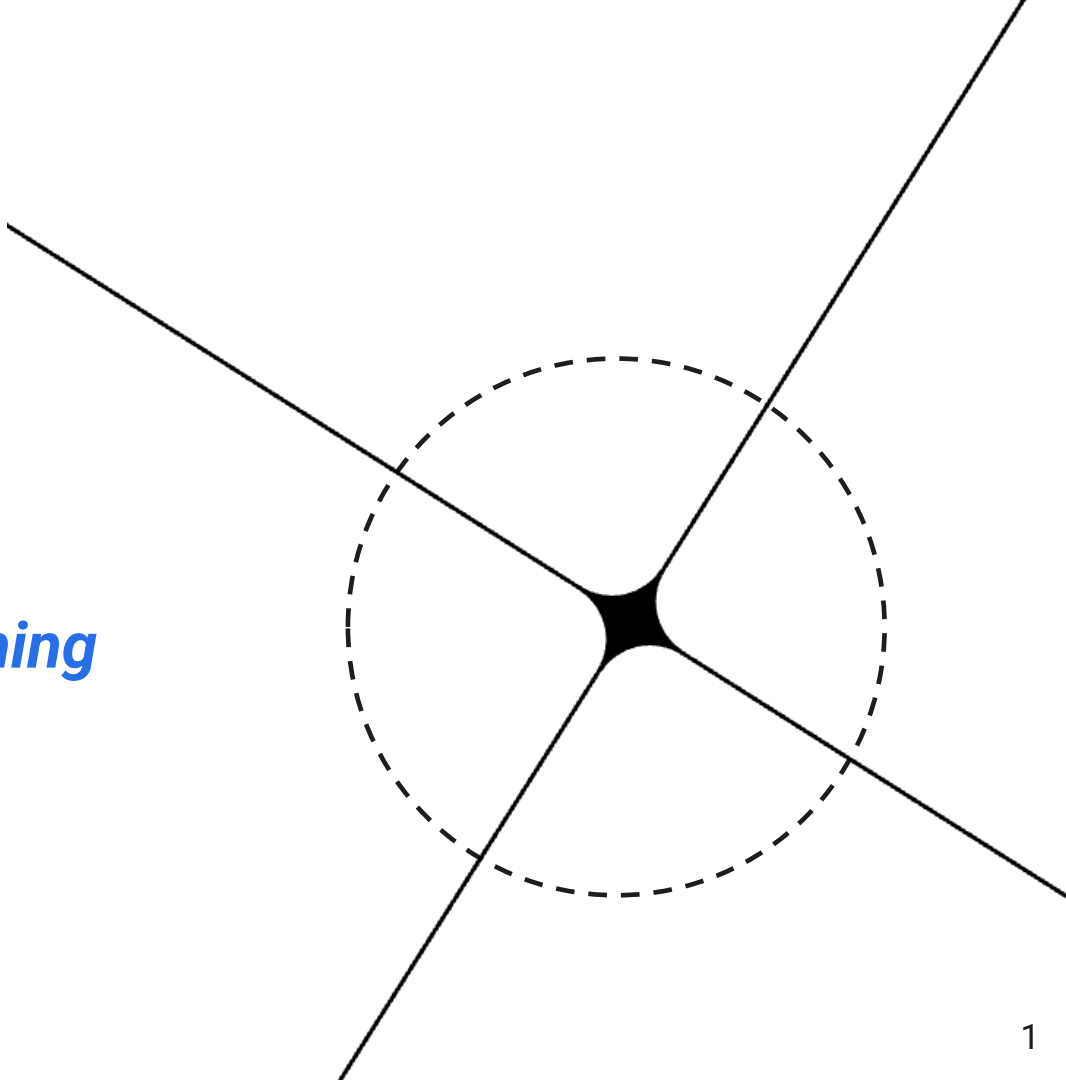
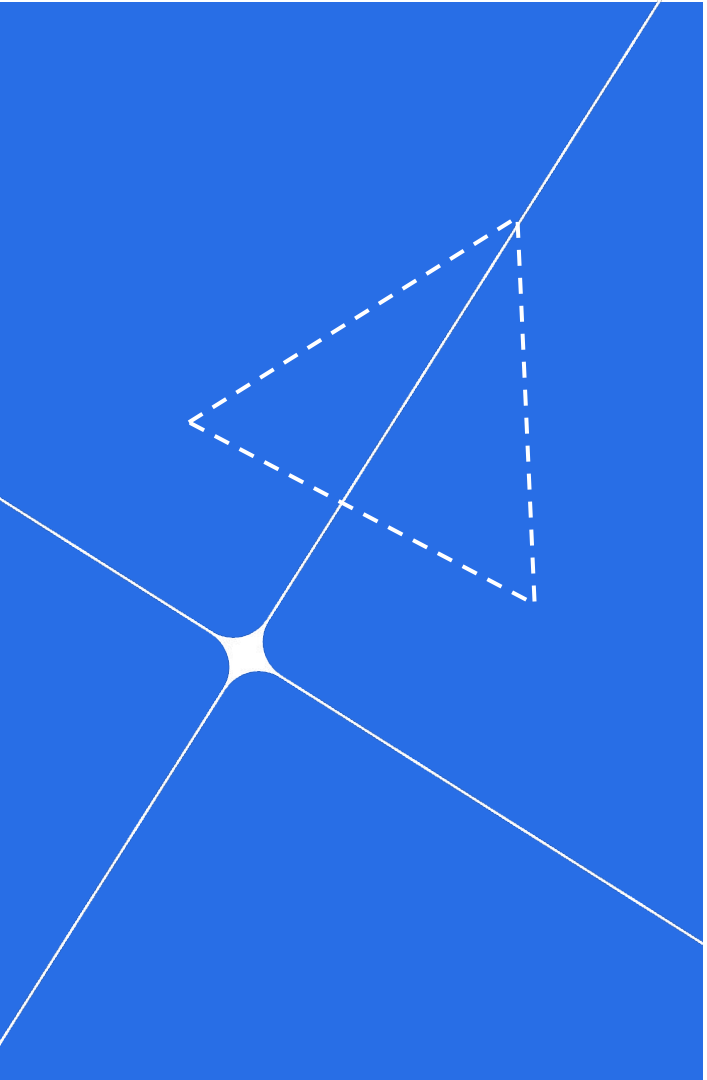
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis



Deep Reinforcement Learning

Julien Romero



Introduction et Intuition

Introduction au Deep Reinforcement Learning (Deep RL)

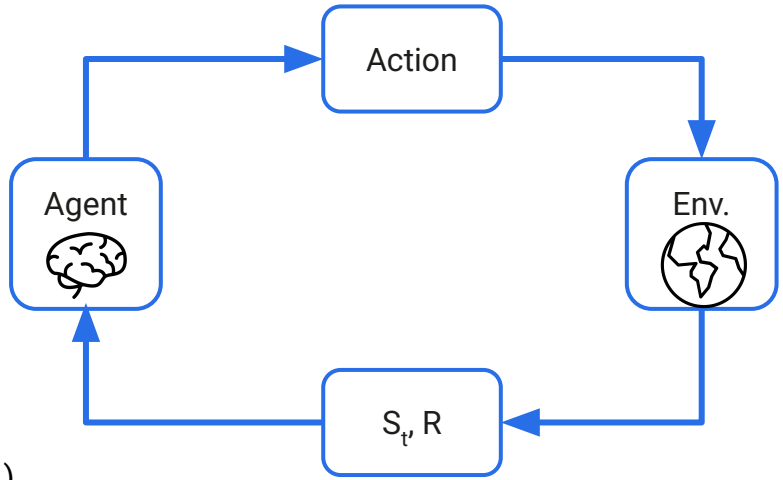
- **Contexte du cours :** Après la vision, l'audio et la génération, nous abordons la **prise de décision**.
- **Motivation :** Jusqu'ici, nos modèles étaient des *observateurs* passifs (classification, régression) ou des créateurs statiques (GenAI).
- **Le saut conceptuel :** En RL, le modèle devient un **Agent**. Il agit sur le monde et en subit les conséquences.
- **Définition :** Apprendre à quoi faire (mapper des situations à des actions) pour maximiser un signal de récompense numérique.
- **Pourquoi "Deep" ?**
  - RL Classique (tabulaire) : Limité à de petits espaces d'états (ex: Tic-Tac-Toe).
  - Deep RL : Utilise des réseaux de neurones pour traiter des entrées complexes (Vision, LiDA, Séries temporelles) et généraliser.
- **Objectif ultime :** Créer des systèmes autonomes capables de résoudre des tâches complexes sans supervision explicite étape par étape.

Le Changement de Paradigme : Supervised vs RL

- **Motivation :** Comprendre pourquoi vos techniques habituelles de gradient descent ne s'appliquent pas directement.
- **Apprentissage Supervisé (Ce que vous connaissez) :**
  - Données : (x,y) étiquetées par un expert.
  - Objectif : Minimiser l'erreur entre la prédiction ŷ et la vérité y.
  - L'agent est "guidé" à chaque exemple (Teacher forcing).
- **Reinforcement Learning (Ce cours) :**
  - Données : Pas de labels y. Seulement un signal scalaire de récompense R (parfois nul, parfois retardé).
  - Interaction : L'agent génère ses propres données en explorant.
- **Feedback :** On ne dit pas à l'agent *ce qu'il aurait dû faire* (pas de "bon label"), on lui dit juste si le résultat est bon ou mauvais.
- **Analogie :**
  - Supervised : Apprendre à conduire avec un instructeur qui touche le volant à chaque erreur.
  - RL : Apprendre à faire du vélo seul. On tombe, on a mal (récompense négative), on ajuste, on tient l'équilibre (récompense positive).

La Boucle Agent-Environnement (The Loop)

- **Concept central :** Tout le RL repose sur cette interaction cyclique à temps discret t.
- **Les composants :**
  - **L'Agent :** Le cerveau (le réseau de neurones). Il maintient une politique π.
  - **L'Environnement :** Tout ce qui est extérieur à l'agent (la physique, le jeu, le marché boursier).
- **Le cycle d'interaction (Step) :**
  1. L'agent observe l'état S_t (State).
  2. L'agent choisit une action At basée sur St.
  3. L'environnement transitionne et renvoie :
     - Une récompense Rt+1 (feedback immédiat).
     - Le nouvel état St+1 (conséquence).
- **Code Conceptuel (Gymnasium style) :**




```
state = env.reset()
while not done:
    action = agent.predict(state)
    next_state, reward, done, info = env.step(action)
    state = next_state # La boucle continue
```

Le Signal de Récompense (Reward Hypothesis)

- **Motivation :** Comment formaliser n'importe quel but complexe en un simple nombre ?
- **The Reward Hypothesis (Sutton & Barto) :** Tous les buts peuvent être décrits par la maximisation de la somme cumulée d'une récompense scalaire attendue.
- **Nature du signal R_t :**
  - C'est un nombre réel scalaire (ex: +1, -10, 0.05).
  - Il définit le *Quoi* (le but), pas le *Comment* (la méthode).
- **Exemples de Design de Récompense (Reward Engineering) :**
  - *Échecs :* +1 si victoire, -1 si défaite, 0 pour tous les autres coups (Reward très "sparse"/rare).
  - *Robotique (Marche) :* +Vitesse avant, -Énergie dépensée, -Chute (Reward "dense").
  - *Trading :* +P&L (Profit and Loss) à chaque instant.
- **Danger :** L'agent exploitera la moindre faille dans la définition de la récompense (Reward Hacking).
  - *Exemple :* Un agent de nettoyage qui "cache" la poussière sous le tapis si on le récompense uniquement pour ne plus *voir* la poussière.

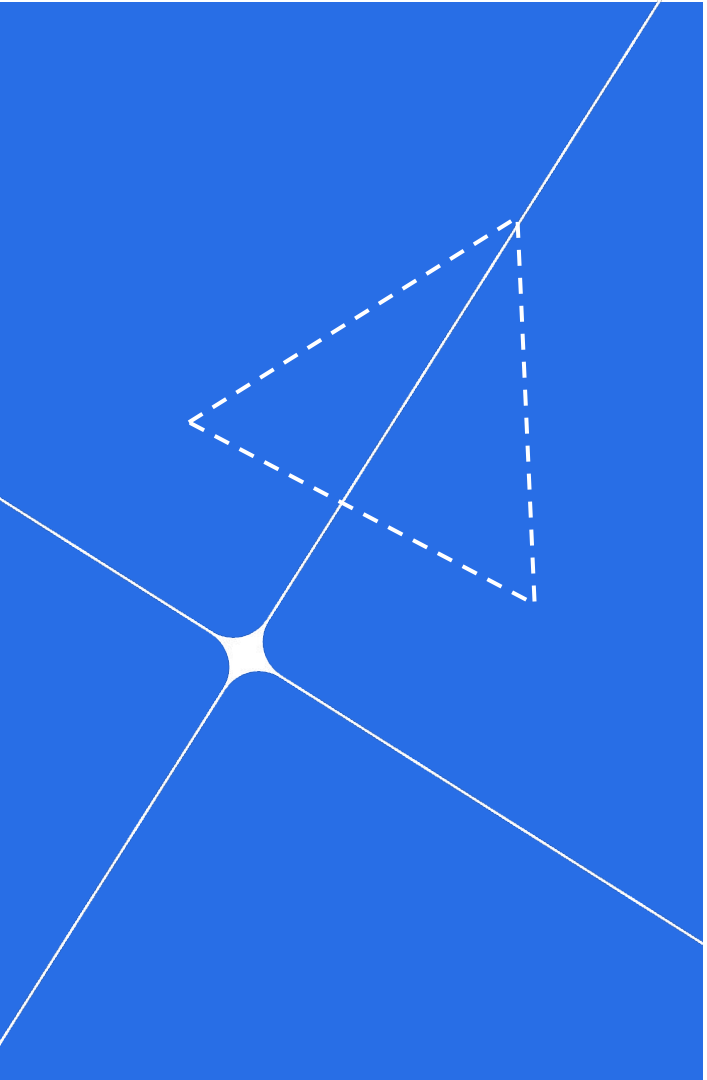
Exemples Concrets : Inputs et Outputs

- **Motivation :** Concrétiser la forme des tenseurs manipulés par nos réseaux.
- **Cas 1 : Atari (Breakout/Pong)**
  - *Input (S_t) :* Images brutes (Pixels). Tenseur [4, 84, 84] (4 frames consécutives empilées pour le mouvement).
  - *Action (A_t) :* Espace Discret. Vecteur One-hot ou index [0: Left, 1: Right, 2: Fire].
  - *Application :* Tests de benchmarks IA, conduite autonome (vision-based).
- **Cas 2 : Contrôle de Datacenter (Google/DeepMind)**
  - *Input (S_t) :* Vecteur de capteurs (Températures, Charge CPU, Vitesse ventilateurs, Pression).
  - *Action (A_t) :* Espace Continu. Réglage des vannes de refroidissement (0% à 100%).
  - *Impact :* Réduction de 40% de la consommation d'énergie pour le refroidissement.
- **Cas 3 : Large Language Models (RLHF)**
  - *Input (S_t) :* Prompt + Texte généré (Tokens).
  - *Action (A_t) :* Token suivant (vocabulaire).
  - *Reward :* Score donné par un modèle de préférence (Reward Model) aligné sur l'humain.

Pourquoi est-ce difficile ? Les Challenges Uniques

- **Motivation :** Ce qui fait échouer un modèle supervisé classique appliqué brutalement au RL.

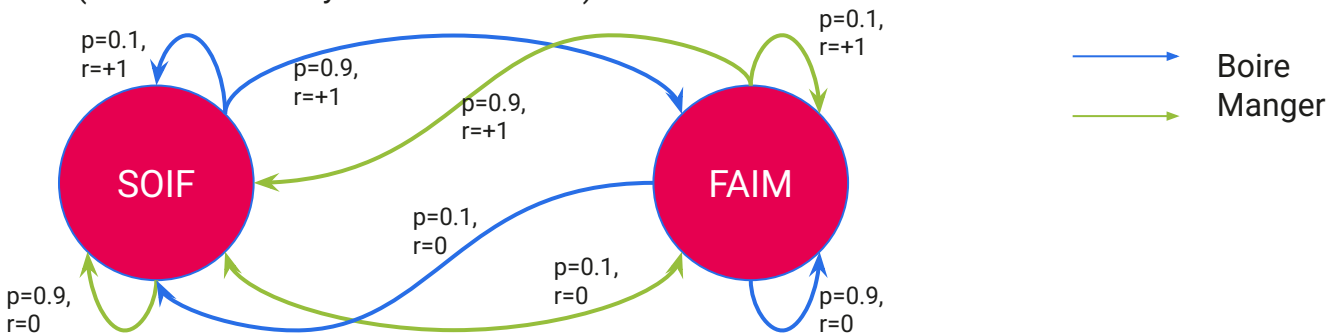
1. **Données Non-IID (Independent and Identically Distributed) :**
   - Les données sont des séquences corrélées (S_{t+1} dépend fortement de S_t).
   - En supervisé, l'ordre des images n'importe pas. En RL, le temps est crucial.
2. **Non-Stationarité :**
   - L'agent apprend → sa politique change → les données qu'il collecte changent.
   - La distribution des données d'entraînement évolue constamment (Moving target).
3. **Credit Assignment Problem (Problème d'attribution du crédit) :**
   - Si je gagne la partie d'échecs au 50ème coup, quel coup a été décisif ? Le 49ème ? Le 12ème ?
   - Le feedback est souvent retardé. Il faut propager la récompense future vers les actions passées.
4. **Exploration vs Exploitation :**
   - Dois-je utiliser ma meilleure stratégie actuelle (Exploitation) ou essayer quelque chose de nouveau pour potentiellement découvrir mieux (Exploration) ?



Les Fondations Mathématiques

Le Cadre Formel : Markov Decision Process (MDP)

- **Motivation :** Nous avons besoin d'un cadre mathématique rigoureux pour modéliser l'interaction Agent-Environnement.
- **Le Tuple (S, A, R, P, γ) :**
  - **State space (S) :** L'ensemble de toutes les situations possibles (ex: position du robot, pixels de l'écran).
  - **Action space (A) :** L'ensemble des mouvements (ex: [Haut, Bas, Gauche, Droite] ou angles moteurs).
  - **Reward function (R) :** R(s, a) ou R(s, a, s'). Le feedback immédiat.
  - **Transition (P) :** La dynamique du monde. Probabilité P(s' | s, a) d'arriver en s' après avoir fait a dans s.
- **La Propriété de Markov :**
  - "Le futur est indépendant du passé sachant le présent".
  - L'état S_t doit contenir *toute* l'information nécessaire pour décider. Si une info manque (ex: vitesse), l'état n'est pas Markoviens (POMDP - Partially Observable MDP).

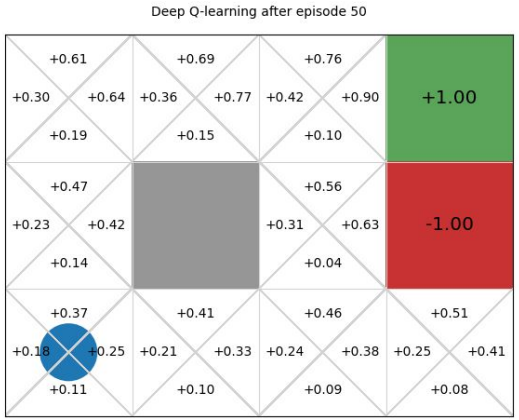


L'Objectif : Return et Discount Factor

- **Motivation :** L'agent ne veut pas juste la récompense *maintenant*, il veut maximiser la somme totale. Mais les séries infinies peuvent diverger.
- **Le Return (G_t) :** La somme cumulée des récompenses futures.
- **L'actualisation (γ - Gamma) :**
  - $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0..\infty} \gamma^k R_{t+k+1}$
  - $\gamma \in [0,1]$: Le "taux d'intérêt" du futur.
- **Intuition de γ :**
  - $\gamma \approx 0$ (Myopic) : L'agent est impulsif, il ne voit que le gain immédiat.
  - $\gamma \approx 1$ (Far-sighted) : L'agent est stratège, il accepte de souffrir maintenant pour gagner plus tard.
  - Mathématiquement : Garantit que la somme converge si l'horizon est infini.

Policy (π) et Value Functions (V & Q)

- **Motivation :** Comment définir le comportement et comment évaluer si une situation est "bonne" ?
- **La Politique (Policy π) :** Le cerveau de l'agent.
  - C'est une distribution de probabilité sur les actions : $\pi(a|s)$.
  - Déterministe : $a=\pi(s)$. Stochastique : $a \sim \pi(\cdot|s)$.
- **Value Function ($V_{\pi(s)}$) :**
  - "Si je suis dans l'état s et que je suis la politique π, combien de récompense vais-je accumuler en moyenne ?"
  - C'est une prédiction long-terme.
- **Action-Value Function ($Q_{\pi(s,a)}$) - CRUCIAL POUR DQN :**
  - "Si je suis dans l'état s, que je fais l'action a, *puis* que je suis la politique π ensuite..."
  - Q(s,a) évalue la paire (Situation, Action). C'est ce que nous allons apprendre.



L'Équation de Bellman (Le Moteur du RL)

- **Motivation :** Comment calculer Q(s,a) sans attendre la fin de l'épisode ? On utilise la récursivité.
- **L'idée clé :** La valeur d'un état est la récompense immédiate + la valeur actualisée de l'état suivant.
- **L'Équation d'Optimalité de Bellman pour Q* :**

$$Q^*(s, a)=E_{s'}[R(s, a) + \gamma \max_{a'} Q^*(s', a')]$$

- **Décomposition :**
  - $R(s, a)$: Réalité immédiate.
  - $\gamma \max_{a'} Q^*(s',a')$: Meilleure estimation future possible.
- **Conséquence :** Cette équation transforme un problème d'optimisation séquentielle complexe en une série de mises à jour locales itératives.
- **Algorithme (Q-Learning Tabulaire) :** On stocke une table géante Q[s,a] et on met à jour chaque case avec l'erreur de Bellman (Différence entre ce qu'on prédisait et ce qu'on vient d'observer).

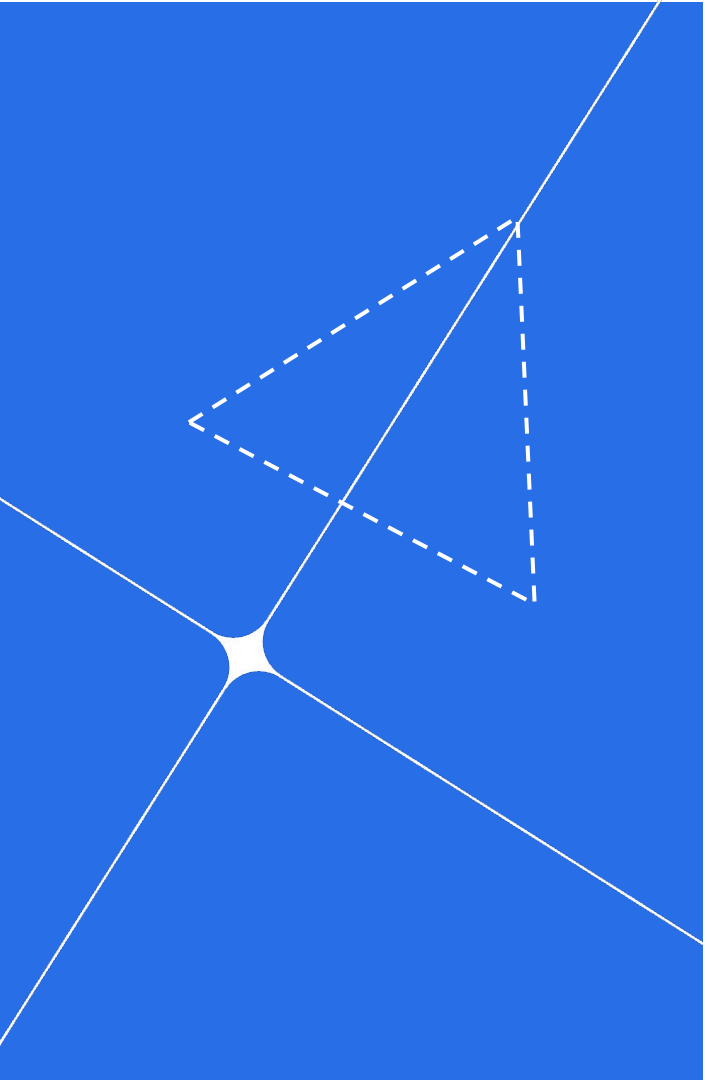
Algorithme du Q-learning tabulaire

```
# Initialisation des hyperparamètres
alpha = 0.1      # Taux d'apprentissage (Learning rate)
gamma = 0.99     # Facteur d'actualisation (Discount factor)
epsilon = 0.1    # Probabilité d'exploration

# Initialisation de la table Q avec des zéros
Q = initialize_q_table(num_states, num_actions) # Dimensions : [nombre_etats, nombre_actions]

for episode in range(total_episodes):
    state = env.reset()
    done = False
    while not done:
        # 1. Sélection de l'action (Stratégie epsilon-greedy)
        if random_uniform(0, 1) < epsilon:
            action = env.action_space.sample()  # Exploration aléatoire
        else:
            action = argmax(Q[state, :])        # Exploitation de la meilleure action connue

        # 2. Exécution de l'action dans l'environnement
        next_state, reward, done, info = env.step(action)

        # 3. Mise à jour de la valeur Q selon l'équation de Bellman
        # On calcule la valeur maximale possible pour l'état suivant
        max_next_q = max(Q[next_state, :])

        # Application de la formule de mise à jour (Temporal Difference)
        Q[state, action] = Q[state, action] + alpha * (reward + gamma * max_next_q - Q[state, action])

        # 4. Transition vers le nouvel état
        state = next_state
```

$$Q^{k+1}(s, a)=(1-\alpha)\ Q^{k}(s', a')) + \alpha\ (R(s, a) + \gamma \max_{a'} Q^{k}(s', a'))$$

Le Dilemme Exploration vs Exploitation

- **Motivation :** Pour apprendre la meilleure stratégie, il faut parfois prendre des risques et essayer des actions inconnues.
- **Le Problème :**
  - Si j'exploite (Max Q) tout le temps : Je reste coincé dans un optimum local (je mange toujours la même pizza correcte).
  - Si j'explore (Random) tout le temps : Je n'optimise rien (je mange n'importe quoi).
- **La Solution Standard : ϵ-Greedy**
  - On définit un hyperparamètre ϵ (ex: 0.1).
  - Avec probabilité ϵ : Choisir une action **aléatoire** (Exploration).
  - Avec probabilité $1-\epsilon$: Choisir $a = \text{argmax } Q(s, a)$ (Exploitation).
- **Decay (Décroissance) :**
  - Au début de l'entraînement, on ne sait rien : $\epsilon=1.0$ (100% aléatoire).
  - Au fil du temps, on réduit ϵ vers 0.01 pour stabiliser la politique.



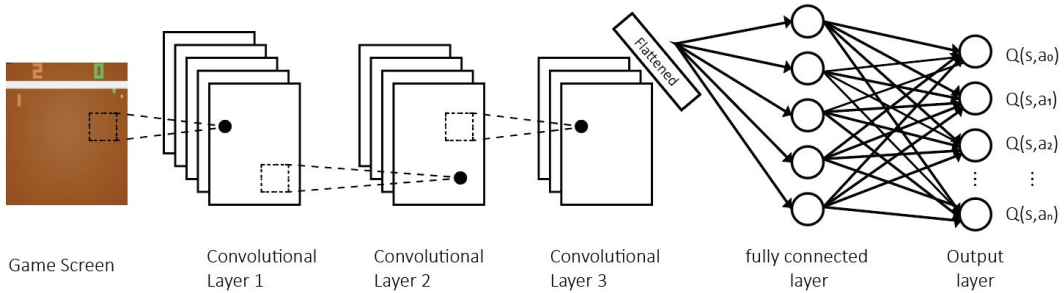
Value-Based Methods - Deep Q-Learning (DQN)

La Malédiction de la Dimensionnalité (Pourquoi le Tabulaire échoue)

- **Motivation :** Comprendre l'impasse des méthodes classiques face au monde réel.
- **Le problème du Tabulaire :** Une table Q[s, a] nécessite une entrée en mémoire pour chaque combinaison d'état et d'action.
- **Le mur de la réalité :**
  - Tic-Tac-Toe : ~5000 états (Facile).
  - Jeu d'échecs : ~10^{43} états.
  - Image de jeu Atari (Pixels 84×84 en niveaux de gris) : 256^{7056} états possibles.
- **La solution du Deep RL :** L'Approximation de Fonction.
- **Le concept :** Au lieu de stocker chaque valeur, nous utilisons un réseau de neurones paramétré par des poids θ pour deviner la valeur : $Q(s, a; \theta) \approx Q^*(s, a)$.
- **L'avantage direct :** La généralisation. Si le réseau voit une image de route légèrement différente de l'entraînement, il peut quand même inférer une bonne Q-Value grâce à ses filtres convolutifs.

Architecture d'un Deep Q-Network (DQN)

- **Motivation :** Comment concevoir l'architecture réseau pour que l'inférence soit ultra-rapide en production ?
- **Design naïf :** Entrée (État + Action) → Réseau → Sortie (1 seule Q-Value).
  - *Inconvénient :* Il faudrait faire une passe avant ("forward pass") pour *chaque* action possible pour trouver le $\max_a Q(s, a)$. Trop lent !
- **Design du DQN (Mnih et al., DeepMind) :**
  - **Entrée :** Uniquement l'état s (ex: tenseur d'images empilées ou vecteur de capteurs).
  - **Réseau :** Feature extractor (CNN pour la vision, ou MLP) suivi de couches denses (Fully Connected).
  - **Sortie :** Un vecteur contenant la Q-Value pour *toutes* les actions possibles simultanément (taille de sortie = nombre d'actions).
- **Attention (Piège classique) :** Pas de fonction d'activation "Softmax" à la fin ! Nous voulons régresser des valeurs absolues (les retours attendus en points), pas des probabilités.

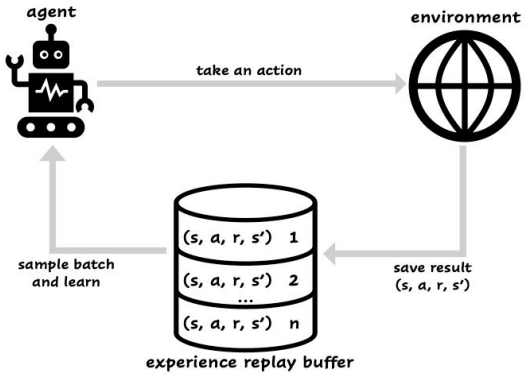


L'Instabilité de l'Apprentissage

- **Motivation :** Pourquoi brancher un réseau de neurones directement sur l'équation de Bellman mène au désastre.
- Si on optimise naïvement le réseau avec les transitions générées en temps réel, la fonction de perte diverge souvent.
- **Cause 1 : Corrélation temporelle des données.**
  - Les états S_t et S_{t+1} d'un robot qui avance sont presque identiques. Le réseau "surapprend" (overfit) sur cette trajectoire très spécifique et oublie le reste (Catastrophic forgetting).
- **Cause 2 : La cible mouvante (Moving Target).**
  - Rappel de la cible de Bellman : $Y = R + \gamma \max_{a'} Q(S', a'; \theta)$.
  - La cible Y dépend des poids θ, qui sont les *mêmes* poids que l'on est en train de mettre à jour via la descente de gradient.
  - À chaque pas d'apprentissage, on déplace la cible que l'on essaie d'atteindre.

Pilier 1 du DQN : L'Experience Replay (Replay Buffer)

- **Motivation :** Comment retrouver l'indépendance statistique (I.I.D) nécessaire au Deep Learning ?
- **Le Concept :** L'agent agit dans l'environnement, mais on ne l'entraîne pas immédiatement sur la dernière action.
- **Le Buffer :** On stocke chaque transition (S, A, R, S', done) dans une grande mémoire circulaire (ex: 1 million de transitions).
- **L'Entraînement :** À chaque itération, on échantillonne aléatoirement un "Mini-Batch" de transitions depuis ce buffer pour faire notre descente de gradient.
- **Avantages majeurs :**
  - Casse la forte corrélation temporelle entre les échantillons.
  - Réutilisation des données (Sample efficiency) : une transition rare mais très instructive (ex: trouver un trésor) sera revue plusieurs fois pendant l'entraînement.



Pilier 2 du DQN : Le Target Network

- **Motivation :** Fixer mathématiquement la "cible mouvante" pour stabiliser la descente de gradient.
- **La Solution :** Maintenir **deux** réseaux de neurones d'architecture identique.
  - **Policy Network (Q_θ) :** Celui qui agit et qui est mis à jour à chaque étape (par gradient).
  - **Target Network (Q_{θ^-}) :** Une copie figée, utilisée **uniquement** pour calculer la cible de Bellman.
- **Calcul de la cible sécurisée :**

$$Y = R + \gamma \max_{a'} Q(S', a'; \theta^-)$$

- **Mise à jour (Sync) :** Périodiquement (ex: tous les 1000 steps), on copie brutalement les poids du Policy Network vers le Target Network ($\theta^- \leftarrow \theta$), ou on utilise un lissage exponentiel (Soft update).
- L'apprentissage devient stable car la cible est fixe pendant de nombreuses itérations.

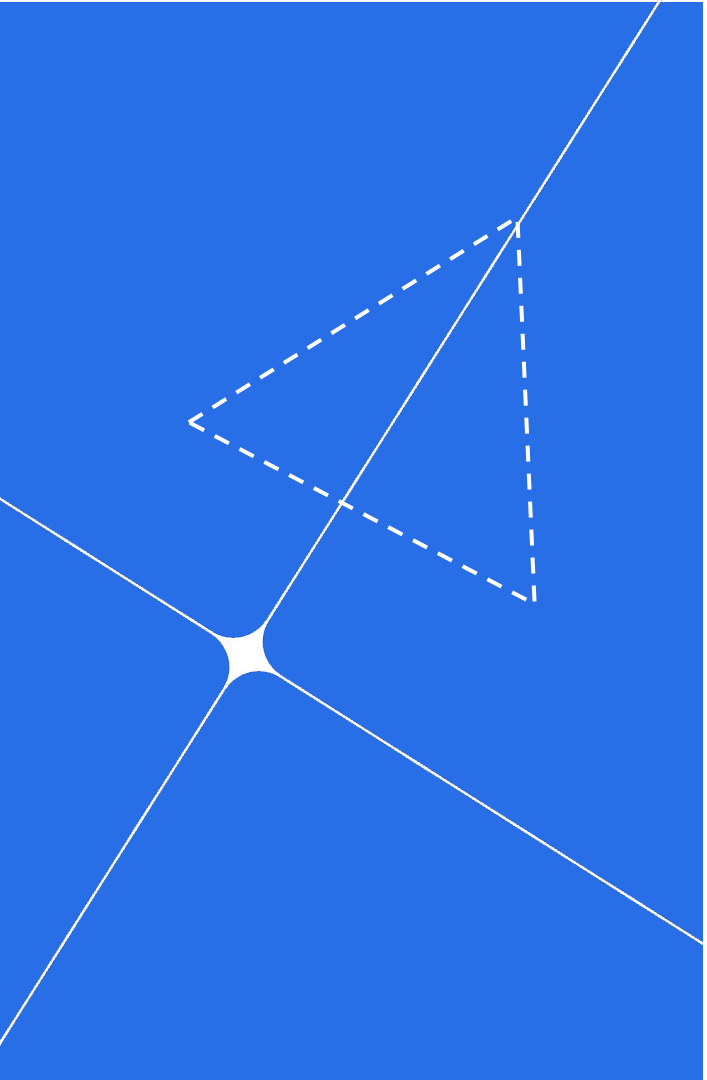
PyTorch en Action : La Loss du DQN

- **Motivation :** Traduire cette architecture conceptuelle en code concret.
- La fonction de perte est généralement la Mean Squared Error (MSE) ou la Huber Loss pour être robuste aux valeurs aberrantes (outliers).
- **Formule mathématique de l'optimisation :**

$$Loss(\theta) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta^-)}_{\text{Target (Fixe)}} - \underbrace{Q(s, a; \theta)}_{\text{Prédiction}}\right)^2\right]$$

- **Snippet PyTorch (Aperçu) :**

```
# 1. Prédictions du Policy Net pour les actions choisies
q_values = policy_net(states).gather(1, actions)

# 2. Valeurs max du Target Net pour l'état suivant (sans gradient)
with torch.no_grad():
    max_next_q_values = target_net(next_states).max(1)[0]

# 3. Calcul de la cible de Bellman (attention si l'épisode est fini)
targets = rewards + gamma * max_next_q_values * (1 - dones)

# 4. Descente de gradient
loss = criterion(q_values, targets.unsqueeze(1))
optimizer.zero_grad()
loss.backward()
optimizer.step()
```



Policy-Based Methods & Actor-Critic (PPO)

Limites des Méthodes Value-Based et Nouvel Objectif

- **Motivation :** DQN est puissant, mais comment contrôler précisément le volant d'une voiture autonome ?
- **Le problème du continu :** DQN requiert de calculer un $\max_a Q(s, a)$. Si l'action est un angle de braquage continu entre -180° et +180°, trouver ce maximum à chaque étape est un problème d'optimisation en soi (très lent).
- **Le problème de la stochasticité :** Parfois, la politique optimale est d'être imprévisible (ex: Pierre-Papier-Ciseaux). DQN produit des politiques déterministes.
- **Le changement de paradigme (Policy-Based) :** Au lieu d'apprendre Q pour en déduire π, on modélise directement la politique π avec un réseau de neurones paramétré par θ : $\pi_\theta(a \mid s)$.
- **L'Output du réseau change :**
  - *Action discrète :* Le réseau sort les probabilités de chaque action (Softmax).
  - *Action continue :* Le réseau sort les paramètres d'une distribution (ex: moyenne μ et écart-type σ d'une Gaussienne), dans laquelle on va échantillonner l'action.

Policy Gradient : L'Intuition Mathématique

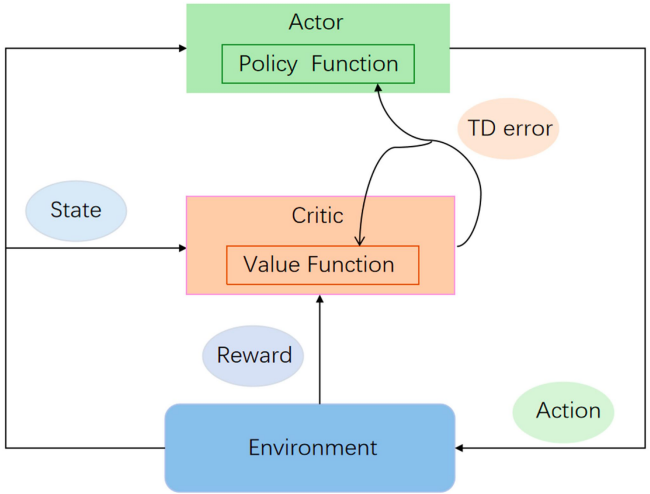
- **Motivation :** Comment mettre à jour les poids θ de notre politique pour maximiser les gains ?
- **L'Objectif (J(θ)) :** Maximiser l'espérance du retour cumulé en suivant la politique π_θ.
- **Le Théorème du Policy Gradient :**
  - On utilise la montée de gradient (Gradient Ascent) car on veut *maximiser* J.
  - Formule centrale : $\nabla_\theta J(\theta) = E_{\pi\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot G_t]$
- **Explication intuitive de la formule :**
  - $\nabla_\theta \log \pi_\theta(a|s)$: La direction dans l'espace des poids pour augmenter la probabilité de l'action a.
  - G_t (Le Return) : Le "juge". Si l'action a mené à une grande récompense ($G_t > 0$), on pousse fort dans cette direction. Si elle a mené à un désastre ($G_t < 0$), la probabilité de cette action est diminuée.
- **Analogie simple :** C'est un apprentissage par essais et erreurs guidé par la récompense finale (Trial and Error).

Réduire la Variance avec la Fonction d'Avantage

- **Motivation :** L'algorithme précédent (REINFORCE) est instable car le retour G_t dépend de trajectoires entières très bruitées (forte variance).
- **Le problème du G_t brut :** Si un agent fait une excellente action, mais que le reste de l'épisode est catastrophique par malchance, l'excellente action sera pénalisée.
- **La Baseline :** Au lieu de demander "Cette action a-t-elle rapporté beaucoup ?", on demande "Cette action a-t-elle fait *mieux que la moyenne* de ce qu'on attend dans cet état ?".
- **L'Advantage Function (A(s,a)) :** $A(s, a) = Q(s, a) - V(s)$
  - $Q(s, a)$: Valeur réelle de l'action choisie.
  - $V(s)$: Valeur moyenne attendue dans l'état s (La Baseline).
- **Conséquence :** Si $A(s, a) > 0$, l'action était meilleure que d'habitude → On augmente sa probabilité. La variance de l'entraînement chute drastiquement.

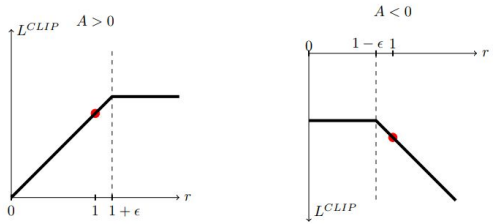
L'Architecture Actor-Critic

- **Motivation :** Pour calculer l'Avantage, nous avons besoin de connaître V(s). Il nous faut donc un réseau qui apprend la valeur, en plus de celui qui apprend la politique.
- **Le concept Actor-Critic :** Fusionner les avantages des méthodes Policy-Based et Value-Based.
- **L'Acteur (Actor) :** Le réseau de la politique $\pi_\theta(a \mid s)$. Il observe l'état et décide quoi faire.
- **Le Critique (Critic) :** Le réseau de valeur $V_\phi(s)$ paramétré par ϕ. Il observe l'état et prédit le score final.
- **La boucle de mise à jour :**
  1. L'Acteur joue une action.
  2. Le Critique évalue cette action (Calcule l'erreur Temporal Difference TD et l'Avantage).
     - TD error = $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
  3. L'Acteur met à jour ses probabilités selon les retours du Critique.
  4. Le Critique met à jour ses prédictions (Loss MSE classique) pour devenir un meilleur évaluateur.



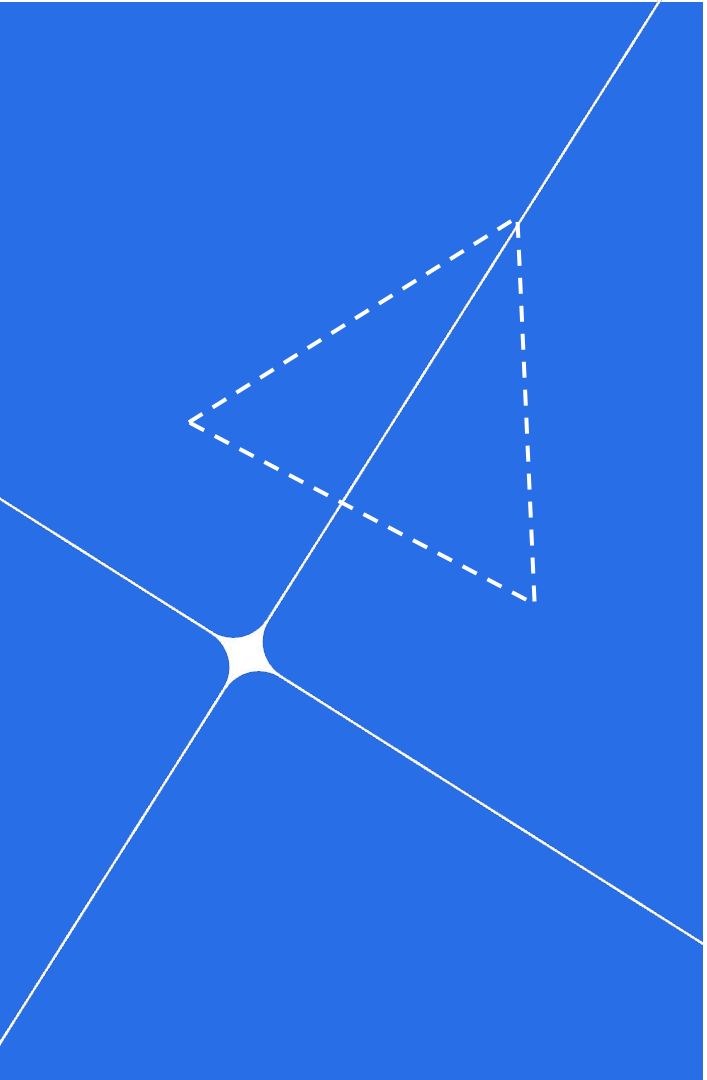
Proximal Policy Optimization (PPO) : L'État de l'Art

- **Motivation :** Un pas de gradient trop grand dans l'espace des politiques peut "casser" l'agent (il oublie comment marcher et ne s'en remet jamais). Comment garantir des mises à jour sûres ?
- **Trust Region (Région de confiance) :** L'idée est d'empêcher la nouvelle politique de trop s'éloigner de l'ancienne politique lors d'une mise à jour.
- **L'intuition de PPO (OpenAI, 2017) :** Le "Clipping" (Écrêtage).
- On calcule le ratio entre la probabilité de l'action sous la *nouvelle* politique et l'*ancienne* politique.
- **Objectif PPO (Surrogate Loss) :** Si ce ratio s'éloigne de 1 (par exemple, sort de l'intervalle [0.8,1.2]), on "coupe" le gradient.
- **Avantage colossal :**
  - Évite les effondrements de performance (Catastrophic drops).
  - Permet de faire plusieurs époques d'entraînement (Multi-epochs) sur le même batch de données sans risque, ce qui améliore massivement l'efficacité des données (Sample efficiency) par rapport aux anciens algorithmes.



PPO en Production : Le Standard de l'Industrie

- **Motivation :** Comprendre pourquoi cet algorithme précis est celui que vous retrouverez le plus souvent dans votre carrière d'ingénieur.
- **Robuste et Généraliste :** PPO fonctionne avec très peu de réglages d'hyperparamètres (Contrairement à DQN ou DDPG, Deep Deterministic Policy Gradient, qui sont très capricieux).
- **Cas d'usage 1 : Robotique et Contrôle.** Boston Dynamics ou les bras robotiques industriels utilisent massivement des variantes de PPO pour des mouvements fluides (actions continues).
- **Cas d'usage 2 : Large Language Models (LLMs).**
  - PPO est le cœur du **RLHF** (Reinforcement Learning from Human Feedback).
  - L'Acteur est le LLM (générateur de texte). Le Critique est un "Reward Model" entraîné sur des préférences humaines.
  - C'est ce qui a transformé un modèle de complétion de texte classique (GPT-3) en un assistant utile et aligné (ChatGPT).



Écosystème et Présentation du Lab

L'API Standard du RL : Gymnasium

- **Motivation :** Pour comparer les algorithmes, il nous faut une interface commune et standardisée pour modéliser *n'importe quel* environnement.
- **Gymnasium (anciennement OpenAI Gym) :** La bibliothèque Python de référence absolue dans l'industrie et la recherche.
- **Le contrat d'interface :** Chaque environnement doit définir deux propriétés fondamentales :
  - observation_space : Le format des entrées (ex: Box pour des tenseurs continus, Discrete pour des entiers).
  - action_space : Les actions permises pour l'agent.
- **Les méthodes essentielles :**
  - env.reset() : Démarre un nouvel épisode et renvoie l'état initial.
  - env.step(action) : Applique une action et renvoie le tuple (observation, reward, terminated, truncated, info).

Exemple générique d'interaction aléatoire :

```
import gymnasium as gym
env = gym.make("CartPole-v1")
obs, info = env.reset()
for _ in range(100):
    action = env.action_space.sample() # Agent aléatoire
    obs, reward, terminated, truncated, info = env.step(action)
    if terminated or truncated:
        obs, info = env.reset()
env.close()
```

Les Frameworks de Deep RL : Stable Baselines3 (SB3)

- **Motivation :** Implémenter PPO ou DQN "from scratch" en PyTorch est un excellent exercice, mais extrêmement sujet aux bugs mathématiques et peu optimisé pour la production.
- **L'approche Ingénieur :** Ne réinventez pas la roue. Utilisez des bibliothèques fiables et testées (comme vous utilisez Transformers d'HuggingFace).
- **Stable Baselines3 :** Une surcouche PyTorch offrant des implémentations de haute qualité des algorithmes RL modernes (PPO, SAC, DQN, A2C).
- **Compatibilité :** SB3 est nativement conçu pour s'emboîter parfaitement avec les environnements Gymnasium.
- **Entraînement en 3 lignes (Littéralement) :**
-

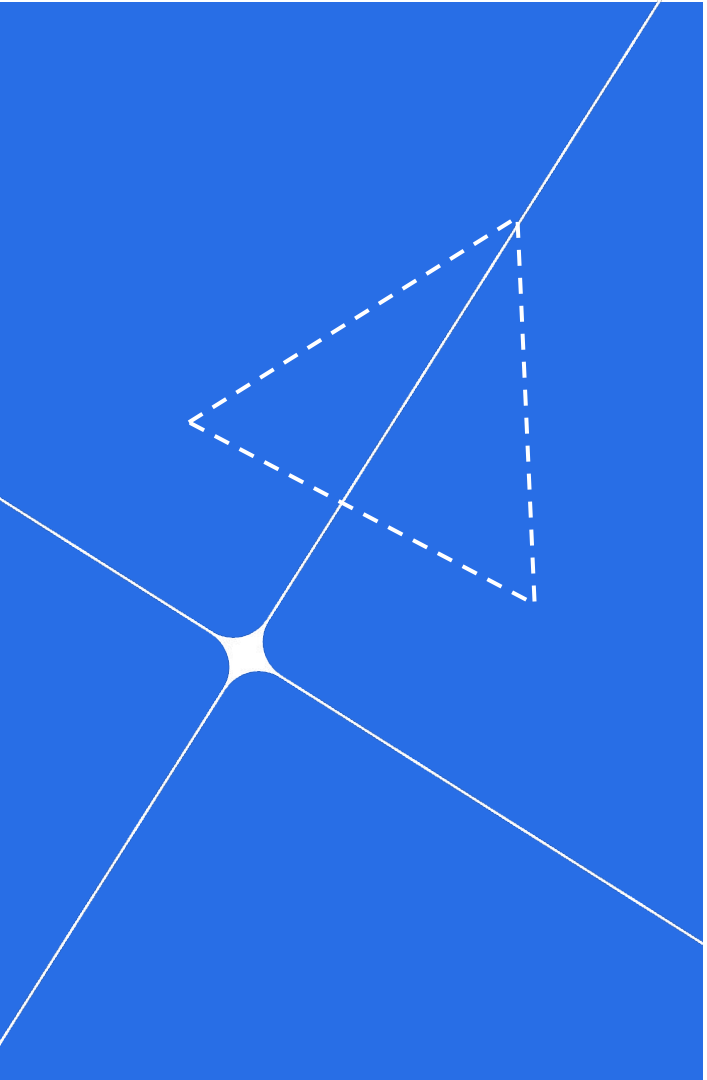
```
from stable_baselines3 import PPO

# 1. Initialisation de l'algorithme (Acteur-Critique intégré)
model = PPO("MlpPolicy", env, verbose=1)

# 2. Lancement de l'entraînement
model.learn(total_timesteps=10000)

# 3. Sauvegarde des poids
model.save("ppo_agent")
```

Infrastructure et Suivi des Expériences

- **Motivation :** En RL, la fonction de perte (Loss) n'est pas toujours un bon indicateur de succès. Comment monitorer correctement notre agent ?
- **La métrique reine :** Le "Mean Episodic Return" (Récompense moyenne par épisode). Si elle monte, l'agent apprend, peu importe le comportement de la Loss.
- **Outils de Logging :** SB3 s'intègre nativement avec **TensorBoard** et Weights & Biases (W&B).
- **Déploiement et Reproductibilité (Vos acquis) :**
  - Le rendu graphique d'environnements (OpenGL, etc.) est capricieux selon les OS.
  - **Docker :** Indispensable pour conteneuriser votre environnement d'entraînement (PyTorch + Gymnasium + dépendances X11 pour le rendu headless).
  - Vous pourrez ensuite packager l'agent entraîné (.zip ou .pth) derrière une API Flask/FastAPI, exactement comme pour un LLM ou un modèle de vision.

Présentation du Lab : Votre Premier Agent Autonome

- **Motivation :** Passer de la théorie à la pratique et observer concrètement les défis temporels du Deep RL.
- **L'objectif du TP :** Entraîner de bout en bout un agent capable de résoudre un problème de contrôle dynamique classique (ex: poser un module lunaire en douceur).
- **Stack Technique :** Python, PyTorch, Gymnasium, Stable Baselines3.
- **Les missions du jour :**
  1. Explorer manuellement les espaces d'états et d'actions pour comprendre les tenseurs en jeu.
  2. Implémenter et lancer un entraînement PPO complet.
  3. Monitorer l'apprentissage et identifier les phases d'exploration vs exploitation.
  4. **Générer une vidéo :** Enregistrer le comportement de votre modèle final en inférence (sans entraînement) pour valider visuellement sa stratégie.
- **Conseil final :** L'entraînement RL est stochastique par nature. Si votre agent échoue, modifiez légèrement l'architecture ou les hyperparamètres (comme le *learning rate* ou *gamma*) avant de conclure à un bug de code.



En route vers le TP