alpha_k \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(s_{t,k}, a_{t,k})\right)\left(\sum_{t=1}^{T} r_{t,k}\right)$
**end**
```

**Algorithm 8:** REINFORCE

## Variance reduction

To reduce the variance of the estimator used in REINFORCE and make the learning process faster and more stable, we can :

① Compute several trajectories and average the gradient estimator.

② Use Reward-to-go : We use only the rewards from the future, i.e. the update becomes

$$\theta \leftarrow \theta + \alpha \sum_{t=1}^{T} \nabla \log \pi_\theta(s_t, a_t) \sum_{u=t}^{T} r_u$$

This is valid because of the Expected Grad-Log-Prob (EGLP) lemma (obtained with the log derivative trick) :

$$\mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(s_t, a_t) r_u\right] = 0$$

③ Add a baseline : $R(\tau) \to R(\tau) - b$.
Examples of baselines :

- Natural (average of episodes):
  $b = \frac{1}{n}\sum_{i=1}^{n}\sum_{t=1}^{T} r(s_{t,i}, a_{t,i})$
- Variance-optimal:
  $b = \min \mathrm{Var}(X) = \frac{\mathbb{E}_{\pi_\theta}\left[(\nabla_\theta \log \pi_\theta(\tau))^2 R(\tau)\right]}{\mathbb{E}_{\pi_\theta}\left[(\nabla_\theta \log \pi_\theta(\tau))^2\right]}$
- Reward-to-go : $b = \frac{1}{n}\sum_{i=1}^{n}\sum_{t=\mathbf{u}}^{T} r(s_{t,i}, a_{t,i})$
- Value function (most common) : $b = V^{\pi_\theta}(s_t)$, often approximated by a neural network $V_\Phi(s_t)$ trying to minimize the MSE :

$$\Phi_k = \arg\min_\Phi \mathbb{E}_{s_t, R\widehat{R}_t \sim \pi_k}\left[(V_\Phi(s_t) - \hat{R}_t)^2\right]$$

A REINFORCE algorithm with averaging of several trajectories, Reward-to-go, and the use of a baseline is often called Vanilla Policy Gradient (VPG). However at this point the definitions get fuzzy and differ depending on sources, which may lead to confusions.

It is possible to adapt PG methods for infinite episodes by computing the discounted return on $n$ steps. However, these methods do not work for on-line learning (without knowing the future).

*See also A2C/A3C, TRPO, PPO*

# Function Approximation

Computing the true value of functions such as $V$ or $Q$ is costly: under the PAC framework, it scales as $S \times A$ where $S$ and $A$ are the dimension of respectively the state and action spaces. For continuous state-action spaces (like most real-world applications), it becomes infeasible.
Instead we use function approximation on parametrized value functions $V_\theta$ and $Q_\theta$, which requires modifying the update steps accordingly :

**Monte-Carlo :**

$$\theta \leftarrow \theta + \alpha \left(R - Q_\theta(s_t, a_t)\right) \nabla_\theta Q_\theta(s, a)$$

**SARSA :**

$$\theta \leftarrow \theta + \alpha \left(r_t + \lambda Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)\right) \nabla_\theta Q_\theta(s_t, a_t)$$

**Q-Learning :**

$$\theta \leftarrow \theta + \alpha \left(r_t + \lambda \max_b Q_\theta(s_{t+1}, b) - Q_\theta(s_t, a_t)\right) \nabla_\theta Q_\theta(s_t, a_t)$$

To make it possible, we need neural networks, because they excel at approximating functions. Then, a few tweaks are needed to guarantee a stable and efficient learning, which leads to the DQN algorithm.

## Deep Q-Networks

DQN implements two tricks :

- **Experience replay** : We store experiences $(s, a, r, s')$ if a replay buffer $B$, from which we can sample the experiences used for the update. This is because successive experiences are strongly correlated, which affects the convergence rate and the stability. If $B$ is too small, the algorithm overfits the few last experiences, and if $B$ is too large, it may slow down the learning.
- **Target network** : in Q-Learning we use a target $r + \lambda \max_b Q(s', b)$ to compute the update. But computing this target with the Q-function we are approximating (which evolves quickly) makes the training unstable. Instead we use a second network $Q_\Phi$ called the target network, which "follows" the evolution of $Q_\theta$ by updating only every $C$ steps. In the algorithm below, beware of the difference between $Q_\theta$ and $Q_\Phi$.

```
Initialize $\theta, \Phi \in \mathbb{R}^n, B, s_1$
**for** $t \geqslant 1$ **do**
    $\pi_t$ : $\epsilon$-greedy policy w.r.t $Q_\theta$
    Observe and store $(s_t, a_t, r_t, s_{t+1})$ in $B$
    Sample $k$ experiences $(s_i, a_i, r_i, s'_i)$ from $B$
    **for** $1 \leqslant i \leqslant k$: **do**
        $y_i = \begin{cases} r_i & \text{if episode stops} \\ r_i + \lambda \max_b Q_\Phi(s'_i, b) & \text{otherwise} \end{cases}$
        $\theta \leftarrow \theta + \alpha \left(y_i - Q_\theta(s_i, a_i)\right) \nabla_\theta Q_\theta(s_i, a_i)$
    **end**
    Every C steps : $\Phi \leftarrow \theta$ or $\Phi \leftarrow \tau\Phi + (1 - \tau)\theta$
**end**
```

**Algorithm 9:** Deep Q-Networks

*See also DDPG, TD3, SAC*

# Actor-Critic Methods

Actor-Critic methods try to take the best of both worls between PG and Function approximation.
The PG update is very convenient for large or even continuous action spaces, but suffers from high variance since you need to sample a trajectory from a policy. Consequently AC methods try to incorporate Function Approximation in the PG update by using value functions instead of the baseline and/or the return. This approach allows to balance the bias of the Value-function approximations and the variance of policy sampling.
In the Actor-Critic framework, the actor corresponds to the entity (most of the time a neural network) computing the policy, and the critic to the entity computing the target, thus assessing how good the policy is.

*See also A2C/A3C, TRPO, PPO, DDPG, TD3, SAC*

# More Algorithms

## A2C/A3C

(Asynchronous) Advantage Actor-Critic (A2C / A3C) are working implementations of VPG (which suffers from too much variance to work in actual settings). It is basically the same algorithm, where we use the value function as a baseline, which leads to the following update :

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla \log \pi_\theta(s_t, a_t) A^{\pi_\theta}(s_t, a_t)$$

with $A^\theta_\Phi(s,a) = R(\tau_{\pi_\theta}) - V^{\pi_\theta}_\Phi(s)$ being the Advantage function, intuitively representing how much better or worse a certain action is compared to other actions (relatively to the current policy).

```
Initialize $\theta_0, \Phi_0 \in \mathbb{R}^n$
foreach episode $1 \leqslant k \leqslant n$ do
    Generate $\tau_k = (s_{1,k}, a_{1,k}, r_{1,k}, ..., s_{T_k,k}, a_{T_k,k}, r_{T_k,k})$
    for $t = T_k, T_{k-1}, ..., 1$ do
        Update the actor network :
        $\theta_{k+1} =$
        $\theta_k + \alpha \sum_{t=1}^{T} \nabla_{\theta_k} \log \pi_{\theta_k}(s_{t,k}, a_{t,k}) A^{\theta_k}_{\Phi_k}(s_{t,k}, a_{t,k})$
        Update the critic network (typically with MSE):
        $\Phi_{k+1} = \Phi_k + \left(R_t(\tau_k) - V_{\Phi_k}(s_{t,k})\right) \nabla_{\Phi_k} V_{\Phi_k}(s_{t,k})$
    end
end
```

**Algorithm 10:** A2C

For infinite episodes, we can use the discounted Temporal Difference target instead of the total reward :

$$A^{\pi_\theta}(s,a) = \sum_{i=0}^{k} \gamma^i r_{t+i} - V^{\pi_\theta}(s)$$

However, the algorithm doesn't work for on-line learning.

In the original A3C paper, the training was made with several agents at the same time (hence asynchronous), which all update separately the centralized network. However, more recent sources affirm that the asynchronous behavior is not needed, and waiting for every update to then average the gradients is actually preferable (larger batch size). This is called A2C.

Remark : for better exploration, we can add an entropy term to the update so that enough randomness is guaranteed.

## TRPO

In A2C/A3C we update the policy by choosing a set of parameters in a neighborhood of the old policy's parameters. But even a small shift in the parameter space might cause a radical change on the policy.

TRPO (Trust Region Policy Optimization) allows to avoid this issue by by choosing the new policy in a neighborhood of the old one directly in the policy space (the so-called trust region). The ”proximity” between the two distributions is guaranteed by a threshold on **KL-divergence**.

However, just like VPG and A2C/A3C, TRPO keeps the drawback of possibly being stuck in a local maximum, and does not work for on-line learning.

The theoretical TRPO updates consists in finding the parameters maximizing the **surrogate advantage** (a function depicting how well the new policy performs compared to the old one), while satisfying a hard constraint on KL-divergence.

For more details, the maths behind TRPO are explained in the annex.

```
Initialize $\theta_0, \Phi_0 \in \mathbb{R}^n$
for $k \geqslant 0$ do
    Generate a set of trajectories $\mathcal{D}_k$ using $\pi_{\theta_k}$
    Compute rewards-to-go $R_t$
    Compute advantage estimates $A_t$ using the critic
    Estimate policy gradient as
        $g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_{\theta_k} \log \pi_\theta(a_t|s_t) A_t$
    Use the conjugate gradient method[1] : $x_k \approx H_k^{-1} g_k$
    Update the policy with backtracking line search[1]:
        $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{x_k^T H_k x_k}} x_k$
    Update the critic network
end
```

**Algorithm 11:** TRPO

[1] : *See annex*

## PPO

PPO is a simplification of TRPO (i.e. seeks to choose the new policy in a trust region in the policy space), but it works as well as TRPO in practive. There are two versions of PPO :

- **PPO-penalty** : KL-divergence is involved, like in TRPO, but it uses a penalty coefficient (modified over time) in the optimization problem rather than a hard constraint.
- **PPO-clip** : Instead of using KL-divergence, we clip the surrogate advantage function if the new policy is too far from the old one. While it does not guarantee to stay in the trust region, it does a pretty good job, and it is possible to use early stopping on KL-divergence during the optimization process to ensure it. We'll focus on this one here.

The update is given by :

$$\theta_{k+1} = \arg\max_\theta \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \left[\mathcal{L}_c(s,a,\theta_k,\theta)\right]$$

where $\mathcal{L}_c$ is a clipped version of the surroagte advantage from TRPO :

$$\mathcal{L}_c(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \right.$$
$$\left. \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right)$$

```
Initialize $\theta_0, \Phi_0 \in \mathbb{R}^n$
for $k \geqslant 0$ do
    Generate a set of trajectories $\mathcal{D}_k$ using $\pi_{\theta_k}$
    Compute rewards-to-go $R_t$
    Compute advantage estimates $A_t$ using the critic
    Update the policy by maximizing the objective :
        $\theta_{k+1} = \underset{\theta}{\text{argmax}} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \mathcal{L}_c(s_t, a_t, \theta_k, \theta)$
     typically via stochastic gradient ascent with Adam.
    Update the critic network
end
```

**Algorithm 12:** PPO-clip

## DDPG - MADDPG

### DDPG

We would like to adopt the Q-Learning framework in order to adapt A2C to on-line learning. However this is not directly possible with large or continuous action-spaces, since in Q-Learning we need to compute the max over all actions.

Deterministic Policy Gradients (DPG) solves this issue by approximating the optimal action $a^*$ in state $s$ by a neural network $\mu_\theta(s) \approx a^*$, making $\mu$ a deterministic policy. However, to adapt the algorithm to real-world problems, we need to apply the tricks from Deep Q-Networks, namely Replay Buffer and Target Networks. The derived algorithm is called Deep Deterministic Policy Gradients (DDPG).

Finally, to boost the learning phase, we can add a zero-mean gaussian noise $\mathcal{N}$ to the policy $\mu_\theta(s)$, and optionally begin the training with a random exploration for a few steps, before coming back to DDPG exploration. Moreover, we can choose when to update the networks to further tune the performance. Updating at the end of an episode or when this episode has reached a maximum length is common.

```
Initialize $\theta, \Phi, \theta_{tg}, \Phi_{tg} \in \mathbb{R}^n, B, s_1$
for $t \geqslant 1$ do
    Noisy Deterministic policy :
      $\mu_{noisy} = \text{clip}(\mu_\theta(s) + \epsilon, a_{low}, a_{high})$ where $\epsilon \sim \mathcal{N}$
    Observe and store $(s_t, a_t, r_t, s_{t+1})$ in $B$
    if $s_{t+1}$ is terminal or ep. reached $max\_len$ then
        Set $s_{t+2} = s$ where s is a starting state.
        Sample $k$ experiences $(s_i, a_i, r_i, s'_i)$ from $B$
        for $1 \leqslant i \leqslant k$: do
            $y_i = \begin{cases} r_i & \text{if end} \\ r_i + \lambda Q_{\Phi_{tg}}(s'_i, \mu_{\theta_{tg}}(s'_i)) & \text{otherwise} \end{cases}$
            Update critic $\Phi$ by gradient descent :
              $\Phi \leftarrow \Phi + \alpha\,(y_i - Q_\Phi(s_i, a_i))\,\nabla_\Phi Q_\Phi(s_i, a_i)$
            Update actor $\theta$ by gradient ascent :
              $\theta \leftarrow \theta + \beta\,\nabla_\theta Q_\Phi(s_i, \mu_\theta(s_i))$
        end
        Update targets : $\begin{cases} \Phi_{tg} \leftarrow \tau\Phi_{tg} + (1-\tau)\Phi \\ \theta_{tg} \leftarrow \tau\theta_{tg} + (1-\tau)\theta \end{cases}$
    end
end
```

**Algorithm 13:** DDPG

## Multi-Agent DDPG

This section provides an extension of DDPG for multi-agent problems. In such environments, several agent must compete, cooperate or come up with a mixed strategy to pursue a global objective. This is not possible with traditional algorithms, such as DDPG, because of two major issues :

- The environment is not stationary from the point of view of one agent, which makes Q-learning approaches unreliable, and prevents the use of a replay buffer.
- When several agents are in play, the variance of policy gradients methods drastically increases.

Multi-Agent DDPG (MADDPG) adds modifications to the DDPG algorithm:

- **MA Learning algorithm** : We adopt centralized training and decentralized execution, which means the agent has access to all the information of every agent during training but can only use its own incomplete information during testing. However, we give the additional info only to the critic, as the Q-function cannot contain different info at training and testing time without further assumptions. The new training update is obtained by using a centralized Q-function taking as input all the actions taken, and some additional information $x$ (typically containing all the observations of each agent).
- **Policy Ensembling** (Optional): To reduce variance, especially with competing agents, a trick is to train each agent on several sub-policies, of which one is randomly selected at each episode. We have to store them in a different replay buffer for each sub-policy, and then update every policy from the corresponding replay buffer.
- **Policy Inference** (Optional) : If we want to discard the assumption of knowing the policies of the other agents, we can approximate them by using a neural network $\hat{\mu}_i^j = \mu_{\Phi_i^j}$.

```
For Each agent :
Initialize $\theta, \Phi, \theta_{tg}, \Phi_{tg} \in \mathbb{R}^n, B, s_1$(and optionnaly $\Phi_i^j$)
for $t \geqslant 1$ do
    Noisy Deterministic policy :
      $\mu_{noisy} = \text{clip}(\mu_\theta(s) + \epsilon, a_{low}, a_{high})$ where $\epsilon \sim \mathcal{N}$
    Observe and store $(s_t, a_t, r_t, s_{t+1})$ in $B$
    if $s_{t+1}$ is terminal or ep. reached $max\_len$ then
        Set $s_{t+2} = s$ where s is a starting state.
        Sample $k$ experiences $(s_i, a_i, r_i, s'_i)$ from $B$
        for $1 \leqslant i \leqslant k$: do
            if Without Policy Inference then
                Collect $x = (o_1, ..., o_n)$
                $y_i = \begin{cases} r_i & \text{if end} \\ r_i + \lambda Q^{\text{cent.}}_{\Phi_{tg}}(x, (a_i)_{1 \leqslant i \leqslant n}) & \text{else} \end{cases}$
            end
            if With Policy Inference then
                Collect $x = o_1$
                $y_i = \begin{cases} r_i & \text{if end} \\ r_i + \lambda Q^{\text{cent.}}_{\Phi_{tg}}(x, (\hat{\mu}_i^j)_{1 \leqslant i \leqslant n}) & \text{else} \end{cases}$
                Update estimates by gradient descent :
                  $\Phi_i^j \leftarrow \phi_i^j + \gamma\,(\log\hat{\mu}_{ij} + \delta H(\hat{\mu}_{ij}))$ where $H$
                  is the entropy of the policy distribution.
            end
            Update critic $\Phi$ by gradient descent :
              $\Phi \leftarrow \Phi + \alpha\,(y_i - Q_\Phi(s_i, a_i))\,\nabla_\Phi Q_\Phi(s_i, a_i)$
            Update actor $\theta$ by gradient ascent :
              $\theta \leftarrow \theta + \beta\,\nabla_\theta Q_\Phi(s_i, \mu_\theta(s_i))$
        end
        Update targets : $\begin{cases} \Phi_{tg} \leftarrow \tau\Phi_{tg} + (1-\tau)\Phi \\ \theta_{tg} \leftarrow \tau\theta_{tg} + (1-\tau)\theta \end{cases}$
    end
end
```

**Algorithm 14:** MADDPG

# TD3

DDPG suffers from a common unstability which worsens the performance : in some cases it may overestimate the Q-function, leading to broken policies. Twin Delayed DDPG (TD3) implements three tricks to enhance DDPG's performance :

- **Clipped Double Q-Learning** : We train two Q-functions instead of one, and only keep the smaller one to compute the target.
- **Delayed Policy Update** : We update the policy less frequently than the Q-function (typically half as many times).
- **Target Policy Smoothing** : We add noise to the target action so that the Q-function is smoother.

```
Initialize $\theta, \Phi_1, \Phi_2, \theta_{tg}, \Phi_{1,tg}, \Phi_{2,tg} \in \mathbb{R}^n, B, s_1$
for $t \geqslant 1$ do
    Noisy Deterministic policy :
      $\mu_{noisy} = \text{clip}(\mu_\theta(s) + \epsilon, a_{low}, a_{high})$ where $\epsilon \sim \mathcal{N}$
    Observe and store $(s_t, a_t, r_t, s_{t+1})$ in $B$
    if $s_{t+1}$ is terminal or ep. reached $max\_len$ then
        Set $s_{t+2} = s$ where s is a starting state.
        Sample $k$ experiences $(s_i, a_i, r_i, s'_i)$ from $B$
        for $1 \leqslant i \leqslant k$: do
            $a'(s'_i) = \text{clip}(\mu_{\theta_{tg}}(s'_i) + \text{clip}(\epsilon, -c, c), a_{low}, a_{high})$
              where $\epsilon \sim \mathcal{N}(0, \sigma)$
            $y_i = \begin{cases} r_i & \text{if end} \\ r_i + \lambda \min_{j=1,2} Q_{\Phi_{j,tg}}(s'_i, a'(s'_i)) & \text{otherwise} \end{cases}$
            Update critics $\Phi_j (j = 1, 2)$ by gradient descent :
              $\Phi_j \leftarrow \Phi_j + \alpha(y_i - Q_{\Phi_j}(s_i, a_i))\nabla_{\Phi_j} Q_{\Phi_j}(s_i, a_i)$
            if $i$ mod $policy\_delay = 0$ then
                Update actor $\theta$ by gradient ascent :
                  $\theta \leftarrow \theta + \beta\,\nabla_\theta Q_{\Phi_1}(s_i, \mu_\theta(s_i))$
            end
        end
        Update targets : $\begin{cases} \Phi_{tg} \leftarrow \tau\Phi_{tg} + (1-\tau)\Phi \\ \theta_{j,tg} \leftarrow \tau\theta_{j,tg} + (1-\tau)\theta_j \end{cases}$
    end
end
```

**Algorithm 15:** TD3

# SAC

Soft Actor-Critic (SAC) optimizes a stochastic policy (like TRPO, PPO) in an off-policy fashion (using a DDPG-like strategy), making it a near-SOTA algorithm working with indinite episodes and continous action-spaces. It also uses clipped double Q-Learning, just like TD3. Consequently four neural networks are trained : the policy $\pi_\theta$, two Q-functions $Q_{\Phi_1}$ and $Q_{\Phi_2}$ and the V-function $V_\psi$. SAC uses two new tricks :

- **Entropy regularization** : We want to control the expected return / entropy (= randomness) trade-off, so that we pick an appropriate strategy for the exploration vs exploitation dilemma. To do that we add an entropy term in the reward, thus changing the return and subsequently V, Q and the Bellman's equations : $R \rightarrow R + \alpha \underset{x \sim \pi}{\mathbb{E}}[\pi(\cdot|s_t)]$. $\alpha$ controls the importance of the entropy term, and increasing it incentivizes exploitation at the expense of exploration.
- **Reparameterization trick** : When sampling actions from the policy distribution, we reparametrize the gaussian distribution using its mean and std (outputs of the nn), so that the distribution over actions (depending on the policy parameters) is replaced with a distribution over a gaussian noise. Moreover we use a squashing function (tanh) to bound sampled actions in a finite range :

$$\tilde{a} = \tilde{a}_\theta(s, \xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \cdot \xi\right), \xi \sim \mathcal{N}$$

```
Initialize $\theta, \Phi_1, \Phi_2, \psi, \psi_{tg} \in \mathbb{R}^n, B, s_1$
for $t \geqslant 1$ do
    Observe and store $(s_t, a_t, r_t, s_{t+1})$ in $B$, where $a_t \sim \pi_\theta(\cdot|s_t)$
    if $s_{t+1}$ is terminal or ep. reached $max\_len$ then
        Set $s_{t+2} = s$ where $s$ is a starting state.
        Sample $k$ experiences $(s_i, a_i, r_i, s'_i)$ from $B$
        for $1 \leqslant i \leqslant k$: do
            Compute Q-function target :
            $y_i^Q = \begin{cases} r_i & \text{if end} \\ r_i + \lambda V_{\psi_{tg}}(s'_i) & \text{otherwise} \end{cases}$
            Sample $\tilde{a} \sim \pi_\theta(\cdot|s_i)$
            Compute V-function target
            $y_i^V = \min_{j=1,2} Q_{\Phi_j}(s_i, \tilde{a}) + \alpha \log \pi(\tilde{a}|s)$
            Update Q-critics $\Phi_j$ by gradient descent :
            $\Phi_j \leftarrow \Phi_j + \alpha(y_i^Q - Q_{\Phi_j}(s_i, a_i))\nabla_{\Phi_j} Q_{\Phi_j}(s_i, a_i)$
            Update V-critic $\psi$ by gradient descent :
            $\psi \leftarrow \psi + \beta(y_i^V - V_\psi(s_i))\nabla_\psi V_\psi(s_i)$
            Sample $\tilde{a} \sim \pi_\theta(\cdot|s_i)$
            Update actor $\theta$ by gradient ascent :
            $\theta \leftarrow \theta + \gamma \nabla_\theta \left(Q_{\Phi_1}(s_i, \tilde{a}) - \eta \log \pi_\theta(\tilde{a}|s)\right)$
        end
        Update target : $\psi_{tg} \leftarrow \tau\psi_{tg} + (1 - \tau)\psi$
    end
end
```

**Algorithm 16:** SAC

## Visuals

### Standard algorithms

### Taxonomies of RL-algorithms

# Annex

## Maths behind TRPO

The KL-divergence represents the "proximity" of a distribution $P$ to a reference $Q$ :

$$D_{KL}(P||Q) = \int_x P(x)\log\left(\frac{P(x)}{Q(x)}\right)$$

Intuitively it is the expectation of the logarithmic difference between $P$ and $Q$, using the probabilities of $P$.

The theoretical TRPO update is :

$$\theta_{k+1} = \arg\max_{\theta} \ \mathcal{L}(\theta_k, \theta) \qquad \text{s.t. } \overline{D}_{KL}(\theta||\theta_k) \leq \delta$$

where $\mathcal{L}(\theta_k, \theta)$ is the **surrogate advantage**, and measures how $\pi_\theta$ performs relatively to $\pi_{\theta_k}$:

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

and $\overline{D}_{KL}(\theta||\theta_k)$ is an average KL-divergence:

$$\overline{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{E}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right]$$

The idea is to get a linear optimization problem using Taylor expansion :

$$\theta_{k+1} = \arg\max_{\theta} \ g^T(\theta - \theta_k) \qquad \text{s.t. } \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k) \leq \delta.$$

where $g$ is the gradient of $\mathcal{L}$ and $H$ the hessian matrix of $\overline{D}_{KL}$. Then, using the Lagrangian dual problem, we can solve it:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}}H^{-1}g.$$

This corresponds to the update of an algorithm called Natural Policy Gradient (NPG). But TRPO strictly ensures the KL constraint, which may not be satisfied due to Taylor approximations, by using **backtracking line search** :

$$\theta_{k+1} = \theta_k + \alpha^j\sqrt{\frac{2\delta}{g^T H^{-1} g}}H^{-1}g,$$

where $0 \leqslant \alpha \leqslant 1$ and $j$ is the smallest integer allowing $\pi_{\theta_{k+1}}$ to satisfy the KL constraint. Intuitively, if we went out of the trust region because of approximations, we take a few steps back, but still in the same direction.

Finally, since computing and storing $H^{-1}$ is extremely costly for large action-spaces, we use the **conjugate gradient method** to directly approximate the value of $H^{-1}g$. This method solves the linear problem $Hx = g$ and converges faster than gradient descent to do so, while only requiring function able to compute $Hx$ :

$$Hx = \nabla_\theta\left(\left(\nabla_\theta \overline{D}_{KL}(\theta||\theta_k)\right)^T x\right),$$