I. Algorithme REINFORCE

1. Environnement

Nous décrivons notre environnement à partir d'un ensemble d'états, noté $\mathcal{S}$, et d'un ensemble d'actions noté $\mathcal{A}$. Nous utilisons un environnement de type MDP (Markov Decision Process) : la transition vers l'état $s' \in \mathcal{S}$ ne dépend que de l'état courant $s \in \mathcal{S}$ et de l'action choisie $a \in \mathcal{A}$. Cela sous-entend que la connaissance des évènements précédents n'est pas utile pour déterminer la transition vers un nouvel état.

Dans notre univers, la transition vers un nouvel état s' après une action a depuis un état s est un processus stochastique. Cela signifie que l'action a appliquée depuis un état s peut amener vers différents états. Nous modélisons ce phénomène en associant une probabilité P à chaque transition $s, a \rightarrow$ s' que nous notons :

$$P(\, s' \mid s\,,\, a\,)$$

Cette notation reprend la notation habituelle désignant la probabilité d'arriver à l'état s' sachant l'état s et l'action a. Cette fonction définit une distribution de probabilités pour chaque couple état/action :

$$\forall s \in \mathcal{S}, \forall\, a \in \mathcal{A} \;:\; \sum_{s' \in \mathcal{S}} P(s'|s,a) = 1$$

2. Récompenses et trajectoires

Dans ce chapitre, nous supposons qu'il existe une fonction récompense $\mathcal{R}$ qui associe à toute transition $s, a \rightarrow$ s' une valeur réelle correspondant à un gain. Ainsi, nous pouvons écrire :

$$r \;= \mathcal{R}(s, a, s')$$

Dans d'autres univers, la récompense pourrait être aléatoire. Cependant, dans ce chapitre, nous gardons l'hypothèse d'une récompense déterministe.

On peut décrire un scénario comme une série de transitions. Pour cela, on utilise un indice t comme marqueur de temps, indice qui s'incrémente à chaque action. Ainsi, on définit une ***expérience*** comme une paire état/action : (s_t, a_t) et on définit une ***trajectoire*** $\boldsymbol{\tau}$ comme une séquence d'expériences :

$$\tau = (s_0, a_0)(s_1, a_1) \dots (s_k, a_k)$$

Nous notons $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$ la récompense obtenue à l'étape t. De cette façon, nous définissons la ***récompense associée à une trajectoire*** $\boldsymbol{\tau}$:

$$R(\tau) = \sum_{t=0}^{k} r_t$$

Remarque : r_t n'est pas correctement défini pour $t = k$ car s_{k+1} est inconnu. Ceci n'est pas un réel problème car suivant le problème étudié, ce cas particulier se traite facilement.

3. Politique

On appelle ***politique***, notée π, une méthode indiquant à une IA quel choix elle doit faire. Une politique peut s'énoncer de manière déterministe : se déplacer à gauche. Si on associe des probabilités aux trois actions : se déplacer à gauche, se déplacer à droite et tirer, notre politique déterministe se modélise par un triplet de probabilités : (100%, 0, 0). Toute évolution de cette politique se traduit par une nouvelle politique associée à un triplet (100%, 0, 0), (0, 100%, 0) ou (0, 0, 100%). De telles variations de 0→100% ou de 100%→0 ne sont pas différentiables. Par conséquent, nous ne pouvons pas utiliser la méthode du gradient pour mettre en place un algorithme d'apprentissage. Pour avoir des politiques différentiables et rendre l'apprentissage possible, nous utilisons des politiques stochastiques. Une ***politique stochastique*** associe une probabilité à chaque action. Par exemple, si la politique optimale est (10%,90%) et que la politique de départ est fixée à (50%,50%), la phase d'apprentissage va successivement faire évoluer la politique courante de cette façon : (45%,55%), (40%,60%), (37%,63%), (30%,70%), …, (12%,88%), (11%,89%), (11%,89%) puis (10%,90%).

Comment choisir une action avec une politique stochastique ? Supposons que nous ayons trois actions possibles et une politique stochastique leur associant les probabilités de 60%, 30% et 10%. Choisir une action peut s'effectuer de la manière suivante : tirer un nombre aléatoire dans l'intervalle [0,100[. Si le tirage se trouve dans l'intervalle [0,60[, alors la première action est choisie. Si le tirage se trouve dans l'intervalle [60,90[, on sélectionne la deuxième action. Lorsque le tirage se trouve dans l'intervalle [90,100[, la dernière action est choisie.

Remarque : une politique stochastique peut générer un très grand nombre de trajectoires différentes.

Remarque : le choix d'une politique stochastique peut paraître déroutant. Habituellement, une stratégie ou un choix de jeu s'énonce de manière déterministe : entourer l'adversaire ou récolter de l'or et fabriquer des combattants. Pourtant dans certains jeux, la politique optimale est stochastique. Par exemple, dans le jeu Pierre / Feuille / Ciseaux, une politique visant à présenter uniquement la Pierre serait un échec car l'adversaire n'aurait qu'à présenter la Feuille à chaque fois pour gagner. La politique optimale associe à chaque possibilité une probabilité de 1/3. Le joueur doit choisir à chaque fois une option en tirant aléatoirement une de ces options.

Remarque : les politiques déterministes peuvent être représentées par des politiques stochastiques, ces dernières peuvent donc être vues comme une généralisation. En effet, une politique stochastique peut prendre des valeurs extrêmes comme (100%,0) ou (100%,0) correspondant à une politique déterministe. Cependant, la phase d'apprentissage ne permettra pas d'atteindre exactement (100%,0) mais plutôt (99.999%,0.001%).

4. REINFORCE

L'algorithme REINFORCE a pour objectif de construire une politique π qui pour chaque état associe à chaque action une probabilité d'être choisie. L'action finalement retenue sera choisie aléatoirement ceci en tenant compte des probabilités données par la politique.

Remarque : dans un jeu d'arcade par exemple, les actions possibles sont en nombre fini et correspondent à l'appui sur les diverses touches du joystick, du clavier ou de la souris. Par conséquent, une politique π, pour un état donné, associe à chaque bouton une probabilité d'être cliqué.

La notation $\boldsymbol{\tau \sim \pi}$ désigne une trajectoire τ générée à partir de la politique π.

L'objectif de l'algorithme REINFORCE est de construire une politique optimale en faisant évoluer ses paramètres d'apprentissage θ. Une telle politique est notée : $\boldsymbol{\pi_\theta}$. L'espérance des gains de cette politique notée $\boldsymbol{J(\pi_\theta)}$ est définie ainsi :

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \quad (1)$$

Cette formule correspond à l'espérance des récompenses, cette valeur traduit la récompense moyenne obtenue sur une infinité de trajectoires issues de la politique π_θ. Ainsi, nous pouvons modéliser le problème d'apprentissage d'une politique optimale de la manière suivante :

$$\max_\theta J(\pi_\theta)$$

5. Optimisation

Nous allons utiliser la méthode du gradient pour optimiser la fonction $J(\pi_\theta)$. Certains ont peut-être l'habitude d'utiliser cette méthode pour minimiser une fonction alors qu'ici nous utilisons une maximisation. Ce n'est pas très grave, car il suffit de remarquer que :

$$\max_\theta J(\pi_\theta) = \min_\theta -1 \times J(\pi_\theta)$$

Pour résoudre ce problème d'optimisation, il faut pouvoir calculer le gradient de J : $\nabla_\theta J(\pi_\theta)$. Pour cela, nous allons présenter différentes propriétés.

a) Probabilité d'une trajectoire

Pour passer de l'état s_t à l'état s_{t+1} avec l'action a_t il faut que l'action a_t soit choisie suivant la politique π et que la transition nous amène vers l'état s_{t+1} ce qui en termes de probabilité s'écrit :

$$P(s_{t+1}|s_t, a_t) \times \pi_\theta(a_t|s_t)$$

Pour une trajectoire τ et une politique π_θ donnés, la probabilité d'apparition de cette trajectoire τ s'écrit finalement :

$$P(\tau|\theta) = \prod_{t=0}^{k} P(s_{t+1}|s_t, a_t) \times \pi_\theta(a_t|s_t) \quad (2)$$

b) L'astuce de la log-dérivation

On peut écrire : $\nabla_\theta P(\tau|\theta) = P(\tau|\theta) \times \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)}$. Or, comme *log'(f(x)) = f'(x)/f(x)*, on obtient finalement :

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta) \times \nabla_\theta \log\big(P(\tau|\theta)\big) \quad (3)$$

c) Log-probabilité d'une trajectoire

En utilisant l'équation (2), on obtient

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{k} (\nabla_\theta \log P(s_{t+1}|s_t, a_t) + \nabla_\theta \log \pi_\theta(a_t|s_t))$$

La probabilité $P(s_{t+1}|s_t, a_t)$ ne dépend pas des paramètres θ ce qui permet de simplifier l'équation :

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{k} \nabla_\theta \log \pi_\theta(a_t|s_t) \qquad (4)$$

Nous pouvons maintenant déterminer la formule du gradient de l'algorithme REINFORCE :

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] && \text{Définition de l'espérance, l'intégrale vient du fait de l'infinité de trajectoires possibles} \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) && \text{Permutation du gradient et de l'intégrale} \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && \text{Utilisation de (3)} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta) R(\tau)] && \text{Ceci correspond à la définition de l'espérance.} \\
\nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] && \text{Utilisation de (4)}
\end{aligned}$$

Pour conclure, il suffit de repermuter le gradient et le grand sigma pour obtenir :

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

6. Intérêt

Pourquoi cette formule est-elle intéressante ? Lorsque l'on déroule une expérience, nous construisons itérativement la trajectoire $\tau = (s_0, a_0)(s_1, a_1) \ldots (s_k, a_k)$. On peut à chaque étape

- conserver la valeur de la probabilité $\pi_\theta(\mathrm{a_t}, \mathrm{s_t})$ associée à l'action retenue $\mathrm{a_t}$ à l'état $\mathrm{s_t}$
- conserver la récompense r_t obtenue

Une fois la séquence terminée, il suffit de calculer la grandeur :

$$Loss(\theta) = -R(\tau) \times \sum_{t=0}^{k} \log\left(\pi_\theta(\mathrm{a_t}, \mathrm{s_t})\right)$$

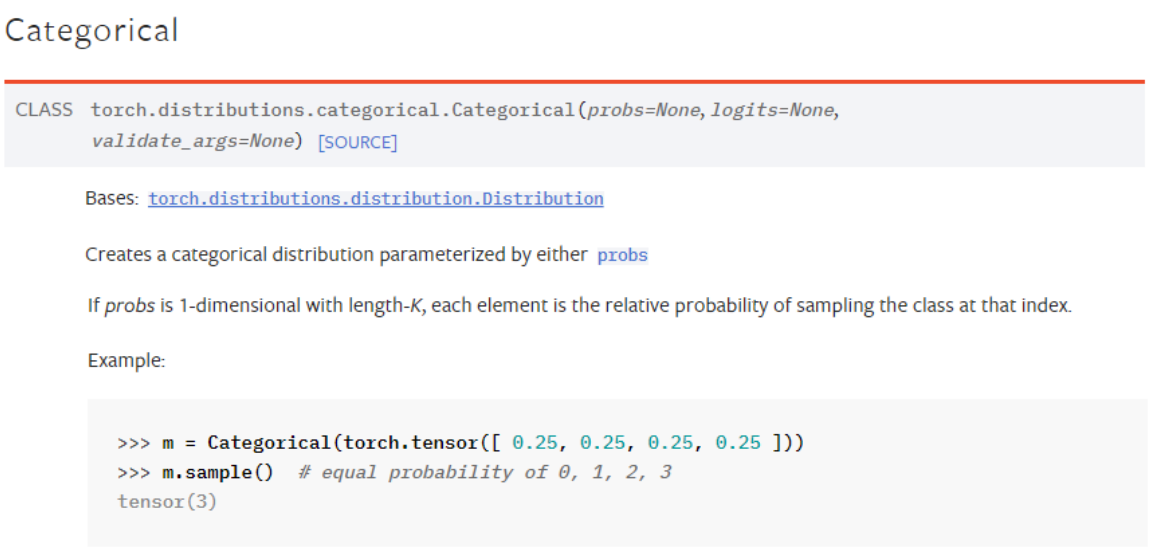
Nous remarquons la présence du signe – dans la formule de $Loss$ car nous devons passer d'une maximisation à une minimisation. Si la politique π_θ est modélisée par un réseau de neurones et que l'ensemble de ces calculs ont été effectués avec une librairie comme PyTorch, il suffit alors de

demander le calcul du gradient sur le tenseur de $Loss$ pour obtenir la valeur de $\nabla_\theta J(\pi_\theta)$. Il est alors possible d'utiliser les optimiseurs fournis par PyTorch pour appliquer la méthode d'optimisation.

Remarque : la formule principale en orange utilise un calcul d'espérance établi sur une infinité de trajectoires. Mais dans la pratique, nous effectuerons seulement une ou quelques simulations pour estimer la valeur de $\nabla_\theta J(\pi_\theta)$ car chaque simulation peut prendre beaucoup de temps.

7. Mise en place

Pour modéliser la politique π_θ, nous utilisons un réseau de neurones. A chaque état, la politique π_θ associe à l'ensemble des actions possibles un vecteur de probabilités. Dans la librairie PyTorch, la classe torch.distributions.Categorical permet de construire un objet pouvant effectuer un tirage aléatoire à partir d'un vecteur de probabilités donné.



Categorical

CLASS torch.distributions.categorical.Categorical(*probs=None*, *logits=None*, *validate_args=None*) [SOURCE]

Bases: torch.distributions.distribution.Distribution

Creates a categorical distribution parameterized by either probs

If *probs* is 1-dimensional with length-*K*, each element is the relative probability of sampling the class at that index.

Example:

```
>>> m = Categorical(torch.tensor([ 0.25, 0.25, 0.25, 0.25 ]))
>>> m.sample()  # equal probability of 0, 1, 2, 3
tensor(3)
```

La résultat obtenu par l'appel à la fonction sample() correspond à l'indice i de la i-ème action sélectionnée. Il faut ensuite calculer la log-probabilité associée à ce choix. L'objet de type Categorical permet d'effectuer cela en utilisant la fonction :

m.log_prob(index_action)

A la fin de l'expérience, nous lançons la rétropropagation depuis le tenseur $Loss(\theta)$ ce qui permet d'effectuer un pas de descente du gradient pour l'optimiseur PyTorch :

```
optimiser.zero_grad()
LossTheta.backward()
optimizer.step()
```

8. Amélioration

Le principe mis en place par l'algorithme REINFORCE consiste à augmenter les probabilités des actions sélectionnées durant une trajectoire proportionnellement à la récompense totale obtenue.

Cependant, cette idée a une faiblesse visible dans la formule $Loss(\theta)$. En effet, si une récompense importante est acquise en début de trajectoire, les actions effectuées en fin de parcours, qui n'ont peut-être générée aucun gain, se trouvent promues au même titre que les actions ayant généré des gains importants, alors qu'elles n'ont pas contribué à cette réussite.

Si un gain important est acquis par des actions effectuées en fin de trajectoire, nous ne savons pas quoi conclure des actions effectuées en début de trajectoire. En effet, peut-être sont-elles peu utiles ou peut-être que ces actions ont permis d'atteindre un état précis nécessaire pour pouvoir effectuer les actions finales très bénéfiques. Nous préférons partir du principe qu'il faut quand même promouvoir les actions initiales.

Ainsi, nous allons modifier le facteur en $R(\tau)$ dans la formule de $Loss(\theta)$ pour que l'action courante soit promue proportionnellement aux gains obtenus à la suite de cette action. Les gains obtenus précédemment ne participeront pas à sa promotion. Ainsi avec :

$$R_t(\tau) = \sum_{u=t}^{k} r_u$$

Nous obtenons finalement

$$Loss(\theta) = -\sum_{t=0}^{k} R_t(\tau) \times \log\left(\pi_\theta(a_t, s_t)\right)$$

II. L'environnement OpenAI-GYM

1. Présentation

Site officiel : https://gym.openai.com/

Gym est une boîte à outils permettant d'entraîner des agents/IA à marcher, à se déplacer ou à jouer à des jeux d'arcade comme Pong ou Space Invaders.

L'apprentissage par renforcement (Reinforcement Learning - RL) est le sous-domaine de l'apprentissage automatique (machine learning) qui étudie comment un agent peut apprendre à atteindre des objectifs dans un environnement complexe et incertain.

La librairie Gym propose un environnement standardisé pour tester et comparer différents algorithmes de RL. De plus, à l'image de l'apprentissage supervisé où de nombreuses données labélisées sont utilisées par les algorithmes d'IA pour leur apprentissage, Gym fournit un environnent robuste permettant aux algorithmes de RL d'effectuer de nombreuses simulations afin d'apprendre en continue.

2. Installation

Nous avons testé l'environnement Gym sous Python 3.8.8 - 64 bits. Voici la commande d'installation qui nous a permis d'obtenir une version fonctionnelle :

```
pip install gym pyglet Box2D
```

3. Test

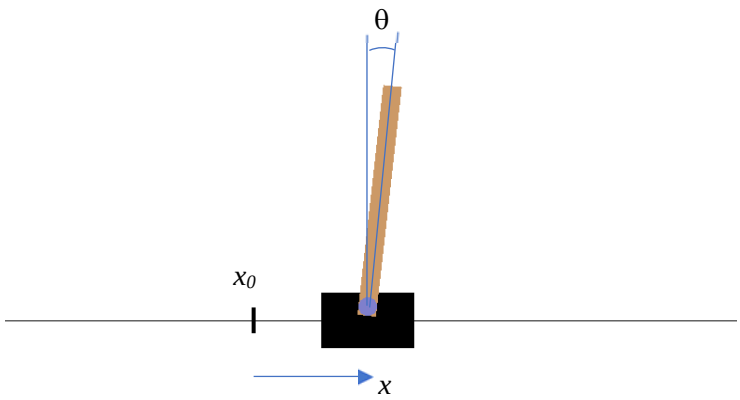
Ouvrez le fichier intitule : CartPole keyboard.py et lancez-le. Vous pouvez jouer au jeu CartPole qui consiste à garder en équilibre une tige debout sur un chariot en mouvement. Par défaut le charriot glisse vers la gauche, mais si vous appuyez sur la barre espace, il va glisser vers la droite.

III. CartPole

1. Présentation

Le jeu CartPole est un des jeux les plus simples pour l'apprentissage par renforcement. L'environnement permet de déplacer un chariot sur la gauche ou sur la droite. L'extrémité d'une tige est fixée par un axe à ce chariot. Cette tige disposée verticalement au démarrage subit les mouvements latéraux du chariot. Déséquilibrée, elle pivote autours de son axe, l'objectif est de faire en sorte qu'elle reste en position verticale. Voici une présentation de cet environnement :

- Objectif : faire en sorte que la tige reste le plus vertical possible pendant 200 itérations.
- Etat : un vecteur de 4 valeurs indiquant : la position horizontale du chariot, l'angle de la tige par rapport à la verticale ainsi que les dérivées de ces deux grandeurs.
- Actions : déplacer le chariot sur la gauche (0) ou sur la droite (1). Remarquez qu'il n'est pas possible de laisser le chariot immobile !
- Récompense : +1 à chaque itération.
- Arrêt :
    - Lorsque la tige fait par rapport à la verticale un angle supérieur à 12°.
    - Lorsque le chariot quitte la zone de jeu.
    - Après 200 itérations.



2. Démarrage

Ouvrez le fichier d'exemple nommé CartPole Reinforce. Cet exemple lance l'environnement Gym et effectue plusieurs simulations. Il dispose d'une IA basique qui envoie le charriot sur la droite lorsqu'il est à gauche de sa position de départ et inversement. Cette méthode, qui ne tient pas compte de l'état de la tige, permet cependant d'atteindre un score entre 20 et 40. Lorsque la simulation se termine, le score final est affiché et une nouvelle simulation est lancée dans la foulée.

Durant la simulation, on utilise la fonction step(id_action) pour choisir l'action à effectuer. La fonction step() retourne ensuite le vecteur de valeurs associé au nouvel état du jeu, la récompense obtenue et un booléen indiquant si la simulation s'est achevée.

3. Mise en place de la politique

Créez un réseau de neurones en utilisant la forme standard pour décrire la politique π_θ. Le réseau sera constitué de 2 couches Linear avec une vingtaine de neurones sur la couche intermédiaire. Les entrées du réseau correspondent aux 4 valeurs associées à l'état du jeu et les sorties du réseau donnent 2 valeurs qui seront transformées en probabilités en utilisant la fonction Softmax. Pour déterminer quelle action va être sélectionnée suivant la politique π_θ, nous utilisons la classe Categorical.

Une fois la politique mise en place, on peut l'utiliser pour effectuer des simulations. Lancez le programme, à ce niveau, le contrôle va être très mauvais car les poids des réseaux ont été initialisés au hasard. Si cette étape fonctionne, la moitié du chemin a été effectué.

4. L'algorithme REINFORCE

L'optimiseur choisi sera un optimiseur Adam avec un learning rate de 0.01 :

```
optimizer = optim.Adam(policy.parameters(), lr=0.01)
```

Pour calculer $Loss(\theta)$, nous devons sauvegarder les informations nécessaires durant la simulation. Voici un squelette de l'algorithme que vous devez construire :

```
Boucle sur 1000 simulations

    # gestion d'une simulation
    state = env.reset()
    done = False
    while not done :
        action, log_prob = π(state)
        state, reward, done, _ = env.step(action)
        sauvegarde reward et log_prob

    # Calcul de la Loss
    …

    # Descente du gradient
    optimizer.zero_grad()
    Loss.backward()
    optimizer.step()
```

Affichez le score obtenu toutes les 20 simulations afin de monitorer la convergence. La convergence devrait être obtenue avant 1000 simulations.

Vous pouvez désactiver la visualisation des simulations pour gagner du temps mais vous pouvez garder une visualisation toutes les 20 simulations pour examiner les progressions faites par l'IA.