# Course Materials for GEN-AI

*Northeastern University*

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

**Instructor: Ramin Mohammadi**
r.mohammadi@northeastern.edu

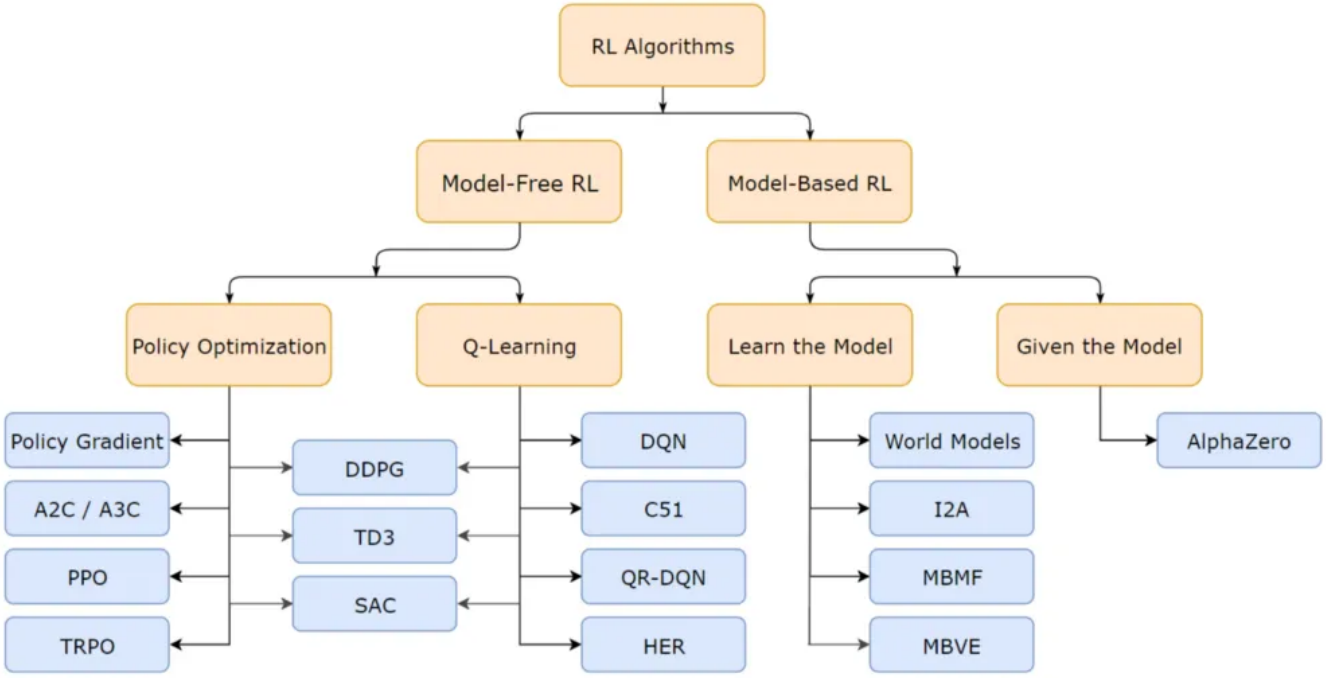
*Thank you for your understanding and collaboration.*

---

Content sourced from multiple references - see Reference section for details

# Reinforcement Learning

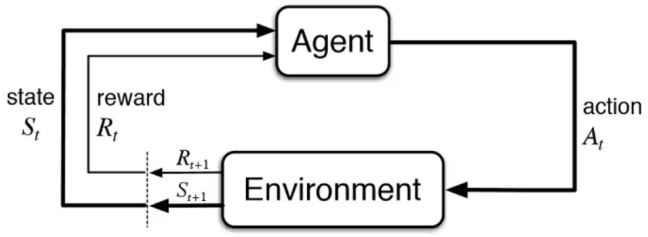
## 1 Introduction




Reinforcement Algorithms Model Types

### 1.1 Reinforcement Learning Definition (Wikipedia)

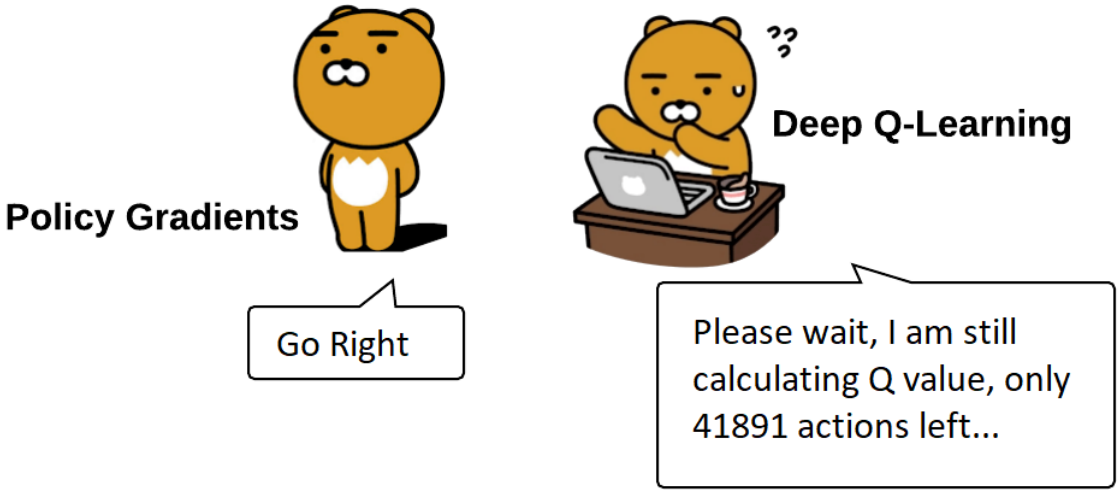
Reinforcement learning is an area of machine learning inspired by behavioral psychology, concerned with how software agents take actions in an environment to maximize some notion of cumulative reward.




- Positive Reinforcement: e.g., Food.
- Negative Reinforcement: e.g., Hunger.

## 2 Policy Gradients

In contrast to action-value methods which learn the values of actions based on their estimated action values and select actions accordingly, this discussion introduces methods that learn a parameterized policy capable of selecting actions independently of a value function. Although a value function can still be utilized to learn the policy parameters, it is not essential for action selection.




We define the policy with parameters $\theta \in \mathbb{R}^d$ as $\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$, representing the probability of choosing action $a$ in state $s$ at time $t$ with policy parameters $\theta$. If a learned value function is also employed, its weight vector is denoted by $w \in \mathbb{R}^d$, and used in the function $\tilde{v}(s,w)$.

The focus is on methods for learning the policy parameters $\theta$ by leveraging the gradient of a scalar performance measure $J(\theta)$, which quantifies the policy's effectiveness.

Policy gradient algorithms try to solve the optimization problem

$$\max_{\theta} J(\pi_\theta) \doteq \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$$

Where:

- $\pi_\theta$: The policy parameterized by $\theta$.
- $\tau = \{(s_0,a_0),(s_1,a_1),\dots\}$: A trajectory sampled from the policy.
- $s_t$: The state at time $t$.
- $a_t$: The action taken at time $t$ based on $\pi_\theta(a_t \mid s_t)$.
- $r(s_t,a_t)$: The reward received after taking action $a_t$ in state $s_t$.
- $\gamma \in [0,1)$: The discount factor, which determines the importance of future rewards.

These methods aim to maximize $J(\theta)$, adjusting $\theta$ using a gradient ascent approach, approximated by:

$$\theta_{t+1} = \theta_t + \alpha\nabla J(\theta_t),$$

where $\alpha$ is the learning rate, enhancing the policy's performance incrementally. by taking stochastic gradient ascent on the policy parameters $\theta$, using the *policy gradient*

1. Action-Value Function:

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[ \sum_{k=t}^{\infty} \gamma^{k-t} r_k | s_t, a_t \right],$$

2. State-Value Function:

$$V_\pi(s_t) = \mathbb{E}_\pi \left[ \sum_{k=t}^{\infty} \gamma^{k-t} r_k | s_t \right],$$

$$\text{where} \quad a_t \sim \pi(a_t \mid s_t),\ s_{t+1} \sim P(s_{t+1} \mid s_t, a_t) \quad \text{for } t \geq 0.$$

The term $\nabla J(\theta_t) \in \mathbb{R}^d$ represents a stochastic estimate that approximates the gradient of the performance measure with respect to its parameter $\theta_t$. Methods adhering to this approach are termed *policy gradient methods*. These methods focus on optimizing the policy parameter $\theta$, irrespective of whether they utilize an approximate value function.

**Limitations of policy gradients:**

- Sample efficiency is poor
  - Because recycling old data to estimate policy gradients is hard
- Distance in parameter space $\neq$ distance in policy space!
  - What is policy space? For the tabular case, the set of matrices

$$\Pi = \left\{ \pi : \pi \in \mathbb{R}^{|S| \times |A|}, \sum_a \pi_{sa} = 1,\ \pi_{sa} \geq 0 \right\}$$

- Policy gradients take steps in parameter space
  - Step size is hard to get right as a result

Moreover, methods that develop approximations for both the policy and the value functions are known as *actor-critic methods*. In this context, the 'actor' refers to the component that learns the policy, and the 'critic' refers to the component that learns the value function, typically a state-value function.

## 2.1 Policy Approximation:

In policy gradient methods, policies are parameterized as $\pi(a \mid s, \theta)$, where $\theta \in \mathbb{R}^d$, and the policy is differentiable with respect to $\theta$. This ensures that the gradient $\nabla_\theta \pi(a \mid s, \theta)$ can be computed for all states $s$ and actions $a$.

To maintain sufficient exploration and avoid deterministic policies, $\pi(a \mid s, \theta)$ is typically constrained such that $\pi(a \mid s, \theta) \in (0, 1)$, often achieved by parameterizing the policy with a softmax function or using distributions like Gaussian for continuous action spaces.

$$\pi(\theta, a, s) = \frac{e^{h(s,a;\theta)}}{\sum_{a'} e^{h(s,a';\theta)}} \tag{1}$$

- Parameterizing policies with a softmax function allows approximate policies to approach deterministic behavior more naturally compared to $\epsilon$-greedy methods, which inherently maintain an $\epsilon$ probability of selecting a random action. In the softmax approach, action-value estimates guide action selection probabilities that can potentially converge to values allowing for deterministic decisions as preferences for the optimal actions are infinitely favored over suboptimal ones.

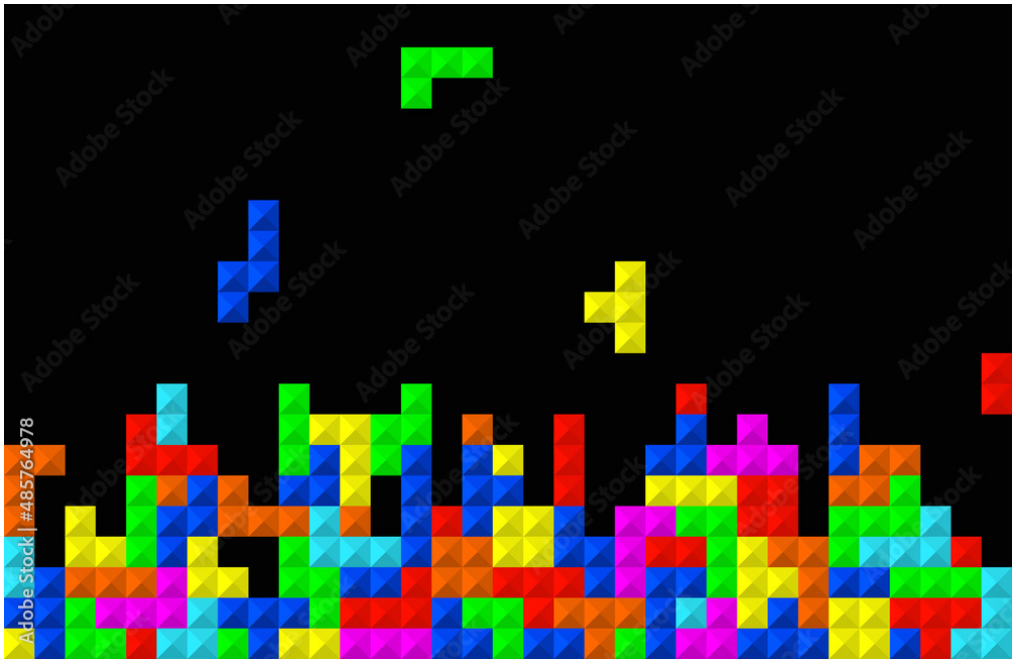
**Example:** In a robot navigation task, where a robot must choose paths based on their safety and speed, the softmax approach can initially allow exploration of various paths but gradually focus more on the safest and quickest routes as it learns their true value, eventually behaving deterministically by always choosing the best known path.

- Furthermore, softmax parameterization facilitates the selection of actions with arbitrary probabilities, which is crucial in environments that require nuanced decision-making strategies. This flexibility is missing in traditional action-value methods, which typically aim for the highest expected value without the ability to model complex probabilities required for optimal stochastic policies.

**Example:** In poker, a game with incomplete information and high stakes, using softmax parameterization allows a player model to adapt its play style by varying the probabilities of bluffing, betting, or folding based on the learned preferences of opponent behavior and hand strength. This dynamic approach enables the player to handle a range of game situations more effectively than simply choosing the action with the highest expected return.

- One notable advantage of policy parameterization over action-value methods is that the policy may be simpler to approximate. The complexity of policy and action-value functions can vary; in some cases, the action-value function is simpler and thus easier to approximate, whereas in other situations, the policy itself is simpler. For scenarios where the policy is simpler, policy-based methods often learn faster and can achieve a superior asymptotic policy.

**Example:** In games like Tetris, policy-based methods can rapidly learn effective strategies for placing tetrominoes to clear lines efficiently, outperforming methods that need to approximate complex action-value functions for each possible piece placement and rotation.




Moreover, the choice of policy parameterization is a strategic method to incorporate prior knowledge about the desired form of the policy into the reinforcement learning system. This is often crucial for guiding the learning process towards more practical and theoretically sound strategies.

**Example:** In autonomous driving, prior knowledge about traffic laws and safe driving practices can be encoded directly into the policy structure, guiding the learning process to focus not only on efficiency but also on compliance and safety.

## 2.2 Policy Gradient Algorithm

The Policy Gradient Theorem offers a solid theoretical foundation for the efficient computation of gradients needed for policy optimization, emphasizing the advantages of policy parameterization:

- Advantages:
  - Finds the best *Stochastic Policy* (Optimal Deterministic Policy, produced by other RL algorithms).
  - Naturally *explores* due to Stochastic Policy representation.
  - Effective in high-dimensional or continuous action spaces.
  - Small changes in $\theta \implies$ small changes in $\pi$, and in state distribution.
  - This avoids the convergence issues seen in argmax-based algorithms.
- Disadvantages:
  - Typically converge to a local optimum rather than a global optimum.
  - Policy Evaluation is typically inefficient and has high variance.
  - Policy Improvement happens in small steps $\implies$ slow convergence.

- **Policy Gradient:**
  - Provides an analytic expression for the gradient of performance with respect to the policy parameter $\theta$, simplifying the computation of the gradient for gradient ascent.

$$J(\pi_\theta) \doteq \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$$

$$J(\pi_\theta) \doteq \sum_{t=0}^{\infty}\gamma^t \mathbb{E}_{\tau\sim\pi_\theta}[r_t]$$

$$J(\theta) = \sum_{s\in S}\mu^{\pi_\theta}(s)V^{\pi_\theta}(s) = \sum_{s\in S}\mu^{\pi_\theta}(s)\sum_{a\in A}\pi_\theta(a \mid s)Q^{\pi}(s,a),$$

  - $\mu^{\pi_\theta}(s) = \lim_{t\to\infty} P(s_t = s \mid s_0, \pi_\theta)$ where $\mu(s)$ denotes the state distribution under policy $\pi$ which is the probability that $s_t = s$ when starting from $s_0$ and following policy $\pi_\theta$ for $t$ steps.
  - It is natural to expect policy-based methods are more useful in the continuous space. Because there is an infinite number of actions and (or) states to estimate the values for and hence value-based approaches are way too expensive computationally in the continuous space.
  - state distribution under policy denotes the probability of reaching different states in an environment when following a policy that is being updated using a policy gradient algorithm.

    **Example:** Consider a 2x2 grid where an agent starts in the top-left corner (state $s_0$) and can move right or down based on a softmax policy. The probabilities of the agent's movements are determined by action preferences, which are set as follows: moving right has higher preference from $s_0$ and moving down from the right-top corner ($s_1$). Here, we show the hypothetical state distribution $\mu(s)$ as an array:

| State | $\mu(s)$ |
|---|---|
| $s_0$ | 0.4 |
| $s_1$ | 0.3 |
| $s_2$ | 0.2 |
| $s_3$ | 0.1 |

    Here, $s_0$ is the starting state (top-left corner), $s_1$ is top-right, $s_2$ is bottom-left, and $s_3$ is bottom-right. The probabilities reflect the likelihood of the agent being in each state under the softmax policy that favors moving right from $s_0$ and $s_1$ and down from $s_2$.

- **Policy Gradient Theorem:**
  - The gradient is given by:

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} \mu^\pi(s) \sum_{a \in A} \nabla_\theta \pi_\theta(a \mid s) Q^\pi(s, a),$$

  - Note: $\mu^\pi(s)$ depends on $\theta$, but there's no $\nabla_\theta \mu^\pi(s)$ term in $\nabla_\theta J(\theta)$.
  - So we can simply sample simulation paths, and at each time step, we calculate $(\nabla_\theta \log \pi(s, a; \theta)) \cdot Q^\pi(s, a)$ (probabilities implicit in paths).
  - Note: $\nabla_\theta \log \pi(s, a; \theta)$ is the Score function (Gradient of log-likelihood).
  - We will estimate $Q^\pi(s, a)$ with a function approximation $Q(s, a; w)$.
  - This numerical estimate of $\nabla_\theta J(\theta)$ enables **Policy Gradient Ascent**.
  - **Challenges and Adaptations:**
    * Adjusting the policy parameter affects both action selections and the distribution of states.
    * Policy parameterization allows for incorporating structural insights into the policy, facilitating effective learning.

## 2.3 Learning Approaches

Policy gradients can optimize the policy using both **on-policy** and **off-policy learning** approaches.

**On-policy learning:**

The agent samples actions based on its current policy and uses the rewards from these actions to directly update the policy parameters.

**Off-policy learning:**

The agent learns from a separate "behavior policy," which may explore more widely, and uses this data to update its own policy.

## Key Points about Policy Gradients:

- **Direct policy optimization:** Unlike value-based methods, policy gradients directly learn the policy function. The policy function maps states to actions and optimizes its parameters to maximize expected rewards.
- **Gradient descent:** The policy is updated by following the gradient of the expected return, moving towards actions that lead to higher rewards.
- **Model-free approach:** Policy gradients can be utilized in environments where the full state transition dynamics are unknown.

# Overview of Policy Gradient Methods

### 1- REINFORCE

REINFORCE is a fundamental policy gradient algorithm that estimates the policy gradient using the likelihood ratio method. It is often considered the foundational approach for other policy gradient techniques. This method directly utilizes entire episodes of interaction to update the policy based on the returns achieved, thereby adjusting the policy parameters in the direction of higher cumulative rewards.

### 2- Actor-Critic

The Actor-Critic method enhances the basic policy gradient approach by integrating two components:

- The **Actor** updates the policy distribution in the direction suggested by the Critic.
- The **Critic** evaluates the action taken by the Actor by computing the value function, which estimates the expected future rewards.

This combination allows for more stable learning by providing more consistent updates based on better reward estimates.

### 3- Proximal Policy Optimization (PPO)

PPO is a widely-used modern policy gradient algorithm that incorporates a clipped objective function to constrain the extent of policy updates. This mechanism effectively prevents excessively large updates to the policy, which can lead to destabilization of the learning process. PPO aims to take the biggest possible improvement step on policy while avoiding excessive updates that could result in performance degradation.

# 3 REINFORCE: (On Policy)

The Reinforce algorithm (REINFORCE), also called Monte-Carlo policy-gradient, is a policy-gradient algorithm that uses an estimated return from an entire episode to update the policy parameter $\theta$.

**Algorithm 1** REINFORCE Algorithm

**Require:** Initial policy parameters $\theta \in \Re^d$ (e.g., to 0)
**Ensure:** Learned policy $\pi_\theta$

**Input:** a dierentiable policy parameterization $\pi(a|s, \theta)$
**Parameters:** Learning rate $\alpha > 0$, discount $\gamma \in [0, 1]$
**while** not converged **do**
  Generate episode $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T)$ following $\pi_\theta$ (Monte-Carlo sampling)
  **for** $t = 0, 1, \ldots, T$ **do**
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \log \pi(a_t|s_t, \theta)$
  **end for**
**end while**
**return** $\pi_w$ =0

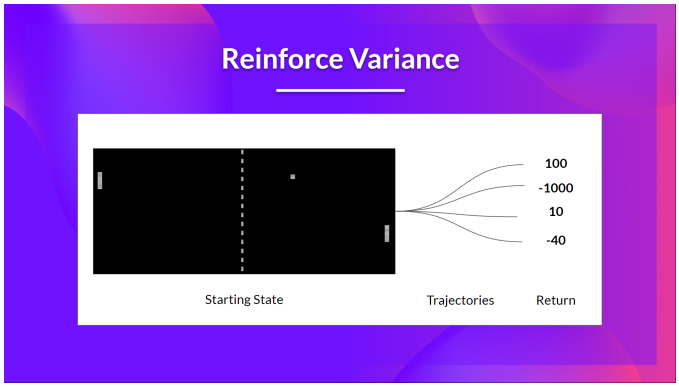
## Advantages

- Simple implementation: REINFORCE is straightforward to implement as it does not require complex architectures or bootstrap estimates.

- Bias-free: Since the updates are based on complete episodes, the gradient estimate is unbiased.

### Disadvantages

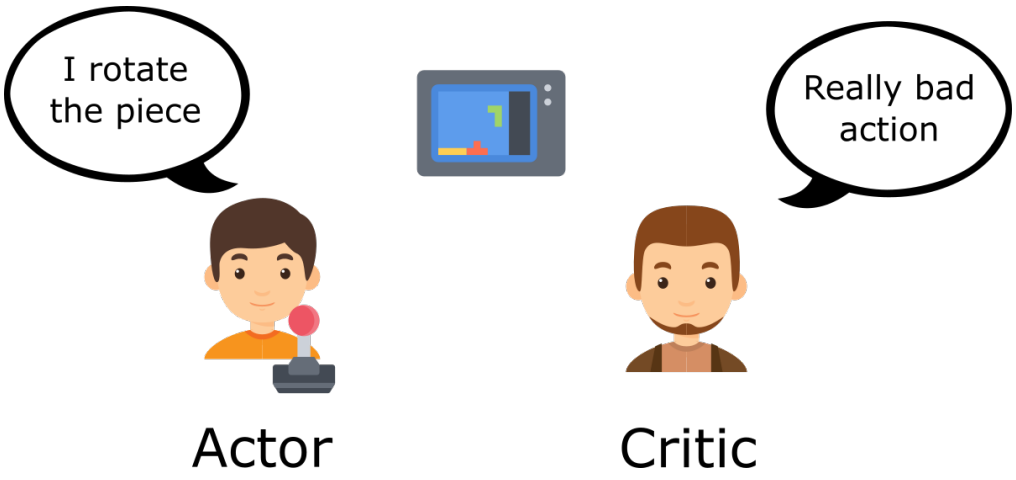
- High variance: The algorithm suffers from high variance in its gradient estimates due to Monte Carlo sampling, which can lead to unstable training.




  Since trajectories can lead to different returns due to stochasticity of the environment (random events during episode) and stochasticity of the policy. Consequently, the same starting state can lead to very different returns. Because of this, the return starting at the same state can vary significantly across episodes.

- Inefficiency: Needs full episodes before updates can be made, making it less efficient compared to methods that bootstrap.

## 4 Actor-Critic Method: (On Policy)




The Actor-Critic method is a type of Reinforcement Learning algorithm that combines the concepts of policy gradient (actor) and value function approximation (critic). It aims to balance the policy learning with value estimation to create a more stable and efficient learning process.

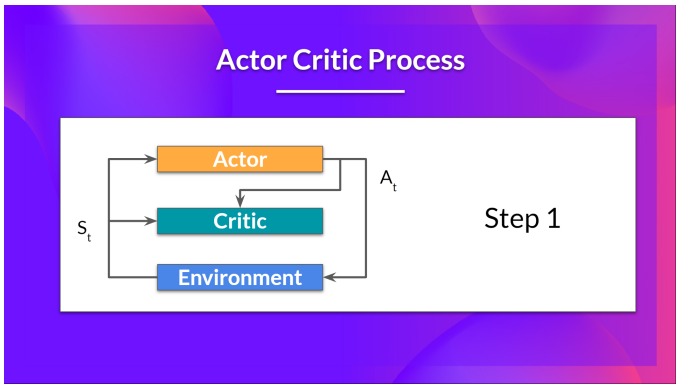
Actor-Critic method aims to reducing the variance of Reinforce algorithm and training our agent faster and better by using a combination of policy-based and value-based methods.

### Algorithm Structure

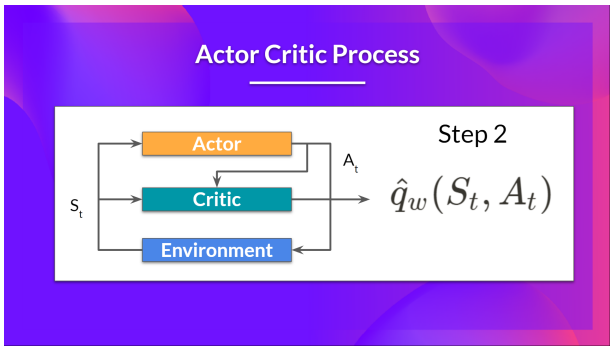
The Actor-Critic architecture involves two main components:

- **Actor**: Responsible for learning the policy function $\pi(a|s, \theta)$, which maps states to a probability distribution over actions.
- **Critic**: Estimates the value function $V(s, v)$ or the action-value function $Q(s, a, w)$, which assesses the quality of the actions taken by the actor. Where v and w are parameters of both models.

We don't know how to play at the beginning, so we try some actions randomly.

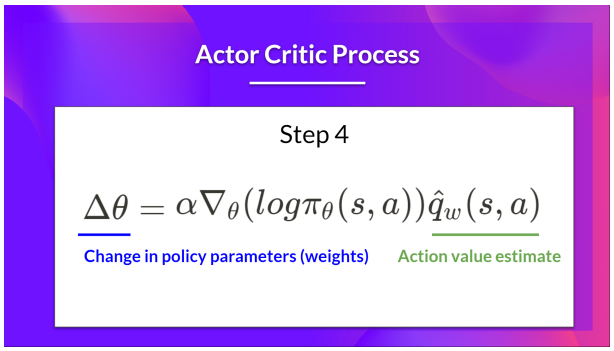



The Critic takes that action also as input and, using $S_t$ and $A_t$, computes the value of taking that action at that state: the Q-value.

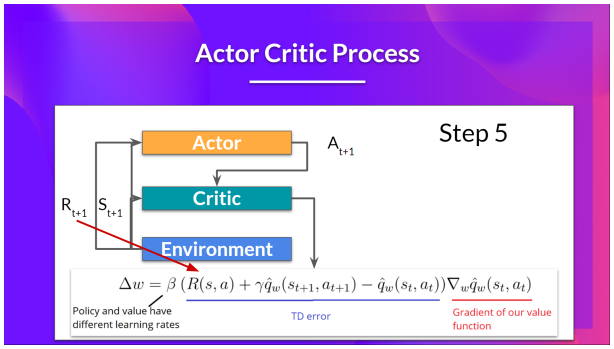



The action $A_t$ performed in the environment outputs a new state $S_{t+1}$ and a reward $R_{t+1}$.


Learning from this feedback, we'll update your policy and be better at playing that game.




On the other hand, our friend (Critic) will also update their way to provide feedback so it can be better next time.



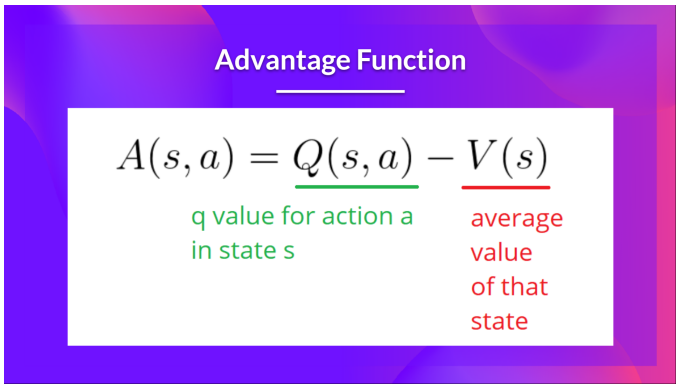

This is the idea behind Actor-Critic. We learn two function approximations:

- A policy that controls how our agent acts: $\pi(a|s,\theta)$
- A value function to assist the policy update by measuring how good the action taken is: $\hat{q}_w(s,a)$

## Using Advantage Function

> We can stabilize learning further by using the Advantage function as Critic instead of the Action value function.The idea is that the Advantage function calculates how better taking that action at a state is compared to the average value of the state.

It's subtracting the mean value of the state from the state-action pair:




In other words, this function calculates the extra reward we get if we take this action at that state compared to the mean reward we get at that state.

The extra reward is what's beyond the expected value of that state.

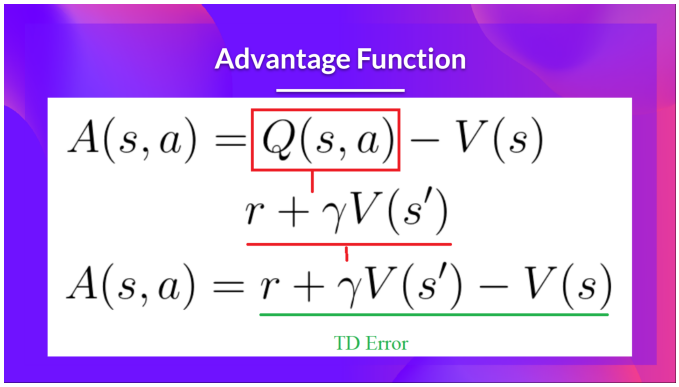
- If $A(s,a) > 0$: our gradient is pushed in that direction.
- If $A(s,a) < 0$ (our action does worse than the average value of that state), our gradient is pushed in the opposite direction.

$$J(\theta) = \sum_{s \in S} \mu^{\pi_\theta}(s) V^{\pi_\theta}(s) = \sum_{s \in S} \mu^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a \mid s) \cdot A(s,a,w,v),$$

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} \mu^{\pi}(s) \sum_{a \in A} \nabla_\theta \pi_\theta(a \mid s) \cdot A(s,a,w,v),$$

## 4.1 ACTOR-CRITIC-TD-ERROR

The problem with implementing this advantage function is that it requires two value functions — $Q(s,a)$ and $V(s)$. Fortunately, we can use the TD error as a good estimator of the advantage function.




This approach requires only one set of critic parameters v.

**Algorithm 2** ACTOR-CRITIC-TD-ERROR(·)

Initialize Policy params $\theta \in \mathbb{R}^m$ and State VF params $v \in \mathbb{R}^n$ arbitrarily
**for** each episode **do**
  Initialize $s$ (first state of episode)
  $P \leftarrow 1$ (discount factor)
  **while** $s$ is not terminal **do**
    $a \sim \pi(s, \cdot; \theta)$
    Take action $a$, observe $r, s'$
    $\delta \leftarrow r + \gamma V(s'; v) - V(s; v)$
    $v \leftarrow v + \alpha_v \cdot \delta \cdot \nabla_v V(s; v)$
    $\theta \leftarrow \theta + \alpha_\theta \cdot P \cdot \delta \cdot \nabla_\theta \log \pi(s, a; \theta)$
    $P \leftarrow \gamma P$
    $s \leftarrow s'$
  **end while**
**end for**=0

# 5 Trust Region Policy Optimization (TRPO) (On Policy)

As we saw, Policy gradient algorithms try to solve the optimization problem

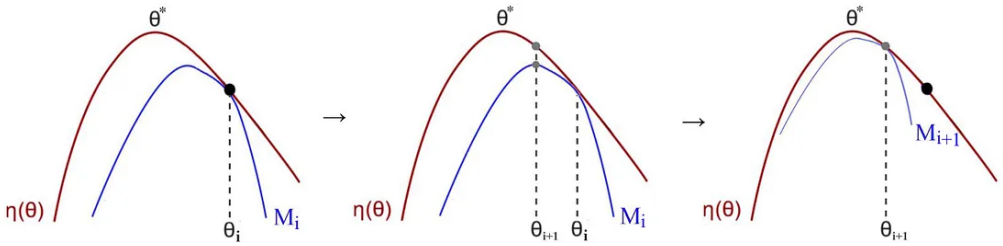
$$\theta J(\pi_\theta) \doteq \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$$

$$J(\theta) = \sum_{s\in S}\mu^{\pi_\theta}(s)V^{\pi_\theta}(s) = \sum_{s\in S}\mu^{\pi_\theta}(s)\sum_{a\in A}\pi_\theta(a \mid s)\cdot A(s,a,w,v),$$

Advantage Function:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

## Refresher: How can we optimize a policy to maximize the rewards?




The Minorize-Maximization (MM) algorithm iteratively maximizes a lower bound function $M$ (represented by the blue line below) that locally approximates the expected reward $\eta$.

In order to solve:

- First, we start with an initial policy guess and construct a lower bound $M$ for $\eta$ at this policy.
- We then optimize $M$ and use its optimal policy as the next guess.
- At the new policy estimate, we approximate a new lower bound and repeat the iterations until convergence.
- To ensure the effectiveness of this approach, the lower bound function $M$ must be easier to optimize than $\eta$.

There are two major optimization methods: line search (such as gradient descent) and the trust region approach.

### Line Search

Gradient descent is an easy, fast, and simple method for optimizing an objective function. This is why it remains widely used in deep learning, even though more accurate methods are available.

Line search first selects the steepest descent direction and then moves forward by a step size. However, this strategy can fail in reinforcement learning (RL). Consider a robot hiking up *Angels Landing*. As illustrated below, the ascent is determined by selecting the direction first. If the step size is too small, reaching the peak will take an impractically long time. Conversely, if the step size is too large, the robot may fall off a cliff. Even if the robot survives the fall, it will land in a region significantly lower than its previous position.

Since policy gradient methods are primarily on-policy, they select actions from the current state. As a result, after a fall, exploration resumes from a poor state with a suboptimal policy, severely degrading performance.

**Trust Region:**

In the trust region approach, we first define the maximum step size that we are willing to explore (represented by the yellow circle above). Then, we identify the optimal point within the trust region and resume the search from there.

$$J(\theta) = \sum_{s \in S} \mu^{\pi_{\text{old}}}(s) \sum_{a \in A} \pi_\theta(a \mid s) \hat{A}_{\text{old}}(s, a)$$

$$= \sum_{s \in S} \mu^{\pi_{\text{old}}}(s) \sum_{a \in A} \left( \frac{\pi(a \mid s)}{\pi(a \mid s)} \pi_\theta(a \mid s) \hat{A}_{\text{old}}(s, a) \right) \quad \text{(importance sampling)}$$

$$= \mathbb{E}_{s\sim\mu^{\pi_{\theta_{\text{old}}}},a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)}\hat{A}_{\text{old}}(s,a)\right].$$

where $\hat{A}$ is stimated advantage rather than the true advantage function because the true rewards are usually unknown.

**What is the maximum step size in a trust region?**

What determines the maximum step size in a trust region? In this method, we begin with an initial guess and may dynamically adjust the region size. For instance, if the divergence between the new and current policy increases significantly, we can shrink the region (or expand it otherwise). To avoid poor decisions, it is beneficial to reduce the trust region when the policy undergoes excessive changes.

In TRPO, we limit how far we can change our policy in each iteration through the KL-divergence:

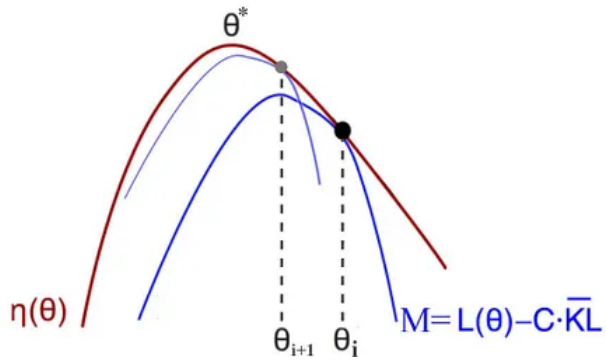
- KL divergence quantifies how much the new policy new policy differs from the old policy.
- A small KL divergence means the policies are similar, while a large KL divergence indicates significant changes.
- TRPO enforces a trust region by constraining the average KL divergence between the new and old policies to ensure stability.

$$D_{KL}(\pi_{\theta_{\text{old}}}(.|s) \,\|\, \pi_{\theta_{\text{new}}}(.|s)) = \mathbb{E}_{s\sim\mu_{\pi_{\text{old}}}}\left[\log\frac{\pi_{\theta_{\text{old}}}(.|s)}{\pi_{\theta_{\text{new}}}(.|s)}\right]$$

$$J(\theta) = \mathbb{E}_{s\sim\mu^{\pi_{\theta_{\text{old}}}},a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)}\hat{A}_{\text{old}}(s,a)\right].$$

$$M = J(\theta) - C.D_{KL}(\pi_{\theta_{\text{old}}}(.|s) \,\|\, \pi_{\theta_{\text{new}}}(.|s)) \text{ (Lower bound)}$$

Given that M is the lower bound:




here J is the expected advantage function; the expected rewards minus a baseline $V(s)$. We use the advantage function instead of the expected reward because it reduces the variance of the estimation.

$$\text{maximize}_\theta\, \mathbb{E}_t\left[\frac{\pi_{\theta_{\text{new}}}(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right]$$

$$\text{subject to} \quad \mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}}\left[D_{\text{KL}}\big(\pi_{\text{old}}(\cdot \mid s) \,\|\, \pi_{\text{new}}(\cdot \mid s)\big)\right] \le \delta$$

**Equivalent of:**

$$\text{maximize}_\theta\, \mathbb{E}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)}\hat{A}_t\right] - \beta\mathbb{E}_t\left[\text{KL}\left(\pi_{\text{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right)\right].$$

**Algorithm 1** Policy iteration algorithm guaranteeing non-decreasing expected return $\eta$

Initialize $\pi_0$.
**for** $i = 0, 1, 2, \ldots$ until convergence **do**
Compute all advantage values $A_{\pi_i}(s, a)$.
Solve the constrained optimization problem

$$\pi_{i+1} = \arg\max_{\pi} \left[ L_{\pi_i}(\pi) - C D_{\mathrm{KL}}^{\max}(\pi_i, \pi) \right]$$

where $C = 4\epsilon\gamma/(1-\gamma)^2$

and $L_{\pi_i}(\pi) = \eta(\pi_i) + \sum_s \rho_{\pi_i}(s) \sum_a \pi(a|s) A_{\pi_i}(s, a)$

**end for**

# 6 Proximal Policy Optimization (PPO) (On Policy)

PPO is a policy gradient method that addresses the key challenges of training stability and sample efficiency in reinforcement learning. It modifies the standard policy gradient objective to prevent too large policy updates, thus maintaining training stability. To do this, it uses a ratio that indicates the difference between our current and old policy and clip this ratio from a specific range $[1 - \epsilon, 1 + \epsilon]$.

We want to avoid having too large policy updates, for two reasons:

1. We know empirically that smaller policy updates during training are more likely to converge to an optimal solution.
2. A too big step in a policy update can result in falling "off the cliff" (getting a bad policy) and having a long time or even no possibility to recover.

So with PPO, we update the policy conservatively. To do so, we need to measure how much the current policy changed compared to the former one using a ratio calculation between the current and former policy. And we clip this ratio in a range $[1 - \epsilon, 1 + \epsilon]$, meaning that we remove the incentive for the current policy to go too far from the old one (hence the proximal policy term).

## Algorithm Overview

PPO aims to take the largest possible step to improve its policy while ensuring the new policy is not too far from the old one. This is achieved by utilizing a clipped objective function, which limits the update at each step using:

$$L(s, a, \theta_k, \theta) = \min\left(r_t(\theta)A^{\theta_k}(s, a),\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ (ratio function) It's the probability of taking action $a_t$ at state $s_t$ in the current policy divided by the previous one.
  - If $r_t(\theta) > 1$, the action $a_t$ at state $s_t$ is more likely in the current policy than in the old policy.
  - If $r_t(\theta)$ is between 0 and 1, the action is less likely for the current policy than for the old one.

  So, this probability ratio is an easy way to estimate the divergence between the old and current policy.
- $A^{\theta_k}(s, a)$ is the advantage estimate at time $t$ which can be calculated using:

$$\begin{aligned} A^{\theta_k}(s, a) &= Q(s, a) - V(S) \\ &= r + \gamma V(s') - V(s) \end{aligned}$$

- $\epsilon$ is a hyperparameter, typically set to 0.1 or 0.2, which determines the bounds for clipping.

## Advantage Function Behavior:

**Advantage is positive:** Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, (1+\epsilon)\right) A^{\theta_{old}}(s, a).$$

Because the advantage is positive, the objective will increase if the action becomes more likely—that is, if $\pi_\theta(a|s)$ increases. But the min in this term puts a limit to how much the objective can increase.
Once $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_{old}}(a|s)$, the min kicks in and this term hits a ceiling of $(1+\epsilon)A^{\theta_{old}}(s, a)$. Thus: the new policy does not benefit by going far away from the old policy.

**Advantage is negative:** Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to:

$$L(s, a, \theta_k, \theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}(1-\epsilon)\right) A^{\theta_{old}}(s, a)$$

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if $\pi_\theta(a|s)$ decreases. But the max in this term puts a limit to how much the objective can increase.
Once $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_{old}}(a|s)$, the max kicks in and this term hits a ceiling of $(1-\epsilon)A^{\theta_{old}}(s, a)$. Thus, again: the new policy does not benefit by going far away from the old policy.

## Pros

- **Training Stability**: The clipping mechanism effectively limits the policy update, which helps in maintaining stability during training.
- **Sample Efficiency**: PPO can reuse data multiple times through multiple epochs of stochastic gradient descent on the same batch of data, improving sample efficiency.

## Cons

- **Tuning Required**: Finding the right balance for the clipping parameter $\epsilon$ and other hyperparameters can be challenging.
- **Complexity in Implementation**: The algorithm's structure, involving multiple mini-batch updates per data sample, can be complex to implement correctly.

## Example

Imagine a robot learning to navigate through a complex maze. PPO would allow the robot to explore various paths and quickly learn effective strategies without making drastic changes in behavior that could lead to unstable training outcomes.

**Algorithm 1** PPO-Clip

Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
**for** $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

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

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

# References

[1] Ali Ghodsi. *DataScienceCoursesUW*.

[2] HuggingFace. “q-learning-recap.”.