Machine learning

Apprentissage renforcé

Table des matières

Formalisation d'un problème d'apprentissage par renforcement

L’interaction agent-environnement dans un processus de décision markovien

Le processus de décision de Markov (PDM) permet de modéliser pratiquement n'importe quel environnement complexe.

Souvent, la dynamique de l’environnement est cachée et inaccessible à l'agent ; cependant, les agents n'ont pas besoin de connaître le PDM précis d'un problème pour apprendre des comportements robustes. Mais la connaissance des PDM est essentielle pour le programmeur car les agents sont généralement conçus en supposant qu'un PDM, même inaccessible, tourne sous le capot.

Processus de décision de Markov

La propriété de Markov stipule que le futur ne dépend que du présent et non du passé. Le processus de Markov consiste en une séquence d'états qui obéissent strictement à la propriété de Markov.

Lorsqu'un problème RL satisfait à la propriété de Markov, c'est-à-dire que le futur ne dépend que de l'état actuel s et de l'action a, mais pas du passé, il est formulé comme un processus de décision de Markov (PDM).

Pour le moment, nous pourrions considérer qu'un PDM consiste essentiellement en cinq-tuplets <S,A,R,p,γ> où les symboles signifient :

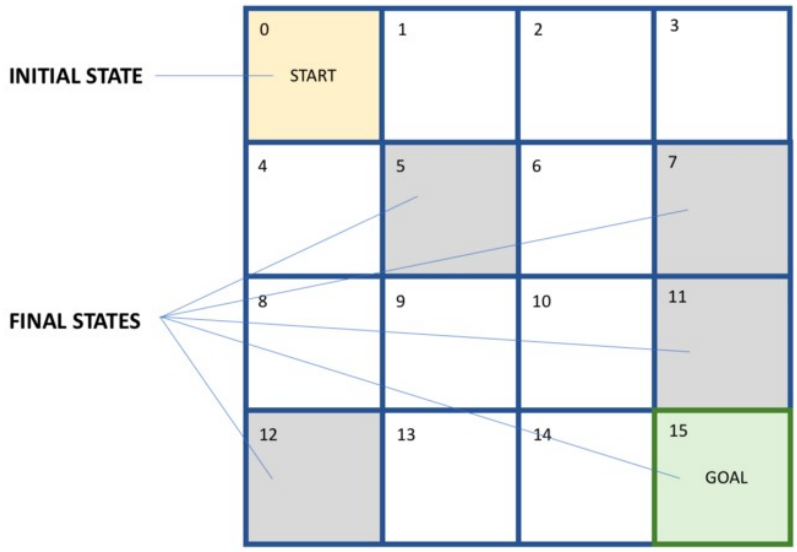
- S - un ensemble d'états
- A - un ensemble d'actions
- R - fonction de récompense
- p - fonction de transition
- γ - facteur d'actualisation

Lorsque nous voulons faire référence à l'état s de l'agent au temps t, nous utiliserons la notation St.

Pour faire référence à un état, une action ou une récompense, nous utiliserons des lettres majuscules et minuscules que nous utiliserons à notre convenance.

États

Un état est une configuration unique et autonome du problème. L'ensemble de tous les états possibles est appelé l'espace d'état. Il existe des états individuels en tant qu'états de départ ou états terminaux. Dans le problème du lac gelé, l'espace d'état de l'environnement est composé de 16 états, il n'y a qu'un seul état initial (l'état 0) et cinq états terminaux (états 5, 7, 11, 12 et 15)



Actions

A chaque état, l'Environnement met à disposition un ensemble d'actions, un espace d'action, à partir duquel l'agent choisira une action. L'agent influence l'environnement par le biais de ces actions, et l'environnement peut changer d'état en réponse à l'action entreprise par l'agent. L'environnement fait

connaître à l'avance l'ensemble des actions disponibles. Dans l'environnement du lac gelé, il y a quatre actions disponibles dans tous les états : HAUT, BAS, DROITE ou GAUCHE.

Maintenant que nous avons présenté les états et les actions, nous pouvons nous rappeler la propriété de Markov. La probabilité de l'état suivant St+1, étant donné l'état actuel St et l'action actuelle At à un moment donné t, sera la même que si vous lui donnez l'historique complet des interactions. En d'autres termes, la probabilité de passer d'un état à un autre en deux occasions distinctes, compte tenu de la même action, est la même, quels que soient tous les états ou actions précédents rencontrés avant ce point. Dans l'exemple du lac gelé, nous savons qu'à partir de l'état 2, l'agent ne peut passer qu'à l'état 1, 3, 6 ou 2, et ceci est vrai que l'état précédent de l'agent ait été 1, 3, 6 ou 2. En d'autres termes, vous n'avez pas besoin de l'historique des états visités par l'agent pour quoi que ce soit.

Nous pouvons classer les espaces d'action en deux types :

- L'espace d'action discret : Lorsque notre espace d'action est constitué d'actions qui sont discrètes. Par exemple, dans l'environnement du lac gelé, notre espace d'action se compose de quatre actions discrètes : haut, bas, gauche, droite, et on l'appelle donc un espace d'action discret. Un environnement discret est un environnement dans lequel l'espace d'action de l'environnement est discret.
- Espace d'action continu : Lorsque notre espace d'action est constitué d'actions qui sont continues. Par exemple, lorsque nous conduisons une voiture, nos actions ont des valeurs continues, telles que la vitesse de la voiture, ou les degrés dont nous avons besoin pour faire tourner le volant, et ainsi de suite. Un environnement continu est un environnement où l'espace d'action de l'environnement est continu.

Fonction de transition

Le passage d'un état à un autre dans lequel l'Agent arrivera (et l'Environnement change d'état) est décidé par la fonction de transition, qui indique la probabilité de passer d'un état à l'état suivant, et est notée p. Elle représente la dynamique à une étape de l'Environnement, c'est-à-dire la probabilité de l'état et des récompenses suivants, étant donné l'état et l'action actuels.

L'environnement répond à l'agent au pas de temps t ; il ne prend en compte que l'état et l'action au pas de temps précédent t-1. Dans sa façon de répondre à l'agent, l’environnement ne se soucie pas des états qui ont été présentés à l'agent plus d'une étape auparavant. Il ne tient pas compte des actions que l'agent a effectuées avant la dernière, ni du montant de la récompense que l’agent perçoit.

Pour cette raison, nous pouvons définir complètement comment l'Environnement décide de l'état et de la récompense en spécifiant la fonction de transition p comme nous l'avons fait ici. La fonction p définit la dynamique du PDM.

En résumé, soulignons que lorsque nous avons un problème réel à l'esprit, nous devrons spécifier le PDM afin de définir formellement le problème, et donc la fonction p. L'agent connaîtra les états, les actions et les récompenses, ainsi que le facteur d'actualisation. Mais la fonction p sera inconnue de l'agent. Bien qu'il ne dispose pas de cette information, l'agent devra quand même apprendre comment atteindre son objectif en interagissant avec l'environnement.

En fonction de l'environnement, les agents peuvent choisir leurs actions de manière déterministe ou stochastique. Voyons comment est la fonction de transition dans les deux cas.

Fonction de transition pour un environnement déterministe

Imaginons l'exemple d'un lac gelé qui n'est pas une surface glissante. Nous pouvons créer cet environnement avec l'argument is_slippery=False pour créer l'environnement en mode déterministe :

```
env = gym.make('FrozenLake-v0', is_slippery=False)
print(env.env.P)
```

Dans ce cas, la probabilité à l'instant t de l'état suivant St+1 étant donné l'état actuel St et l'action At est toujours de 1. En d'autres termes, il s'agit d'un environnement déterministe dans lequel il y a toujours un seul état suivant possible pour l'action. Dans ce cas, nous pouvons considérer la fonction de transition comme une simple table de consultation d'une matrice à deux dimensions (2D). Dans notre exemple du lac gelé, nous pourrions l'obtenir avec env.env.P qui produit la fonction sous forme de dictionnaire. Dans chaque état, à chaque action possible (0,1,2,3) correspond une liste, où chaque élément de la liste est un tuple montrant la probabilité de transition vers l'état, l'état suivant, la récompense, et si le jeu se termine là ou non (terminé= Vrai si l'état suivant est un TROU ou le BUT).

```
{
0: {0: [(1.0, 0, 0.0, False)],
    1: [(1.0, 4, 0.0, False)],
    2: [(1.0, 1, 0.0, False)],
    3: [(1.0, 0, 0.0, False)]},

1: {0: [(1.0, 0, 0.0, False)],
    1: [(1.0, 5, 0.0, True)],
    2: [(1.0, 2, 0.0, False)],
    3: [(1.0, 1, 0.0, False)]},
.
.
.
14: {0: [(1.0, 13, 0.0, False)],
     1: [(1.0, 14, 0.0, False)],
     2: [(1.0, 15, 1.0, True)],
     3: [(1.0, 10, 0.0, False)]},

15: {0: [(1.0, 15, 0, True)],
     1: [(1.0, 15, 0, True)],
     2: [(1.0, 15, 0, True)],
     3: [(1.0, 15, 0, True)]}
}
```

St : état source

At : action

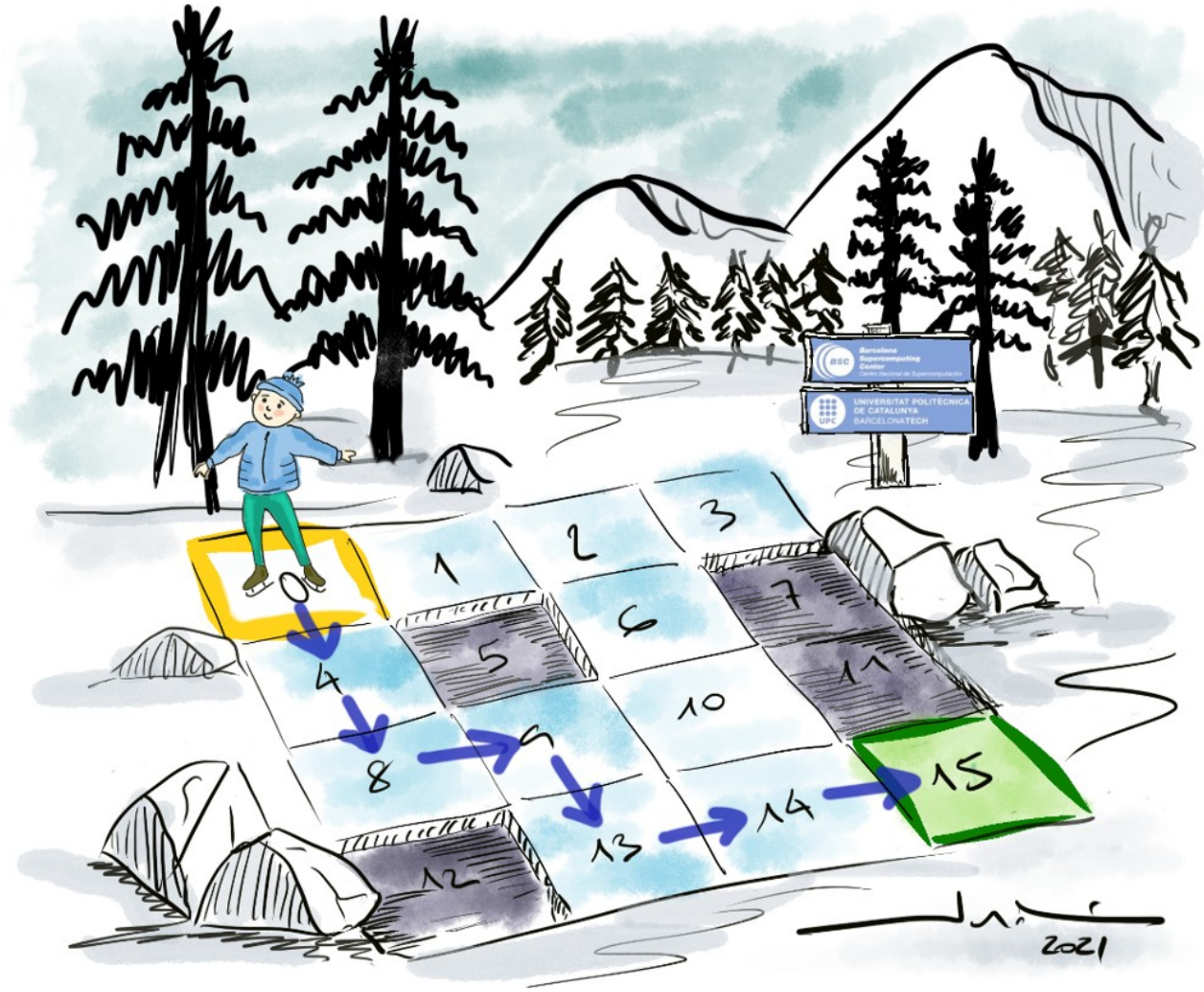
probabilité que l'action aboutisse à l'état cible

St+1 : état cible

récompense

état final OUI/NON

Par exemple, dans cet environnement "non glissant", si nous exécutons la séquence/le "bon plan" indiqué dans le dessin suivant, l'agent arrivera sain et sauf à la fin :



Ici, l'environnement réagit de manière déterministe aux actions de l'agent, mais on peut envisager un environnement qui réagit de manière stochastique aux actions de l'agent pour simuler le glissement.

Fonction de transition pour un environnement stochastique

Dans un environnement stochastique, à l'instant t, la fonction de transition p fait correspondre un tuple de transition (St, At, St+1) à la probabilité correspondante de la transition p d'un état source St à l'état cible St+1 en prenant l'action At.

Pour capturer tous les détails de l'environnement et les réactions possibles aux actions de l'agent, la fonction de transition ne peut pas être représentée par une matrice 2D comme dans le cas de l'environnement déterministe. Nous avons besoin d'une matrice 3D avec les dimensions état source, action et espace cible, où chaque élément indique la probabilité de transition d'un état source St à un état cible St+1 étant donné l'action At.

Pour vérifier que nous parlons bien d'une matrice 3D, nous pouvons obtenir la fonction de transition comme précédemment, mais maintenant pour l'environnement glissant :

```
{
0: {
   0: [
        (0.3333333333333333, 0, 0.0, False),
        (0.3333333333333333, 0, 0.0, False),
        (0.3333333333333333, 4, 0.0, False)],

   1: [
        (0.3333333333333333, 0, 0.0, False),
        (0.3333333333333333, 4, 0.0, False),
        (0.3333333333333333, 1, 0.0, False)],

   2: [
        (0.3333333333333333, 4, 0.0, False),
        (0.3333333333333333, 1, 0.0, False),
        (0.3333333333333333, 0, 0.0, False)],

   3: [
        (0.3333333333333333, 1, 0.0, False),
        (0.3333333333333333, 0, 0.0, False),
        (0.3333333333333333, 0, 0.0, False)]},
.
.
.
14: {
   0: [
        (0.3333333333333333, 10, 0.0, False),
        (0.3333333333333333, 13, 0.0, False),
        (0.3333333333333333, 14, 0.0, False)],

    1: [
        (0.3333333333333333, 13, 0.0, False),
        (0.3333333333333333, 14, 0.0, False),
        (0.3333333333333333, 15, 1.0, True)],

    2: [
        (0.3333333333333333, 14, 0.0, False),
        (0.3333333333333333, 15, 1.0, True),
        (0.3333333333333333, 10, 0.0, False)],

    3: [
        (0.3333333333333333, 15, 1.0, True),
        (0.3333333333333333, 10, 0.0, False),
        (0.3333333333333333, 13, 0.0, False)]},

15: {
    0: [(1.0, 15, 0, True)],

    1: [(1.0, 15, 0, True)],

    2: [(1.0, 15, 0, True)],

    3: [(1.0, 15, 0, True)]}
}
```

St : état source

At : action

probabilité que l'action aboutisse à l'état cible

St+1 : état cible

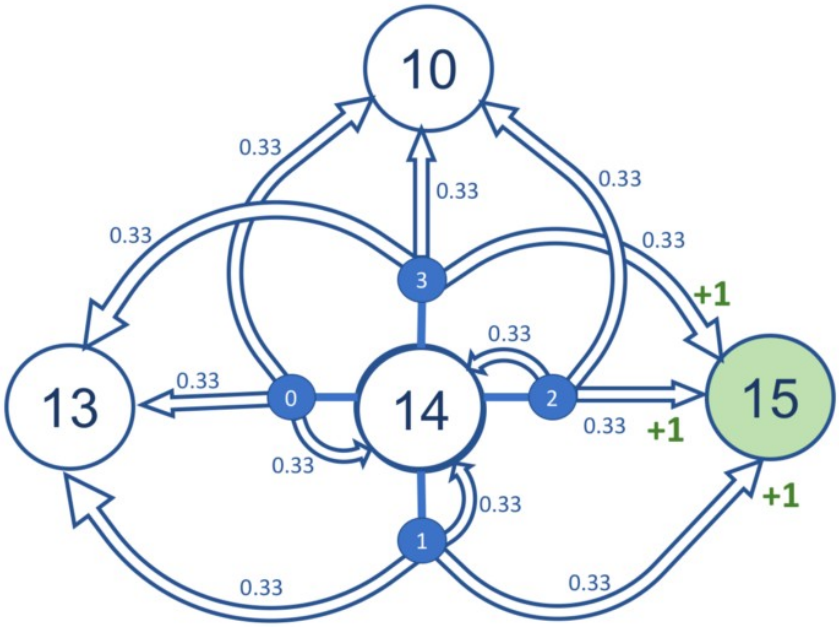
récompense

état final OUI/NON

Cette fonction de transition signifie qu'il y a 33,3 % de chances que nous passions à l'état vraiment prévu et 66,6 % de chances que nous passions à des directions orthogonales. Il y a également une chance que nous restions à l'état d'où nous venons si nous sommes près du mur.

On peut représenter visuellement l'environnement sous la forme d'un graphe dont :

- les nœuds représentent les états
- les bords, étiquetés avec des probabilités (et des récompenses), représentent une transition possible d'un état à l'autre.



Maintenant, si nous exécutons plusieurs fois la même séquence d'actions du "bon plan", nous pouvons constater qu'elle ne se comporte pas de manière déterministe, donnant des résultats très différents.

Fonction de récompense et facteur d'actualisation

Une fois qu'une action est effectuée, l'environnement délivre une récompense comme mesure de la qualité des transitions en utilisant une fonction de récompense.

Fonction de récompense

La fonction de récompense est généralement désignée par r(s,a) ou r(s,a,s'). Elle représente la récompense que notre agent obtient lorsqu'il passe de l'état s à l'état s' en effectuant une action a. Il s'agit simplement d'une valeur scalaire que l'agent obtient à chaque pas de temps (ou à chaque nombre fixe de pas de temps) de l'environnement (elle peut être positive ou négative, grande ou petite).

Il est important de noter que le mot "renforcement" et "apprentissage par renforcement" est un terme issu des sciences comportementales. Il fait référence à un stimulus délivré immédiatement après un comportement afin de rendre ce comportement plus susceptible de se reproduire à l'avenir. Le fait que ce nom soit emprunté n'est pas une coïncidence. Nous devons concevoir la récompense comme

un mécanisme de rétroaction qui indique à l'agent qu'il a choisi l'action appropriée. La récompense sera notre façon de dire à l'agent qu'il fait un "bon travail" ou un "mauvais travail".

Prenons l'exemple d'un agent qui souhaite apprendre à s'échapper d'un labyrinthe. Quels signaux de récompense encourageront l'agent à s'échapper du labyrinthe aussi vite que possible ? La récompense peut être de -1 pour chaque pas de temps que l'agent passe à l'intérieur du labyrinthe. Lorsque l'agent s'échappe, il reçoit une récompense de +10, et l'épisode se termine. Considérons maintenant un agent qui souhaite apprendre à tenir en équilibre une assiette de nourriture sur sa tête. Quels signaux de récompense encourageront l'agent à maintenir l'assiette en équilibre le plus longtemps possible ? Par exemple, il peut y avoir une récompense de +1 pour chaque pas de temps où l'agent garde l'assiette en équilibre sur sa tête. Si l'assiette tombe, l'épisode se termine.

En résumé, la récompense donne à l'agent un feedback sur son succès, et la récompense obtenue devrait renforcer le comportement de l'agent de manière positive ou négative. Néanmoins, elle ne reflète que le succès de l'activité récente de l'agent et non tous les succès obtenus par l'agent jusqu'à présent. Ce qu'un agent essaie d'obtenir, c'est la plus grande récompense accumulée au cours de sa séquence d'actions, et nous avons besoin d'un autre feedback, le gain potentiel ou discounted return. Mais avant d'introduire le gain potentiel, il faut aborder le dernier composant d'un PDM, le facteur d'actualisation.

Facteur d'actualisation

La tâche que l'agent tente de résoudre peut avoir ou non une fin naturelle. Les tâches qui ont une fin naturelle, comme un jeu, sont appelées tâches épisodiques. La séquence de pas de temps entre le début et la fin d'une tâche épisodique est appelée un épisode. L'environnement Frozen-Lake présente des tâches épisodiques parce qu'il y a des états terminaux ; il y a des états de but et d'échec clairs. À l'inverse, les tâches qui n'ont pas de fin naturelle sont appelées tâches continues, comme l'apprentissage de la marche avant.

En raison de la possibilité de séquences infinies d'étapes temporelles, nous avons besoin d'un moyen d'actualiser la valeur des récompenses dans le temps ; autrement dit, nous avons besoin d'un moyen de dire à l'agent qu'il vaut mieux obtenir des +1 tôt que tard. Ainsi, nous utilisons généralement une valeur réelle positive inférieure à un pour actualiser de manière exponentielle la valeur des récompenses futures. Plus la récompense est loin dans le futur, moins elle a de valeur dans le présent.

Ce paramètre est appelé facteur d'actualisation, gamma, ou taux d'actualisation désigné par γ, et sa plage est [0,1]. L'agent utilise le facteur d'actualisation pour ajuster l'importance des récompenses dans le temps. Plus l'Agent reçoit des récompenses tardives, moins elles sont intéressantes pour les calculs actuels. En d'autres termes, l'Agent est plus intéressé par l'obtention de récompenses qui arrivent plus tôt et sont plus probables que les récompenses qui arrivent plus tard et sont moins probables.

Configuration du problème

Le cycle d'apprentissage par renforcement le plus primitif peut se résumer ainsi : à chaque pas de temps t, l'agent reçoit un état s, et choisit une action a, en suivant une stratégie (une politique). Selon la dynamique de l'environnement, l'agent reçoit une récompense scalaire r et passe à l'état

suivant s'. Dans un problème épisodique, ce processus se poursuit jusqu'à ce que l'agent atteigne un état terminal, puis il recommence.

Épisode et trajectoire

L'agent interagit avec l'environnement en effectuant certaines actions, en partant de l'état initial et en atteignant l'état final. Cette interaction Agent-Environnement à partir de l'état initial S0 jusqu'à l'état final est appelée un épisode.

Mais parfois, nous pouvons être intéressés par une partie seulement de l'interaction Agent-Environnement. A un pas de temps arbitraire t, l'interaction entre l'agent et l'environnement du PDM a évolué comme une séquence d'états, d'actions et de récompenses. Chaque étape est définie comme une transition :

$$\text{transition } k = (s_k, a_k, r_{k+1})$$

Nous définissons une trajectoire, juste une partie d'un épisode, composée par une séquence de transitions. Une trajectoire est légèrement plus flexible qu'un épisode car il n'y a pas de restrictions sur sa longueur ; elle peut correspondre à un épisode complet ou juste à une partie d'un épisode. Nous représentons une trajectoire par τ :

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$$

Nous désignons la longueur de la trajectoire par un H (ou h) où H signifie Horizon

Récompense à long terme ou gain potentiel – discounted return

En général, les agents sont conçus pour maximiser la récompense cumulative qu'ils reçoivent à long terme. Maintenant que nous avons introduit le gamma, nous pouvons définir comment le gain potentiel est calculé à partir du gamma et des récompenses. La récompense à long terme, notée Gt, au pas de temps t, est défini ainsi :

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Pour chaque pas de temps t, nous calculons Gt comme une somme de récompenses ultérieures, mais les récompenses plus éloignées sont multipliées par le facteur d'actualisation élevé à la puissance du nombre de pas dont nous sommes éloignés.

- Si gamma est égal à 1, le gain potentiel est juste égal à la somme de toutes les récompenses ultérieures.
- Si gamma est égal à 0, le gain potentiel sera juste la récompense immédiate sans aucun état ultérieur.

Lorsque nous fixons le facteur d'actualisation à une petite valeur (proche de 0), nous accordons plus d'importance aux récompenses immédiates qu'aux récompenses futures. Lorsque nous fixons le facteur d'actualisation à une valeur élevée (proche de 1), cela implique que nous accordons plus d'importance aux récompenses futures qu'aux récompenses immédiates. Les valeurs extrêmes ne

sont utiles que dans des cas particuliers, et la plupart du temps, gamma est fixé à une valeur intermédiaire.

Un agent accorde de l'importance aux récompenses immédiates et futures en fonction des tâches. Dans certaines tâches, les récompenses futures sont plus souhaitables que les récompenses immédiates et vice versa. Dans une partie d'échecs, par exemple, le but est de battre le roi de l'adversaire. Accordons plus d'importance à la récompense immédiate, qui est acquise par des actions telles que notre pion battant toute pièce d'échec adverse. L'agent apprendra probablement à réaliser ce sous-objectif au lieu d'apprendre l'objectif important.

Les actions ont des conséquences à court et à long terme, et l'agent doit comprendre les effets complexes de ses actions sur l'environnement. Comme nous le verrons tout au long de cette série, le gain potentiel aidera l'agent dans cette tâche : l'agent choisira toujours une action dans le but de maximiser le gain potentiel. Mais comme nous le verrons, en général, un agent ne peut pas prédire avec une certitude totale ce que sera la récompense future, et donc la récompense à long terme, il doit donc s'appuyer sur une prédiction ou une estimation selon la famille de méthodes que nous utiliserons pour apprendre. C'est là que les concepts clés de fonction de politique et de fonction de valeur entrent en jeu.

Politique

Voyons comment l'Agent prend la décision d'atteindre ses objectifs : trouver une séquence d'actions qui maximisera le gain potentiel actualisé G pendant un épisode. En d'autres termes, l'Agent doit avoir un plan, une séquence d'actions allant de l'état START à l'état GOAL.

Nous avons précédemment présenté un "bon plan" pour l'exemple du lac gelé, qui semble intuitivement le meilleur. Mais lorsque nous l'exécutons dans un environnement stochastique, même le meilleur des plans tombe, car les actions entreprises ne fonctionneront pas toujours comme prévu. Rappelez-vous que dans l'environnement du lac gelé, les actions non intentionnelles ont une probabilité élevée : 66,6% contre 33,3%.

Et en particulier, il arrive qu'en raison de la stochasticité de l'environnement, notre Agent atterrisse sur une cellule non couverte par notre plan. Alors ? Que faire ? Eh bien, nous avons besoin d'un plan pour chaque état possible, un "plan universel", une politique qui couvre tous les états possibles.

Une politique définit le comportement de l'agent dans un environnement. C'est la stratégie (par exemple, un ensemble de règles) que l'agent utilise pour déterminer la prochaine action en fonction de l'état actuel. Généralement désignée par ,une politique est une fonction qui détermine la prochaine action a à entreprendre étant donné un état s.

La politique que nous venons de mentionner est appelée politique déterministe et on la note :

$$\pi (s)$$

Elle indique à l'agent d'effectuer une action particulière a dans un état s. Ainsi, la politique déterministe fait correspondre à l'état une action particulière.

Mais en général, nous aurons affaire à des politiques plus générales, et qui seront définies comme une probabilité et non comme une action concrète. En d'autres termes, il s'agit d'une politique stochastique qui a une distribution de probabilité sur les actions qu'un agent peut prendre à un état donné. Ainsi, au lieu d'effectuer la même action à chaque fois que l'agent visite l'état, l'agent

effectue des actions différentes à chaque fois sur la base d'une distribution de probabilité renvoyée par la politique stochastique. Une politique stochastique est généralement désignée par :

$$\pi (a | s)$$

Comme nous le verrons plus loin dans cette série, la politique peut changer à mesure que l'agent acquiert de l'expérience au cours du processus d'apprentissage. Par exemple, l'agent peut partir d'une politique aléatoire, où la probabilité de toutes les actions est uniforme : entre-temps, l'agent devrait apprendre à optimiser sa politique pour atteindre une meilleure politique. Lors d'une itération initiale, l'agent effectue une action aléatoire dans chaque état et tente d'apprendre si l'action est bonne ou mauvaise en fonction de la récompense obtenue. Au cours d'une série d'itérations, l'agent apprendra à effectuer de bonnes actions dans chaque état, ce qui donne une récompense positive. Finalement, l'agent apprendra une bonne politique ou une politique optimale.

Les politiques stochastiques sont classées en deux grandes familles :

- Les politiques catégoriques : lorsque l'espace d'action est discret, et que la politique utilise une distribution de probabilité catégorique sur l'espace d'action. C’est le cas pour l'environnement du lac gelé.
- Les politiques gaussiennes : lorsque notre espace d'action est continu, et que la politique utilise une distribution de probabilité gaussienne sur l'espace d'action.

Même pour des environnements raisonnablement simples, nous pouvons avoir une grande variété de politiques. Nous avons alors besoin d'une méthode pour trouver automatiquement les politiques optimales.

La politique optimale est celle qui procure à l'agent une bonne récompense et l'aide à atteindre son objectif. Elle indique à l'agent d'effectuer l'action correcte dans chaque état afin qu'il puisse recevoir une bonne récompense. C'est là que les fonctions de valeur entrent en jeu.

Prédire les récompenses futures : Fonctions de valeur

Une fonction de valeur détermine ce qui est bon pour l'agent à long terme, contrairement à la récompense immédiate. Nous avons deux types de fonctions de valeur qui aident à apprendre et à trouver les politiques optimales pour l'agent :

- la fonction état-valeur
- la fonction action-valeur,

La fonction état-valeur, également appelée fonction V, mesure la qualité de chaque état. Elle porte à notre connaissance le gain potentiel que nous pouvons attendre à l'avenir si nous partons de cet état. En d'autres termes, elle indique s'il est bon ou mauvais de se trouver dans un état particulier en fonction du gain potentiel G lorsque nous suivons une certaine politique.

Cependant, le gain potentiel G n'est pas très utile en pratique, car il a été défini pour chaque épisode spécifique, et peut donc varier considérablement, même pour un même état. Mais, si nous allons à l'extrême et calculons l'espérance mathématique $\mathbb{E}[.]$ du gain potentiel pour un état en faisant la moyenne d'un grand nombre d'épisodes, nous obtiendrons une valeur beaucoup plus utile pour la fonction V.

Nous pouvons étendre la définition de la fonction état-valeur, en définissant une valeur pour chaque paire état-action. On parle de fonction action-valeur, également connue sous le nom de fonction Q. Elle nous indique dans quelle mesure il est bon ou mauvais de prendre une action spécifique parmi l'ensemble des actions que nous pouvons choisir à partir de l'état dans lequel nous sommes.

s, s', S, S'	states in an Environment
a, A	an action
r, R	a reward
t	we use t to denote discrete time step (natural number including zero)
T	final time step of an episode, or of the episode including time step t
a_t, A_t	action at time t
s_t, S_t	state at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
r_t, R_t	reward at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
$\mathcal{S}$	set of all nonterminal states in a MDP
$\mathcal{S}^+$	set of all states, including the terminal state in a MDP
$\lvert\mathcal{S}\rvert$	number of elements in set $\mathcal{S}$
$\mathcal{A}(s)$	set of all actions available in state s
$\mathcal{R}$, $r()$	reward function
$r(s,a)$	expected immediate reward from state s after action a
$r(s,a,s')$	expected immediate reward on transition from s to s' under action a
$\mathcal{P}$, $p()$	transition function in a MDP
$p(s' \mid s,a)$	probability of transition to state s', from state s taking action a
$p(s', r \mid s,a)$	probability of transition to state s' with reward r, from state s and action a
γ	discount factor (where $0 \leq \gamma \leq 1$)

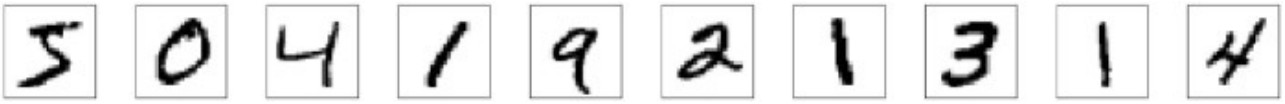
A MDP is defined by $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma >$

τ	trajectory
h, H	horizon, the legth of a trajectory
transition k=	(s_k, a_k, r_{k+1})
trajectory	$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$
$\doteq$	equality relationship that is true by definition
G_t	return following time t
	discounted return at time t $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
π	policy (decision-making rule)
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\|s)$	probability of taking action a in state s under *stochastic* policy π

Les réseaux de neurones artificiels

Classification des chiffres manuscrits

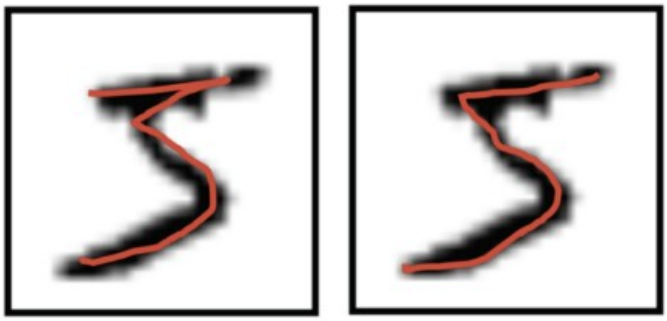
Nous allons créer un modèle mathématique pour identifier des chiffres manuscrits :



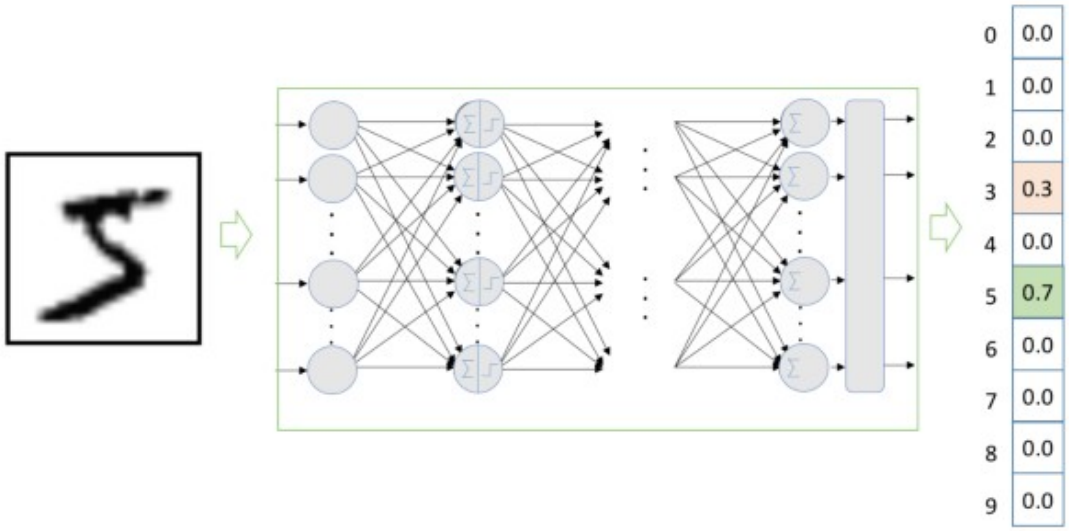
L'objectif est de créer un réseau neuronal qui, étant donné une image, puisse identifier le nombre représenté. Par exemple, si nous donnons au réseau la première image, nous nous attendons à ce qu'il réponde qu'il s'agit d'un 5, la suivante d'un 0, la suivante d'un 4, et ainsi de suite.

Problème de classification avec incertitude

Nous avons affaire à un problème de classification. Le modèle doit classer une image entre 0 et 9. Mais parfois,une part de doute peut venir perturber la reconnaissance, même pour un lecteur humain. Par exemple, l’image suivante représente un 5 ou un 3 ?



Pour restituer son classement, le réseau neuronal retourne un vecteur à 10 positions indiquant la probabilité de chacun des dix chiffres possibles :



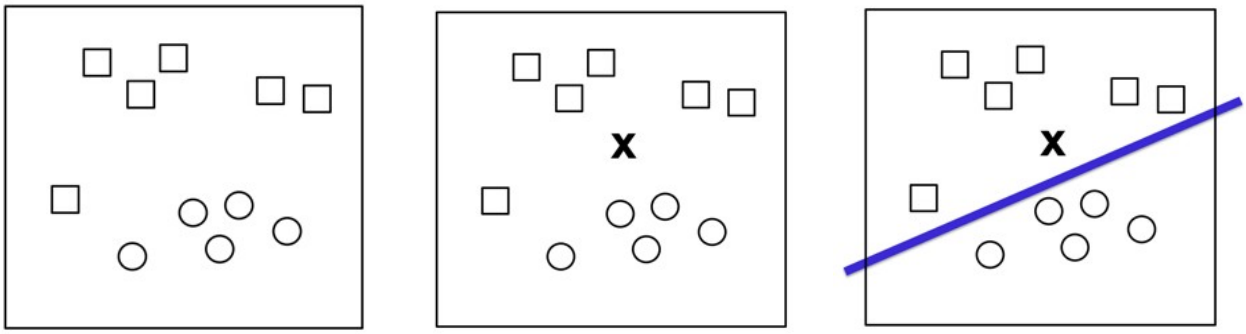
Format et manipulation des données

Le jeu de données MNIST contient 60 000 images de chiffres faits à la main. Ce jeu de données d'images en noir et blanc (les images contiennent des niveaux de gris) a été normalisé à 28×28 pixels. Nous allons effectuer une transformation pour passer de cette entrée à 2 dimensions à un vecteur à une dimension. En d'autres termes, nous allons représenter la matrice de 28×28 nombres par un vecteur de 784 nombres, via une concaténation ligne par ligne.

L'étiquetage sera réalisé par un vecteur de 10 positions, où la position correspondant au chiffre qui représente l'image contient un 1, les autres positions valant toutes 0. Ce processus est connu sous le nom de **one-hot encoding**. Par exemple, le nombre 7 sera codé comme suit :

Le composant de base : le neurone

Afin de montrer ce qu'est un neurone artificiel simple, imaginons un exemple simple où nous avons un ensemble de points dans un plan bidimensionnel et où chaque point est déjà étiqueté "carré" ou "cercle". Étant donné un nouveau point "x", nous voulons savoir quelle étiquette lui correspond. Une approche courante consiste à tracer une ligne qui sépare les deux groupes et à utiliser cette ligne comme classificateur :



Dans ce cas, les données d'entrée seront matérialisées par un vecteur (x1, x2) indiquant les coordonnées du point dans cet espace bidimensionnel. Pour indiquer si le point doit être classé comme "carré" ou "cercle", notre « fonction » renverra 0 ou 1, selon si le point se situe au-dessus ou au-dessous de la ligne. Elle peut être définie par :

$$y = w_1 x_1 + w_2 x_2 + b$$

Ou d'une manière plus générale :

$$y = W * X + b$$

Pour classer les éléments d'entrée X, qui dans notre cas sont bidimensionnels, nous devons « apprendre » un vecteur de poids W = (w1, w2) de la même dimension que les vecteurs d'entrée, et un biais b. Ce processus relève de ce qu'on appelle un neurone artificiel qui fait le produit scalaire de son vecteur de poids W et du vecteur d'entrée X, pour au final ajouter le biais b. Le résultat de l'opération est ensuite passé à une fonction non linéaire « d'activation » pour produire un résultat binaire 0 ou 1.

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ si } z \geq 0 \\ 0 \text{ si } z < 0 \end{cases}$$

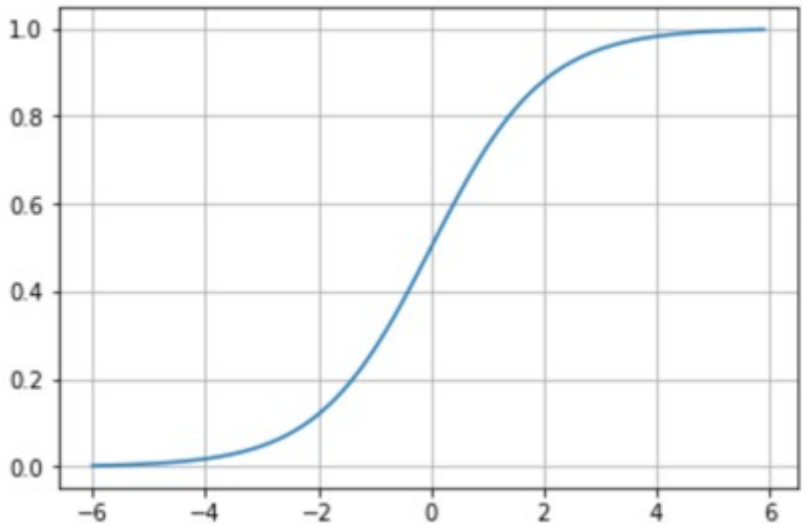
Il existe diverses fonctions d'activation. Pour cet exemple, nous utiliserons la fonction sigmoïde :

$$y = \frac{1}{1+e^{-z}}$$

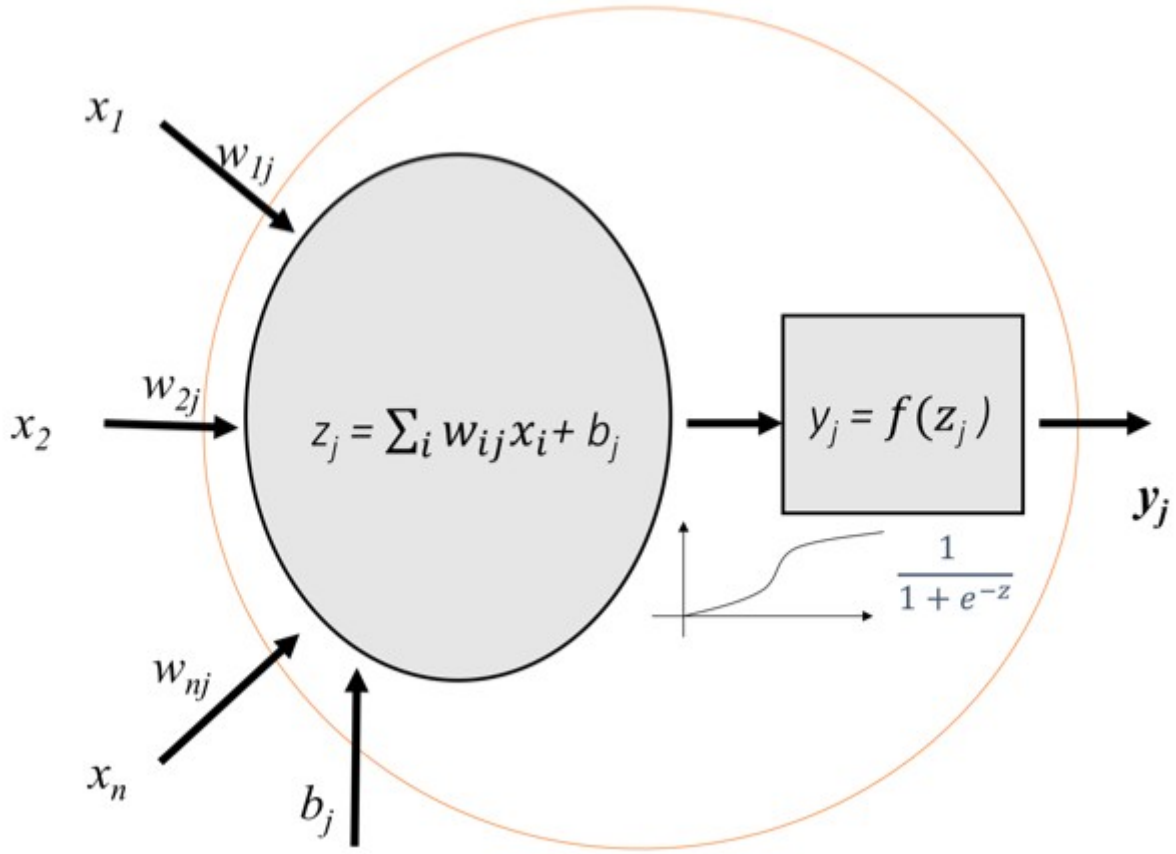
Cette fonction tend toujours à donner des valeurs proches de 0 ou de 1 :

- Si z est raisonnablement grande et positive, la fonction sigmoïde prend la valeur de 1.
- Si z est raisonnablement grande et négative, la fonction sigmoïde a une valeur proche de 0.

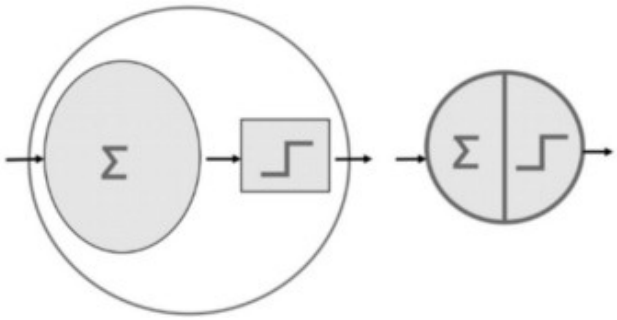
Graphiquement, la fonction sigmoïde présente cette forme :



Nous venons de présenter l'architecture la plus simple qu'un réseau neuronal puisse avoir, celle du Perceptron, inventée en 1957 par Frank Rosenblatt :



Voici 2 visualisations simplifiées qu'on peut couramment trouver pour représenter un perceptron :

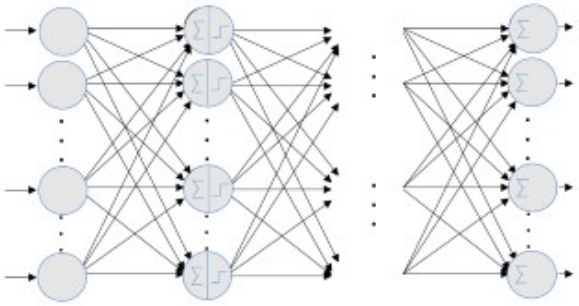


Perceptron multicouche

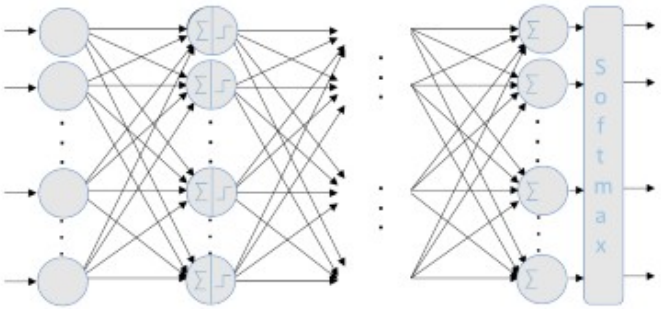
La littérature fait référence à un perceptron multicouche (MLP) pour désigner des réseaux neuronaux qui ont :

- une couche d'entrée,
- une ou plusieurs couches composées de perceptrons, appelées couches cachées
- et une couche finale avec plusieurs perceptrons appelée couche de sortie.

En général, on parle de Deep Learning lorsque le réseau de neurones qui constitue le modèle est composé de plusieurs couches cachées :



Les MLP sont souvent utilisés pour la classification, et plus particulièrement lorsque les classes sont exclusives (on parle de classes exclusives lorsque la limite supérieure d'une classe est la même que la limite inférieure de la classe suivante) comme dans le cas de la classification des images de chiffres (en classes de 0 à 9). Dans ce cas, la couche de sortie renvoie la probabilité d'appartenance à chacune des classes, souvent grâce à une fonction appelée softmax :



Dans une première approche très réductrice, on peut considérer la fonction softmax comme une généralisation de la fonction sigmoïde en vue de procéder à la classification de plus de deux classes.

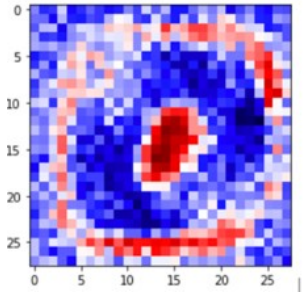
Fonction d'activation Softmax

Pour une image donnée, un tel modèle pourra prédire distinguer un 5, par exemple en étant sûr à 70% qu'il s'agit d'un 5. Il pourrait aussi s’agir d’un 3 avec une probabilité de 20% et il pourrait y avoir une certaine probabilité que ce soit un tout autre chiffre. Bien que nous soyons amenés à considérer que l’on a affaire à un 5 puisqu'il s'agit de la probabilité la plus élevée établie par notre modèle, cette approche consistant à utiliser une distribution de probabilités peut nous donner une meilleure idée de la confiance que nous avons dans notre prédiction. C'est une bonne chose car les chiffres sont tracés à la main, et il est certain que pour beaucoup d'entre eux, nous ne pouvons pas procéder à une reconnaissance avec 100% de certitude.

Par conséquent, pour cet exemple de classification, nous obtiendrons un vecteur de 10 probabilités, exprimées entre 0 et 1, correspondant chacune à un des 10 chiffres à reconnaître et dont la somme fait 1.

Pour parvenir à un tel résultat, notre réseau neuronal peut mettre en œuvre une couche de sortie utilisant la fonction d'activation softmax, qui calcule « l'évidence » qu'une certaine image appartienne à une classe particulière, ces éléments d'évidence étant ensuite convertis en probabilités d'appartenance à chacune des classes possibles.

Une approche pour mesurer la preuve qu'une certaine image appartient à une classe particulière consiste à faire une somme pondérée de la preuve de l'appartenance de chacun de ses pixels à cette classe. Prenons le cas du nombre 0 et supposons que nous ayons un modèle comme celui présenté ci-dessous :



Les pixels en rouge représentent des poids négatifs, réduisant la preuve d'appartenance à la classe cible. Les pixels en bleu représentent des poids positifs, augmentant la preuve d'appartenance à la classe. La couleur blanche représente la valeur neutre.

Par conséquent, l'utilisation d'une métrique basée sur l'addition lorsque nous sommes dans la zone bleue et la soustraction lorsque nous sommes dans la zone rouge semble raisonnable.

Une fois que les preuves d'appartenance à chacune des 10 classes ont été calculées, elles doivent être converties en probabilités. Pour cela, softmax utilise la valeur exponentielle des preuves calculées et les normalise ensuite de façon à ce que la somme soit égale à 1, formant ainsi une distribution de probabilité. La probabilité d'appartenir à la classe i est :

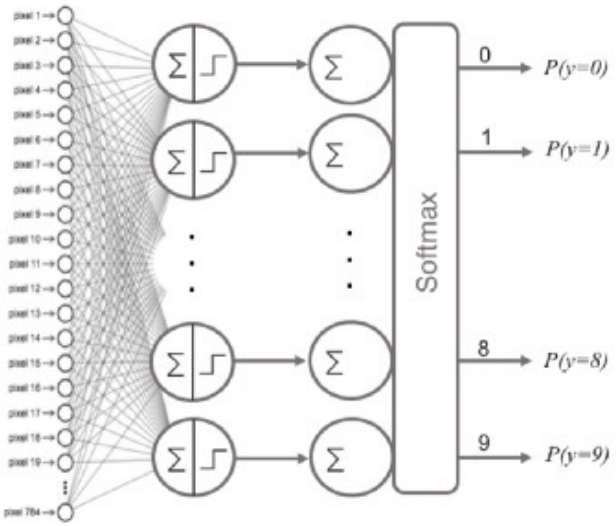
$$Softmax_i = \frac{e^{evidencia_i}}{\sum_j e^{evidencia_j}}$$

Intuitivement, l'effet obtenu avec l'utilisation des exponentielles est qu'une unité de preuve supplémentaire a un effet multiplicateur et une unité de moins a l'effet inverse.

Si notre prédiction est bonne, il y aura une seule entrée dans le vecteur avec une valeur proche de 1, tandis que les autres entrées seront proches de 0. Si notre prédiction est faible, il y aura plusieurs étiquettes possibles, qui auront plus ou moins la même probabilité.

Modèle de réseau neuronal pour les chiffres manuscrits

Nous allons définir un réseau de neurones très simple à deux couches :



L’entrée du réseau est de taille 784 (28×28).

La couche suivante de 10 neurones équipés d’une fonction d'activation sigmoïde "distille" les données d'entrée pour obtenir les 10 sorties souhaitées requises en entrée de la couche suivante.

La couche finale est une couche softmax de 10 neurones, retournant 10 valeurs, chacune d’entre elles représentant la probabilité que l'image du chiffre actuel appartienne à un des 10 chiffres possibles.

Processus d'apprentissage

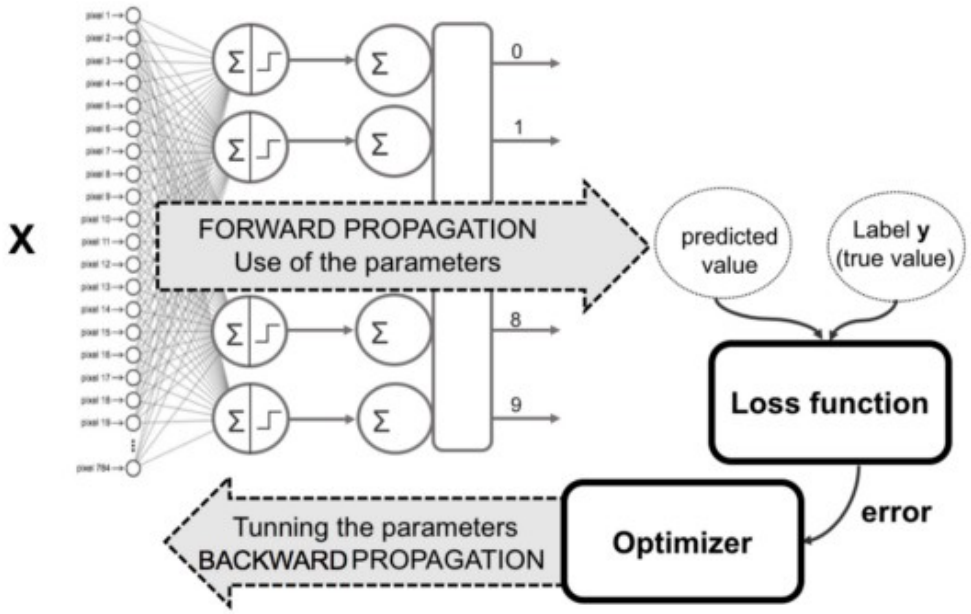
L'entraînement de notre réseau neuronal, c'est-à-dire l'apprentissage des valeurs de nos paramètres (poids W et biais b), est l’essence même du Deep Learning. Ce processus d'apprentissage est un processus itératif avec une phase forward/aller, ou propagation vers l’avant, et une phase backward/retour appelée rétro-propagation de l'information.

La propagation vers l'avant se produit lorsque les données traversent l'ensemble du réseau neuronal pour calculer les prédictions. Il s'agit de faire passer les données d'entrée à travers le réseau de telle sorte que tous les neurones appliquent leur transformation aux informations qu'ils reçoivent des neurones de la couche précédente et les envoient aux neurones de la couche suivante. Le résultat de prédiction est obtenu lorsque les données ont traversé toutes les couches.

Une fonction **loss** est alors utilisée pour estimer l'erreur, c’est-à-dire pour comparer et mesurer à quel point notre résultat de prédiction est bon ou mauvais par rapport au résultat correct. Rappelez-vous que nous sommes dans un environnement d'apprentissage supervisé et que nous avons connaissance de la valeur attendue (étiquette). Idéalement, nous voulons que le coût soit nul, c'est-à-dire sans divergence entre la valeur estimée et la valeur attendue. Au fur et à mesure de l'apprentissage du modèle, les poids d’interconnexion des neurones sont progressivement ajustés jusqu'à l'obtention des meilleurs prédictions possibles.

Une fois calculée, l’erreur est propagée en arrière : c’est la rétro-propagation : https://fr.wikipedia.org/wiki/R%C3%A9tropropagation_du_gradient

En partant de la couche de sortie, l’erreur se propage vers tous les neurones de la couche précédente (on parle de couche cachée) et contribuant à la sortie. On doit préciser que les neurones ne reçoivent qu'une fraction de l’erreur totale, fonction de la contribution relative de chaque neurone à la sortie. Ce processus est répété, couche par couche, jusqu'à ce que tous les neurones du réseau aient reçu le signal décrivant leur contribution relative à l’erreur totale. Maintenant que nous avons réparti ces informations, nous pouvons ajuster les poids des connexions entre les neurones.



Nous pouvons considérer le processus d'apprentissage comme un problème d'optimisation globale où les paramètres (poids et biais) doivent être ajustés de manière à minimiser la fonction loss choisie. Dans la plupart des cas, ces paramètres ne peuvent être résolus analytiquement, mais en général, ils peuvent être bien approchés avec un optimiseur (algorithmes d'optimisation itérative – cf descente de gradient, adam pour adaptative momentum estimation , BFGS pour Broyden-Fletcher-Goldfarb-Shanno).

Pour en savoir plus sur les optimiseurs :

- https://penseeartificielle.fr/meilleur-optimizer-ranger-radam-lookahead-fastai/
- https://www.datarobot.com/blog/introduction-to-optimizers/

Cette technique modifie les poids par petits incréments à l'aide du calcul de la dérivée (ou gradient) de la fonction loss, ce qui nous donne une indication sur la direction à adopter pour "descendre" vers le minimum global ; ceci est fait en général par itérations successives en faisant travailler le réseau sur des lots de données (on parle de batch).

Entropie croisée ou Cross-Entropy

Il existe un large éventail de fonctions loss pour notre modèle de réseau neuronal. Par exemple, le lecteur connaît certainement la fonction Mean Squared Error (MSE) couramment utilisée pour la

régression. Pour la classification, la fonction loss habituellement utilisée est l'entropie croisée, qui permet de mesurer la différence entre deux distributions de probabilité.

En Deep Learning, l'entropie croisée, ou log loss, mesure les performances d'un modèle de classification dont la sortie est une valeur de probabilité comprise entre 0 et 1.

L'entropie croisée augmente à mesure que la probabilité prédite s'écarte de l'étiquette réelle. Un modèle parfait aurait une perte logarithmique de 0. Dans un problème de classification binaire, où le nombre de classes est de 2, l'entropie croisée peut être calculée comme suit :

$$-(y\log(p)+(1-y)\log(1-p))$$

Pour en revenir à un problème de classification multi-classes, nous calculons les valeurs de log loss pour chaque étiquette de classe par observation et additionnons le résultat :

$$-\sum_{c=0}^{C} y_{o,c}\log(p_{o,c})$$

avec :

- C : nombre de classes (10 dans notre cas)
- log : fonction logarithme naturel
- y : indicateur binaire (0 ou 1) précisant si l'étiquette de classe c est la classification correcte de l'observation o
- p : probabilité prédite que l'observation o soit de la classe c

La retro-propagation dans la pratique

On initialise un tenseur result avec des valeurs aléatoires. A ce stade, result n'a pas de gradients.

```
import torch

result = torch.randn(target.shape[0], target.shape[1], requires_grad=True)
print("\nrandom result")
print(result)
print("\ngradient sur result")
print(result.grad)
```

```
random result
tensor([[-0.6163,  0.8821],
        [ 1.7030, -1.0074],
        [-1.2193,  0.5078],
        [ 0.0218,  0.2029]], requires_grad=True)

gradient sur result
None
```

On crée un tenseur objectif :

```
target = torch.tensor([[1.12, 0.34],
                       [4.90, 2.34],
                       [7.45, 6.12],
                       [0.23, 1.12]])
```

On choisit une fonction loss et on calcule l'erreur entre result et target

```
loss = torch.nn.MSELoss()
loss_calculation = loss(result, target)
print("\nloss MSE")
print(loss_calculation)

print("\nrecalcul manuel du loss MSE")
temp = (result - target)**2
manuel = torch.sum(temp)/(target.shape[0]*target.shape[1])
print(manuel)
```

```
loss MSE
tensor(16.5341, grad_fn=<MseLossBackward0>)

recalcul manuel du loss MSE
tensor(16.5341, grad_fn=<DivBackward0>)
```

On retro-propage l'erreur. Après cette rétro-propagation, result dispose d'un gradient.

```
loss_calculation.backward()
print("\ngradient sur result après calcul du loss")
print(result.grad)
```

```
gradient sur result après calcul du loss
tensor([[-0.4341,  0.1355],
        [-0.7993, -0.8369],
        [-2.1673, -1.4030],
        [-0.0521, -0.2293]])
```

Comment le réseau de neurones effectue ses calculs ?

On initialise un réseau neurones avec 2 paramètres en entrée et 2 paramètres en sortie :

```
import torch

inputSize = 2
outputSize = 2

model = torch.nn.Sequential(
        torch.nn.Linear(inputSize, outputSize)
    )
```

```
print("\npoids du réseau")
print(model[0].weight)
print("\nbiais du réseau")
print(model[0].bias)
```

```
poids du réseau
Parameter containing:
tensor([[-0.2086,  0.2867],
        [ 0.6439, -0.0158]], requires_grad=True)

biais du réseau
Parameter containing:
tensor([-0.1744,  0.4714], requires_grad=True)
```

On initialise un tenseur à fournir au réseau, tous ses paramètres étant égaux à 2 :

```
v = torch.tensor([2.0,2.0])
```

On applique le réseau sur ce vecteur et on recalcule la sortie manuellement :

```
print("\ncalcul manuel de la sortie du réseau")
for j in range(outputSize):
    a = model[0].weight[j,0]
    b = model[0].weight[j,1]
    print(a*v[0]+b*v[1]+model[0].bias[j])

print("\nsortie du réseau")
print(model(v))
```

```
calcul manuel de la sortie du réseau
tensor(-0.0181, grad_fn=<AddBackward0>)
tensor(1.7277, grad_fn=<AddBackward0>)

sortie du réseau
tensor([-0.0181,  1.7277], grad_fn=<AddBackward0>)
```

Lorsqu'on travaille en mode batch, les calculs sont effectués en parallèle :

```
batch = torch.randn(12, outputSize)
print("\nbatch and output via network")
print(batch)
estim = model(batch)
print(estim)
```

```
batch and output via network
tensor([[-1.8003,  1.0406],
        [ 0.4858,  2.2814],
        [-0.4864,  0.1789],
```

```
       [-0.3649,  0.3436],
       [ 0.1032, -0.7312],
       [-1.8067, -0.9846],
       [-0.4972,  0.4988],
       [-0.0793, -0.0797],
       [ 0.6836, -0.1530],
       [-0.3562, -1.6578],
       [ 0.0381, -1.2380],
       [ 1.0437, -0.8716]])
tensor([[ 4.9946e-01, -7.0425e-01],
       [ 3.7841e-01,  7.4821e-01],
       [-2.1650e-02,  1.5533e-01],
       [ 1.9859e-04,  2.3102e-01],
       [-4.0560e-01,  5.4937e-01],
       [-7.9888e-02, -6.7644e-01],
       [ 7.2311e-02,  1.4335e-01],
       [-1.8073e-01,  4.2155e-01],
       [-3.6087e-01,  9.1394e-01],
       [-5.7550e-01,  2.6816e-01],
       [-5.3734e-01,  5.1540e-01],
       [-6.4204e-01,  1.1571e+00]], grad_fn=<AddmmBackward0>)
tensor(1.6930, grad_fn=<MseLossBackward0>)
```

On peut tirer aléatoirement des cibles, choisir une fonction loss et rétro-propager les gradients :

```
target = torch.randn(12, outputSize)
loss = torch.nn.MSELoss()
loss_calculation = loss(estim, target)
print(loss_calculation)
loss_calculation.backward()
print("\ngradient sur poids et biais après rétro-propagation")
print(model[0].weight.grad)
print(model[0].bias.grad)
```

```
tensor(1.6930, grad_fn=<MseLossBackward0>)
gradient sur poids et biais après rétro-propagation
tensor([[ 0.0930,  0.2081],
       [ 0.5913, -0.2758]])
tensor([-0.1402, -0.0980])
```

On a désormais une meilleure idée de la mécanique cachée derrière les réseaux de neurones. On a utilisé Pytorch mais on pourrait faire la même chose avec Tensorflow

Apprentissage par renforcement à l'aide de la méthode de l'entropie croisée

La méthode Cross-Entropy

On peut à ce stade passer à notre premier cas pratique : un algorithme dit évolutif qui utilise l’entropie croisée évoquée précédemment. Le principe de base est simple : certains individus sont échantillonnés dans une population, et seuls les individus "élites" régissent les caractéristiques des générations futures.

Pour mémoire, l’entropie croisée de 2 distributions p et q sur le même espace de probabilité x est :

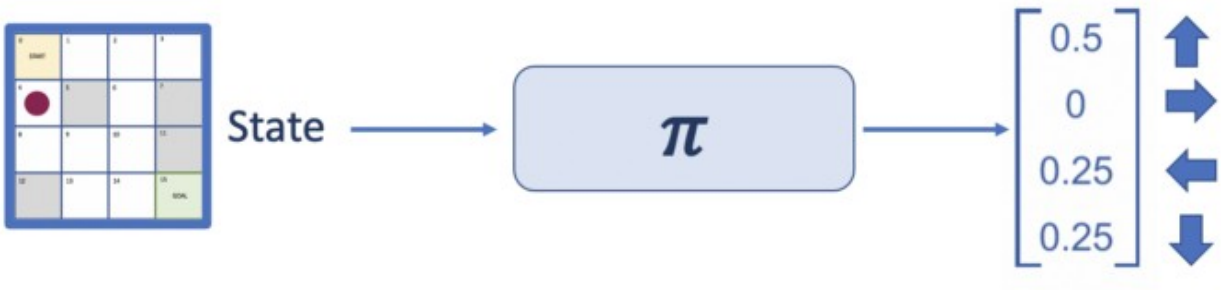
$$\mathrm{H}(p, q) = -\sum_{x} p(x)\, \log q(x).$$

Essentiellement, la méthode de l'entropie croisée consiste à prendre un ensemble d'entrées, à observer les sorties qui en résultent, à choisir les entrées qui ont conduit aux meilleures sorties et à régler l'agent jusqu'à ce que nous soyons satisfaits des sorties produites.

Vue d'ensemble

Une politique, notée π(a|s), indique quelle action l'agent doit entreprendre pour chaque état observé. Nous allons considérer que la politique sera produite par un réseau de neurones, constituant le coeur de notre agent. Lorsqu’on entraîne un réseau de neurones à produire la politique, on parle de méthodes basées sur la politique.

En pratique, la politique est généralement représentée comme une distribution de probabilités sur les actions (que l'agent peut entreprendre dans un état donné), ce qui la rend très similaire à un problème de classification, le nombre de classes étant égal au nombre d'actions que nous pouvons effectuer. Dans notre cas, la sortie de notre réseau neuronal est un vecteur représentant la distribution de probabilité des actions, comme le montre visuellement la figure suivante :



Il s’agit d’une politique stochastique, qui renvoie une distribution de probabilité sur les actions et non une action unique déterministe.

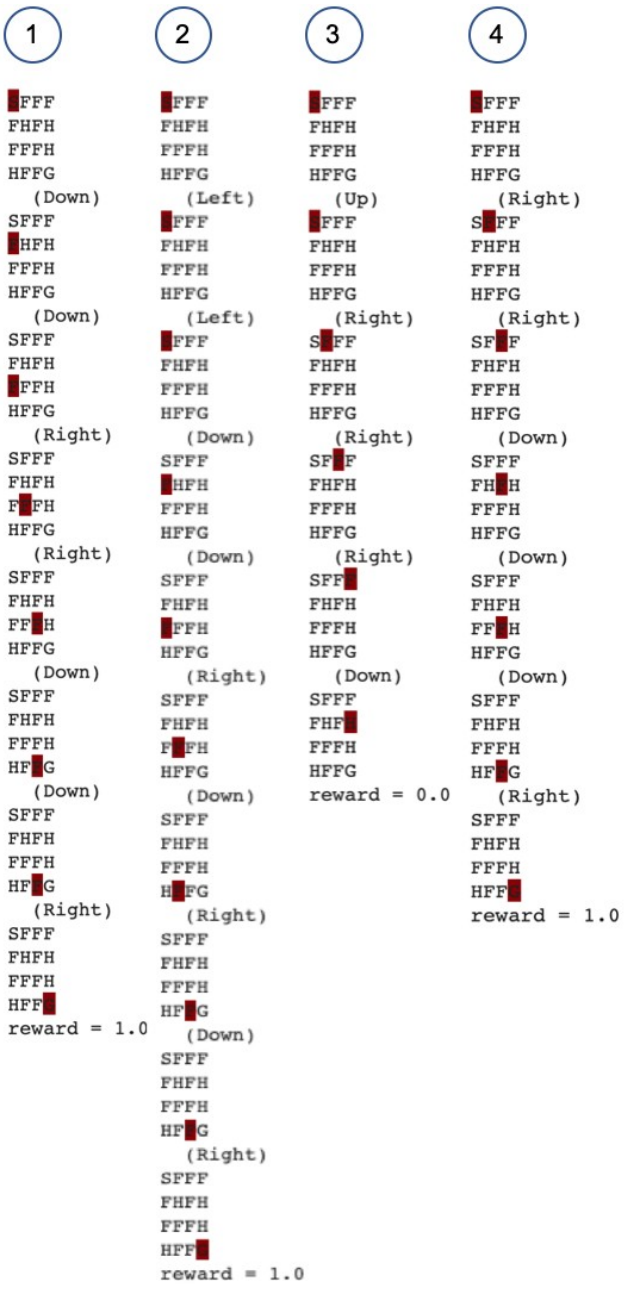
Avant toute chose, nous initialisons au hasard une politique, une distribution de probabilité. Ensuite, nous améliorons notre politique en jouant quelques épisodes, puis en ajustant notre politique (modification des poids du réseau neuronal) de manière à ce qu'elle soit plus efficace. Nous répétons ensuite ce processus afin que notre politique s'améliore progressivement. C'est la base de la méthode de l'entropie croisée.

Ensemble de données d’entraînement

Puisque nous allons considérer un réseau neuronal comme le cœur de l’ agent, nous devons trouver un moyen pour lui fournir un ensemble de données d'entraînement, qui comprend des données d'entrée et leurs étiquettes respectives.

Pour ce faire, nous allons considérer le problème comme un problème d'apprentissage supervisé où les états observés sont considérés comme les caractéristiques (données d'entrée) et les actions constituent les étiquettes.

Au cours de la vie de l'agent, son expérience est présentée sous forme d'épisodes. Chaque épisode est une séquence d'observations d'états que l'agent a obtenus de l'environnement, d'actions qu'il a émises et de récompenses pour ces actions. Le cœur de la méthode de l'entropie croisée est de rejeter les mauvais épisodes et de s'entraîner sur les meilleurs, mais comment trouver les meilleurs ? Imaginons que notre agent ait joué plusieurs épisodes de ce type. Pour chaque épisode, nous pouvons calculer le gain potentiel (récompense totale) que l'agent a obtenu. Rappelez-vous qu'un agent essaie d'accumuler autant de récompense totale que possible en interagissant avec l'environnement.



Une fois encore, pour simplifier, nous utiliserons l'exemple du lac gelé. Pour comprendre ce qui se passe, nous devons examiner de plus près la structure de récompense de l'environnement du lac gelé. Nous obtenons la récompense de 1,0 uniquement lorsque nous atteignons l'objectif, et cette récompense ne dit rien sur la qualité de chaque épisode. Était-il rapide et efficace ? ou avons-nous fait de nombreux tours sur le lac avant d'entrer par hasard dans la dernière cellule ? Nous ne le savons pas ; c'est juste une récompense de 1,0 et c'est tout.

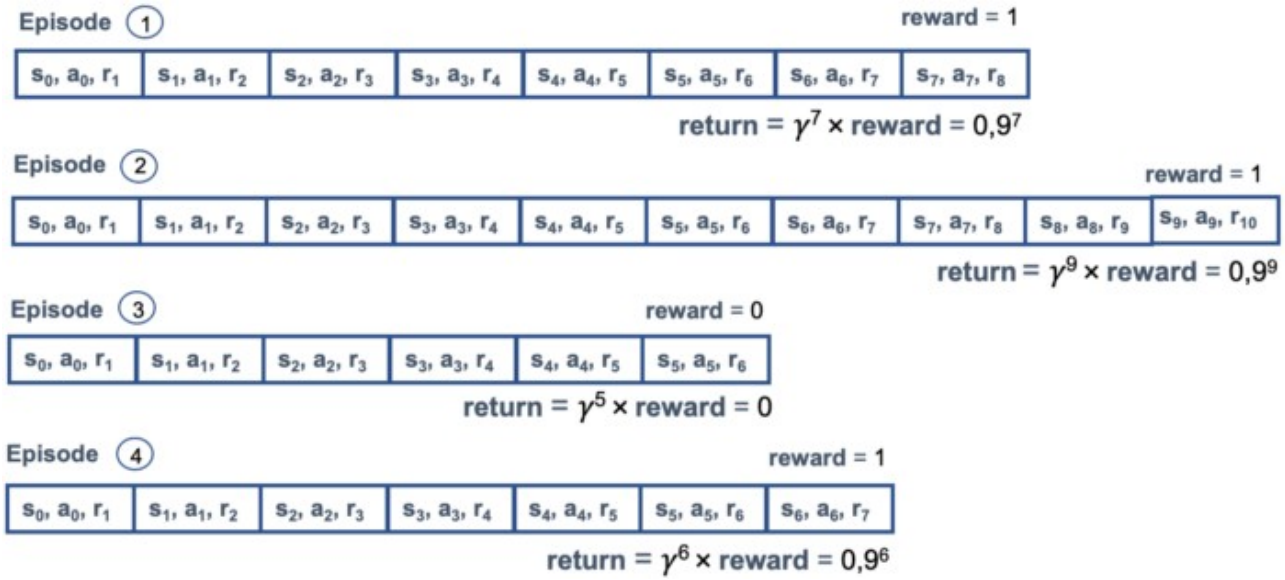
Imaginons que nous ayons déjà programmé l'agent et que nous l'utilisions pour créer 4 épisodes, que nous pouvons visualiser avec la méthode .render()

Notez qu'en raison du caractère aléatoire de l'environnement et de la manière dont l'agent sélectionne les actions à entreprendre, les épisodes ont une longueur différente et présentent également des récompenses différentes. Il est évident qu'un épisode dont la récompense est de 1,0

est meilleur qu'un épisode dont la récompense est de 0,0. Qu'en est-il des épisodes qui se terminent par la même récompense ?

Il est clair que nous pouvons considérer certains épisodes comme "meilleurs" que d'autres, par exemple le troisième est plus court que le deuxième. Pour cela, le facteur d'actualisation gamma est très utile. Nous pouvons utiliser = 0,9. Dans ce cas, le rendement actualisé sera égal à la récompense r (1,0 ou 0,0) obtenue à la fin de l'épisode au pas de temps t par gamma à la puissance t.

Illustrons ces quatre épisodes par un diagramme où chaque cellule représente l'étape de l'agent, une transition, dans l'épisode, et le rendement actualisé correspondant :



Nous pouvons constater que le rendement actualisé des épisodes courts sera plus élevé que celui des épisodes longs.

Algorithme d'entropie croisée

Le méthode Cross-Entropy est simple : elle génère des lots d'épisodes, élimine les mauvais épisodes d'un lot pour entraîner le réseau neuronal de l'agent sur de meilleurs épisodes. Pour décider lesquels jeter, nous utilisons la méthode percentile de numpy sur le percentile 30, ce qui signifie que nous ne gardons que les 30 % qui ont fait mieux que 70 % des autres.

https://fr.wikipedia.org/wiki/Centile

A mesure que nous utilisons de nouveaux lots d'épisodes d'élite, le réseau neuronal apprend à répéter les actions qui conduisent à ce que les résultats de ses décisions soient toujours meilleures. L'agent est entrainé jusqu'à ce qu'un certain seuil de récompense soit atteinte en moyenne pour le lot d'épisodes.

Un pseudocode de la méthode peut être décrit par les étapes suivantes :

0. Initialiser le modèle de réseau neuronal de l'agent
1. Créer un lot d'épisodes joués dans l'environnement en utilisant notre modèle d'agent actuel.
2. Calculer le gain potentiel pour chaque épisode et décider d'une limite de rendement en utilisant un percentile de toutes les récompenses.
3. Éliminer tous les épisodes dont le rendement est inférieur à la limite de rendement.

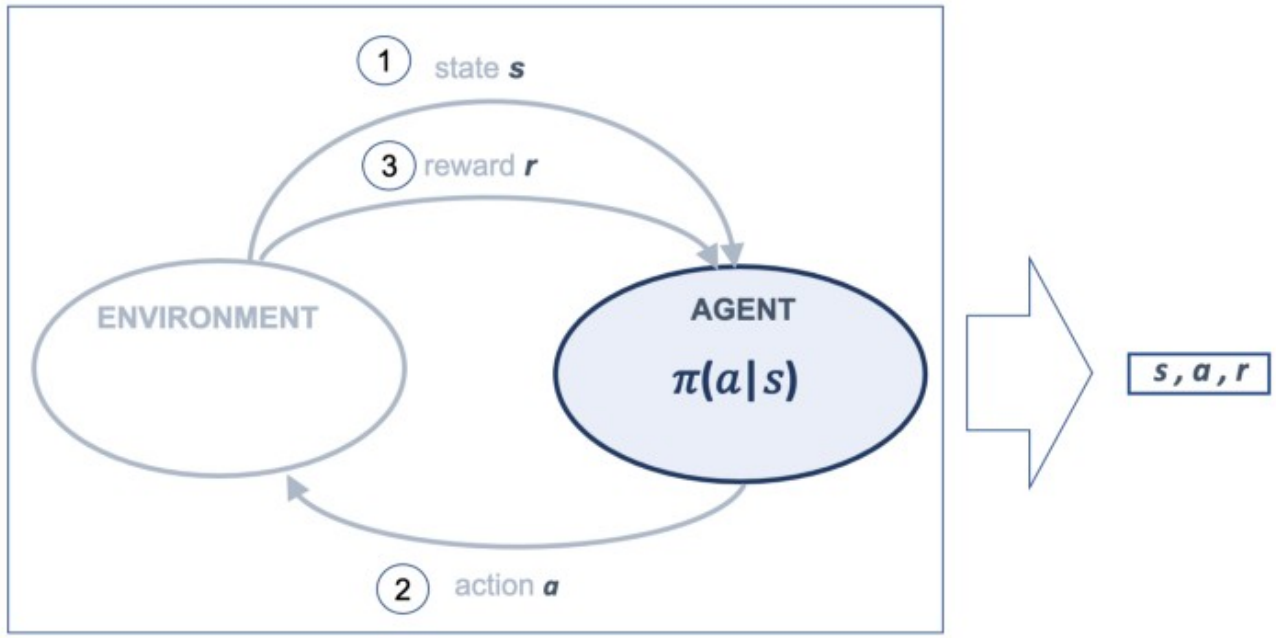
4. Former le réseau neuronal de l'agent en utilisant les étapes de l'épisode, c'est-à-dire les transitions <s,a,r>) des épisodes "élites" restants, en utilisant l'état s comme entrée et les actions émises a comme étiquette.
5. Répéter à partir de l'étape 1 jusqu'à ce que nous soyons satisfaits de la récompense moyenne pour le lot d'épisodes.

Une variante de la méthode consiste à conserver les épisodes "d'élite" pendant une période plus longue. La version par défaut de l'algorithme échantillonne des épisodes de l'environnement, s'entraîne sur les meilleurs, et les jette. Cependant, lorsque le nombre d'épisodes réussis est faible, les épisodes "d'élite" peuvent être conservés plus longtemps, en les gardant pendant plusieurs itérations pour s'entraîner sur eux.

L'environnement

L'environnement est la source de données à partir de laquelle nous allons créer le jeu de données qui sera utilisé pour entraîner le réseau neuronal de notre agent. L'agent démarre à partir d'une politique aléatoire, où la probabilité de toutes les actions est uniforme. Au cours de son apprentissage, l'agent espère apprendre des données obtenues de l'environnement pour optimiser sa politique et atteindre la politique optimale.

Les données provenant de l'environnement sont des étapes d'épisodes qui doivent être exprimées par des tuples de la forme <s,a,r> (état, action et récompense) obtenus à chaque étape du temps, comme indiqué dans le schéma suivant :



https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_06_07_Cross_Entropy.ipynb

```
import numpy as np
import torch
import torch.nn as nn
import gym
import gym.spaces
```

```
env = gym.make(‘FrozenLake-v0’, is_slippery=False)
```

Notre espace d'état est discret, ce qui signifie qu'il s'agit simplement d'un nombre compris entre zéro et quinze inclus (notre position actuelle dans la grille). L'espace d'action est également discret, de zéro à trois.

```
print(env.observation_space)
print(env.observation_space.n)
print(env.action_space)
print(env.action_space.n)

Discrete(16)
16
Discrete(4)
4
```

Notre réseau neuronal s'attend à un vecteur de nombres. Pour l'obtenir, nous pouvons appliquer l'encodage traditionnel des entrées discrètes, ce qui signifie que l'entrée de notre réseau comportera 16 nombres avec zéro partout sauf à l'indice que nous allons encoder.

Pour faciliter le code, nous surchargeons la classe ObservationWrapper de Gym au sein d’une classe fille OneHotWrapper :

```
class OneHotWrapper(gym.ObservationWrapper):
  def __init__(self, env):
   super(OneHotWrapper, self).__init__(env)
   self.observation_space = gym.spaces.Box(0.0, 1.0, (env.observation_space.n, ), dtype=np.float32)

  def observation(self, observation):
    r = np.copy(self.observation_space.low)
    r[observation] = 1.0
    return r

env = OneHotWrapper(env)
```

Après cette transformation, observation_space ne dispose plus de la propriété n, il faut passer par shape

```
print(env.observation_space)
print(env.observation_space.shape[0])
print(env.action_space)
print(env.action_space.n)

Box(16,)
16
Discrete(4)
4
```

L'agent

Le cœur de notre modèle est un réseau neuronal banal à une couche cachée de 32 neurones utilisant une fonction d'activation sigmoïde.

```
obs_size = env.observation_space.shape[0]
n_actions = env.action_space.n
HIDDEN_SIZE = 32
net= nn.Sequential(
    nn.Linear(obs_size, HIDDEN_SIZE),
    nn.Sigmoid(),
    nn.Linear(HIDDEN_SIZE, n_actions)
)
import torch.optim as optim
objective = nn.CrossEntropyLoss()
optimizer = optim.Adam(params=net.parameters(), lr=0.001)
```

Le réseau neuronal prend une seule observation de l'environnement comme vecteur d'entrée et sort un nombre pour chaque action que nous pouvons effectuer, une distribution de probabilité sur les actions. Pour cela, nous utiliserons la fonction softmax qui permet de transformer un score (calculé par le réseau neurones) en probabilité. L'intérêt de cette fonction est qu'elle est différentiable, et donc compatible avec l'algorithme du gradient.

Une façon directe de procéder serait d'inclure la non-linéarité softmax après la dernière couche. Nous essayons d'éviter d'appliquer la softmax pour augmenter la stabilité numérique du processus d’entraînement. Plutôt que de calculer la softmax et ensuite la perte d'entropie croisée, nous utilisons dans cet exemple la classe PyTorch nn.CrossEntropyLoss, qui combine à la fois la softmax et l'entropie croisée en une seule expression, plus stable numériquement. CrossEntropyLoss requiert des valeurs brutes et non normalisées du réseau neuronal (également appelées logits).

Pour un état observé reçu de l’environnement, l’agent décide de l’action à effectuer :

- en transmettant l’état au réseau neuronal pour qu’il calcule la distribution de probabilité,
- en effectuant un échantillonnage aléatoire à partir de cette distribution de probabilité :

```
sm = nn.Softmax(dim=1)

def select_action(state):
    state_t = torch.FloatTensor(np.array([state]))
    act_probs_t = sm(net(state_t))
    act_probs = act_probs_t.data.numpy()[0]
    action = np.random.choice(len(act_probs), p=act_probs)
    return action
```

La fonction nécessite dans un premier temps de transformer l'état en un tenseur à destination du réseau neuronal. À chaque itération, nous convertissons notre observation courante (tableau Numpy de 16 positions) en un tenseur PyTorch et le passons au modèle pour obtenir des probabilités d'action. Rappelez-vous que notre modèle de réseau neuronal a besoin de tenseurs comme données d'entrée. Par convention, le suffixe _t indique que la variable est un tenseur.

Une conséquence de l'utilisation de nn.CrossEntropyLoss est que nous devons appliquer softmax chaque fois que nous devons obtenir des probabilités à partir de la sortie de notre réseau neuronal.

Nous devons convertir le tenseur de sortie (rappelez-vous que le modèle et la fonction softmax renvoient des tenseurs) en un tableau NumPy. Ce tableau aura la même structure 2D que l'entrée, avec la dimension du lot sur l'axe 0, nous devons donc récupérer le premier élément du lot pour obtenir un vecteur 1D de probabilités d'action.

Enfin, nous utilisons la distribution de probabilité des actions pour obtenir l'action réelle pour l'étape actuelle en échantillonnant cette distribution à l'aide de la fonction random.choice() de NumPy.

On utilise la fonction softmax de pytorch mais elle est assez facile à implémenter.

https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html

```
def mysoftmax(x):
    """
    x : tenseur vectoriel pytorch
    """
    denominateur = np.sum(np.exp(x).numpy())
    result = np.exp(x)/denominateur
    return result

print("\napplication de softmax pour obtenir la distribution de probabilité")
print(mysoftmax(net(state_t).squeeze(0).detach()))
```

```
application de softmax pour obtenir la distribution de probabilité
tensor([0.2243, 0.1558, 0.3252, 0.2947])
```

Nota : On utilise detach() pour pouvoir transformer les tenseurs en objet numpy et ainsi faire appel aux méthodes numpy.sum() et numpy.exp()

Les commandes ci-dessous permettent de comprendre les opérations tensorielles, notamment l'effet de la méthode squeeze.

2 cas de figures :

- on applique squeeze sur un tenseur sans lui donner d'argument. squeeze() supprime toutes les dimensions dont la taille est égale à 1
- on fournit à squeeze le numéro de la dimension i. squeeze(i) supprime cette dimension si sa taille est égale à 1, sinon ne procède à aucune modification

```
state = env.reset()
state_np = np.array([state])
print("l'état au format numpy")
print((state_np))
state_t = torch.FloatTensor(state_np)
print("\nl'état en tenseur pytorch")
print((state_t))
```

```
print("\nla sortie du réseau neurones")
print(net(state_t))
print("\nsuppression de la première dimension")
print(net(state_t).squeeze(0))
print("\nrajout d'une dimension à la fin après avoir supprimé la première dimension")
print(net(state_t).squeeze(0).unsqueeze(-1))
print("\nretour au tenseur obtenu après suppression de la première dimension")
print(net(state_t).squeeze(0).unsqueeze(-1).squeeze(-1))
```

```
l'état au format numpy
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

l'état en tenseur pytorch
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

la sortie du réseau neurones
tensor([[-0.1147, -0.4787,  0.2568,  0.1585]], grad_fn=<AddmmBackward0>)

suppression de la première dimension
tensor([-0.1147, -0.4787,  0.2568,  0.1585], grad_fn=<SqueezeBackward1>)

rajout d'une dimension à la fin après avoir supprimé la première dimension
tensor([[-0.1147],
        [-0.4787],
        [ 0.2568],
        [ 0.1585]], grad_fn=<UnsqueezeBackward0>)

retour au tenseur obtenu après suppression de la première dimension
tensor([-0.1147, -0.4787,  0.2568,  0.1585], grad_fn=<SqueezeBackward1>)
```

Constantes, principales variables et helpers

```
BATCH_SIZE = 100
GAMMA = 0.9
PERCENTILE = 30
REWARD_GOAL = 0.8

from collections import namedtuple

Episode = namedtuple('Episode', field_names=['reward', 'steps'])
EpisodeStep = namedtuple('EpisodeStep', field_names=['observation', 'action'])

iter_no = 0
reward_mean = 0
full_batch = []
batch = []
episode_steps = []
episode_reward = 0.0
state = env.reset()
```

Nous définissons deux helpers de type namedtuple de la bibliothèque collections standard :

- EpisodeStep : représente une étape unique que notre agent a effectuée au cours de l'épisode, stocke l'état observé de l'environnement et l'action effectuée par l'agent. La récompense n'est pas enregistrée car elle est toujours de 0,0 sauf pour la dernière transition. Rappelez-vous que nous utiliserons les étapes des épisodes "d'élite" comme données d'entraînement.
- Épisode : Il s'agit d'un épisode unique stocké avec une récompense totale actualisée et une collection d'EpisodeStep.

Boucle d'entraînement

La boucle d'entraînement de l'algorithme d'entropie croisée répète 4 étapes principales jusqu'à ce que nous soyons satisfaits du résultat :

1. Jouer des d'épisodes
2. Calculer le gain potentiel pour chaque épisode et la valeur limite du gain potentiel au-delà de laquelle un épisode est considéré comme faisant partie de l'élite
3. Eliminer tous les épisodes qui ne font pas partie de l'élite.
4. Entraîner le réseau neuronal en utilisant les étapes des épisodes "d'élite".

L'entraînement est arrêté lorsqu'on atteint un certain seuil de récompense de 80% indiqué dans la variable REWARD_GOAL….

Jouer des épisodes

On génère les lots avec les épisodes :

```
while reward_mean < REWARD_GOAL:
    action = select_action(state)
    next_state, reward, episode_is_done, _ = env.step(action)
    episode_steps.append(EpisodeStep(observation=state,action=action))
    episode_reward += reward
    if episode_is_done: # Episode finished
        batch.append(Episode(reward=episode_reward, steps=episode_steps))
        next_state = env.reset()
        episode_steps = []
        episode_reward = 0.0
        <STEP 2>
        <STEP 3>
        <STEP 4>
    state = next_state
```

Les principales variables utilisées sont :

- **batch** accumule la liste des Episodes (BATCH_SIZE=100).
- **episode_steps** accumule la liste des étapes de l'épisode en cours.
- **episode_reward** maintient un compteur de récompense pour l'épisode en cours. Dans notre cas, nous n'avons de récompense qu'à la fin de l'épisode, mais l'algorithme est décrit pour

une situation plus générale où nous pouvons avoir des récompenses pas seulement à la dernière étape.

La liste des étapes de l'épisode est étendue avec une paire (observation, action). Il est important de noter que nous sauvegardons l'état observé qui a été utilisé pour choisir l'action (mais pas l'observation next_state retournée par l'environnement comme résultat de l'action)

La récompense est ajoutée à la récompense totale de l'épisode en cours

Lorsque l'épisode actuel est terminé, si on est tombé dans un trou ou que nous avons atteint l'objectif, nous devons ajouter l'épisode finalisé au batch, en sauvegardant la récompense totale et les étapes que nous avons franchies.

Ensuite, nous réinitialisons notre environnement tout comme les variables episode_steps et episode_reward pour pouvoir aborder le prochain épisode

Calculer le gain potentiel pour chaque épisode et estimer le gain potentiel limite

```
if len(batch) == BATCH_SIZE:
    reward_mean = float(np.mean(list(map(lambda s: s.reward, batch))))
    elite_candidates= batch
    ExpectedReturn = list(map(lambda s: s.reward * (GAMMA **len(s.steps)), elite_candidates))
    reward_bound = np.percentile(ExpectedReturn, PERCENTILE)
```

Si le batch est plein, nous calculons :

- la récompense moyenne qui sert pour interrompre la boucle d'apprentissage
- le gain potentiel pour tous les épisodes du batch
- la récompense limite à utiliser pour filtrer les épisodes "élites" qui serviront à entraîner le réseau neuronal

Pour obtenir la récompense limite, nous utilisons la fonction percentile de NumPy qui, à partir de la liste des valeurs et du percentile souhaité, calcule la valeur du percentile. Dans ce code, nous utiliserons les 30% supérieurs des épisodes (indiqués par la variable PERCENTILE) pour créer les épisodes "élites".

Éliminer tous les épisodes dont le gain potentiel est inférieur à la limite.

```
train_obs = []
train_act = []
elite_batch = []
for example, discounted_reward in zip(elite_candidates, ExpectedReturn):
    if discounted_reward > reward_bound:
        train_obs.extend(map(lambda step: step.observation, example.steps))
        train_act.extend(map(lambda step: step.action, example.steps))
        elite_batch.append(example)
full_batch=elite_batch
state=train_obs
acts=train_act
print("**************************************************")
```

Pour chaque épisode du batch, nous vérifions que l'épisode a une récompense totale supérieure à notre limite, et si c'est le cas, nous complétons :

- la liste des états
- la liste des actions
- la trace des épisodes d'élite, qui nous servira plus tard lorsque nous chercherons à conserver les épisodes "d'élite" plus longtemps pour les réutiliser d'un entraînement sur batch sur l'autre

Entraîner le réseau neuronal à l'aide des étapes des épisodes élites

```
        if len(full_batch) :
            state_t = torch.FloatTensor(np.array(state))
            acts_t = torch.LongTensor(np.array(acts))
            optimizer.zero_grad()
            action_scores_t = net(state_t)
            loss_t = objective(action_scores_t, acts_t)
            loss_t.backward()
            optimizer.step()
            print("{}: loss={:.3f}, reward_mean={:.3f}".format(iter_no, loss_t.item(), reward_mean))
            iter_no += 1

        batch = []
```

Une fois les variables converties en tenseur, les gradients du réseau neuronal sont mis à zéro, et les états observés lui sont transmis pour qu'il calcule les scores d'action.

		scores d'actions tenseur action_scores_t renvoyé par le réseau neurones
np.array(state)	[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]	Tensor([[-0.2219, 0.5608, -0.0553, 0.3762], [-0.2312, 0.4945, -0.0787, 0.3542], [-0.2219, 0.5608, -0.0553, 0.3762], [-0.2312, 0.4945, -0.0787, 0.3542], [-0.2184, 0.5210, -0.0713, 0.3675], [-0.2185, 0.5201, -0.0518, 0.3675], [-0.1973, 0.5346, -0.0330, 0.4057], [-0.2253, 0.5810, -0.0937, 0.3503]], grad_fn=<AddmmBackward0>)
acts	[1, 3, 1, 1, 2, 2, 1, 2]	

Les scores sont transmis à la fonction objective, qui calcule l'entropie croisée entre la sortie du réseau neuronal et les actions entreprises par l'agent. Rappelez-vous que nous ne prenons en compte que les actions "d'élite". L'idée est de renforcer notre réseau neuronal pour qu'il effectue les actions "d'élite" qui ont conduit à de bonnes récompenses. A l'issu de ce calcul de l'entropie croisée, on dispose d'un tenseur loss_t

Enfin, on calcule les gradients sur loss_t à l'aide de la méthode backward() et on ajuste les paramètres de notre réseau neuronal à l'aide de la méthode steps() de l'optimiseur :

Avec 32 neurones, il faut près de 700 entraînements sur batch pour dépasser le REWARD_GOAL

Si on passe à 128 neurones, la convergence est presque deux fois plus rapide.

Que se passe-t-il si nous changeons la fonction d'activation ? par exemple, une ReLU au lieu d'une Sigmoïde ? Moins de 200 entraînements sur batch suffisent.

En plus de changer l'architecture du réseau neuronal, nous pouvons également améliorer l'algorithme lui-même, en conservant les épisodes "d'élite" plus longtemps.

Pour cela, il suffit de modifier une seule ligne du code au niveau de l’étape 2:

```
        elite_candidates= full_batch + batch
        #elite_candidates= batch
```

Avec cette modification, on résout le problème en 125 entraînements sur batch

L’équation de Bellman - les fonctions V et Q

Agents basés sur la valeur

L'objectif de l'agent est de trouver une séquence d'actions qui maximise le gain potentiel pendant un épisode ou toute la vie de l'agent, selon la tâche.

Dans une tâche continue, cette somme est infinie. Nous pouvons résoudre ce problème à l'aide du discount parametrer qui se situe dans la plage [0:1]. La formule pour le gain potentiel G à l'étape t est la suivante :

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{j=0}^{T} \gamma^j r_{t+j+1}$$

Bien que la somme soit toujours infinie, si γ<1, alors Gt aura une valeur finie. Si γ=0, l'Agent n'est intéressé que par la récompense immédiate et écarte la récompense à long terme. Inversement, si γ=1, l'Agent considère toutes les récompenses futures égales à la récompense immédiate.

Nous pouvons réécrire cette équation avec une relation récursive :

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + + \gamma^{T-1} r_T \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma\, r_{t+3} + \gamma^2 r_{t+4} + \cdots + + \gamma^{T-2} r_T) \\ &= r_{t+1} + \gamma(G_{t+1}) \end{aligned}$$

Presque tous les algorithmes d'apprentissage par renforcement demandent aux agents d'estimer de fonctions de valeur - fonctions d'états ou de paires état-action. Ce sont les agents dits "à valeur". Une fonction de valeur estime combien il est bon pour l'agent d'être dans un état donné ou combien il est bon d'effectuer une action donnée dans un état donné, le tout par rapport à des façons particulières d'agir, appelées politiques, dénotées π

La fonction V : la valeur de l'état

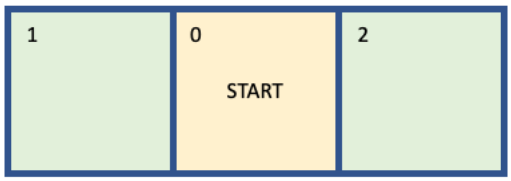
> La fonction V, également appelée fonction état-valeur, ou simplement V, mesure à quel point il est bon ou mauvais d'être dans un état particulier lorsqu'on suit une politique π.

En d'autres termes, nous pouvons définir la fonction V comme une récompense totale attendue que l'on peut obtenir de l'état. De manière formelle, la valeur de Vπ(s) est :

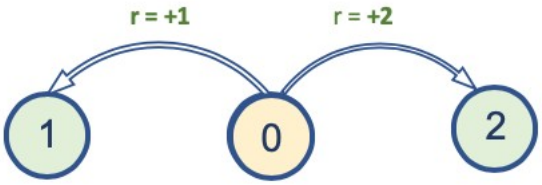
$$V_\pi(s) = \mathbb{E}_\pi[G_t | s = s_t] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \,|s = s_t\right]$$

Elle décrit la valeur attendue du gain potentiel G, au pas de temps t en partant de l'état s au temps t et en suivant ensuite la politique . On utilise l'espérance $\mathbb{E}[.]$ dans cette définition car la fonction de transition de l'environnement pourrait agir de manière stochastique.

Pour clarifier un peu plus le concept, considérons un environnement simple avec trois états :



- L'état initial 0 de l'agent
- L'état final 1 dans lequel se trouve l'agent après avoir exécuté l'action "gauche" de l'état initial. La récompense obtenue est r=+1
- L'état final 2 dans lequel se trouve l'agent après avoir exécuté l'action "droite". La récompense obtenue est r=+2.



L'environnement est toujours déterministe - chaque action réussit, et nous commençons toujours par l'état 0. Une fois que nous avons atteint l'état 1 ou l'état 2, l'épisode se termine.

Maintenant, la question est de savoir quelle est la valeur de l'état 0 (indiquée par V(0)). Un détail important est que la valeur d'un état est toujours calculée (dépendante) en fonction d'une politique que notre agent suit. Même dans un environnement simple, notre agent peut avoir différents comportements, dont chacun aura sa propre valeur pour l'état 0. Considérons les 4 politiques suivantes :

1. L'agent va toujours à gauche
2. L'agent va toujours à droite
3. L'agent va à gauche avec une probabilité de 0,5 et à droite avec une probabilité de 0,5.
4. L'agent va à gauche avec une probabilité de 0,2 et à droite avec une probabilité de 0,8.

Pour la politique 1, la valeur de l'état 0 est V(0)=1,0 (il va à gauche une fois, obtient +1 et l'épisode se termine).

Pour la politique 2, la valeur de l'état 0 est V(0)=2.0 (il va à droite une fois, il obtient +2 et l'épisode se termine).

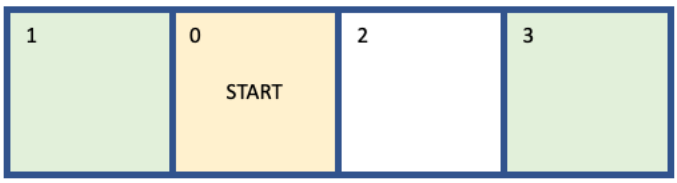
Pour la politique 3, la valeur de l’état 0 est V(0)=1,0 x 0,5 + 2,0 x 0,5 = 1,5.

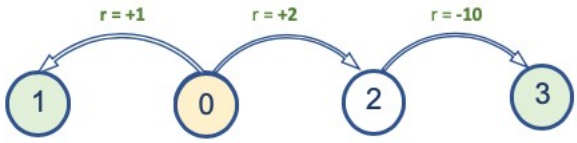
Pour la politique 4, la valeur de l’état 0 est V(0)=1,0 x 0,2 + 2,0 x 0,8 = 1,8.

Étant donné que l'objectif de l'agent est d'obtenir la récompense totale la plus élevée possible, la politique optimale pour cet agent dans cet environnement simple à une étape est la politique 2.

Mais l'exemple précédent peut donner une fausse impression que nous devrions "être greedy" et toujours prendre l'action avec la récompense la plus élevée. Malheureusement, ce n'est pas si simple. Par exemple, étendons notre environnement précédent avec un autre état qui est atteignable

depuis l'état 2. L'état 2 n'est plus un état terminal mais une transition vers l'état 3, avec une (très) mauvaise récompense de -10 :





Pour la politique 1 : V(0)=+1.0

Pour la politique 2 : V(0)= 2.0 + (-10.0) = -8.0

Pour la politique 3 : V(0)=1.0 * 0.5 + (2.0+(-10.0))* 0.5 = -3.5

Pour la politique 4 : V(0)=1.0 * 0.2 + (2.0+(-10.0)) * 0.8 = -6.2

Ainsi, la meilleure politique pour ce nouvel environnement est maintenant la politique 1.

La fonction Q ou fonction action valeur

Q définit la valeur d'une action a dans l'état s sous une politique π, dénotée par Qπ(s,a) , comme le gain potentiel G en partant de s, en prenant l'action a, et en suivant ensuite la politique π.

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \,|S_t = s, A_t = a\right]$$

Maintenant que nous avons défini Q et V, formalisons leur relation. Nous désignons par π (a | s) la probabilité qu'une politique, π, choisisse une action, a, étant donné un état courant, s. Notons que la somme des probabilités de toutes les actions sortantes de l'état s est égale à 1 :

$$\sum_a \pi(a|s) = 1$$

Nous pouvons affirmer que la fonction état-valeur est équivalente à la somme des fonctions action-valeur de toutes les actions sortantes (de s) a, multipliée par la probabilité politique de choisir chaque action :

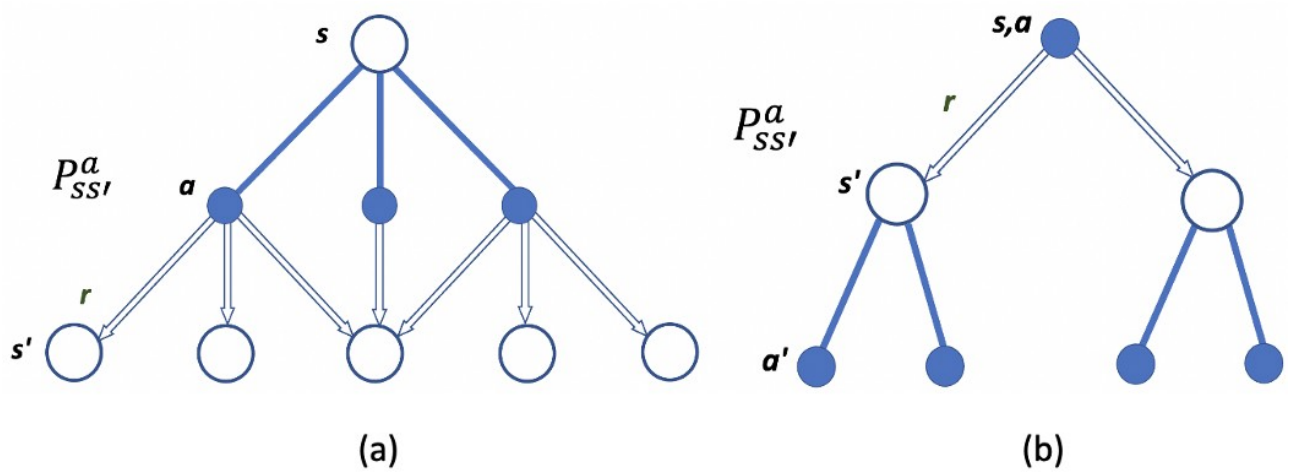
$$V_\pi(s) = \sum_a \pi(a|s) \cdot Q_\pi(s,a)$$

L'équation de Bellman

L'équation de Bellman apparaît partout dans la littérature sur l'apprentissage par renforcement et constitue l'un des éléments centraux de nombreux algorithmes d'apprentissage par renforcement. Elle permet de décomposer la fonction de valeur en deux parties, la récompense immédiate et les valeurs futures actualisées.

Cette équation simplifie le calcul de la fonction de valeur, de sorte qu'au lieu de faire la somme sur plusieurs pas de temps, nous pouvons trouver la solution optimale d'un problème complexe en le décomposant en sous-problèmes récursifs plus simples et en trouvant leurs solutions optimales.

Pour faciliter la compréhension de la formulation dans les sections suivantes, les deux diagrammes suivants résument une convention des noms donnés aux variables et à leur relation :



Backup diagrams pour (a) Vπ(s) et (b) Qπ(s,a).

Dans ces diagrammes, P désigne la probabilité que l'action a, émise dans l'état s, aboutisse dans l'état s' (avec une récompense r).

Équation de Bellman pour la fonction État-valeur

Nous avons déjà vu que nous pouvions définir le gain potentiel G en termes récursifs.

Voyons maintenant comment définir de manière récursive l'équation de Bellman pour la fonction état-valeur :

$$V_\pi(s) = \sum_a \pi(a|s) \cdot \sum_{s'} P^a_{ss'}(r(s,a) + \gamma V_\pi(s'))$$

Cette équation nous indique comment trouver la valeur d'un état s suivant une politique π.

Nous pouvons intuitivement distinguer 2 termes :

- la récompense immédiate attendue de l'état suivant, r(s,a),
- la valeur d'un état successeur Vπ(s') pondérée par le facteur d'actualisation .

L'équation ci-dessus exprime également la stochasticité de l'Environnement avec la somme sur les probabilités de politique.

L'équation de Bellman est importante car elle nous donne la possibilité de décrire la valeur d'un état s, Vπ(s), avec la valeur de l'état s' Vπ(s'), et avec une approche itérative, nous pouvons calculer les valeurs de tous les états.

Malheureusement, dans la plupart des scénarios, nous ne connaissons pas la probabilité P et la récompense r, nous ne pouvons donc pas résoudre le PDM en appliquant directement l'équation de Bellman. Mais nous pouvons pour les trouver par expérience.

Équation de Bellman pour la fonction Action-valeur

Dans l'état s, après avoir exécuté l'action a, on obtient une récompense r(s,a) et il y a une certaine probabilité que nous arrivions dans l'état s'. Une fois que l'on est dans cet état s', on choisit une action a' parmi les possibles qui s'offrent à nous et on peut espérer le gain potentiel Qπ(s',a') avec une probabilité π(a' | s') et un facteur d'actualisation .

Pour la fonction action-valeur, l'équation de Bellman s'écrit donc de la manière suivante:

$$Q_\pi(s,a) = \sum_{s'} P^a_{ss'}(r(s,a) + \gamma \cdot \sum_{a'} \pi(a'|s') \cdot Q_\pi(s',a'))$$

De la même manière que la fonction état-valeur, cette équation nous indique comment trouver récursivement la valeur d'une paire état-action suivant une politique .

Nous avons montré que la fonction état-valeur V(s') est égale à la somme sur toutes les actions possibles a' des produits de la probabilité que la politique sélectionne l'action a' dans l'état s' par la valeur de la fonction action-valeur Q(s',a')

La formule précédente peut donc être exprimée comme suit :

$$Q_\pi(s,a) = \sum_{s'} P^a_{ss'}(r(s,a) + \gamma V_\pi(s'))$$

Politique optimale

L'objectif de l'agent est de maximiser la récompense cumulative totale à long terme. La politique qui maximise la récompense cumulative totale est appelée politique optimale.

Fonction de valeur optimale

Une politique π′ est définie comme étant meilleure qu'une politique π si et seulement si :

$$V\pi'(s) \geq V\pi(s) \text{ pour tous les états } s$$

Une politique optimale $\pi*$ satisfait $\pi* \geq \pi$ pour toutes les politiques π.

Une politique optimale est garantie d'exister mais peut ne pas être unique. Cela signifie qu'il peut y avoir différentes politiques optimales, mais qu'elles partagent toutes les mêmes fonctions de valeur, les fonctions de valeur "optimales".

La fonction de valeur optimale est celle qui donne la valeur maximale par rapport à toutes les autres fonctions de valeur (suite à l'utilisation d'autres politiques).

Résoudre un PDM revient donc à trouver la fonction de valeur optimale. Ainsi, mathématiquement, la fonction état-valeur optimale peut être exprimée comme suit :

$$V_*(s) = \max_{\pi} V_{\pi}(s)$$

Dans la formule ci-dessus, V∗(s) nous indique quelle est la récompense maximale pour l'état s que nous pouvons obtenir du système.

De même, la fonction de valeur état-action optimale indique la récompense maximale que nous allons obtenir si nous sommes dans l'état s et que nous effectuons l'action a à partir de là :

$$Q_*(s,a) = \max_{\pi} Q_{\pi}(s,a)$$

Vu la définition de V(s) et de Q(s,a), pour la politique optimale *, on peut écrire que la valeur d'un certain état est égale à la valeur de l'action maximale que nous pouvons exécuter à partir de cet état :

$$V_*(s) = \max_{a} Q_*(s,a)$$

L'équation d'optimalité de Bellman

Bellman a prouvé que la fonction de valeur optimale d'un état s est égale à l'action a, qui nous donne la récompense immédiate espérée maximale possible, plus la récompense à long terme actualisée pour l'état suivant s' :

$$V_*(s) = \max_{a} \sum_{s'} P^a_{ss'}(r(s,a) + \gamma V_*(s'))$$

Bellman a également prouvé que la fonction de valeur de l'état optimal dans l'état s et en prenant l'action a est :

$$Q_*(s,a) = \sum_{s'} P^a_{ss'}(r(s,a) + \gamma \max_{a'} Q_*(s',a'))$$

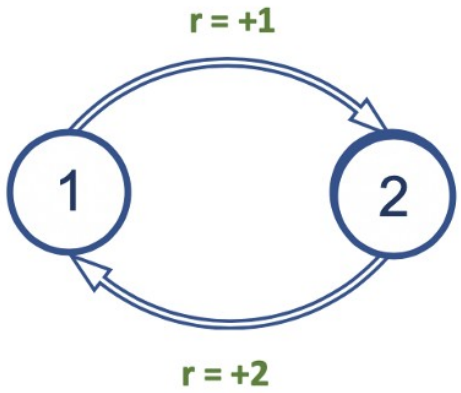
L'algorithme d'itération des valeurs

Estimation des Transitions et des Récompenses à partir de l'expérience de l'Agent

nous avons présenté les agents basés sur la valeur et examiné l'équation de Bellman, l'un des éléments centraux de nombreux algorithmes d'apprentissage par renforcement. Nous allons nous intéresser désormais à la méthode d'itération des valeurs pour calculer les valeurs V et Q requises par les agents basés sur les valeurs.

Calcul de la fonction V dans un environnement avec des boucles

Jusqu'à présent, nous n'avions pas de boucles dans les transitions et la façon de calculer les valeurs des états était claire. Voyons comment ces cas sont résolus avec un environnement simple avec deux états, l'état 1 et l'état 2, qui présente le diagramme de transition des états de l'environnement suivant :



Nous n'avons que deux transitions possibles : à partir de l'état 1, nous ne pouvons entreprendre qu'une action qui nous mène à l'état 2 avec une récompense de +1 et à partir de l'état 2, nous ne pouvons entreprendre qu'une action qui nous ramène à l'état 1 avec une récompense de +2. Ainsi, la vie de notre agent évolue dans une séquence infinie d'états en raison de la boucle infinie entre les deux états. Quelle est la valeur des deux états ?

Supposons que nous ayons un facteur d'actualisation γ<1, disons 0,9, et rappelons que la valeur optimale de l'état est égale à celle de l'action qui nous donne la récompense immédiate espérée maximale possible, plus la récompense à long terme actualisée pour l'état suivant :

$$V_*(s) = \max_a \sum_{s'} P_{ss'}^a (r(s,a) + \gamma V_*(s'))$$

Comme il n'y a qu'une seule action disponible dans chaque état, nous pouvons donc simplifier la formule précédente comme suit :

$$V_*(s) = r(s,a) + \gamma V_*(s')$$

Si nous commençons à l'état 1, la séquence d'états sera [1,2,1,2,1,2, ...], et puisque chaque transition de l'état 1 à l'état 2 nous donne une récompense de +1 et chaque transition inverse nous donne une récompense de +2, la séquence de récompenses sera [+1,+2,+1,+2,+1,+2, ...]. Par conséquent, la formule précédente pour l'état 1 devient :

$$V(1) = 1 + \gamma(2 + \gamma\ (1 + \gamma\ (2 + \ \ldots))) = \sum_{i=0}^{\infty} 1\gamma^{2i} + 2\gamma^{2i+1}$$

En toute rigueur, il est impossible de calculer la valeur exacte de notre état, mais avec un facteur d'actualisation γ= 0,9, la contribution d'une nouvelle action diminue avec le temps. Par exemple, pour le balayage i=37 le résultat de la formule est 14,7307838, pour le balayage i=50 le résultat est 14,7365250 et pour le balayage i=100 le résultat est 14,7368420. Cela signifie que nous pouvons arrêter le calcul à un moment donné (par exemple à i=50) et obtenir une bonne estimation de la valeur V, dans ce cas V(1) = 14,736.

L'algorithme d'itération de valeurs

L'exemple précédent peut être utilisé pour obtenir l'essentiel d'une procédure plus générale appelée l'algorithme d'**I**tération de **V**aleur **(VI)**. Celui-ci nous permet de calculer numériquement les valeurs des états des processus de décision de Markov, avec des probabilités de transition et des récompenses connues.

L'idée derrière l'algorithme d'itération de valeur est de fusionner une étape d'évaluation de politique tronquée (comme dans l'exemple précédent) et une amélioration de politique dans le même algorithme.

Fondamentalement, l'algorithme d'itération des valeurs calcule la fonction de valeur d'état optimale en améliorant de manière itérative l'estimation de V(s). L'algorithme initialise V(s) à des valeurs aléatoires arbitraires. Il met à jour de manière répétée les valeurs de Q(s, a) et V(s) jusqu'à ce qu'elles convergent. L'itération de valeurs est garantie pour converger vers les valeurs optimales. Le pseudo-code suivant exprime l'algorithme proposé :

Initialize V(s) to arbitrary values
Repeat until V(s) converge
    For all states
        For all actions
            $Q(s,a) \leftarrow \sum_{s'} P^{a}_{ss'}(r(s,a) + \gamma V(s'))$
        $V(s) \leftarrow \max_{a} Q(s,a)$

cf http://www.incompleteideas.net/book/first/ebook/node44.html

Estimation des transitions et des récompenses

En pratique, cette méthode d'itération des valeurs présente plusieurs limites. Tout d'abord, l'espace d'état doit être discret et suffisamment petit pour effectuer de multiples itérations sur tous les états. Ce n'est pas un problème pour notre environnement de lac gelé, mais dans un problème général d'apprentissage par renforcement, ce n'est pas le cas.

Un autre problème pratique essentiel découle du fait que pour mettre à jour l'équation de Bellman, l'algorithme doit connaître la probabilité des transitions et la récompense pour chaque transition de l'environnement.

Rappelez-vous que dans notre exemple du lac gelé, nous observons l'état, décidons d'une action, et ce n'est qu'ensuite que nous obtenons la prochaine observation et la récompense pour la transition, mais nous ne connaissons pas ces informations à l'avance. Que pouvons-nous faire pour les obtenir ?

Heureusement, ce que nous pouvons avoir est l'historique de l'interaction de l'agent avec l'environnement. La réponse à la question précédente consiste donc à utiliser l'expérience de notre agent comme une estimation des deux inconnues. Nous allons voir ci-dessous comment nous pouvons y parvenir.

Estimation des récompenses

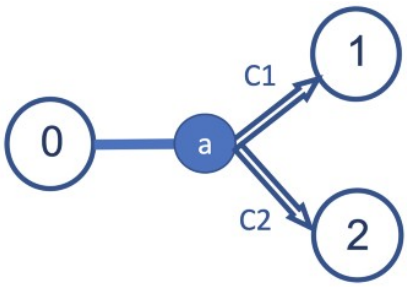
Estimer les récompenses est la partie la plus facile puisque les récompenses peuvent être utilisées telles quelles. Nous devons simplement nous souvenir de la récompense que nous avons obtenue lors de la transition de s à s' en utilisant l'action a.

Estimation des transitions

L'estimation des transitions est également facile, par exemple en maintenant des compteurs pour chaque tuple dans l'expérience de l'agent (s, a, s') et en les normalisant.

Par exemple, nous pouvons créer un tableau simple qui conserve les compteurs des transitions expérimentées. La clé de la table peut être un composite "état" + "action", (s, a), et les valeurs de chaque entrée sont les informations sur les états cibles, s', et le nombre de fois où nous avons vu chaque état cible, c.

A titre d’exemple, imaginons que pendant l'expérience de l'agent, dans un état donné s0, il a exécuté une action a plusieurs fois et qu'il se retrouve c1 fois dans l'état s1 et c2 fois dans l'état s2. Le nombre de fois où il est passé dans chacun de ces états est stocké dans notre table de transition. C'est-à-dire que l'entrée (s,a) dans la table contient {s1 : c1, s2 : c2}. Peut-être pouvez-vous visualiser plus facilement les informations contenues dans la table pour cet exemple :



Il est facile ensuite d'utiliser ce tableau pour estimer les probabilités de nos transitions. La probabilité que l'action nous mène de l'état 0 à l'état 1 est c1 / (c1 + c2) et que l'action nous mène de l'état 0 à l'état 2 c2 / (c1 + c2).

Par exemple, imaginons qu'à partir d'un état 0, nous exécutons l'action 1 dix fois, et qu'après 4 fois, elle nous mènera à l'état 1, et après 6 fois, elle nous mènera à l'état 2. Pour cet exemple particulier, l'entrée avec la clé (0, 1) dans ce tableau contient {1 : 4, 2 : 6}. Et cela représente que la probabilité de passage de l'état 0 à l'état 1 est de 4/10, soit 0,4 et celle de l'état 0 à l'état 2 de 6/10, soit 0,6.

Avec ces informations estimées à partir de l'expérience de l'agent, nous disposons déjà de toutes les informations nécessaires pour pouvoir appliquer l'algorithme d'itération des valeurs.

Itération de valeur pour la fonction V

La fonction V en pratique pour l’environnement de lac gelé

https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_10_VI_Algorithm_for_V.ipynb

Importons les bibliothèques utilisées et définissons les constantes principales :

```
import gym
import collections
from torch.utils.tensorboard import SummaryWriter
ENV_NAME="FrozenLake-v0"
GAMMA = 0.9
TEST_EPISODES = 20
N =100
REWARD_GOAL = 0.9
```

Datastructure de l'agent

On conservera les informations dans les objets suivants, qui sont créés dans le constructeur de l’agent :

	Nom de variable
Dictionnaire des récompenses La clé est le composite "état source" + "action" + "état cible". La valeur est la récompense immédiate.	rewards
Dictionnaire contenant les compteurs des transitions vécues La clé est le composite "état" + "action" La valeur est un autre dictionnaire qui associe l'état cible à un nombre de fois où nous l'avons vu.	transits
Dictionnaire état valeur associant un état à la valeur calculée de cet état = valeur de la fonction état valeur V	values
Etat actuel de l'agent	state

Algorithme d’entraînement

La logique générale de notre algorithme d'apprentissage est simple. Tant que l'objectif de récompense souhaité n'est pas atteint, nous exécutons les étapes suivantes :

1. Jouer N étapes aléatoires de l'environnement pour remplir les tables rewards et transits.
2. Après ces N étapes, exécuter une étape d'itération de valeur sur tous les états, mettant à jour la table values.

3. Jouer ensuite plusieurs épisodes complets pour vérifier les améliorations en utilisant la table values mise à jour.
4. Si la récompense moyenne pour ces épisodes de test est supérieure à la limite souhaitée, nous arrêtons l'entraînement

Classe agent

Dans le constructeur de la classe d'agent, nous créons un environnement que nous utiliserons pour les échantillons de données, nous obtenons notre première observation et nous définissons des tableaux pour les récompenses, les transitions et les valeurs :

```
class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state = self.env.reset()
        self.rewards = collections.defaultdict(float)
        self.transits = collections.defaultdict(collections.Counter)
        self.values = collections.defaultdict(float)
```

Jouer des étapes aléatoires

L'estimation des transitions et des récompenses sera obtenue grâce à l'historique de l'interaction de l'agent avec l'environnement. On utilise pour celà la méthode play_n_random_steps qui joue N étapes aléatoires de l'environnement, **remplissant les tables rewards et transits** avec des expériences aléatoires.

```
def play_n_random_steps(self, count):
    for _ in range(count):
        action = self.env.action_space.sample()
        new_state, reward, is_done, _ = self.env.step(action)
        self.rewards[(self.state, action, new_state)] = reward
        self.transits[(self.state, action)][new_state] += 1
        if is_done:
            self.state = self.env.reset()
        else:
            self.state = new_state
```

Notez que nous n'avons pas besoin d'attendre la fin de l'épisode pour commencer l'apprentissage ; il nous suffit d'effectuer N étapes et de nous souvenir de leurs résultats. C'est l'une des différences entre l'itération de valeur et la méthode crossentropy, qui nécessite des épisodes complets pour apprendre.

Valeur de l'action

La méthode suivante calcule la fonction Q, la valeur d'une action à partir d'un état en utilisant les tableaux transits, rewards et values. Nous l'utiliserons à deux fins :

- sélectionner la meilleure action à effectuer à partir de l'état

- calculer la nouvelle valeur de l'état avec l'algorithme d'itération des valeurs.

```
def calc_action_value(self, state, action):
    target_counts = self.transits[(state, action)]
    total = sum(target_counts.values())

    action_value = 0.0
    for tgt_state, count in target_counts.items():
        reward = self.rewards[(state, action, tgt_state)]
        val = reward + GAMMA * self.values[tgt_state]
        action_value += (count / total) * val
    return action_value
```

Tout d'abord, à partir de la table transits, nous extrayons les compteurs de transition pour l'état et l'action que la méthode reçoit comme arguments. Nous additionnons tous les compteurs pour obtenir le nombre total de fois où nous avons exécuté l'action à partir de l'état.

Ensuite, nous calculons la contribution de chaque état cible en utilisant l'équation de Bellman :

$$Q(s,a) = \sum_{s'} P_{ss'}^{a}(r(s,a) + \gamma V(s'))$$

Cette contribution est égale à la récompense immédiate plus la valeur actualisée pour l'état cible et on multiplie cette somme par la probabilité de cette transition (compteur individuel divisé par la valeur totale calculée précédemment). Nous ajoutons le résultat de chaque itération à la variable qui sera retournée : action_value

Sélection de la meilleure action

Afin de sélectionner la meilleure action à partir d'un état donné, nous avons la méthode select_action qui utilise la méthode précédente calc_action_value pour prendre la décision :

```
def select_action(self, state):
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.calc_action_value(state, action)
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

Cette méthode itère sur toutes les actions possibles dans l'environnement, calcule la valeur de chaque action et renvoie l'action dont la valeur Q est la plus élevée.

Fonction d'itération de valeur

La méthode value_iteration boucle simplement sur tous les états de l'environnement, calcule pour chaque état les valeurs pour chaque action, et met à jour la valeur de chaque état avec la valeur maximale obtenue.

```
def value_iteration(self):
   for state in range(self.env.observation_space.n):
      state_values = [ self.calc_action_value(state, action) for action in range(self.env.action_space.n) ]
      self.values[state] = max(state_values)
```

La boucle d'apprentissage

Tout d'abord, on crée :

- un environnement de test pour ne pas modifier l’environnement utilisé pour les entraînements et le recueil les données aléatoires,
- une instance de la classe Agent avec son propre environnement intégré,
- le SummaryWriter pour TensorBoard, et quelques variables que nous utiliserons :

```
test_env = gym.make(ENV)
agent = Agent()
writer = SummaryWriter()
iter_no = 0
best_reward = 0.0
```

En théorie, la méthode value_iteration nécessite un nombre infini d'itérations pour converger exactement vers l'obtention de la fonction de valeur optimale. En pratique, nous pouvons nous arrêter lorsque nous atteignons un certain seuil de récompense REWARD_GOAL, et que nous considérons implicitement que la fonction de valeur ne change plus que légèrement au cours d'une itération de la boucle d'apprentissage.

```
while best_reward < REWARD_GOAL:
      agent.play_n_random_steps(N)
      agent.value_iteration()
      iter_no += 1
      reward_test = 0.0
      for _ in range(TEST_EPISODES):
         total_reward = 0.0
         state = test_env.reset()
         while True:
            action = agent.select_action(state)
            new_state, new_reward, is_done, _ = test_env.step(action)
            total_reward += new_reward
            if is_done: break
            state = new_state
         reward_test += total_reward
      reward_test /= TEST_EPISODES
      writer.add_scalar("reward", reward_test, iter_no)
      if reward_test > best_reward:
         print("Best reward updated %.2f at iteration %d " % (reward_test, iter_no))
         best_reward = reward_test
```

ÉTAPE 1 : on joue N pas aléatoires en appelant la méthode plays_n_random_steps.

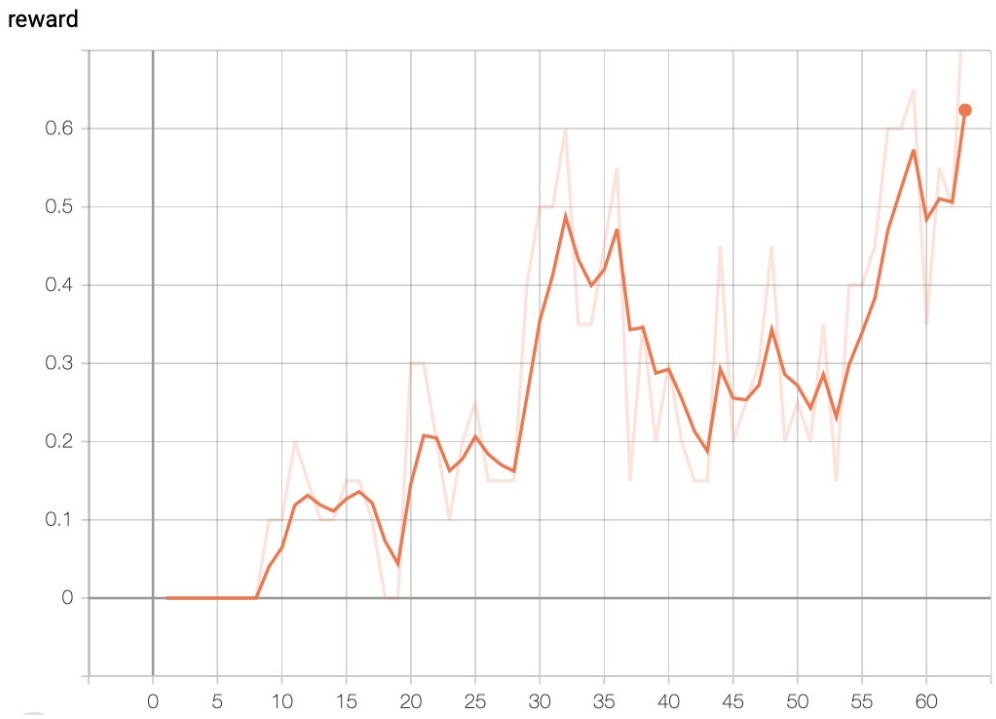
ÉTAPE 2 : on effectue un balayage d'itération de valeur sur tous les états en appelant la méthode value_iteration.

ÉTAPE 3 : Nous jouons ensuite plusieurs épisodes complets pour vérifier les améliorations en utilisant la table de valeurs mise à jour. Pour cela, on utilise agent.select_action pour trouver la meilleure action à entreprendre et on l'exécute dans l'environnement test_env, afin de vérifier les améliorations de l'agent et on écrit les données dans TensorBoard afin de suivre l'évolution de la meilleure récompense moyenne.

Bilan

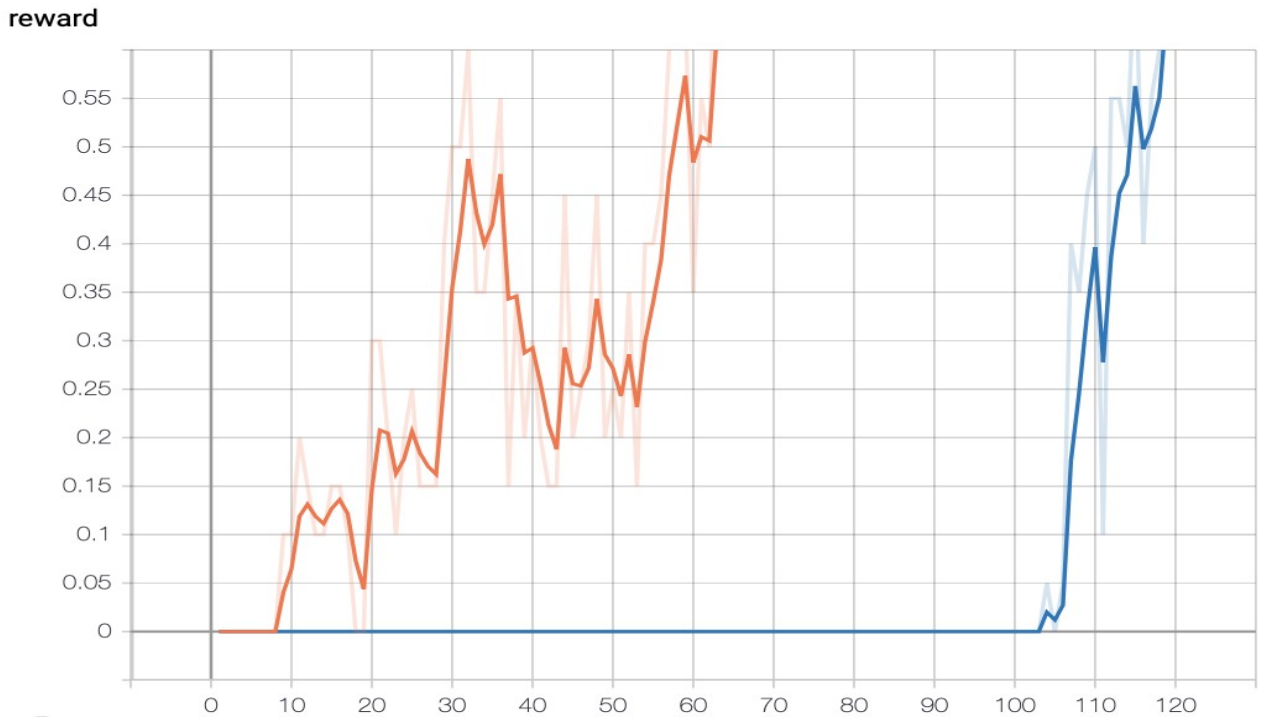
L'agent construit peut apprendre d'un environnement glissant :

```
new_test_env = gym.make('FrozenLake-v0', is_slippery=True)
state = new_test_env.reset()
new_test_env.render()
is_done = False
iter_no = 0
while not is_done:
    action = Select_action(state)
    new_state, reward, is_done, _ = new_test_env.step(action)
    new_test_env.render()
    state = new_state
    iter_no +=1
print("reward = ", reward)
print("iterations =", iter_no)
```



Il faut quelques secondes au maximum pour trouver une bonne politique qui résout l'environnement dans 80% des exécutions.

Nous pouvons appliquer cet algorithme à une version plus grande de FrozenLake : FrozenLake8x8-v0. Cette plus grande version se résout en beaucoup plus d'itérations, et, selon les graphiques de TensorBoard, on doit attendre le premier épisode réussi (il doit avoir au moins un épisode réussi pour commencer à apprendre de la table values), puis on atteint très rapidement la convergence. Le graphique ci-dessous compare la dynamique de la récompense pendant l'entraînement entre FrozenLake-4x4 et 8x8 :



Itération de valeur pour la fonction Q

Seules des modifications mineures sont nécessaires pour adapter le code d'itération de valeur mis en œuvre pour la fonction V au cas des valeurs d'action et à la fonction Q. Le changement le plus évident concerne notre tableau de valeurs. Dans le cas précédent, nous conservions la valeur de l'état, de sorte que la clé dans le dictionnaire était juste un état. Maintenant, nous devons stocker les valeurs de la fonction Q, qui a deux paramètres : état et action, donc la clé dans la table de valeur est maintenant un composite.

Les structures de données qui conserveront nos tables et les fonctions que nous utiliserons dans la boucle d'apprentissage, sont les mêmes que dans le cas de la fonction V.

Les principaux changements sont les suivants :

- la table **values** devient un n dictionnaire qui fait maintenant correspondre une paire état-action à la valeur calculée de cette action.
- Nous n'avons plus besoin de la fonction calc_action_value, puisque nos valeurs d'action sont stockées dans la table **values**.
- la méthode **value_iteration()** de l'agent ne peut plus se contenter d'être un simple wrapper demandant à calc_action_value() de réaliser le travail d'approximation de Bellman

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a')$$

Le code de la méthode **value_iteration()** devient le suivant :

```
def value_iteration_for_Q(self):
    for state in range(self.env.observation_space.n):
        for action in range(self.env.action_space.n):
            action_value = 0.0
            target_counts = self.transits[(state, action)]
            total = sum(target_counts.values())
            for tgt_state, count in target_counts.items():
                key = (state, action, tgt_state)
                reward = self.rewards[key]
                best_action = self.select_action(tgt_state)
                val = reward + GAMMA * \
                      self.values[(tgt_state, best_action)]
                action_value += (count / total) * val
            self.values[(state, action)] = action_value
```

Le code ressemble beaucoup à celui de calc_action_value dans l'implémentation précédente.

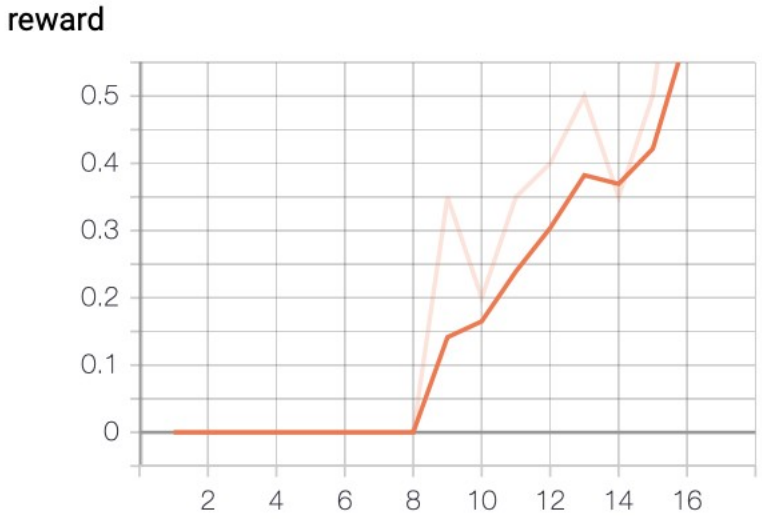
Pour un état et une action donnés, il calcule la valeur de l'action en utilisant les informations recueillies par la fonction play_n_random_steps qui joue N étapes aléatoires de l'environnement, remplissant les tables **rewards** et **transits** avec des expériences aléatoires.

Cependant, dans l'implémentation précédente, la fonction V étant stockée dans la table values, nous nous contentions d'extraire les valeurs de cette table. Nous ne pouvons plus le faire, donc nous devons appeler la méthode select_action, qui choisit pour nous la meilleure action avec la plus grande valeur Q, et nous prenons cette valeur Q comme valeur de l'état cible.

```
def select_action(self, state):
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.values[(state, action)]
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

Cette méthode select_action est implémentée différemment, elle n'appelle plus la méthode calc_action_value mais itère sur les actions et recherche leurs valeurs dans la table values.

Le code de la boucle d'apprentissage et celui pour tester le client et tracer les résultats dans TensorBoard restent inchangés.



Les valeurs Q sont beaucoup plus pratiques à utiliser, car il est beaucoup plus simple pour l’agent de prendre des décisions concernant les actions basées sur les valeurs Q que sur les valeurs V. Dans le cas des valeurs Q, pour choisir l'action basée sur l'état, l'agent doit simplement calculer la valeur Q pour toutes les actions disponibles en utilisant l'état actuel et choisir l'action avec la valeur Q la plus grande.

Pour faire de même en utilisant les valeurs V, l'agent doit connaître non seulement les valeurs mais aussi les probabilités des transitions. En pratique, nous les connaissons rarement à l'avance, et l'agent doit donc estimer les probabilités de transition pour chaque paire action-état.

Dans la méthode d'itération des valeurs pour la fonction V, cette dépendance à la probabilité ajoute une charge supplémentaire pour l'agent. Cela dit, il est important de connaître cette méthode car elle constitue un élément essentiel des méthodes avancées.

Les méthodes de Monte Carlo

La plus simple des méthodes classiques de l'apprentissage par renforcement pour estimer la valeur d'une politique π consiste à exécuter plusieurs épisodes avec cette politique en collectant des centaines de trajectoires, puis à calculer des moyennes pour chaque état. Cette méthode d'estimation des fonctions de valeur est appelée prédiction de Monte Carlo (MC).

Mais comment estimer la politique optimale et comment gérer le dilemme exploration-exploitation ?

Monte Carlo versus programmation dynamique

Nous avons présenté une solution au PDM appelée programmation dynamique, dont Richard Bellman est le pionnier. L'équation de Bellman nous permet de définir la fonction de valeur de manière récursive et peut être résolue avec l'algorithme d'itération de valeur. En résumé, la programmation dynamique fournit une base pour l'apprentissage par renforcement, mais nous devons parcourir tous les états à chaque itération (leur taille peut croître de manière exponentielle et l'espace d'état peut être très grand ou infini). La programmation dynamique nécessite également un modèle de l'environnement, en particulier la connaissance de la probabilité de transition d'état

$$p\,(s', r \mid s, a)$$

En revanche, les méthodes de Monte Carlo consistent à apprendre par l'expérience. Toute valeur attendue peut être approchée par des moyennes d'échantillons, tout ce que nous devons faire est jouer un tas d'épisodes, rassembler les retours et en faire la moyenne. Les méthodes de Monte Carlo sont en fait un ensemble d'alternatives à l'algorithme de base. Elles ne concernent que les tâches épisodiques, où l'interaction s'arrête lorsque l'agent rencontre un état terminal. En d'autres termes, nous supposons que l'expérience est divisée en épisodes, et que tous les épisodes finissent par se terminer, quelles que soient les actions sélectionnées.

Il est important de noter que les méthodes de Monte Carlo ne nous donnent une valeur que pour les états et les actions que nous avons rencontrés, et si nous ne rencontrons jamais un état, sa valeur est inconnue.

Méthodes de Monte Carlo

La politique optimale $\pi*$ spécifie, pour chaque état de l'environnement s, comment l'agent doit sélectionner une action a afin de maximiser la récompense G. L'agent structure sa recherche d'une politique optimale en estimant d'abord la fonction action-valeur optimale $q*$, et une fois $q*$ connue, $\pi*$ est rapidement obtenue.

Au départ, l'agent initialise une politique de base de type aléatoire équiprobable, une politique stochastique qui, à partir de chaque état, choisit aléatoirement avec une égale probabilité dans l'ensemble des actions disponibles. L'agent utilise cette politique, interagit avec l'environnement pour recueillir quelques épisodes, puis consolide les résultats pour arriver à une meilleure politique.

Dans la pratique, l’agent estime la fonction action-valeur à l'aide d'une table que nous appellons Q-table. Cette table de base des méthodes de Monte Carlo comporte une ligne pour chaque état et une colonne pour chaque action. **Q(s,a) est l'entrée correspondant à l'état s et à l'action a.**

On parle de méthodes Monte Carlo de prédiction. Nous concentrerons notre explication sur la fonction action-valeur Q, mais on pourrait aussi estimer la fonction état-valeur V.

L'algorithme commence par collecter de nombreux épisodes avec la politique. Chaque entrée de la table Q correspond à un état et une action particuliers et une entrée de la Q-table contient la valeur moyenne des gains potentiels Gt obtenus par l'agent lorsqu'il s'est trouvé dans cet état et a choisi cette action.

Une paire état-action (St, At) pouvant être visitée plus d'une fois dans un épisode, il existe deux versions de l'algorithme, selon que l'on comptabilise toutes les visites d'une même paire état/action pour le calcul de la valeur moyenne des gains potentiels ou que l'on ne considère que la première visite dans chaque épisode.

- Prédiction MC à chaque visite
- Prédiction MC à la première visite dans chaque épisode.

La loi des grands nombres garantit la convergence de l'une ou l'autre des méthodes vers la vraie fonction de valeur.

Nous allons nous intéresser au problème du BlackJack. Avec ce jeu, peu importe le choix de la méthode (première visite ou non) car le même état ne se répète jamais dans un même épisode. Le pseudocode de l'algorithme de première visite est le suivant :

Input: policy π, $num\_episodes$
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ ***to*** $num\_episodes$ **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $returns\_sum(S_t, A_t) \leftarrow returns\_sum(S_t, A_t) + G_t$
  end
end
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
return Q

La variable num_épisodes indique le nombre d'épisodes à collecter par l'agent et outre la table Q, il il y a deux autres tableaux importants. Tous trois sont structurés de manière identique : une ligne pour chaque état et une colonne pour chaque action.

- N est une table gardant la trace du nombre de (premières) visites que nous avons faites à chaque paire état-action.
- returns_sum est une table gardant la trace de la somme des récompenses obtenues après les (premières) visites à chaque paire état-action.

Après chaque épisode, N et returns_sum sont mises à jour pour stocker les informations fournies par l'épisode. L'estimation finale de Q est faite après que tous les épisodes aient été collectés.

Règles du BlackJack

Le blackjack est un populaire jeu de cartes de casino. Le but est d'obtenir des cartes dont la somme des valeurs numériques est la plus grande possible sans dépasser 21. Si l'on dépasse 21, on perd sa main (bust), tandis que si les deux parties ont 21, on fait match nul (draw). Nous allons considérer une version simplifiée dans laquelle chaque joueur joue indépendamment contre le croupier.

- Du 2 au 9 : chaque carte a sa propre valeur nominale.
- Les 10, Valets J, Dames Q et Rois K valent 10 et sont appelés les « Bûches ».
- Les as valent 1 ou 11 selon le jeu du joueur Si votre main ne dépasse pas 21, l'as compte 11 et on parle d'as utilisable (usable ace). Si au contraire elle le dépasse, l'as compte pour 1; la valeur de l'As étant toujours calculée à votre avantage.
- La main appelée « Blackjack » est composée d'un as et d'une carte valant 10, pour un total de 21, reçues dès le début. Si le joueur atteint 21 points avec plus de deux cartes, on compte 21 points et non Blackjack

Les deux parties, le joueur et le croupier, reçoivent deux cartes. Les deux cartes du joueur sont face visible, tandis qu'une seule des cartes du croupier est face visible. Après avoir vu ses cartes et la première carte du croupier, le joueur peut choisir de demander une carte supplémentaire (hit) ou pas et de rester sur son résultat (stand).

Le croupier révèle alors sa deuxième carte - si la somme est inférieure à 17, il continue à tirer des cartes jusqu'à ce qu'il atteigne 17, après quoi il reste sur son résultat.

Nous utiliserons l'environnement d'OpenAI, Blackjack-v0. Chaque état est un 3-tuple de :

- la somme actuelle du joueur $\in\{0,1,...,31\}$. 0,1,2 et 3 ne sont pas atteignables. 31 est atteint lorsque le joueur a 2 cartes valant 10 et un as. Il s'agit d'un repère pour l'algorithme car dans la pratique lorsqu'on a cette main, on ne fait que 21.
- la carte visible du croupier $\in\{1,...,10\}$
- un booléen indiquant si le joueur possède ou non un as utilisable (usable ace).

L'agent a deux actions potentielles :

- stick/stand/stay (action 0) : ne prendre aucune autre carte
- hit (action 1) : prendre une autre carte du croupier.

Chaque partie de blackjack est un épisode. Des récompenses terminales de +1, -1 et 0 sont accordées pour les gains, les pertes et les matchs nuls, respectivement. Il n'y a pas de récompense accordée lors des étapes non terminales d'une partie/épisode, nous ne faisons pas de discounting et gamma vaut 1 ; par conséquent, ces récompenses terminales sont également les gains potentiels.

Nous commençons par importer les bibliothèques nécessaires :

```
import sys
import gym
import numpy as np
from collections import defaultdictenv = gym.make('Blackjack-v0')
print(env.observation_space)
print(env.action_space)
```

Il y a 704 états différents dans l'environnement correspondant à 32 x 11 x 2 et deux actions possibles stick ou hit

```
Tuple(Discrete(32), Discrete(11), Discrete(2))
Discrete(2)
```

Le code suivant permet de jouer en prenant des décisions aléatoires :

```
state = env.reset()
while True:
      print(state)
      action = env.action_space.sample()
      state, reward, done, info = env.step(action)
      if done:
          print('Reward: ', reward, '\n')
         break
```

Voici un exemple de jeu type :

```
(21, 4, True)
(13, 4, False)
(14, 4, False)
Reward:  -1.0
```

Méthode de Monte Carlo à chaque visite

Considérons une politique où le joueur décide d'une action en se basant uniquement sur son score actuel. Si la somme des cartes est supérieure à 18, il considère avec une probabilité de 75 % qu'il est dangereux d'accepter une nouvelle carte. Si la somme des cartes qu'il a est inférieure ou égale à 18, il estime avec une probabilité de 75 % pouvoir demander une nouvelle carte.

- si la somme est supérieure à 18, nous choisissons l'action Stick avec une probabilité de 75%
- si la somme est égale ou inférieure à 18, nous choisissons l'action Hit avec une probabilité de 75%

Cette première approche de la politique est très rustique. Elle n'utilise qu'une seule information parmi les trois que l'état contient, et ne se soucie ni de la carte visible du croupier ni de savoir si nous avons un as utilisable. La fonction generate_episode échantillonne un épisode selon cette politique :

```
def generate_episode(env):
   episode = []
   state = env.reset()
   while True:
      probs = [0.75, 0.25] if state[0] > 18 else [0.25, 0.75]
      action = np.random.choice(np.arange(2), p=probs)
      next_state, reward, done, info = env.step(action)
      episode.append((state, action, reward))
      state = next_state
      if done:
         break
      return episode
```

Le code utilise la politique π décidée et renvoie un épisode sous la forme d'une liste de tuples de (état, action, récompense).

- episode[i][0] = état au pas de temps i
- episode[i][1] = action au pas de temps i
- episode[i][2] = récompense au pas de temps i+1.

Pour mieux comprendre la structure de données des épisodes, on peut jouer quelques parties :

```
for i in range(10):
    print(generate_episode(env))


[((19, 3, True), 1, 0.0), ((19, 3, False), 0, 1.0)]
[((14, 5, False), 1, -1.0)]
[((15, 10, False), 0, -1.0)]
[((15, 10, False), 1, 0.0), ((17, 10, False), 1, -1.0)]
[((13, 2, False), 1, -1.0)]
[((13, 4, False), 1, 0.0), ((14, 4, False), 0, -1.0)]
[((11, 10, False), 1, 0.0), ((21, 10, False), 0, 0.0)]
[((15, 10, False), 1, -1.0)]
[((9, 9, False), 1, 0.0), ((16, 9, False), 0, 1.0)]
[((13, 7, False), 1, 0.0), ((18, 7, False), 1, 0.0), ((21, 7, False), 1, -1.0)]
```

Nous allons coder l’algorithme MC à chaque visite sous la forme d’une fonction nommée mc_prediction. Cette fonction prend comme arguments l'instance de l’environnement, le nombre d'épisodes à générer, la fonction de génération des épisodes et le taux d'actualisation (valeur par défaut 1). Elle renvoie en sortie le Q-table, un dictionnaire de tableaux unidimensionnels qui constitue notre estimation de la fonction action-valeur.

```
def mc_prediction(env, num_episodes, generate_episode, gamma=1.0):
    returns_sum = defaultdict(lambda: np.zeros(env.action_space.n))
    N = defaultdict(lambda: np.zeros(env.action_space.n))
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    for episode in range(1, num_episodes+1):

        if episode % 10000 == 0: # monitor progress
            print("\repisode {}/{}.".format(episode, num_episodes), end="")
            sys.stdout.flush()

        episode = generate_episode(env)
        states, actions, rewards = zip(*episode)

        discounts = np.array([gamma**i for i in range(len(rewards)+1)])

        for i, state in enumerate(states):
            returns_sum[state][actions[i]] += sum(rewards[i:]*discounts[:-(1+i)])
            N[state][actions[i]] += 1.0
            Q[state][actions[i]] = returns_sum[state][actions[i]] / N[state][actions[i]]
    return Q
```

La fonction mc_prediction :

- initialise les dictionnaires N, returns_sum, et Q

- génère des épisodes générés à l'aide de la fonction generate_episode qui utilise pour cela la politique. Chaque épisode est une liste de tuples d'états, d'actions et de récompenses et la commande zip permet de séparer les états, les actions et les récompenses.
- parcourt la trajectoire de l'épisode et pour chaque état/action rencontré, met à jour N[état][action] en incrémentant sa valeur de 1 et returns_sum[état][action] en lui ajoutant la valeur du gain potentiel obtenu.
- met à jour le tableau Q estimé, dès que les valeurs mises à jour de returns_sum et N sont connues

Nota sur les indices pour comprendre le calcul des gains potentiels :

- states, action et rewards sont des listes de taille n
- discounts est de taille n+1 par construction
- rewards[i:] est de taille n-i
- discounts[:-(1+i)] est aussi de taille n-i

i	i+1	…...	n-1 = i + (n – 1 – i)
	On a 2 éléments : i et i + 1		On a n -1 – i + 1 éléments Soit n – i éléments

discounts[:-1] est de taille n+1 -1 = n
discounts[:-(i+1)] est de taille n + 1 – (i + 1) = n – i

Visualisation de la fonction état valeur

On peut calculer la fonction état valeur pour apprécier les états qui ont le plus de valeur. Il faut pour cela utiliser la politique et la table Q produite :

```
State_Value_table={}
for state, actValues in Q.items():
    State_Value_table[state]= (state[0]>18)*(np.dot([0.75, 0.25],actValues)) + (state[0]<=18)*(np.dot([0.75, 0.25],actValues))
```

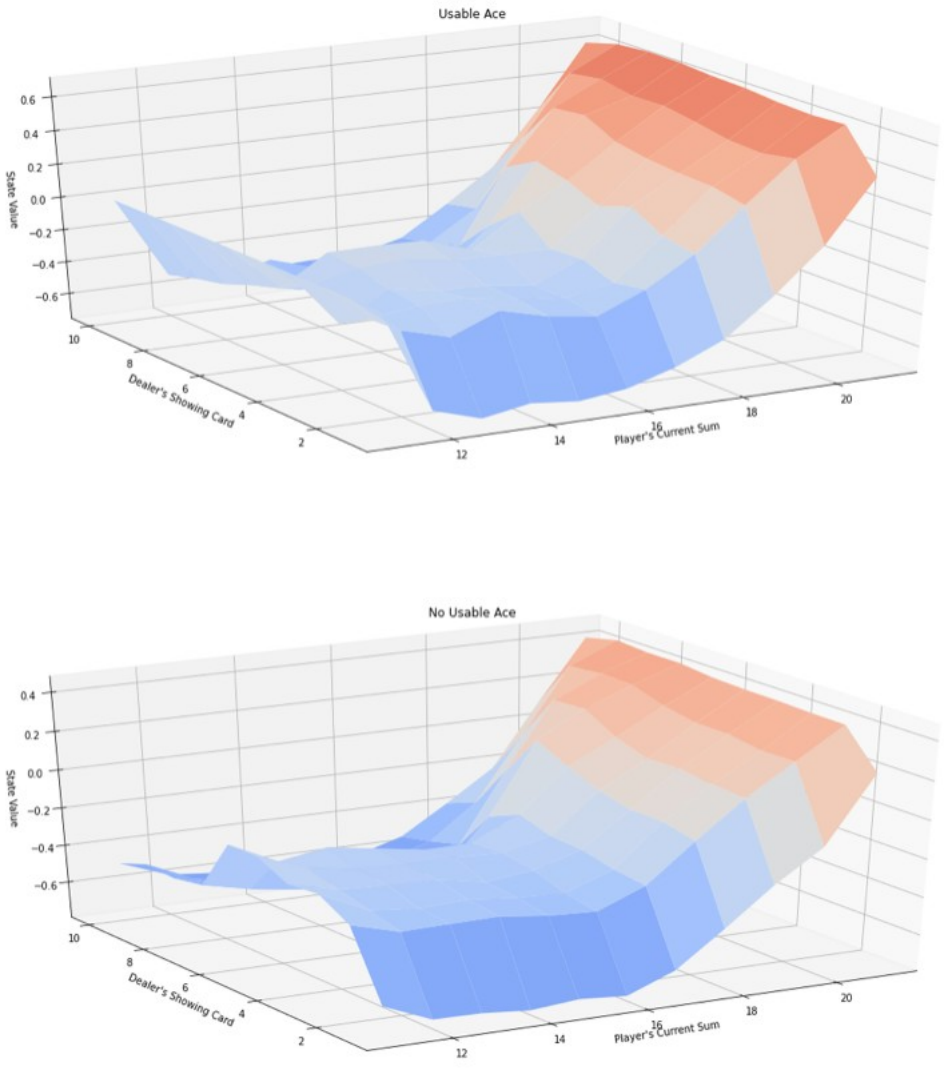
On peut utiliser la fonction suivante pour produire des graphiques 3D :

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

def plot_blackjack_values(V):

    def get_Z(x, y, usable_ace):
        if (x,y,usable_ace) in V:
            return V[x,y,usable_ace]
        else:
            return 0

    def get_figure(usable_ace, ax):
        x_range = np.arange(11, 22)
```

```
    y_range = np.arange(1, 11)
    X, Y = np.meshgrid(x_range, y_range)

    Z = np.array([get_Z(x,y,usable_ace) for x,y in zip(np.ravel(X), np.ravel(Y))]).reshape(X.shape)

    surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.coolwarm, vmin=-1.0, vmax=1.0)
    ax.set_xlabel('Player\'s Current Sum')
    ax.set_ylabel('Dealer\'s Showing Card')
    ax.set_zlabel('State Value')
    ax.view_init(ax.elev, -120)

fig = plt.figure(figsize=(20, 20))
ax = fig.add_subplot(211, projection='3d')
ax.set_title('Usable Ace')
get_figure(True, ax)
ax = fig.add_subplot(212, projection='3d')
ax.set_title('No Usable Ace')
get_figure(False, ax)
plt.show()
```

On produit les graphiques avec la commande suivante :

```
plot_blackjack_values(State_Value_table)
```



Il y a deux graphiques

Le premier correspond au cas où nous avons un as utilisable et le second au cas où nous n’en avons pas.

Dans les deux cas, nous voyons que les valeurs d'état les plus élevées correspondent au moment où la somme du joueur est autour de 20 ou 21

Cela semble évident car c’est dans ce cas que nous avons le plus de chances de gagner la partie.

Méthode de Monte Carlo à la première visite

```
for s, a, r in episode:
    first_occurence_idx = next(i for i,x in enumerate(episode) if x[0] == s)
    G = sum([x[2]*(gamma**i) for i,x in enumerate(episode[first_occurence_idx:])])
    returns_sum[s][a] += G
    N[s][a] += 1.0
    Q[s][a] = returns_sum[s][a] / N[s][a]
```

Cf https://github.com/djbyrne/MonteCarlo/blob/master/Prediciton.ipynb

Exploration vs. Exploitation

Jusqu'à présent, nous avons appris comment un agent peut interagir avec l'environnement en utilisant par exemple une politique aléatoire, pour ensuite capitaliser cette expérience au sein d'une table Q qui propose une estimation de la fonction action-valeur de cette politique.

La question est maintenant de savoir comment utiliser cela dans notre recherche d'une politique optimale.

Politiques greedy

Pour obtenir une meilleure politique, qui n'est pas nécessairement la politique optimale, il suffit de sélectionner pour chaque état l'action qui maximise la table Q. Nous appellerons cette nouvelle politique π' greedy ou avide. Lorsque nous prenons un tableau Q et que nous utilisons l'action qui maximise chaque ligne d'état pour élaborer la politique, nous disons que nous mettons en œuvre une politique avide par rapport au tableau Q.

Il est courant de se référer à l'action sélectionnée comme étant l'action avide. Dans le cas d'un PDM fini, l'estimation de la fonction action-valeur est représentée dans une Q-table. Pour obtenir l'action avide, pour chaque ligne du tableau, il suffit de sélectionner l'action correspondant à la colonne qui maximise la ligne.

Politiques Epsilon-Greedy

Cependant, au lieu de toujours mettre en oeuvre une politique greedy, nous allons utiliser une politique dite Epsilon-Greedy qui a beaucoup de chances de choisir l'action avide, mais qui, avec une probabilité faible mais non nulle, choisit l'une des autres actions à la place. Dans ce cas, on utilise un petit nombre positif epsilon ϵ, qui doit être compris entre zéro et un. Ceci est motivé par le fait qu'un agent doit trouver un équilibre entre :

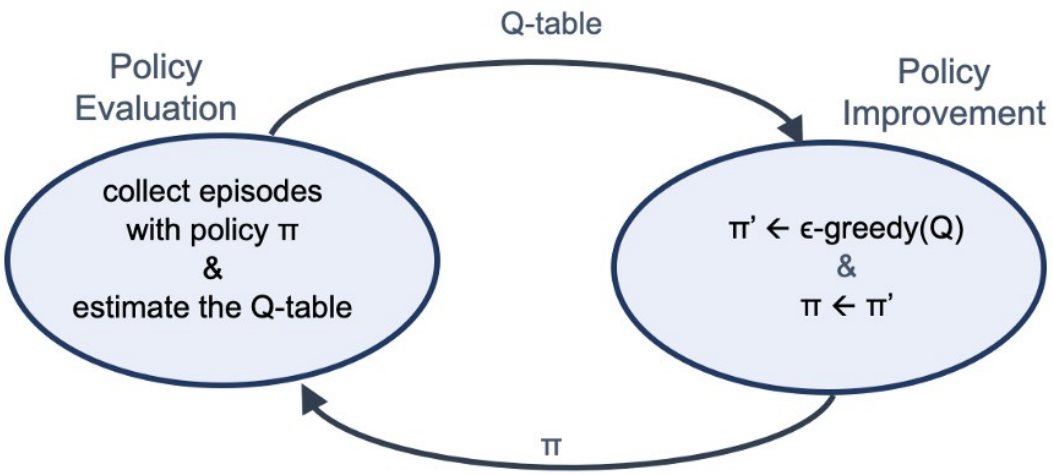
- la volonté de se comporter de manière optimale sur la base de ses connaissances actuelles,
- et le besoin d'acquérir des connaissances nouvelles pour obtenir un meilleur comportement futur.

Contrôle de Monte Carlo

La méthode de contrôle de Monte Carlo permet d'estimer la politique optimale. Elle est en 2 étapes :

- Étape 1 : utiliser la politique π pour construire la table Q
- Étape 2 : amélioration de la politique en la changeant pour qu'elle soit ϵ-greedy par rapport à la table Q (notée ϵ-greedy(Q))

Il est courant de désigner l'étape 1 comme l'évaluation de la politique puisqu'elle est utilisée pour déterminer la fonction action-valeur de la politique. L'étape 2 étant utilisée pour améliorer la politique, nous l'appelons étape d'amélioration de la politique.



En utilisant cette nouvelle terminologie, nous pouvons résumer que notre méthode de contrôle Monte Carlo alterne entre les étapes d'évaluation et d'amélioration de la politique pour trouver la politique optimale π∗ :

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*}$$

avec :

- E indiquant une évaluation complète de la politique
- I indiquant une amélioration complète de la politique

Cependant, certaines questions se posent, comment fixer la valeur de ϵ lors de la construction de politiques ϵ-greedy ?

Dilemme exploration-exploitation

La dynamique de l'environnement étant inconnue de notre agent et son objectif étant de maximiser le gain potentiel, l'agent doit apprendre à connaître l'environnement par l'interaction. A chaque pas de temps, lorsque l'agent choisit une action, il fonde sa décision sur son expérience passée avec l'environnement.

L'instinct pourrait être de sélectionner l'action qui, sur la base de l'expérience passée, maximisera le rendement. Cette politique greedy par rapport à l'estimation de la fonction action-valeur peut toutefois facilement conduire à une convergence vers une politique sous-optimale. Cela peut se

produire parce que, dans les premiers épisodes, les connaissances de l'agent sont assez limitées et qu'il n'envisage pas certaines actions non avides toutefois meilleures en termes de récompenses futures que les actions connues.

Un agent performant ne peut donc pas se limiter à agir avec avidité à chaque pas de temps ; afin de découvrir la politique optimale, il doit continuer à affiner le gain potentiel estimé pour toutes les paires état-action et en même temps, il doit maintenir un certain niveau d'actions avides pour conserver l'objectif de maximiser le rendement aussi rapidement que possible.

En résumé, il doit trouver un moyen d'équilibrer :

- la volonté de se comporter de manière optimale en fonction de ses connaissances actuelles (exploitation)
- et le besoin d'acquérir des connaissances pour obtenir un meilleur jugement (exploration).

Une solution potentielle à ce dilemme est mise en œuvre en modifiant progressivement la valeur de ϵ lors de la construction des politiques ϵ-greedy. Il est logique que l'agent commence son interaction avec l'environnement en essayant diverses stratégies, c’est-à-dire en privilégiant l'exploration plutôt que l'exploitation. Dans cette optique, la meilleure politique de départ est la politique aléatoire équiprobable, car elle a une probabilité égale d'explorer toutes les actions possibles à partir de chaque état. En fixant $\epsilon=1$, on obtient une politique ϵ-greedy qui est équivalente à la politique aléatoire équiprobable.

Aux étapes ultérieures, il est logique de favoriser l'exploitation par rapport à l'exploration, où la politique devient progressivement plus avide par rapport à l'estimation de la fonction action-valeur. En fixant $\epsilon=0$, on obtient la politique avide. Il a été démontré que favoriser initialement l'exploration par l'exploitation et préférer progressivement l'exploitation à l'exploration est une stratégie optimale.

Afin de garantir que le contrôle MC converge vers la politique optimale $\pi*$, nous devons nous assurer que deux conditions sont remplies :

- chaque paire état-action est visitée une infinité de fois,
- et la politique converge vers une politique qui est avide par rapport à l'estimation de la fonction action-valeur Q.

Nous appelons ces conditions Greedy in the Limit with Infinite Exploration qui garantissent que l'agent continue à explorer pour tous les pas de temps, et que l'agent exploite progressivement plus et explore moins.

Une façon de satisfaire ces conditions est de modifier la valeur de ϵ, en la faisant décroître progressivement. Cependant, il faut être très prudent en fixant le taux de décroissance de ϵ. Déterminer la meilleure décroissance n'est pas trivial et nécessite un peu d'alchimie, c'est-à-dire d'expérience.

Le contrôle Monte Carlo et les méthodes Temporal-Difference

Nous allons améliorer les méthodes de contrôle de Monte Carlo pour estimer la politique optimale. Dans l'algorithme précédent de contrôle de Monte Carlo, nous collectons un grand nombre d'épisodes pour construire la Q-table. Ensuite, une fois que les valeurs de la table Q ont convergé, nous utilisons la table pour proposer une politique améliorée.

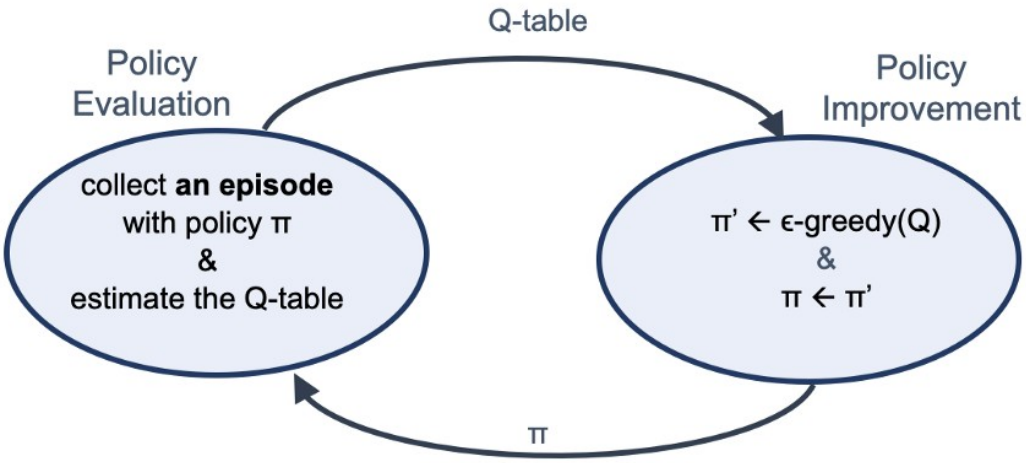
Cependant, les méthodes de prédiction Monte Carlo peuvent être mises en œuvre de manière incrémentielle, épisode par épisode, et c'est ce que nous allons faire. Même si la politique est mise à jour avant que les valeurs de la table Q n'approchent précisément la fonction action-valeur, cette estimation de moindre qualité contient néanmoins suffisamment d'informations pour aider à proposer des politiques successivement meilleures.

De plus, la table Q peut être mise à jour à chaque pas de temps au lieu d'attendre la fin de l'épisode en utilisant les méthodes de différence temporelle.

Améliorations du contrôle de Monte Carlo

Nous avons présenté comment l'algorithme de contrôle de Monte Carlo collecte un grand nombre d'épisodes pour construire la Q-table (étape d'évaluation de la politique). Ensuite, une fois que la Q-table se rapproche de la fonction action-valeur q_π, l'algorithme utilise la table pour proposer une politique améliorée π' qui est ϵ-greedy par rapport à la Q-table (indiquée comme ϵ-greedy(Q)), ce qui donnera une politique meilleure que la politique originale π (étape d'amélioration de la politique).

Peut-être serait-il plus efficace de mettre à jour la table Q après chaque épisode ? Oui, nous pourrions modifier l'étape d'évaluation de la politique pour mettre à jour la Q-table après chaque épisode d'interaction. Ensuite, la table Q mise à jour pourrait être utilisée pour améliorer la politique. Cette nouvelle politique pourrait alors être utilisée pour générer l'épisode suivant, et ainsi de suite :



La variante la plus populaire de l'algorithme de contrôle MC qui met à jour la politique après chaque épisode (au lieu d'attendre pour mettre à jour la politique que les valeurs de la table Q aient complètement convergé après de nombreux épisodes) est le contrôle MC à alpha constant.

Contrôle MC à alpha constant

Dans cette variante du contrôle MC, au cours de l'étape d'évaluation de la politique, l'agent collecte un épisode

$$S_0, A_0, R_1, \ldots, S_T$$

en utilisant la politique la plus récente π.
Une fois l'épisode terminé à l'étape T, pour chaque étape t, la paire état-action correspondante (St, At) est modifiée à l'aide de l'équation de mise à jour suivante :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

où Gt est le gain potentiel au pas de temps t, et Q(St,At) est l'entrée de la table Q correspondant à l'état St et à l'action At.
D'une manière générale, l'idée de base de cette équation de mise à jour est que l'élément Q(St,At) de la table Q contient l'estimation par l'agent du gain potentiel attendu si l'environnement est dans l'état St et que l'agent choisit l'action At.

Si le gain potentiel Gt n'est pas égal au gain potentiel attendu contenu dans Q(St,At), nous ajustons la valeur de Q(St,At) pour qu'elle corresponde un peu plus au gain potentiel Gt. L'ampleur de la modification que nous apportons à Q(St,At) est contrôlée par l'hyperparamètre α.

Nous devons toujours fixer la valeur de α à un nombre supérieur à zéro et inférieur (ou égal) à un. Dans les cas les plus extrêmes :

- Si α=0, alors l'estimation de la fonction action-valeur n'est jamais mise à jour par l'agent.
- Si α=1, alors l'estimation de la valeur finale pour chaque paire état-action est toujours égale au dernier retour vécu par l'Agent.

Fixer la valeur d'Epsilon

Le comportement aléatoire est meilleur au début de l'entraînement lorsque notre approximation de la table Q est mauvaise, car il nous donne des informations plus uniformément distribuées sur les états de l'Environnement. Cependant, au fur et à mesure que notre formation progresse, le comportement aléatoire devient inefficace, et nous voulons utiliser notre approximation de la table Q pour décider comment agir. Les politiques Epsilon-Greedy ont été introduites à cette fin, pour disposer d'une méthode effectuant un mélange de deux comportements extrêmes qui est juste la commutation entre le hasard et la politique Q en utilisant l'hyperparamètre de probabilité ϵ. En faisant varier ϵ , nous pouvons sélectionner le ratio d'actions aléatoires.

Nous définirons qu'une politique est ϵ-greedy par rapport à une estimation de la fonction action-valeur Q si pour chaque état,

- avec la probabilité 1-ϵ, l'agent sélectionne l'action greedy, et
- avec une probabilité ϵ, l'agent choisit une action uniformément au hasard dans l'ensemble des actions disponibles (non greedy et greedy).

Ainsi, plus ϵ est grand, plus vous avez de chances de choisir une des actions non greedy.
Pour construire une politique π qui est ϵ-greedy par rapport à l'estimation actuelle de la fonction action-valeur Q, nous définirons mathématiquement la politique comme suit

si l'action a maximise Q(s,a) :	Sinon :
$1\text{-}\epsilon + \epsilon/\lvert A(S)\rvert$	$\epsilon/\lvert A(S)\rvert$

pour chaque s∈S et a∈A(s).

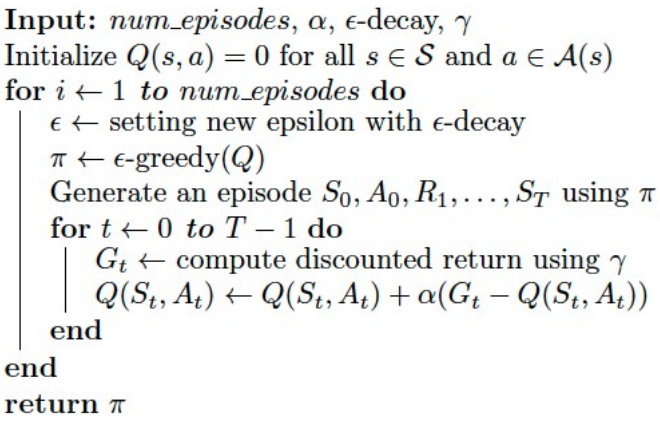
Dans cette équation, il est inclus un terme supplémentaire ϵ/|A(s)| pour l'action optimale (|A(s)| est le nombre d'actions possibles) car la somme de toutes les probabilités doit être égale à 1. Notez que

si nous additionnons les probabilités d'effectuer toutes les actions non optimales, nous obtiendrons (|A(s)|-1)×ϵ/|A(s)|, et en ajoutant cela à 1-ϵ+ϵ/|A(s)| , la probabilité de l'action optimale, la somme donne un.

Rappelez-vous que pour garantir que le contrôle MC converge vers la politique optimale $\pi*$, nous devons assurer les conditions Greedy in the Limit with Infinite Exploration qui assurent que l'agent continue à explorer pour tous les pas de temps, et que l'agent exploite progressivement plus et explore moins. Nous avons présenté qu'une façon de satisfaire ces conditions est de modifier la valeur de ϵ, en la faisant décroître progressivement, lors de la spécification d'une politique ϵ-greedy. La pratique habituelle est de commencer avec ϵ = 1,0 (100% d'actions aléatoires) et de le diminuer lentement jusqu'à une petite valeur ϵ > 0 (dans notre exemple, nous utiliserons ϵ = 0,05) . En général, ceci peut être obtenu en introduisant un facteur ϵ-decay avec une valeur proche de 1 qui multiplie les ϵ à chaque itération.

Pseudocode

Nous pouvons résumer toutes les explications précédentes avec ce pseudocode pour l'algorithme de contrôle MC à α constant qui guidera notre mise en œuvre de l'algorithme :



Une implémentation simple du contrôle MC à α constant

Nous allons écrire une implémentation du contrôle MC à α constant pour aider un agent à récupérer la politique optimale dans le cadre du jeu de Blackjack.

https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_13_14_Monte_Carlo.ipynb

Contrôle MC à α constant pour l'environnement Blackjack

Nous avons déjà implémenté une politique où le joueur choisit toujours stick si la somme de ses cartes dépasse 18 pour l'environnement BlackJack. Dans ce cas, la fonction generate_episode échantillonne les épisodes en utilisant cette politique définie par le programmeur.
Ici, au lieu d'utiliser une politique codée en dur par le programmeur, l'algorithme de contrôle MC estimera et renverra une politique optimale, ainsi que la table Q.

```
env = gym.make('Blackjack-v0')
num_episodes=1000000
alpha = 0.02
eps_decay=.9999965
gamma=1.0
policy, Q = MC_control(env, num_episodes, alpha, eps_decay, gamma)
```

La politique est un dictionnaire dont :

- la clé correspond à un état : un 3-tuple indiquant la somme actuelle du joueur, la carte visible du croupier et si le joueur possède ou non un as utilisable
- la valeur indique l'action que l'agent choisit après avoir observé cet état en suivant cette politique.

L'autre dictionnaire est le Q-table Q où :

- la clé correspond à un état
- la valeur est un tableau de dimension égale au nombre d'actions (2 dimensions dans notre cas) dont chaque élément est la valeur estimée de chaque action.

Cet algorithme de contrôle MC prend en entrée les arguments suivants :

- env : L'instance d'un environnement OpenAI Gym.
- num_épisodes : Le nombre d'épisodes qui sont générés.
- alpha : Le paramètre de taille de pas pour le pas de mise à jour.
- eps_decay : Le facteur de décroissance pour la mise à jour du paramètre epsilon.
- gamma : Le taux d'actualisation.

Définir la valeur d'Epsilon

Il est important de spécifier une politique ϵ-greedy pour garantir que le contrôle MC converge vers la politique optimale $\pi*$.
Avec le code suivant qui fixe la valeur de ϵ dans chaque épisode et surveille ses évolutions avec un print, nous pouvons vérifier qu'en sélectionnant un eps_decay=0.9999965 nous pouvons obtenir la décroissance graduelle de ϵ :

```
eps_start=1.0
eps_decay=.9999965
eps_min=0.05
epsilon = eps_start
for episode in range(num_episodes):
        epsilon = max(epsilon*eps_decay, eps_min)
        if episode % 100000 == 0:
           print(“Episode {} -> epsilon={}.”.format(episode, epsilon))
```

```
Episode 0 -> epsilon=0.9999965.
Episode 100000 -> epsilon=0.7046851916863968.
Episode 200000 -> epsilon=0.4965829574224525.
Episode 300000 -> epsilon=0.3499358813150294.
Episode 400000 -> epsilon=0.24659549668667013.
Episode 500000 -> epsilon=0.17377280305645773.
Episode 600000 -> epsilon=0.12245554962615056.
Episode 700000 -> epsilon=0.08629291448656881.
Episode 800000 -> epsilon=0.060809551819579116.
Episode 900000 -> epsilon=0.05.
```

Pour commencer, nous initialisons la valeur d'epsilon à un. Ensuite, pour chaque épisode, nous diminuons légèrement la valeur d'Epsilon en la multipliant par la valeur eps_decay. Nous ne voulons pas que la valeur d'Epsilon devienne trop petite car nous voulons nous assurer constamment qu'il y a une petite quantité d'exploration tout au long du processus.

Fonction principale

```
def MC_control(env, num_episodes, alpha, eps_decay, gamma):
    eps_start=1.0
    eps_min=0.05

    nA = env.action_space.n
    Q = defaultdict(lambda: np.zeros(nA))

    epsilon = eps_start

    for episode in range(1, num_episodes+1):

        if episode % 1000 == 0: # monitor progress
            print("\rEpisode {}/{}.".format(episode, num_episodes), end="")
            sys.stdout.flush()

        epsilon = max(epsilon*eps_decay, eps_min)
        episode_generated = generate_episode_from_Q(env, Q, epsilon, nA)
        Q = update_Q(env, episode_generated, Q, alpha, gamma)
    policy = dict((state,np.argmax(actions)) for state, actions in Q.items())
    return policy, Q
```

On commence par initialiser toutes les valeurs de la table Q à zéro. Ainsi, Q est initialisé comme un dictionnaire vide de tableaux numpy dont tous les éléments sont nuls, chaque tableau contenant autant d'éléments qu'il y a d'actions dans l'environnement :

```
nA = env.action_space.n
Q = defaultdict(lambda: np.zeros(nA))
```

Nous jouons une série d'épisodes, et à chaque itération :

- nous calculons le ϵ correspondant,
- nous construisons la politique ϵ-greedy puis générons un épisode en utilisant cette politique ϵ-greedy.
- Enfin, nous mettons à jour la table Q

```
for episode in range(1, num_episodes+1):
      epsilon = max(epsilon*eps_decay, eps_min)
      episode_generated=generate_episode_from_Q(env,Q,epsilon,nA)
      Q = update_Q(env, episode_generated, Q, alpha, gamma)
```

Après avoir terminé la boucle des épisodes, la politique correspondant à la table Q finale est calculée avec le code suivant :

```
policy=dict((state,np.argmax(actions)) for state, actions in Q.items())
```

La politique indique pour chaque état l'action à prendre, ce qui correspond simplement à l'action qui a la valeur d'action maximale dans la table Q.

Génération d'épisodes à l'aide de la table Q et de la politique epsilon-greedy

La construction de la politique ϵ-greedy et la génération d'un épisode utilisant cette politique ϵ-greedy sont regroupées dans la fonction generate_episode_from_Q.

Cette fonction prend en entrée l'Environnement, l'estimation la plus récente de la table Q, la valeur courante d'epsilon et le nombre d'actions. En sortie, elle retourne un épisode.
L'agent utilisera la politique Epsilon-greedy pour sélectionner les actions. Nous avons implémenté cela en utilisant la méthode random.choice de Numpy, qui prend en entrée l'ensemble des actions possibles et les probabilités correspondant à la politique Epsilon-greedy. L'obtention des probabilités d'actions correspondant à la politique ϵ-greedy se fera à l'aide de ce code :

```
def get_probs(Q_s, epsilon, nA):
    policy_s = np.ones(nA) * epsilon / nA
    max_action = np.argmax(Q_s)
    policy_s[max_action] = 1 - epsilon + (epsilon / nA)
    return policy_s
```

Évidemment, si l'état n'est pas déjà dans la Q-table, nous choisissons aléatoirement une action en utilisant la fonction action_space.sample() de l'environnement. Le code complet de cette fonction qui génère un épisode en suivant la politique epsilon-greedy est le suivant :

```
def generate_episode_from_Q(env, Q, epsilon, nA):
    episode = []
    state = env.reset()
    while True:
        probs = get_probs(Q[state], epsilon, nA)
        action = np.random.choice(np.arange(nA), p=probs) if state in Q else env.action_space.sample()
        next_state, reward, done, info = env.step(action)
        episode.append((state, action, reward))
        state = next_state
        if done:
            break
    return episode
```

Mise à jour de la table Q

Une fois que nous avons l'épisode, il suffit de regarder chaque action-état et d'appliquer l'équation de mise à jour :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

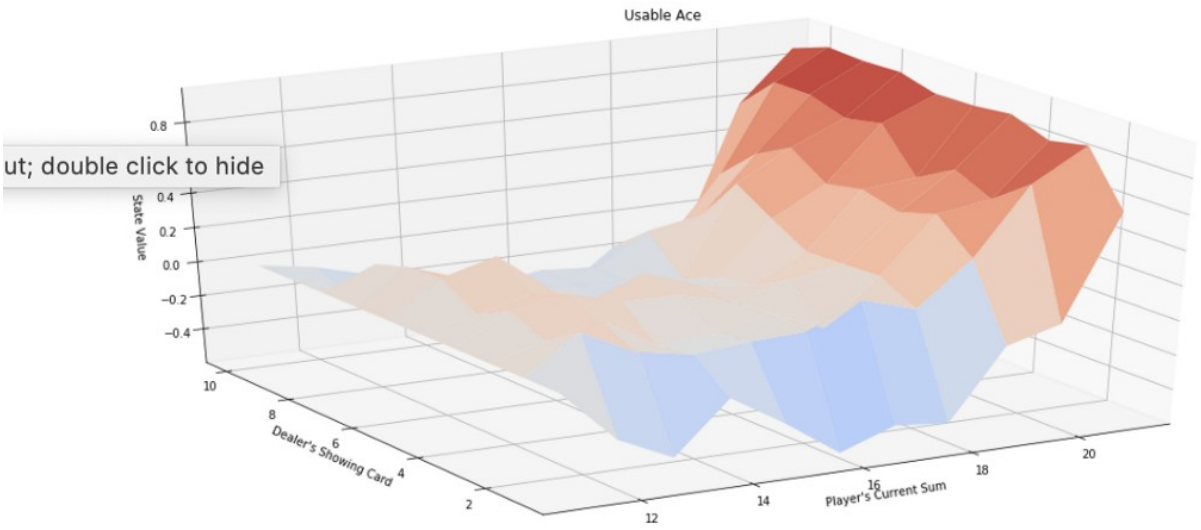
```
def update_Q(env, episode, Q, alpha, gamma):
    states, actions, rewards = zip(*episode)
    discounts=np.array([gamma**i for i in range(len(rewards)+1)])
    for i, state in enumerate(states):
        old_Q = Q[state][actions[i]]
        Q[state][actions[i]] = old_Q + alpha * (sum(rewards[i:]*discounts[:-(1+i)]) - old_Q)
    return Q
```

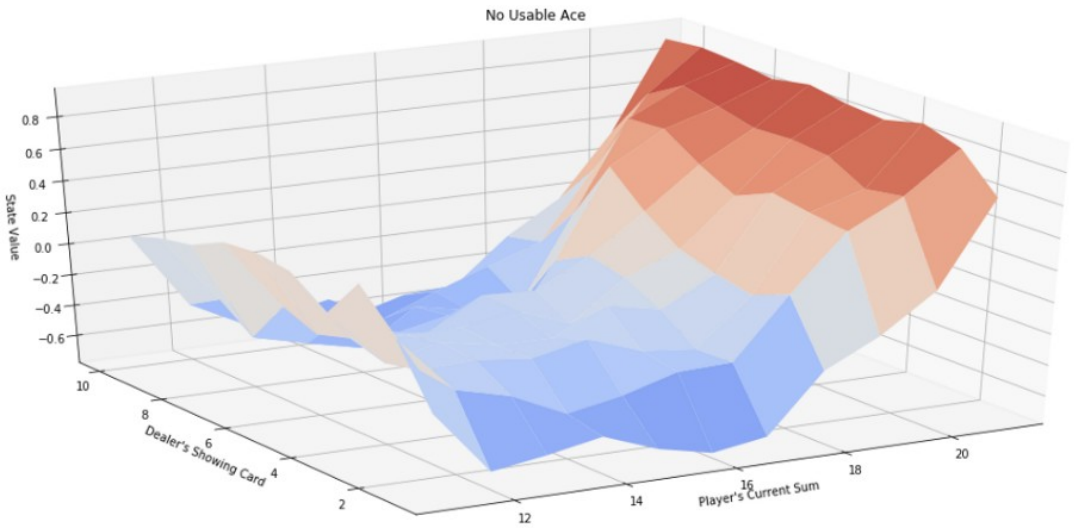
Tracer la fonction état-valeur

Nous pouvons obtenir la fonction état-valeur optimale estimée correspondante et la tracer.

Il y a deux graphiques, selon que nous ayons ou non un as utilisable.

La politique obtenue avec le Contrôle MC à α constant est meilleure puisque les valeurs des états sont beaucoup plus élevées.





Apprentissage par différence temporelle

L'un des principaux inconvénients de la méthode de Monte Carlo (MC) est le fait que l'agent doit attendre la fin d'un épisode pour obtenir le gain potentiel réel avant de pouvoir mettre à jour et améliorer l'estimation de la fonction de valeur. Cela implique que la méthode de Monte Carlo a des propriétés de convergence assez solides, car elle actualise l'estimation de la fonction de valeur en

fonction du gain potentiel réel G, qui est une estimation non biaisée de la véritable fonction de valeur.
Cependant, bien que les gains potentiels réels G soient des estimations assez précises, il s'agit également d'une estimation à variance élevée de la vraie fonction de valeur. Cela est dû au fait que les gains potentiels réels accumulent de nombreux événements aléatoires dans la même trajectoire ; toutes les actions, tous les états suivants, toutes les récompenses sont des événements aléatoires. Cela signifie que le gain potentiel réel recueille tout ce caractère aléatoire pendant plusieurs étapes temporelles. C'est pourquoi nous disons qu'à Monte Carlo, le gain potentiel réel est sans biais, mais avec une variance élevée.
De plus, en raison de la variance élevée des gains potentiels réels, l'échantillonnage à la mode Monte Carlo peut être très inefficace. Tout le caractère aléatoire devient un bruit qui ne peut être atténué que par un grand nombre de trajectoires et d'échantillons de gains potentiels réels. Une façon de réduire le problème de la variance élevée, au lieu d'utiliser le gain potentiel réel, est d'estimer un gain potentiel et d'apprendre à partir des épisodes non terminés.
Cette approche, appelée Temporal Differences (TD), a été présentée par Richard Sutton en 1988 dans un article de recherche. Cet article montrait que, alors que des méthodes telles que MC calculent les erreurs en utilisant les différences entre les gains potentiels prédits et réels, TD était capable d'utiliser la différence entre des prédictions successives dans le temps (d'où le nom d'apprentissage par différence temporelle).
Cette découverte a eu un grand impact sur l'avancement des méthodes d'apprentissage par renforcement, et l'apprentissage TD est le précurseur de techniques telles que SARSA ou DQN.
Il convient de noter que le gain potentiel estimé utilisé dans ces méthodes est une estimation biaisée car nous utilisons une estimation pour calculer une estimation. Cette façon de mettre à jour une estimation à l'aide d'une estimation est appelée bootstrapping et présente une certaine similitude avec ce que font les méthodes de programmation dynamique.
Mais bien qu'il s'agisse d'une méthode d'estimation biaisée, elle présente une variance beaucoup plus faible que la méthode de Monte Carlo. Cela s'explique par le fait que la cible de la TD ne dépend que d'une seule action, d'une seule transition et d'une seule récompense, et qu'il y a donc beaucoup moins d'aléas accumulés. Par conséquent, les méthodes TD apprennent généralement beaucoup plus rapidement que les méthodes MC.

Méthodes à différence temporelle

Comme nous l'avons déjà présenté, les méthodes par différence temporelle (TD) sont une combinaison des idées de Monte Carlo (MC) et de programmation dynamique (DP). Comme les méthodes MC, les méthodes TD peuvent apprendre directement à partir de l'expérience brute sans modèle de la dynamique de l'environnement. Cependant, comme la programmation dynamique, les méthodes TD mettent à jour les estimations en se basant en partie sur d'autres estimations apprises, sans attendre un résultat final.
Rappelez-vous que l'équation de mise à jour du contrôle de Monte Carlo après un épisode complet est la suivante :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{G_t}\ \text{-}\ \mathrm{Q}(\mathrm{S}_t, A_t))$$

Dans cette équation de mise à jour, nous recherchons l'estimation actuelle Q(St,At) dans la table Q et nous la comparons au gain potentiel Gt que nous avons réellement obtenu après avoir visité la paire état-action. Nous utilisons ce nouveau gain potentiel Gt pour rendre notre tableau Q un peu plus précis. La seule différence réelle entre les méthodes TD et les méthodes MC est que les méthodes TD mettent à jour la Q-table à chaque pas de temps au lieu d'attendre la fin de l'épisode.
Il existe trois variantes principales appelées Sarsa, Sarsamax et Expected Sarsa, qui mettent en œuvre trois versions de l'équation de mise à jour. À l'exception de cette nouvelle étape de mise à jour, tout le reste du code est identique à ce que nous avons vu dans la méthode de contrôle de Monte Carlo. Voyons chacun d'entre eux.

Sarsa

Avec cette méthode, nous commençons par initialiser toutes les valeurs d'action à zéro pour construire la politique Epsilon Greedy correspondante. Ensuite, l'agent commence à interagir avec l'environnement et reçoit le premier état S0. Ensuite, il utilise la politique pour choisir l'action A0. Immédiatement après, il reçoit une récompense R1 et l'état suivant S1. Ensuite, l'agent utilise à nouveau la même politique pour choisir l'action suivante A1. Après la séquence

$$S_0, A_0, R_1, S_1, A_1$$

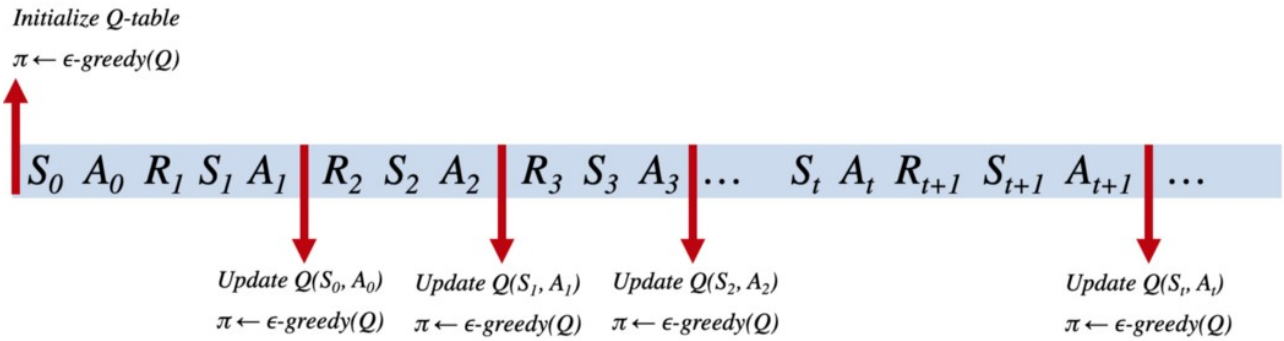
la méthode met à jour la table Q des valeurs d'action correspondant à la paire état-action précédente. Toutefois, au lieu d'utiliser le gain potentiel comme estimation alternative pour la mise à jour de la table Q, nous utilisons la somme de la récompense immédiate R1 et de la valeur actualisée de la paire action-état suivante Q(S1, A1) multipliée par le facteur gamma :

$$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha(\ \boldsymbol{R_1 + \gamma Q(S_1, A_1)}\ - Q(S_0, A_0))$$

L'équation de mise à jour exprimée de manière plus générale est la suivante :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}\ - Q(S_t, A_t))$$

Rappelez-vous qu'à l'exception de cette nouvelle étape de mise à jour, le reste du code est identique à ce que nous avons fait dans le cas du contrôle de Monte Carlo. En particulier, nous utiliserons la politique ϵ-greedy(Q) pour sélectionner les actions à chaque pas de temps.
Le schéma global suivant résume ce qui a été expliqué :

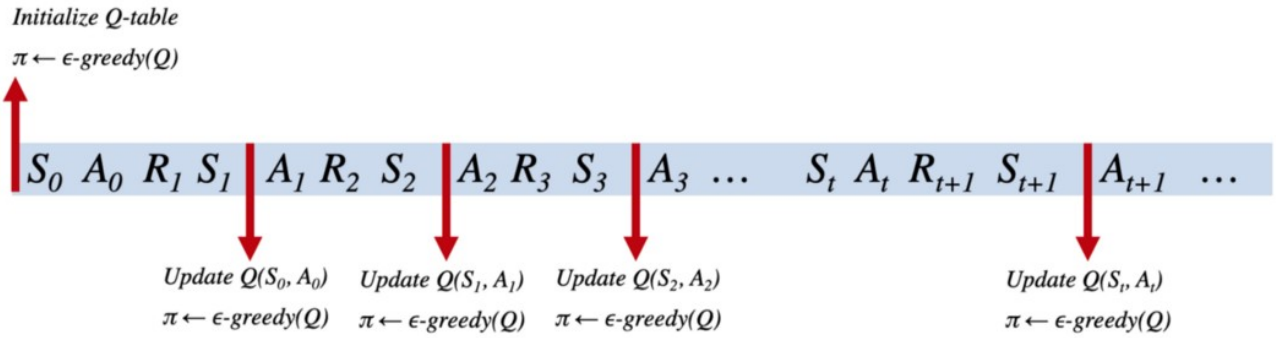


Sarsamax

Une autre méthode de contrôle TD est Sarsamax, également connue sous le nom de Q-Learning, qui fonctionne légèrement différemment de Sarsa.
Cette méthode commence avec les mêmes valeurs initiales pour les valeurs d'action et la politique de Sarsa. L'agent reçoit l'état initial S0, la première action A0 est toujours choisie dans la politique initiale. Mais ensuite, après avoir reçu la récompense R1 et l'état suivant S1, au lieu d'utiliser la même politique pour choisir l'action suivante A1 (l'action qui a été choisie en utilisant la politique Epsilon Greedy), nous allons mettre à jour la politique avant de choisir l'action suivante.
En particulier, pour l'estimation, nous allons considérer l'action en utilisant une politique Greedy, au lieu de la politique Epsilon Greedy. Cela signifie que nous utiliserons l'action qui maximise la valeur Q(s,a) pour une action donnée. Ainsi l'équation de mise à jour pour Sarsamax sera :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1} + \gamma \max_{a \in \mathcal{A}(s)} Q(S_{t+1}, a)}\ - Q(S_t, A_t))$$

où nous nous appuyons sur le fait que l'action greedy correspondant à un état est simplement celle qui maximise les valeurs d'action pour cet état.
Et donc ce qui se passe, c'est qu'après avoir mis à jour la valeur d'action pour le pas de temps zéro en utilisant l'action greedy, nous sélectionnons ensuite A1 en utilisant la politique Epsilongreedy correspondant aux valeurs d'action que nous venons de mettre à jour. Et cela continue lorsque nous avons reçu une récompense et l'état suivant. Ensuite, nous faisons la même chose que précédemment où nous mettons à jour la valeur correspondant à S1 et A1 en utilisant l'action greedy, puis nous sélectionnons A2 en utilisant la politique greedy Epsilon correspondante, et ainsi de suite. En suivant la même schématisation faite avec sarsa, on peut le résumer visuellement comme suit :



Expected Sarsa

Une dernière version de l'équation de mise à jour est la Sarsa attendue. Alors que Sarsamax prend le maximum sur toutes les actions de toutes les paires état-action suivantes possibles, Sarsa attendu utilise la valeur attendue de la paire état-action suivante, où l'attente prend en compte la probabilité que l'agent sélectionne chaque action possible de l'état suivant :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1}} + \gamma \sum_{a \in \mathcal{A}(s)} \boldsymbol{\pi(a|S_{t+1})Q(S_{t+1}, a)}\ - Q(S_t, A_t))$$

Vous trouverez ci-dessous un résumé des trois équations de mise à jour des TD (et MC) :

Monte Carlo	$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{G_t}$ - $Q(S_t, A_t))$
Sarsa	$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1}} + \gamma \boldsymbol{Q(S_{t+1}, A_{t+1})}\ - Q(S_t, A_t))$
Sarsamax	$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1}} + \gamma \max_{a \in \mathcal{A}(s)} \boldsymbol{Q(S_{t+1}, a)}\ - Q(S_t, A_t))$
Expected Sarsa	$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\ \boldsymbol{R_{t+1}} + \gamma \sum_{a \in \mathcal{A}(s)} \boldsymbol{\pi(a|S_{t+1})Q(S_{t+1}, a)}\ - Q(S_t, A_t))$

Les trois méthodes TD convergent vers la fonction action-valeur optimale q∗ (et donnent ainsi la politique optimale π∗) si la valeur de ϵ décroît conformément aux conditions de Greedy in the Limit with Infinite Exploration (GLIE), et si le paramètre de taille de pas α est suffisamment petit.

Méthodes de bootstrapping

Notez que les méthodes TD actualisent leurs estimations en se basant en partie sur d'autres estimations. Elles apprennent une estimation à partir d'une estimation, elles font du bootstrap. Dans les méthodes TD, le gain potentiel à partir d'une paire état-action est estimé alors que dans MC, nous utilisons le gain potentiel exact Gt.
Le bootstrap en apprentissage renforcé peut être interprété comme "l'utilisation de valeurs estimées dans l'étape de mise à jour pour le même type de valeur estimée".

Ni les méthodes MC ni les méthodes TD à une étape ne sont toujours les meilleures. Les méthodes de bootstrapping à n étapes unifient les méthodes MC et les méthodes TD, qui généralisent les deux méthodes de sorte que l'on puisse passer de l'une à l'autre en douceur, selon les besoins d'une tâche particulière. **Pour plus de détails, on pourra consulter le chapitre 7 du manuel Reinforcement Learning : An Introduction de Richard S. Sutton et Andrew G. Barto.**

Méthodes avec ou sans politique d'apprentissage

Les méthodes présentées peuvent être classées selon un des concepts les plus utilisés pour classer les méthodes d'apprentissage par renforcement : les méthodes avec et sans politique.

- Sarsa et Expected Sarsa sont tous deux des algorithmes de contrôle de TD à politique active. Dans ce cas, la même politique (ϵ-greedy) qui est évaluée et améliorée est également utilisée pour sélectionner les actions.

- Sarsamax est une méthode hors politique, où la politique (greedy) qui est évaluée et améliorée est différente de la politique (ϵϵ-greedy) qui est utilisée pour sélectionner les actions.

Apprentissage profond par renforcement

Nous avons présenté des méthodes de solution qui représentent les valeurs d'action dans un petit tableau. Nous avons appelé ce tableau "Q-table". L'idée est désormais d'utiliser des réseaux neuronaux pour élargir la taille des problèmes que nous pouvons résoudre avec l'apprentissage par renforcement, en présentant le Deep Q-Network (DQN), qui représente la fonction action-valeur optimale sous forme de réseau neuronal, au lieu d'un tableau.

Jeux Atari 2600

La méthode d'apprentissage Q-learning précédemment abordée résout le problème en itérant sur l'ensemble des états. Cependant, nous nous rendons souvent compte que nous avons trop d'états à suivre. Un exemple est celui des jeux Atari, qui peuvent avoir une grande variété d'écrans différents, et dans ce cas, le problème ne peut être résolu avec une table Q.
La console de jeu Atari 2600 était très populaire dans les années 1980, et de nombreux jeux de style arcade étaient disponibles pour elle. La console Atari est archaïque par rapport aux normes de jeu d'aujourd'hui, mais ses jeux constituent toujours un défi pour les ordinateurs et un point de référence très populaire dans le cadre de la recherche RL (en utilisant un émulateur).

En 2015, DeepMind a exploité l'algorithme dit Deep Q-Network (DQN) ou Deep Q-Learning qui a appris à jouer à de nombreux jeux vidéo Atari mieux que les humains. L'article de recherche qui le présente, appliqué à 49 jeux différents, a été publié dans la revue Nature :

https://hallab.cs.dal.ca/images/0/00/Minh2015.pdf

L'environnement de jeu Atari 2600 peut être reproduit en utilisant la bibliothèque OpenAI Gym, qui propose plusieurs versions de chaque jeu, notamment l'environnement Pong-v0. L'article de DeepMind dans Nature contenait un tableau avec tous les détails sur les hyperparamètres utilisés

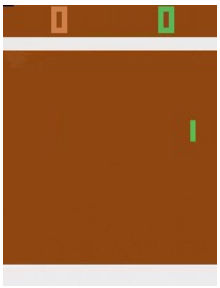
pour entraîner son modèle sur les 49 jeux Atari utilisés pour l'évaluation. Cependant, notre objectif ici est beaucoup plus modeste : nous voulons résoudre uniquement le jeu Pong.

Notebook github : https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_15_16_17_DQN_Pong.ipynb
Inspiré du code de Maxim Lapan : https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter06/02_dqn_pong.py

Nos exemples précédents pour FrozenLake, ou CartPole, n'étaient pas exigeants du point de vue des besoins de calcul, car les observations étaient petites. Cependant, à partir de maintenant, ce n'est plus le cas. La version du code partagée dans ce post converge vers un score moyen de 19.0 en 2 heures (en utilisant un NVIDIA K80).

Pong

Pong est un jeu vidéo d'arcade sur le thème du tennis de table avec des graphismes simples en deux dimensions, fabriqué par Atari et sorti initialement en 1972. Dans Pong, un joueur marque des points si la balle passe devant l'autre joueur. Un épisode est terminé lorsque l'un des joueurs atteint 21 points. Dans la version de Pong de l'OpenAI Gym framework, l'agent est affiché à droite et l'ennemi à gauche.



Un agent (joueur) peut effectuer trois actions dans l'environnement Pong : rester immobile, effectuer une translation verticale vers le haut et une translation verticale vers le bas. Cependant, si nous utilisons la méthode action_space.n, l'environnement semble disposer de 6 actions :

```
import gym
import gym.spaces
DEFAULT_ENV_NAME = “PongNoFrameskip-v4”
test_env = gym.make(DEFAULT_ENV_NAME)
print(test_env.action_space.n)
6
```

Même si l'environnement OpenAI Gym Pong a six actions, trois des six sont redondants (FIRE est égal à NOOP, LEFT est égal à LEFTFIRE et RIGHT est égal à RIGHTFIRE).

```
print(test_env.unwrapped.get_action_meanings())
[‘NOOP’, ‘FIRE’, ‘RIGHT’, ‘LEFT’, ‘RIGHTFIRE’, ‘LEFTFIRE’]
```

Réseau Neurones profond

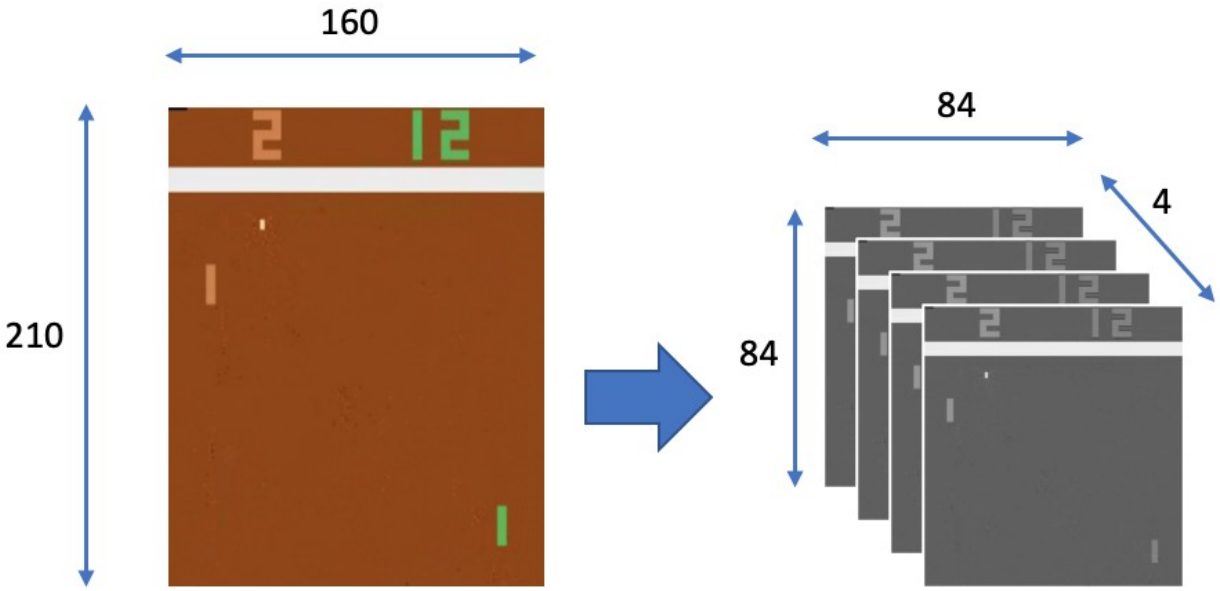
Dans cette nouvelle approche, nous trouvons au coeur de l’agent un réseau neuronal profond au lieu d'un Q-tableau. Il convient de noter que l'agent ne reçoit que des données brutes en pixels, ce qu'un joueur humain verrait à l'écran, sans accès à l'état de jeu sous-jacent, à la position de la balle, des palettes, etc.
En tant que signal de renforcement, l'agent reçoit en retour la variation du score du jeu à chaque pas de temps. Au début, lorsque le réseau neuronal est initialisé avec des valeurs aléatoires, il est vraiment mauvais, mais au fil du temps, il commence à associer les situations et les séquences du jeu avec des actions appropriées et apprend à bien jouer.

Environnement

Les jeux Atari sont affichés à une résolution de 210 par 60 pixels, avec 128 couleurs possibles pour chaque pixel :

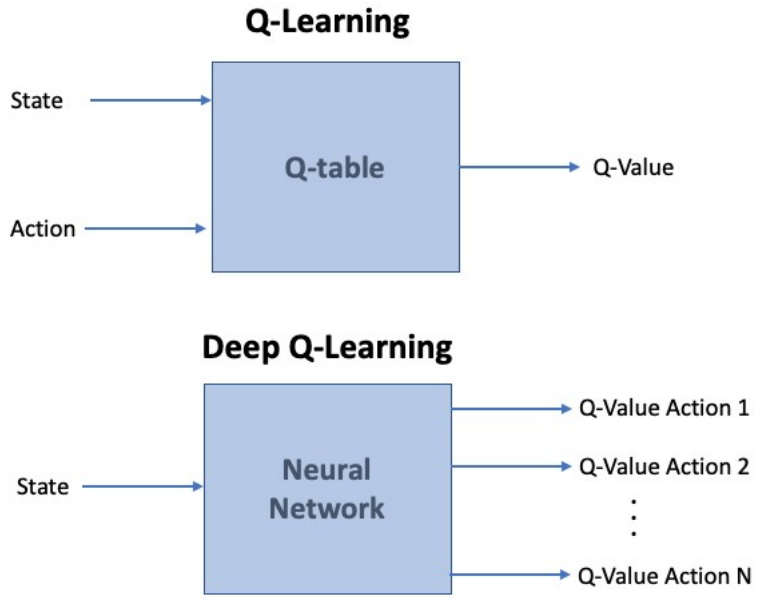
```
print(test_env.observation_space.shape)
(210, 160, 3)
```

Il s'agit toujours techniquement d'un espace d'état discret, mais il est très grand à traiter tel quel et nous pouvons l'optimiser. Pour réduire cette complexité, on effectue un traitement minimal : on convertit les images en niveaux de gris, et on les réduit à un bloc carré de 84 par 84 pixels. Maintenant, réfléchissons bien si avec cette image fixe nous pouvons déterminer la dynamique du jeu. Il y a certainement une ambiguïté dans l'observation, n'est-ce pas ? Par exemple, nous ne pouvons pas savoir dans quelle direction va la balle, ce qui viole la propriété de Markov. La solution consiste à conserver plusieurs observations du passé et à les utiliser comme un état. Dans le cas des jeux Atari, les auteurs de l'article suggèrent d'empiler 4 images successives et de les utiliser comme observation à chaque état. Pour cette raison, le prétraitement empile quatre images ensemble, ce qui donne un espace d'état final de 84 par 84 par 4 :



Sortie du réseau neurones

Contrairement à ce qui a été vu jusqu'à présent avec une seule valeur Q produite à la fois, le réseau Q profond est conçu pour produire en un seul calcul une valeur Q pour chaque action possible dans l'environnement :



Cette approche, qui consiste à calculer toutes les valeurs Q en un seul passage dans le réseau, évite de devoir exécuter le réseau individuellement pour chaque action et permet d'augmenter considérablement la vitesse. Maintenant, nous pouvons simplement utiliser ce vecteur pour entreprendre une action en choisissant celle qui a la valeur maximale.

Architecture du réseau neuronal

L'agent DQN original utilisait la même architecture de réseau neuronal, pour les 49 jeux, qui prend en entrée une image de 84x84x4.
Les images de l'écran sont d'abord traitées par trois couches convolutionnelles. Cela permet au système d'exploiter les relations spatiales et d'utiliser l'espace des règles spatiales. De plus, puisque quatre images sont empilées et fournies en entrée, ces couches convolutionnelles extraient également certaines propriétés temporelles à travers ces images. En utilisant PyTorch, nous pouvons coder la partie convolutive du modèle comme suit :

```
nn.Conv2d(input_shape, 32, kernel_size=8, stride=4),
nn.ReLU(),
nn.Conv2d(32, 64, kernel_size=4, stride=2),
nn.ReLU(),
nn.Conv2d(64, 64, kernel_size=3, stride=1),
nn.ReLU()
```

où input_shape est la forme de l'espace d'observation de l'environnement.
Les couches convolutionnelles sont suivies d'une couche cachée entièrement connectée avec activation ReLU et d'une couche de sortie linéaire entièrement connectée qui produit le vecteur des valeurs d'action :

```
nn.Linear(conv_out_size, 512),
```

```
nn.ReLU(),
nn.Linear(512, n_actions)
```

où conv_out_size est le nombre de valeurs dans la sortie de la couche de convolution.
Le constructeur de la première couche entièrement connectée a besoin de cette valeur qui pourrait être codée en dur car elle ne dépend que de la forme de l'entrée (pour une entrée de 84x84, la sortie de la couche de convolution sera de taille 3136). Cependant, afin de coder un modèle générique (pour tous les jeux) qui peut accepter différentes formes d'entrée, nous utiliserons une fonction simple, get_conv_out qui accepte la forme de l'entrée et applique la couche de convolution à un faux tenseur de cette forme :

```
def get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

conv_out_size = get_conv_out(input_shape)
```

Un autre problème à résoudre est de fournir la sortie de la convolution à la couche entièrement connectée. Mais PyTorch ne dispose pas d'une couche "plus plate" et nous devons remodeler le lot de tenseurs 3D en un lot de vecteurs 1D. On résout ce problème avec la méthode forward(), qui fait appel à la fonction view() des tenseurs.

La fonction view() "remodèle" un tenseur avec les mêmes données et le même nombre d'éléments en entrée, mais avec la forme spécifiée. L'aspect intéressant de cette fonction est qu’on ne précise pas la valeur d’une dimension (une seule au maximum bien entendu) et qu’on indique -1 à la place, elle est déduite des autres dimensions et du nombre d'éléments en entrée.

Par exemple, si nous avons un tenseur de forme (2, 3, 4, 6), qui est un tenseur 4D de 144 éléments, nous pouvons le remodeler en un tenseur 2D avec 2 lignes et 72 colonnes en utilisant view(2,72). On obtient le même résultat avec view(2,-1), puisque 144/ (3*4*6) = 2.

forward() aplatit le tenseur 4D de la partie convolutionnelle (la première dimension est la taille du batch, la deuxième est le canal de couleur, la troisième et la quatrième sont les dimensions de l'image) en un tenseur 2D qui servira d’entrée à nos couches entièrement connectées pour obtenir des valeurs Q pour chaque entrée du batch.

```
import torch
import torch.nn as nn
import numpy as np

class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
```

```
            nn.ReLU()
        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        return self.fc(conv_out)
```

```
DQN(
  (conv): Sequential(
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=6, bias=True)
  )
)
```

Adaptation de l’environnement

Dans l'article de DeepMind, plusieurs transformations (comme la conversion des images en niveaux de gris et leur réduction à un bloc carré de 84 par 84 pixels) sont appliquées à l'interaction avec la plateforme Atari afin d'améliorer la vitesse et la convergence de la méthode. Une technique courante pour opérer ces transformations est de surcharger le wrapper de base de Gym afin d'ajouter des fonctionnalités à l’environnement.

Le code suivant correspond au wrapper FireResetEnv qui appuie sur le bouton FIRE dans les environnements qui l'exigent au démarrage du jeu :

```
class FireResetEnv(gym.Wrapper):
  def __init__(self, env=None):
      super(FireResetEnv, self).__init__(env)
      assert env.unwrapped.get_action_meanings()[1] == ‘FIRE’
      assert len(env.unwrapped.get_action_meanings()) >= 3
```

```
def step(self, action):
      return self.env.step(action)

def reset(self):
      self.env.reset()
      obs, _, done, _ = self.env.step(1)
      if done:
        self.env.reset()
      obs, _, done, _ = self.env.step(2)
      if done:
        self.env.reset()
      return obs
```

Le prochain wrapper dont nous aurons besoin est MaxAndSkipEnv :

```
class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env=None, skip=4):
        super(MaxAndSkipEnv, self).__init__(env)
        self._obs_buffer = collections.deque(maxlen=2)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = None
        for _ in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            self._obs_buffer.append(obs)
            total_reward += reward
            if done:
                break
        max_frame = np.max(np.stack(self._obs_buffer), axis=0)
        return max_frame, total_reward, done, info

    def reset(self):
        self._obs_buffer.clear()
        obs = self.env.reset()
        self._obs_buffer.append(obs)
        return obs
```

D'une part, il permet d'accélérer considérablement l'apprentissage en appliquant la fonction max à N observations (quatre par défaut) et en retournant le résultat comme observation pour l'étape. En effet, sur les images intermédiaires, l'action choisie est simplement répétée et nous pouvons prendre une décision d'action tous les N pas, car traiter chaque image avec un réseau neuronal est une opération assez exigeante, mais la différence entre deux images consécutives est généralement mineure.

D'autre part, il prend le maximum de chaque pixel dans les deux dernières images pour s’en servir comme observation.
https://docs.python.org/fr/3/library/collections.html#collections.deque

Certains jeux Atari ont un effet de scintillement (lorsque le jeu dessine des parties différentes de l'écran sur les images paires et impaires, une pratique normale chez les développeurs de l'Atari 2600 pour augmenter la complexité des sprites du jeu), qui est dû à la limitation de la plate-forme. Pour l'œil humain, ces changements rapides ne sont pas visibles, mais ils peuvent perturber un réseau neuronal.

Rappelez-vous que nous avons déjà mentionné qu'avant d'alimenter le réseau neuronal avec les images, chaque image est réduite de 210x160, avec trois cadres de couleur (canaux de couleur RVB), à une image monochrome de 84x84 en utilisant une conversion colorimétrique en niveaux de gris. Différentes approches sont possibles. L'une d'entre elles consiste à recadrer les parties non pertinentes de l'image, puis à en réduire l'échelle, comme le fait le code suivant :

```
class ProcessFrame84(gym.ObservationWrapper):
    def __init__(self, env=None):
        super(ProcessFrame84, self).__init__(env)
        self.observation_space = gym.spaces.Box(low=0, high=255,
                            shape=(84, 84, 1), dtype=np.uint8)

    def observation(self, obs):
        return ProcessFrame84.process(obs)

    @staticmethod
    def process(frame)
        if frame.size == 210 * 160 * 3:
            img = np.reshape(frame, [210, 160,  3]).astype(np.float32)
        elif frame.size == 250 * 160 * 3:
            img = np.reshape(frame, [250, 160, 3]).astype(np.float32)
        else:
            assert False, “Unknown resolution.”
            img = img[:, :, 0] * 0.299 + img[:, :, 1] * 0.587 + img[:, :, 2] * 0.114
            resized_screen = cv2.resize(img, (84, 110), interpolation=cv2.INTER_AREA)
            x_t = resized_screen[18:102, :]
            x_t = np.reshape(x_t, [84, 84, 1])
            return x_t.astype(np.uint8)
```

Comme nous l'avons déjà évoqué comme solution rapide au manque de dynamique de jeu, la classe BufferWrapper empile plusieurs (généralement quatre) images successives :

```
class BufferWrapper(gym.ObservationWrapper):
    def __init__(self, env, n_steps, dtype=np.float32):
        super(BufferWrapper, self).__init__(env)
        self.dtype = dtype
        old_space = env.observation_space
        self.observation_space =
                gym.spaces.Box(old_space.low.repeat(n_steps,
                axis=0),old_space.high.repeat(n_steps, axis=0),
                dtype=dtype)

    def reset(self):
        self.buffer = np.zeros_like(self.observation_space.low,
        dtype=self.dtype)
```

```
        return self.observation(self.env.reset())

    def observation(self, observation):
        self.buffer[:-1] = self.buffer[1:]
        self.buffer[-1] = observation
        return self.buffer
```

La forme d'entrée du tenseur a un canal de couleur comme dernière dimension, mais les couches de convolution de PyTorch supposent que le canal de couleur est la première dimension. Ce simple wrapper change la forme de l'observation de HWC (hauteur, largeur, canal) au format CHW (canal, hauteur, largeur) requis par PyTorch :

```
class ImageToPyTorch(gym.ObservationWrapper):
    def __init__(self, env):
        super(ImageToPyTorch, self).__init__(env)
        old_shape = self.observation_space.shape
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0,
                                shape=(old_shape[-1],
                                old_shape[0], old_shape[1]),
                                dtype=np.float32)

    def observation(self, observation):
        return np.moveaxis(observation, 2, 0)
```

L'écran obtenu à partir de l'émulateur est codé comme un tenseur d'octets avec des valeurs de 0 à 255, ce qui n'est pas la meilleure représentation pour un NN. Nous devons donc convertir l'image en flottants et remettre à l'échelle les valeurs dans la plage [0,0...1,0]. Ceci est fait par le wrapper ScaledFloatFrame

```
class ScaledFloatFrame(gym.ObservationWrapper):
    def observation(self, obs):
        return np.array(obs).astype(np.float32) / 255.0
```

Enfin, on crée la fonction simple suivante make_env qui construit l’environnement et lui applique tous les wrappers nécessaires :

```
def make_env(env_name):
    env = gym.make(env_name)
    env = MaxAndSkipEnv(env)
    env = FireResetEnv(env)
    env = ProcessFrame84(env)
    env = ImageToPyTorch(env)
    env = BufferWrapper(env, 4)
    return ScaledFloatFrame(env)
```

Les défis de l'apprentissage profond par renforcement

Malheureusement, l'apprentissage par renforcement est plus instable lorsque des réseaux de neurones sont utilisés pour représenter les valeurs des actions. L'entraînement d'un tel réseau nécessite beaucoup de données, mais même dans ce cas, il n'est pas garanti qu'il converge vers la fonction de valeur optimale. En fait, il existe des situations où les poids du réseau peuvent osciller ou diverger, en raison de la forte corrélation entre les actions et les états.
Afin de résoudre ce problème, nous présentons dans cette section deux techniques utilisées par le Deep Q-Network :

- experience replay
- target network

Il existe de nombreux autres trucs et astuces découverts par les chercheurs pour rendre la formation du DQN plus stable et efficace.

Experience replay

Nous essayons d'approximer une fonction complexe et non linéaire, Q(s, a), avec un réseau neuronal. Pour ce faire, nous devons calculer les cibles à l'aide de l'équation de Bellman, puis considérer que nous sommes face à un problème d'apprentissage supervisé. Cependant, l'une des exigences fondamentales de l'optimisation SGD est que les données d'apprentissage soient indépendantes et identiquement distribuées et que, lorsque l'agent interagit avec l'environnement, la séquence de tuples d'expérience puisse être fortement corrélée. L'algorithme naïf d'apprentissage Q qui apprend de chacun de ces tuples d'expérience dans un ordre séquentiel court le risque d'être influencé par les effets de cette corrélation.

Nous pouvons empêcher les valeurs d'action d'osciller ou de diverger de manière catastrophique en utilisant une grande mémoire tampon de notre expérience passée et en échantillonnant les données d'apprentissage à partir de celle-ci, au lieu d'utiliser notre dernière expérience. Cette technique est appelée expérience buffer. Le buffer contient une collection de tuples d'expérience (S, A, R, S′). Les tuples sont progressivement ajoutés au buffer au fur et à mesure que nous interagissons avec l'environnement. L'implémentation la plus simple est un tampon de taille fixe, avec de nouvelles données ajoutées à la fin du buffer de sorte qu'il pousse l'expérience la plus ancienne hors de celui-ci. Le fait d'échantillonner un petit lot de tuples à partir du buffer afin d'apprendre est connu sous le nom de d'expérience replay. En plus de briser les corrélations nuisibles, cela permet d'apprendre davantage de tuples individuels à plusieurs reprises, de nous souvenir des occurrences rares et, en général, de faire un meilleur usage de notre expérience.

Pour résumer, l'idée de base de l’expérience replay est de stocker les expériences passées et d'utiliser un sous-ensemble aléatoire de ces expériences pour mettre à jour le réseau Q, plutôt que de n'utiliser que l'expérience la plus récente. Afin de stocker les expériences de l'agent, nous avons utilisé une structure de données appelée "deque" dans la bibliothèque de collections intégrée de Python. Il s'agit essentiellement d'une liste à laquelle vous pouvez fixer une taille maximale. Ainsi, si vous essayez d'ajouter un élément à la liste et qu'elle est déjà pleine, le premier élément de la liste est supprimé et le nouvel élément est ajouté à la fin de la liste. Les expériences elles-mêmes sont des tuples de [observation, action, récompense, drapeau de réalisation, état suivant] pour conserver les transitions obtenues de l'environnement.

```
Experience = collections.namedtuple(‘Experience’,
        field_names=[‘state’, ‘action’, ‘reward’, ‘done’, ‘new_state’])
```

```
class ExperienceReplay:
  def __init__(self, capacity):
     self.buffer = collections.deque(maxlen=capacity)

  def __len__(self):
     return len(self.buffer)

  def append(self, experience):
     self.buffer.append(experience)

  def sample(self, batch_size):
     indices = np.random.choice(len(self.buffer), batch_size, replace=False)

     states, actions, rewards, dones, next_states = zip([self.buffer[idx] for idx in indices])

     return np.array(states), np.array(actions),
          np.array(rewards,dtype=np.float32),
          np.array(dones, dtype=np.uint8),
          np.array(next_states)
```

Chaque fois que l'agent effectue une étape dans l'environnement, il pousse la transition dans le buffer, ne conservant qu'un nombre fixe d'étapes (10000 transitions dans notre cas). Pour l'entraînement, nous échantillonnons aléatoirement le lot de transitions du buffer, ce qui nous permet de rompre la corrélation entre les étapes suivantes dans l'environnement.
La plupart du code de l'expérience buffer est assez simple : il exploite essentiellement la bibliothèque deque. Dans la méthode sample(), nous créons une liste d'indices aléatoires, puis nous reconditionnons les entrées échantillonnées dans des tableaux NumPy pour faciliter les calculs.

Target network

Dans le Q-Learning, nous mettons à jour une supposition avec une supposition, et cela peut potentiellement conduire à des corrélations néfastes. L'équation de Bellman nous fournit la valeur de Q(s, a) via Q(s', a') . Cependant, les états s et s' ne sont séparés que par un seul pas. Ils sont donc très similaires et il est très difficile pour un réseau neuronal de les distinguer.
La mise à jour des paramètres du réseau neuronal pour rapprocher Q(s, a) du résultat souhaité peut indirectement modifier la valeur produite pour Q(s', a') et les autres états proches. Cela peut rendre notre apprentissage très instable.
Pour rendre l’apprentissage plus stable, il existe une astuce, appelée target network, par laquelle nous conservons une copie de notre réseau neuronal et l'utilisons pour la valeur Q(s', a') dans l'équation de Bellman.
En d'autres termes, les valeurs Q prédites de ce deuxième réseau Q, appelé target network, sont utilisées pour la rétropropagation et l'entraînement du réseau Q principal. Il est important de souligner que le target network ne subit pas lui-même d’apprentissage, mais ses paramètres sont périodiquement synchronisés avec les paramètres du réseau Q principal. L'idée est d’améliorer la stabilité de l’ apprentissage en utilisant des valeurs Q du target network pour former le Q-réseau principal.

Algorithme de Q-Learning profond

Il y a deux phases principales qui s'entrecroisent dans l'algorithme de Deep Q-Learning. La première consiste à échantillonner l'environnement en effectuant des actions et à stocker les tuples avec les expériences observées dans un buffer. L'autre est celle où nous sélectionnons le petit lot de

tuples de cette mémoire, de manière aléatoire, et apprenons à partir de ce lot en utilisant une étape de mise à jour par descente de gradient (SGD).
Ces deux phases ne sont pas directement dépendantes l'une de l'autre et nous pourrions effectuer plusieurs étapes d'échantillonnage puis une étape d'apprentissage, ou même plusieurs étapes d'apprentissage avec différents lots aléatoires. En pratique, vous ne pourrez pas exécuter l'étape d'apprentissage immédiatement. Vous devrez attendre d'avoir suffisamment de tuples d'expériences dans D.
L'algorithme DQN de base est le suivant :

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
  `/* Sample phase`
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from state s using policy ϵ-greedy(Q)
  Agent takes action a, observe reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if *enough experiences in* D **then**
    `/* Learn phase`
    Sample a random *minibatch* of N transitions from D
    for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      end
    end
    Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
    Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every C steps, copy weights from Q to $\hat{Q}$
  end
end

Au début, nous devons :

- créer le réseau principal et le target network,
- initialiser une mémoire vide D
- initialiser l'agent pour gérer l'interaction avec l'environnement.

La mémoire est finie, et on utilisera pour la représenter une file d'attente circulaire qui retient les d tuples d'expérience les plus récents. Nous n'effaçons pas la mémoire après chaque épisode, ce qui nous permet de rappeler et de construire des lots d'expériences à partir de plusieurs épisodes.

Codage de la boucle d'apprentissage

Hyperparamètres et temps d'exécution

L'article de DeepMind dans Nature contenait un tableau avec tous les détails sur les hyperparamètres utilisés pour entraîner son modèle sur 49 jeux Atari. DeepMind a gardé tous ces paramètres identiques pour tous les jeux, mais a entraîné des modèles individuels pour chaque jeu. L'intention de l'équipe était de montrer que la méthode est suffisamment robuste pour résoudre de

nombreux jeux dont la complexité, l'espace d'action, la structure de récompense et d'autres détails varient, en utilisant une seule architecture de modèle et les mêmes hyperparamètres.
Cependant, notre objectif dans ce post est de résoudre uniquement le jeu Pong, un jeu assez simple. Nous avons donc décidé d'utiliser des valeurs de paramètres plus personnalisées pour notre environnement Pong qui converge vers un score moyen de 19,0 en un temps raisonnable (environ deux heures au maximum, fonction du type de GPU).
Rappelez-vous que nous pouvons connaître le type de GPU qui a été assigné à notre environnement d'exécution avec la commande !nvidia-smi.

```
DEFAULT_ENV_NAME = “PongNoFrameskip-v4”
MEAN_REWARD_BOUND = 19.0
gamma = 0.99
batch_size = 32
replay_size = 10000
learning_rate = 1e-4
sync_target_frames = 1000
replay_start_size = 10000
eps_start=1.0
eps_decay=.999985
eps_min=0.02
```

DEFAULT_ENV_NAME identifie l'environnement sur lequel s'entraîner
MEAN_REWARD_BOUND est la limite de récompense pour arrêter l'entraînement. Nous considérerons que le jeu a convergé lorsque notre agent atteindra une moyenne de 19 parties gagnées (sur 21) au cours des 100 dernières parties.

Ensuite :

- gamma est le facteur d'actualisation
- batch_size est la taille du minibatch
- learning_rate est le taux d'apprentissage
- replay_size est la taille du tampon de relecture (nombre maximum d'expériences stockées dans la mémoire de relecture)
- sync_target_frames indique la fréquence à laquelle nous synchronisons les poids du modèle du réseau DQN principal au réseau DQN cible (combien d'images entre les synchronisations).
- replay_start_size le nombre d'images (expériences) à ajouter au buffer avant de commencer l'entraînement.

Agent

A chaque étape, l'agent :

- choisit une action au hasard en phase d’exploration et utilise progressivement de plus en plus la fonction action valeur Q représentée par le réseau neurones
- interagit avec l'environnement
- enregistre le résultat de l'interaction avec l'environnement dans la mémoire

Choose an action a from state s using policy ϵ-greedy(Q)
Agent takes action a, observe reward r, and next state s'
Store transition $(s, a, r, s', done)$ in the experience replay memory D

A l'initialisation de l'agent, nous devons créer l'environnement et la mémoire tampon D.

```
class Agent:
   def __init__(self, env, exp_buffer):
      self.env = env
      self.exp_buffer = exp_buffer
      self._reset()

   def _reset(self):
      self.state = env.reset()
      self.total_reward = 0.0

   def play_step(self, net, epsilon=0.0, device="cpu"):
      done_reward = None

      if np.random.random() < epsilon:
         action = env.action_space.sample()
      else:
         state_a = np.array([self.state], copy=False)
         state_v = torch.tensor(state_a).to(device)
         q_vals_v = net(state_v)
         _, act_v = torch.max(q_vals_v, dim=1)
        action = int(act_v.item())

      new_state, reward, is_done, _ = self.env.step(action)
      self.total_reward += reward

      exp = Experience(self.state,action,reward,is_done,new_state)
      self.exp_buffer.append(exp)
      self.state = new_state
      if is_done:
         done_reward = self.total_reward
         self._reset()
      return done_reward
```

La méthode play_step utilise une politique ϵ-greedy(Q) pour sélectionner les actions à chaque pas de temps :

- si nous tirons un nombre aléatoire inférieur à epsilon (passée en argument), nous prenons l'action aléatoire
- sinon, nous utilisons l'agent nommé net (aussi passé en argument) pour obtenir les valeurs Q de toutes les actions possibles et choisir la meilleure.

Après avoir obtenu l'action, la méthode exécute l'étape de l'environnement pour obtenir l'observation suivante : next_state, reward et is_done.

Enfin, la méthode stocke l'observation dans la mémoire puis évalue si nous avons atteint la fin de l'épisode.

Le résultat de la fonction est soit None soit la récompense totale accumulée si nous avons atteint la fin de l'épisode avec cette étape.

Boucle principale

Nous créons

- notre environnement,
- le réseau neuronal DQN principal que nous allons entraîner,
- notre target network avec la même architecture.
- la mémoire de la taille requise
- l'agent.

Les dernières choses que nous faisons avant de passer à la boucle d'entraînement sont :

- la création d'un optimiseur,
- d'une liste pour les récompenses d'épisodes complets,
- d'un compteur d’étapes
- d'une variable pour suivre la meilleure récompense moyenne atteinte (car chaque fois que la récompense moyenne bat le record, nous sauvegardons le modèle dans un fichier)

```
env = make_env(DEFAULT_ENV_NAME)

net = DQN(env.observation_space.shape, env.action_space.n).to(device)
target_net = DQN(env.observation_space.shape, env.action_space.n).to(device)
buffer = ExperienceReplay(replay_size)
agent = Agent(env, buffer)
epsilon = eps_start
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
total_rewards = []
frame_idx = 0
best_mean_reward = None
```

Au début de la boucle d'apprentissage, nous incrémentons le compteur d'étapes et mettons à jour epsilon. Ensuite, l'agent effectue une étape dans l'environnement.
play_step ne renvoie un résultat différent de None que si cette étape est la dernière étape de l'épisode. Dans ce cas, nous signalons la progression dans la console :

- nombre d’étapes jouées,
- récompense moyenne pour les 100 derniers épisodes,
- valeur actuelle d'epsilon

```
while True:
    frame_idx += 1
    epsilon = max(epsilon*eps_decay, eps_min)
    reward = agent.play_step(net, epsilon, device=device)

    if reward is not None:
        total_rewards.append(reward)
        mean_reward = np.mean(total_rewards[-100:])
        print(“%d: %d games, mean reward %.3f” % (frame_idx, len(total_rewards), mean_reward))
        print(“epsilon %.2f%” % epsilon)
        if best_mean_reward is None or best_mean_reward < mean_reward:
            torch.save(net.state_dict(), DEFAULT_ENV_NAME + “-best.dat”)
            best_mean_reward = mean_reward
            if best_mean_reward is not None:
                print(“Best mean reward updated %.3f” % (best_mean_reward))

        if mean_reward > MEAN_REWARD_BOUND:
            print(“Solved in %d frames!” % frame_idx)
            break
```

```
if len(buffer) < replay_start_size:
    continue

batch = buffer.sample(batch_size)
states, actions, rewards, dones, next_states = batch

states_v = torch.tensor(states).to(device)
next_states_v = torch.tensor(next_states).to(device)
actions_v = torch.tensor(actions).to(device)
rewards_v = torch.tensor(rewards).to(device)
done_mask = torch.BoolTensor(dones).to(device)

state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
next_state_values = target_net(next_states_v).max(1)[0]
next_state_values[done_mask] = 0.0
next_state_values = next_state_values.detach()

expected_state_action_values=next_state_values * gamma + rewards_v
loss_t = nn.MSELoss()(state_action_values, expected_state_action_values)

optimizer.zero_grad()
loss_t.backward()
optimizer.step()

if frame_idx % sync_target_frames == 0:
    target_net.load_state_dict(net.state_dict())
```

Chaque fois que la récompense moyenne des 100 derniers épisodes atteint un maximum, nous le signalons dans la console et enregistrons les paramètres actuels du modèle dans un fichier.
Si cette récompense moyenne dépasse MEAN_REWARD_BOUND (19,0 dans notre cas), nous arrêtons l'entraînement.

Si la taille du buffer est inférieure à une certaine taille (içi 10000 ce qui correspond à un buffer totalement rempli mais on pourrait commencer avec un buffer moins rempli), l'algorithme saute la partie entraînement et on continue de nourrir la mémoire :

```
if len(buffer) < replay_start_size:
    continue
```

Phase d'apprentissage

L'apprentissage est la partie du code située en dessous de l'instruction continue, et qui suit le pseudocode suivant :

Sample a random *minibatch* of N transitions from D
for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
  if $done_i$ **then**
    $y_i = r_i$
  else
    $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
  end
end
Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
Every C steps, copy weights from Q to $\hat{Q}$

La première tache réalisée est l'échantillonnage d'un mini-batch aléatoire de transactions à partir de la mémoire.

On transforme alors les tableaux NumPy individuels issus de la mémoire en tenseurs PyTorch et on les copie sur le GPU (nous partons du principe que le périphérique CUDA est spécifié en argument).
Ce code est inspiré de celui de Maxim Lapan :
https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter06/02_dqn_pong.py
https://github.com/Shmuma/Deep-Reinforcement-Learning-Hands-On

Il est écrit de manière à exploiter au maximum les capacités du GPU en traitant (en parallèle) tous les échantillons par lots avec des opérations vectorielles.

L'instruction suivante transmet les observations à l'agent et calcule les valeurs Q spécifiques pour toutes les actions, pour ne retenir que celles correspondant aux actions réellement prises, en utilisant plusieurs fonctions de manipulation des tenseurs, gather, unsqueeze et squeeze :

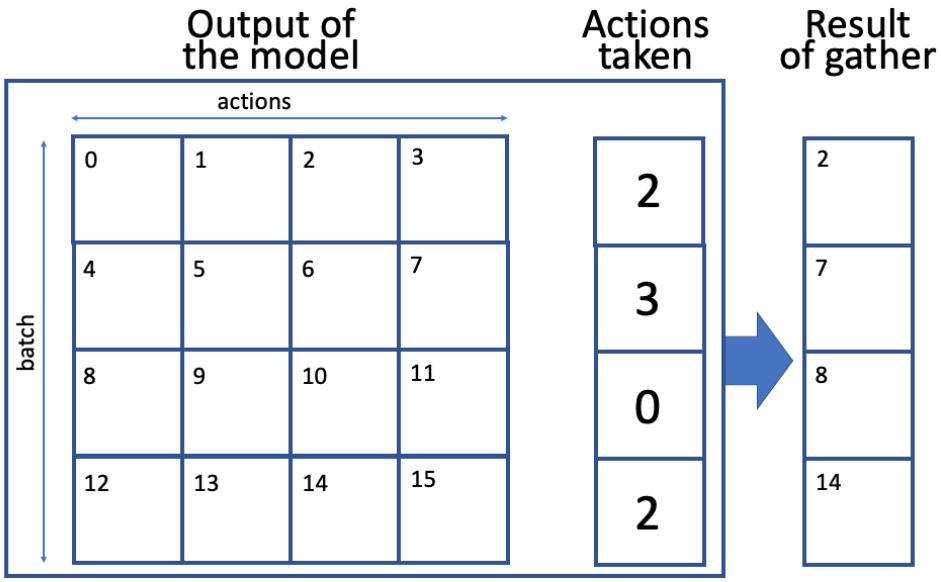
```
state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
```

Pour rappel :

- actions_v est un vecteur, un tenseur 1D contenant les valeurs des actions pour chaque échantillon du batch.
- net(states_v) est de son côté une matrice, un tenseur 2D
  1. axe 0 : numéros des échantillons
  2. axe 1 : numéros des actions

net(states_v)[0;1] est donc la valeur de l'action 1 pour l'échantillon 0

net(states_v) étant un tenseur 2D, il faut donner à la commande gather une matrice 2D d'indices pour pouvoir sélectionner au sein de net(states_v) les valeurs des actions exécutées. On transforme donc actions_v en un tenseur 2D avec la commande unsqueeze(-1) qui ajoute une dimension à la fin du tenseur, pour que la dimension 0 soit bien celle des échantillons.
La commande gather est exécutée sur l'axe des actions, à savoir 1.
Enfin, pour supprimer la dimension supplémentaire que nous avons créée, on applique la fonction squeeze(-1)



Notez que le résultat de gather() sur un tenseur est une opération qui conserve tous les gradients pour pouvoir estimer l'erreur finale.

Maintenant que nous avons calculé les valeurs des actions pour chaque état du batch, nous calculons les actions pour l'ensemble des états suivants. Pour ce faire, rappelez-vous que nous devons utiliser le target network.

Le calcul de la valeur Q maximale se fait le long de l'axe des actions, à savoir 1, et la fonction max renvoyant max and argmax (indice pour lequel le max est atteint) nous ne nous intéressons qu'à l'élément 0 (la première entrée du résultat):

```
next_state_values = target_net(next_states_v).max(1)[0]
```

Si un de nos états suivants correspond à une étape finale de l'épisode, alors notre valeur de l'action n'a pas à avoir de récompense, car il n'y a pas d'état suivant à partir duquel recueillir la récompense. Pour réaliser cette opération en une ligne, on utilise le masque produit par l'instruction done_mask = torch.BoolTensor(dones).to(device)

```
next_state_values[done_mask] = 0.0
```

Bien que nous ne puissions pas entrer dans les détails, il est important de souligner que le calcul de la valeur de l'état suivant par le target network ne doit pas affecter les gradients. Pour ce faire, nous utilisons la fonction detach(), qui en fait une copie déconnectée de l'opération parente, afin d'empêcher les gradients de se propager dans le graphe du target network

En utilisant l'équation de Bellman, nous pouvons désormais calculer le vecteur des valeurs attendues d'état-action pour chaque transition du batch :

```
expected_state_action_values=next_state_values * gamma + rewards_v
```

Nous avons toutes les informations nécessaires pour calculer l'erreur avec la méthode des moindres carrés :

```
loss_t = nn.MSELoss()(state_action_values, expected_state_action_values)
```

La séquence suivante de la boucle met à jour le réseau neuronal principal à l'aide de l'algorithme SGD en minimisant le loss :

```
optimizer.zero_grad()
loss_t.backward()
optimizer.step()
```

Enfin, la dernière ligne du code synchronise les paramètres de notre réseau DQN principal avec le target network à chaque sync_target_frames :

```
if frame_idx % sync_target_frames == 0:
    target_net.load_state_dict(net.state_dict())
```

Double Q reinforcement learning

L'apprentissage Q profond (DQN) a un défaut : il peut être instable en raison d'estimations biaisées des récompenses futures, ce qui ralentit l'apprentissage. L'apprentissage double Q ou de target network peut résoudre ce problème de biais et produire de meilleurs résultats en matière d'apprentissage Q.

L'objectif du réseau neuronal dans l'apprentissage Q profond est d'apprendre la fonction

$$Q(st, at ; \theta t)$$

À un moment donné du jeu/épisode, l'agent se trouve dans un état st. Cet état est introduit dans le réseau, et diverses valeurs Q seront renvoyées pour chacune des actions possibles at de l'état st. Les θt font référence aux paramètres du réseau neuronal (c'est-à-dire à toutes les valeurs de poids et de biais).

L'agent choisit une action en fonction d'une politique epsilon-greedy. Cette politique est une combinaison d'actions choisies au hasard et de la sortie du réseau neuronal Q profond - la probabilité d'une action choisie au hasard diminuant au cours du temps d'apprentissage. Lorsque le réseau Q profond est utilisé pour sélectionner une action, il le fait en prenant la valeur Q maximale retournée sur toutes les actions, pour l'état s. Par exemple, si un agent est dans l'état 1, et que cet état comporte 4 actions possibles que l'agent peut effectuer, il produira 4 valeurs Q. L'action qui a la valeur Q la plus élevée sera choisie parmi toutes les actions. L'action qui a la valeur Q la plus élevée est l'action qui sera sélectionnée. Ceci peut être exprimé comme suit :

$$a = argmax Q(s_t, a; \theta_t)$$

Où l'argmax est effectué sur toutes les actions / nœuds de sortie du réseau neuronal. C'est ainsi que les actions sont choisies dans l'apprentissage Q profond. l’apprentissage est guidé en utilisant l'équation de Bellman / Q-learning :

$$Q_{target} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$$

Pour une action donnée a de l'état st, nous voulons entraîner le réseau à prédire ce qui suit :

- la récompense immédiate pour cette action rt+1, plus
- la récompense actualisée pour la meilleure action possible dans l'état suivant st+1.

Si nous réussissons à entraîner le réseau à prédire ces valeurs, l'agent choisira systématiquement l'action qui donne la meilleure récompense immédiate rt+1 plus les récompenses futures actualisées des états ultérieurs. gamma est le terme d'actualisation, qui accorde moins de valeur aux récompenses futures qu'aux récompenses actuelles (mais généralement de façon marginale).

Dans l'apprentissage Q profond, le jeu est joué de manière répétée et les états, actions et récompenses sont stockés en mémoire sous la forme d'une liste de tuples ou d'un tableau

$$(st, a, rt, st+1).$$

Ensuite, pour chaque étape d’apprentissage, un lot aléatoire de ces tuples est extrait de la mémoire et le Qtarget(st, at) est calculé et comparé à la valeur produite par le réseau actuel Q(st, at). La différence quadratique moyenne entre ces deux valeurs est utilisée comme erreur.

Le problème du deep Q learning

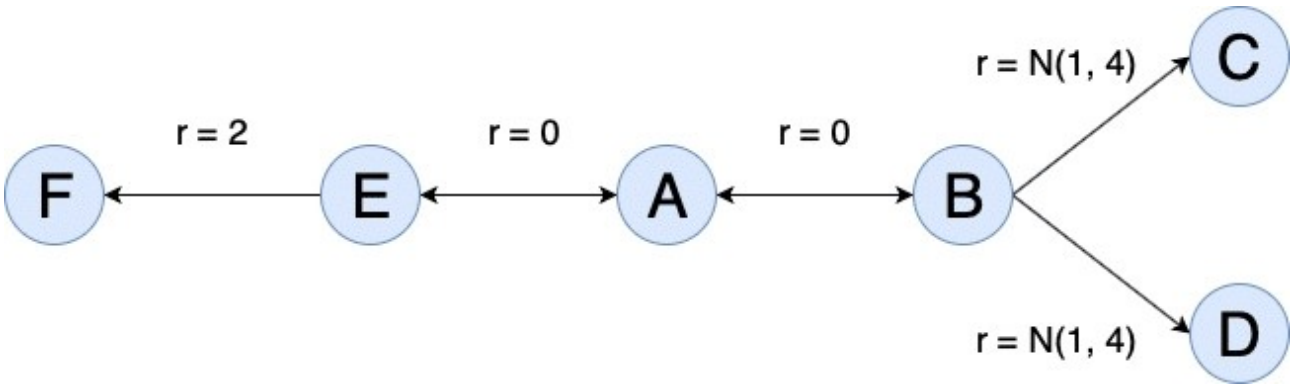
Le problème de l'apprentissage Q profond est lié à la manière dont il fixe les valeurs cibles :

$$Q_{target} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$$

Plus précisément, le problème concerne la valeur max. Cette partie de l'équation est censée estimer la valeur des récompenses pour les actions futures si l'action a est entreprise à partir de l'état actuel st. C'est un peu long, mais considérez qu'il s'agit d'essayer d'estimer les récompenses futures optimales rfuture si l'action a est entreprise.

Le problème est que dans de nombreux environnements, il y a du bruit aléatoire. Par conséquent, lorsqu'un agent explore un environnement, il n'observe pas directement r ou rfuture, mais quelque chose comme r + epsilon, où epsilon est le bruit. Dans un tel environnement, après avoir joué plusieurs fois au jeu, nous espérons que le réseau apprendra à faire des estimations non biaisées de la valeur attendue des récompenses - donc E[r]. S'il peut le faire, nous sommes dans une bonne situation : le réseau devrait choisir les meilleures actions pour les récompenses actuelles et futures, malgré la présence de bruit.

C'est là que l'opération max pose problème : elle produit des estimations biaisées des récompenses futures, et non les estimations non biaisées dont nous avons besoin pour obtenir des résultats optimaux. Un exemple permettra de mieux expliquer ce phénomène. Considérons l'environnement ci-dessous. L'agent commence dans l'état A et à chaque état, il peut se déplacer à gauche ou à droite. Les états C, D et F sont des états terminaux - le jeu se termine lorsque ces points sont atteints. Les valeurs r sont les récompenses que l'agent reçoit lorsqu'il passe d'un état à l'autre.



Toutes les récompenses sont déterministes à l'exception des récompenses lors de la transition des états B vers C et B vers D. Les récompenses pour ces transitions sont tirées au hasard d'une distribution normale avec une moyenne de 1 et un écart type de 4.

Nous savons que les récompenses attendues, E[r], de l'une ou l'autre des actions (B vers C ou B vers D) sont égales à 1 - cependant, il y a beaucoup de bruit associé à ces récompenses. Quoi qu'il en soit, en moyenne, l'agent devrait idéalement apprendre à toujours se déplacer vers la gauche depuis A, vers E et enfin F où r est toujours égal à 2.

Considérons l'expression de Qtarget pour ces cas. Fixons gamma à 0,95. L'expression Qtarget pour se déplacer vers la gauche à partir de A est : Qtarget = 0 + 0,95 * max([0, 2]) = 1,9. Les deux

options d'action de E sont de se déplacer vers la droite (r = 0) ou vers la gauche (r = 2). Le maximum de celles-ci est évidemment 2, et nous obtenons donc le résultat 1,9.

Et dans la direction opposée, en se déplaçant vers la droite depuis A ? Dans ce cas, on obtient Qtarget = 0 + 0,95 * max([N(1, 4), N(1, 4)]). Nous pouvons explorer la valeur à long terme de cette action de "déplacement vers la droite" en utilisant l'extrait de code suivant :

```
import numpy as np
Ra = np.zeros((10000,))
Rc = np.random.normal(1, 4, 10000)
Rd = np.random.normal(1, 4, 10000)
comb = np.vstack((Ra, Rc, Rd)).transpose()
max_result = np.max(comb, axis=1)
print(np.mean(Rc))
print(np.mean(Rd))
print(np.mean(max_result))
```

On crée un essai de 10 000 itérations de ce que le terme max donnera sur le long terme de l'exécution d'un agent Q profond dans cet environnement.

- Ra est la récompense pour le déplacement vers la gauche en direction de A (toujours zéro, d'où np.zeros()).
- Rc et Rd sont tous deux des distributions normales, avec une moyenne de 1 et un écart-type de 4.

En combinant toutes ces options et en prenant le maximum pour chaque essai, on obtient le terme max de chaque essai (max_result). Enfin, les valeurs attendues (c'est-à-dire les moyennes) de chaque quantité sont affichées. Comme prévu, les moyennes de Rc et Rd sont approximativement égales à 1 - la moyenne que nous avons fixée pour leurs distributions. Cependant, la valeur attendue moyenne du terme max est en fait d'environ 3 !

Vous pouvez voir le problème ici. Comme le terme max prend toujours la valeur maximale des tirages aléatoires des récompenses, il a tendance à être biaisé positivement et ne donne pas une véritable indication des valeurs attendues des récompenses pour un mouvement dans cette direction (c'est-à-dire 1). Ainsi, un agent utilisant la méthodologie d'apprentissage Q profond ne choisira pas l'action optimale de A (c'est-à-dire se déplacer vers la gauche) mais aura plutôt tendance à se déplacer vers la droite !

Par conséquent, dans les environnements bruyants, on peut voir que l'apprentissage Q profond aura tendance à surestimer les récompenses. Finalement, l'apprentissage Q profond convergera vers une solution raisonnable, mais il est potentiellement beaucoup plus lent qu'il ne devrait l'être. L'apprentissage Q profond pose un autre problème qui peut entraîner une instabilité dans le processus de formation. Considérez que dans l'apprentissage Q profond, le même réseau choisit la meilleure action et détermine la valeur du choix de ces actions. Il y a ici une boucle de rétroaction qui peut exacerber le problème de surestimation de la récompense mentionné précédemment, et ralentir davantage le processus d'apprentissage. Ce n'est évidemment pas idéal, et c'est pourquoi l'apprentissage Double Q a été développé.

introduction au Double Q reinforcement learning

https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

L'article qui a introduit l'apprentissage Double Q proposait initialement la création de deux réseaux distincts qui prédisaient respectivement QA et QB. Ces réseaux étaient formés sur le même environnement/problème, mais étaient chacun mis à jour de manière aléatoire. Ainsi, disons que 50 % du temps, QA était mis à jour sur la base d'un certain ensemble aléatoire de tuples d'entraînement, et 50 % du temps, QB était mis à jour sur la base d'un ensemble aléatoire différent de tuples d'entraînement. Il est important de noter que lors de la mise à jour de QA, le réseau A obtient une estimation des futures récompenses de la part du réseau B - et non de lui-même. Et la réciproque est vraie. Cette nouvelle approche fait deux choses :

1. Les réseaux A et B sont entraînés sur des échantillons d'entraînement différents, ce qui permet de supprimer le biais de surestimation, car, en moyenne, si le réseau A voit une récompense bruyante élevée pour une certaine action, il est probable que le réseau B verra une récompense plus faible, ce qui annule les effets du bruit.
2. Il y a un découplage entre le choix de la meilleure action et l'évaluation de la meilleure action.

L'algorithme de l'article original est le suivant :

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Une action est d'abord choisie parmi QA(s,.) ou QB(s,.) et les récompenses, l'état suivant, l'action etc. sont stockés dans la mémoire. Ensuite, on choisit aléatoirement soit UPDATE(A) soit UPDATE(B). Pour l'état s', la valeur Q prédite pour toutes les actions de cet état est extraite du réseau A ou B, et l'action ayant la valeur Q prédite la plus élevée est choisie, a*.

Intéressons nous au cas UPDATE(A). La meilleure action a* de l'état suivant s’ est choisie en utilisant le réseau A, la récompense actualisée pour cette action future est calculée en utilisant réseau B. Cela élimine tout biais associé à la argmax du réseau A, et dissocie également le choix des actions de l'évaluation de la valeur de ces actions (c'est-à-dire que cela rompt la boucle de rétroaction). C'est le cœur de l'apprentissage par renforcement Double Q.

Double DQN network

https://arxiv.org/pdf/1509.06461.pdf

Le même auteur a proposé un algorithme mis à jour appelé Double DQN (ou DDQN) et la principale différence est la suppression de la mise à jour aléatoire par rétropropagation de deux réseaux A et B. Il y a toujours deux réseaux impliqués, mais au lieu de les entrainer tous les deux, seul un réseau primaire est entrainé par rétropropagation. L'autre réseau, souvent appelé target network, est périodiquement copié à partir du réseau primaire. L'opération de mise à jour du réseau primaire dans le réseau DQN double ressemble à ce qui suit :

$$Q_{target} = r_{t+1} + \gamma Q(s_{t+1}, argmax Q(s_{t+1}, a; \theta_t); \theta_t^-)$$

ou encore :

$$a* = argmax Q(s_{t+1}, a; \theta_t)$$

$$Q_{target} = r_{t+1} + \gamma Q(s_{t+1}, a*); \theta_t^-)$$

Remarquez que, comme dans l'algorithme précédent, l'action a* ayant la valeur Q la plus élevée dans l'état suivant st+1 est extraite du réseau primaire, qui a des poids θt. Ce réseau primaire est aussi souvent appelé le **réseau online** - c'est le réseau à partir duquel les décisions d'action sont prises. Cependant, remarquez que, lors de la détermination de Qtarget, la valeur actualisée de Q est extraite du t**arget network** avec des poids θt^-. Par conséquent, les actions que l'agent doit entreprendre sont extraites du réseau online, mais l'évaluation des récompenses futures est extraite du target network. Jusqu'à présent, cette étape est similaire à l'étape UPDATE(A) présentée dans l'algorithme Double Q précédent.

La différence dans cet algorithme est que les poids du target network θt^- ne sont pas formés par rétropropagation - ils sont plutôt copiés périodiquement à partir du réseau online. Cela réduit la charge de calcul liée à l'apprentissage de deux réseaux par rétropropagation.

Cette copie peut être :

- soit une "copie dure" périodique, où les poids sont copiés du réseau online vers le target network sans modification, cette copie étant réalisée toutes les X itérations
- soit une "copie douce" à chaque itération, où les valeurs du target network et du réseau online sont mélangées, selon la règle suivante, τ étant une constante assez petite (0.05) :

$$\theta^- = \theta^-(1 - \tau) + \theta\tau$$

Policy-Based Methods - Hill Climbing algorithm

Nous allons présenter ici une classe d'algorithmes qui nous permettent d'approximer directement la fonction de politique π, au lieu de passer par les fonctions de valeurs V ou Q. Rappelez-vous que nous avons défini la politique comme l'entité qui nous dit quoi faire dans chaque état. En d'autres termes, au lieu d'entraîner un réseau qui produit des valeurs d'action, nous entraînerons un réseau pour produire (la probabilité) des actions.

Rappels sur les méthodes fondées sur la valeur

Le sujet central de l'itération de valeur et du Q-learning est la valeur de l'état (désignée par V) ou la valeur de l'état-action (désignée par Q). Rappelez-vous que la valeur est définie comme la récompense totale actualisée que nous pouvons obtenir d'un état ou en émettant une action particulière de l'état. Si la valeur est connue, la décision à chaque étape devient simple et évidente : elle est prise avec avidité (greedy) en termes de valeur, ce qui garantit une bonne récompense totale à la fin de l'épisode. Pour obtenir ces valeurs, nous avons utilisé l'équation de Bellman, qui exprime la valeur de l'étape courante par les valeurs de l'étape suivante (elle fait une prédiction à partir d'une prédiction).

Les méthodes fondées sur la politique

L'apprentissage par renforcement consiste finalement à apprendre une politique optimale, notée π*, à partir de l'interaction avec l'environnement. Jusqu'à présent, nous avons appris avec des méthodes basées sur la valeur, où nous trouvons d'abord une estimation de la fonction action-valeur optimale q* à partir de laquelle nous obtenons la politique optimale π*.

Pour les petits espaces d'état, comme l'exemple du lac gelé présenté dans le post 1, cette fonction de valeur optimale q* peut être représentée dans un tableau, le Q-table, avec une ligne pour chaque état et une colonne pour chaque action. À chaque pas de temps, pour un état donné, il suffit de tirer la ligne correspondante de la table, et l'action optimale est simplement l'action dont la valeur est la plus élevée.

Mais qu'en est-il des environnements avec des espaces d'état beaucoup plus grands, comme l'environnement Pong présenté dans les posts précédents ? Il y a un grand nombre d'états possibles, et cela rendrait le tableau beaucoup trop grand pour être utile en pratique. Nous avons donc utilisé un réseau neuronal pour représenter la fonction action-valeur optimale q*. Dans ce cas, le réseau neuronal reçoit en entrée l'état de l'environnement et renvoie en sortie la valeur de chaque action possible.

Dans les deux cas, que nous utilisions une table ou un réseau de neurones, nous avons dû d'abord estimer la fonction action-valeur optimale avant de pouvoir aborder la politique optimale π*. Une question intéressante se pose alors : peut-on trouver directement la politique optimale sans avoir à traiter au préalable une fonction de valeur ? La réponse est oui, et la classe d'algorithmes permettant d'y parvenir est connue sous le nom de méthodes basées sur la politique.

Avec les méthodes basées sur la valeur, l'agent utilise son expérience de l'environnement pour maintenir une estimation de la fonction action-valeur optimale. La politique optimale est ensuite

obtenue à partir de l'estimation de la fonction action-valeur optimale (par exemple, en utilisant epsilon-greedy).

Au contraire, les méthodes basées sur la politique apprennent directement la politique optimale à partir des interactions avec l'environnement, sans avoir à maintenir une estimation séparée de la fonction de valeur.

L'entropie croisée est un exemple de méthode basée sur une politique. Une politique $\pi(a \mid s)$, indique quelle action l'agent doit entreprendre pour chaque état observé. En pratique, la politique est généralement représentée comme une distribution de probabilité sur les actions (que l'agent peut prendre à un état donné) avec un nombre de classes égal au nombre d'actions que nous pouvons effectuer.

Les méthodes basées sur les politiques offrent quelques avantages par rapport aux méthodes de prédiction de valeur comme DQN. L'un d'eux est que, comme nous l'avons déjà dit, nous n'avons plus à nous soucier de concevoir une stratégie de sélection d'actions comme la politique ϵ-greedy ; au lieu de cela, nous échantillonnons directement les actions de la politique. Et ceci est important ; rappelez-vous que nous avons perdu beaucoup de temps à mettre au point des méthodes pour améliorer la stabilité de l'apprentissage de notre DQN. Par exemple, nous avons dû utiliser l’experience replay et les targets networks. Un réseau à base de politique tend à simplifier une partie de cette complexité.

Approximation de la fonction de politique avec un réseau de neurones

En apprentissage par renforcement profond, il est courant de représenter la politique par un réseau de neurones.

Dans l'exemple du problème d'équilibre chariot-pôle, un chariot est positionné sur une piste sans frottement le long de l'axe horizontal, et un poteau est ancré au sommet du chariot. L'objectif est d'empêcher le poteau de tomber en déplaçant le chariot vers la gauche ou vers la droite, sans tomber de la piste. Le système est contrôlé en appliquant une force de +1 (gauche) ou -1 (droite) au chariot. Le pendule démarre à la verticale, et le but est de l'empêcher de tomber. Une récompense de +1 est fournie pour chaque pas de temps où le poteau reste droit, y compris le dernier pas de l'épisode. L'épisode se termine lorsque le poteau s'écarte de plus de 15 degrés de la verticale, ou que le chariot se déplace de plus de 2,4 unités du centre.

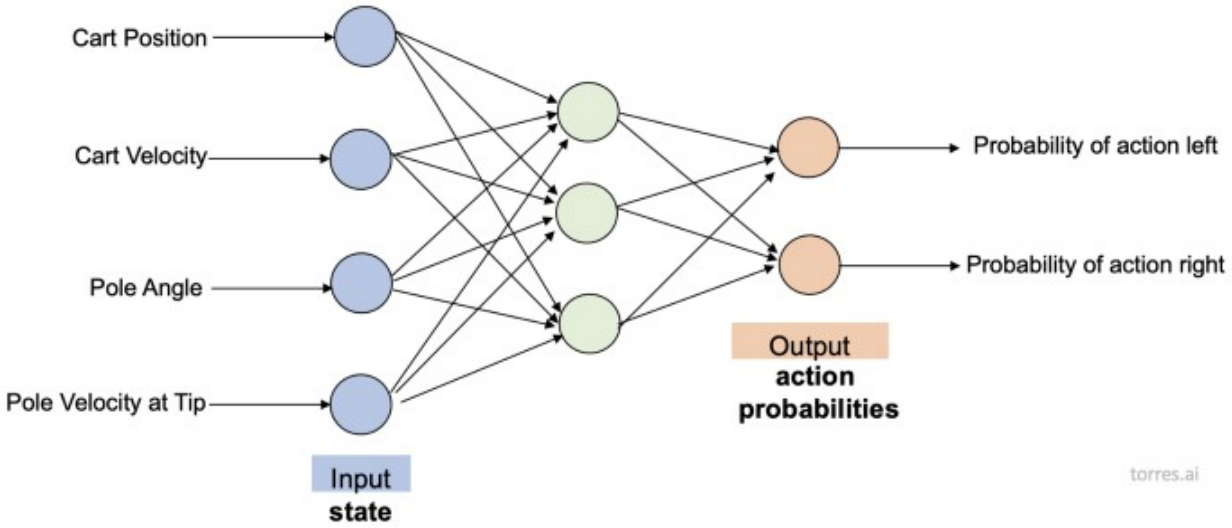
L'espace d'observation de cet environnement à chaque point temporel est un tableau de 4 nombres. À chaque pas de temps, vous pouvez observer sa position, sa vitesse, son angle et sa vitesse angulaire. Ce sont les états observables de ce monde :

https://github.com/openai/gym/wiki/CartPole-v0

Remarquez les valeurs minimales (-Inf) et maximales (Inf) pour la vitesse du chariot et la vitesse du pôle à la pointe. Puisque l'entrée dans le tableau correspondant à chacun de ces indices peut être un nombre réel quelconque, cela signifie que l'espace d'état est infini !

À chaque état, le chariot n'a que deux actions possibles : se déplacer vers la gauche ou vers la droite. En d'autres termes, l'espace d'état du pôle-chariot a quatre dimensions de valeurs continues, et l'espace d'action a une dimension de deux valeurs discrètes.

Nous pouvons construire un réseau neuronal qui se rapproche de la politique prenant un état en entrée. Dans cet exemple, la couche de sortie aura deux nœuds qui renvoient, respectivement, la probabilité de chaque action. En général, si l'environnement possède un espace d'action discret, comme dans cet exemple, la couche de sortie possède un nœud pour chaque action possible et contient la probabilité que l'agent choisisse chaque action possible.



L'agent alimente l'état actuel de l'environnement, puis échantillonne à partir des probabilités d'actions calculées par le réseau (gauche ou droite dans ce cas) pour sélectionner sa prochaine action.
L'objectif est de déterminer les valeurs appropriées pour les poids du réseau représentés par θ (Thêta). θ encode la politique selon laquelle pour chaque état que nous passons dans le réseau, il renvoie les probabilités d'action où l'action optimale a le plus de chances d'être sélectionnée. Les actions choisies influencent les récompenses obtenues qui sont utilisées pour obtenir le gain potentiel.
Rappelez-vous que l'objectif de l'agent est toujours de maximiser le gain potentiel. Dans notre cas, désignons le gain potentiel par J. L'idée principale est qu'il est possible d'écrire le gain potentiel J comme une fonction de θ. Nous verrons plus tard comment exprimer cette relation, J(θ), d'une manière plus "mathématique" pour trouver les valeurs des poids qui maximisent le rendement attendu.

Méthodes sans dérivation

Les poids du réseau neuronal sont initialement définis sur des valeurs aléatoires. L'agent met à jour les poids au fur et à mesure qu'il interagit avec l'environnement. Cette section donne un aperçu des approches possibles pour optimiser ces poids, les méthodes sans dérivation, également appelées méthodes d'ordre zéro.

Les méthodes sans dérivation recherchent directement dans l'espace des paramètres le vecteur de poids qui maximise les gains potentiels obtenus par une politique, en évaluant seulement certaines

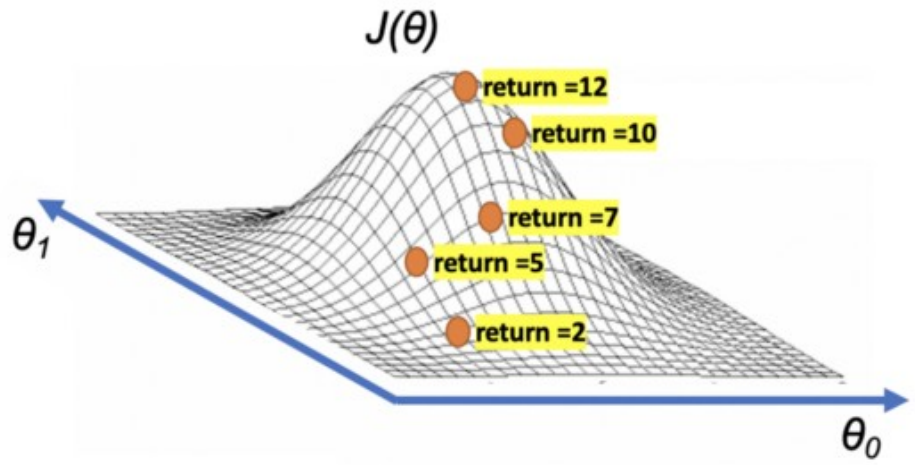
positions de l'espace des paramètres, sans les dérivées qui calculent les gradients. Expliquons l'algorithme le plus simple de cette catégorie, dénommé Hill Climbing, et qui nous aidera à comprendre par la suite le fonctionnement des méthodes de type Policy-Gradient.

Hill Climbing

https://en.wikipedia.org/wiki/Hill_climbing

Le Hill Climbing est un algorithme itératif qui peut être utilisé pour trouver les poids θ d'une politique optimale. Il s'agit d'un algorithme relativement simple que l'agent peut utiliser pour améliorer progressivement les poids θ de son réseau de politiques tout en interagissant avec l'environnement.

Comme son nom l'indique, intuitivement, on peut visualiser que l'algorithme élabore une stratégie pour atteindre le point le plus haut d'une colline, où θ indique les coordonnées de l'endroit où l'on se trouve à un moment donné et G indique l'altitude à laquelle on se trouve à ce point :



Hill Climbing: L'objectif de l'agent est de maximiser le gain potentiel J. Les poids du réseau neuronal pour cet exemple sont θ = (θ1,θ2).

Cet exemple visuel représente une fonction de deux paramètres, mais la même idée s'applique à plus de deux paramètres. L'algorithme commence par une estimation initiale de la valeur de θ (ensemble aléatoire de poids). Nous recueillons un seul épisode avec la politique qui correspond à ces poids θ et enregistrons ensuite le gain potentiel G.
Ce retour est une estimation de ce à quoi ressemble la surface à cette valeur de θ. Ce ne sera pas une estimation parfaite car il est peu probable que le gain potentiel que nous venons de collecter soit égal au gain potentiel attendu. En effet, en raison du caractère aléatoire de l'environnement (et de la politique, si elle est stochastique), il est fort probable que si nous collectons un deuxième épisode avec les mêmes valeurs pour θ, nous obtiendrons probablement une valeur différente pour le gain potentiel G. Mais en pratique, même si le gain potentiel (échantillonné) n'est pas une estimation parfaite du gain potentiel attendu, il s'avère souvent assez bon.
À chaque itération, nous perturbons légèrement les valeurs (en ajoutant un peu de bruit aléatoire) de la meilleure estimation actuelle des poids θ, pour obtenir un nouvel ensemble de poids candidats que nous pouvons essayer. Ces nouveaux poids sont ensuite utilisés pour collecter un épisode et calculer le gain potentiel. Si les nouvelles pondérations nous donnent un gain potentiel supérieur à

notre meilleure estimation actuelle, nous concentrons notre attention sur cette nouvelle valeur, puis nous recommençons en proposant itérativement de nouvelles politiques dans l'espoir qu'elles soient plus performantes que la politique existante. Si ce n'est pas le cas, nous revenons à notre dernière meilleure estimation de la politique optimale et nous itérons jusqu'à ce que nous obtenions la politique optimale.
Maintenant que nous avons une compréhension intuitive du fonctionnement de l'algorithme Hill Climbing, nous pouvons le résumer avec le pseudocode suivant :

1. Initialiser la politique π avec des poids aléatoires θ
2. Initialiser θbest (notre meilleure estimation pour les poids θ)
3. Initialiser Gbest (le meilleur gain potentiel G que nous avons obtenu jusqu'à présent)
4. Collecter un seul épisode avec θbest, et enregistrer le rendement G
5. Si G > Gbest alors θbest ← θ et Gbest ← G
6. Ajouter un peu de bruit aléatoire à θbest, pour obtenir un nouvel ensemble de poids θ.
7. Répétez les étapes 4 à 6 jusqu'à ce que l'environnement soit résolu.

Dans notre exemple, nous avons supposé une surface avec un seul maximum, configuration qui convient bien à l'algorithme Hill-climbing. Notez qu'avec plus d'un maximum local, si l'algorithme commence à un mauvais endroit, il peut converger vers le maximum le plus bas.

Coding Hill Climbing

https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_18_Policy_Based_Methods.ipynb

Cette section va explorer une implémentation de Hill-Climbing appliquée à l'environnement Cartpole, basée sur le pseudocode précédent. Le modèle de réseau de neurones ici est simple, il n'utilise qu'une matrice de forme [4x2] (state_space x action_space), pas besoin de tenseur, de PyTorch ni même de GPU.

Comme toujours, nous commençons par importer les bibliothèques nécessaires et créer l'environnement :

```
import gym
import numpy as np
from collections import deque
import matplotlib.pyplot as plt
env = gym.make('CartPole-v0')
from IPython import display
```

La politique π (initialisée avec des poids aléatoires θ) peut être codée comme suit :

```
class Policy():
    def __init__(self, s_size=4, a_size=2):
        # 1. Initialize policy π with random weights
        self.θ = 1e-4*np.random.rand(s_size, a_size)

    def forward(self, state):
        x = np.dot(state, self.θ)
        return np.exp(x)/sum(np.exp(x))
```

```
    def act(self, state):
        probs = self.forward(state)
        action = np.argmax(probs) # deterministic policy
        return action
```

Pour visualiser l'effet de l’entraînement, nous affichons les poids θ avant et après l’entraînement et visualisons comment l'agent applique la politique :

```
def watch_agent():
  env = gym.make('CartPole-v0')
  state = env.reset()
  rewards = []
  img = plt.imshow(env.render(mode='rgb_array'))
  for t in range(2000):
    action = policy.act(state)
    img.set_data(env.render(mode='rgb_array'))
    plt.axis('off')
    display.display(plt.gcf())
    display.clear_output(wait=True)
    state, reward, done, _ = env.step(action)
    rewards.append(reward)
    if done:
        print("Reward:", sum([r for r in rewards]))
        break
  env.close()

policy = Policy()
print ("Policy weights θ before train:\n", policy.θ)
watch_agent()
```

```
Policy weights θ before train:
 [[6.30558674e-06 2.13219853e-05]
 [2.32801200e-05 5.86359967e-05]
 [1.33454380e-05 6.69857175e-05]
 [9.39527443e-05 6.65193884e-05]]
Reward: 9.0
```

La fonction suivante entraîne l'agent :

```
def hill_climbing(n_episodes=10000, gamma=1.0, noise=1e-2):
    scores_deque = deque(maxlen=100)
    scores = []

    #2. Initialize θbest
    Gbest = -np.Inf
    #3. Initialize Gbest
    θbest = policy.θ
    for i_episode in range(1, n_episodes+1):
        rewards = []
        state = env.reset()
        while True:
        #4.Collect a single episode with θ,and record the return G
```

```
            action = policy.act(state)
            state, reward, done, _ = env.step(action)
            rewards.append(reward)
            if done:
                break
        scores_deque.append(sum(rewards))
        scores.append(sum(rewards))
        discounts = [gamma**i for i in range(len(rewards)+1)]
        G = sum([a*b for a,b in zip(discounts, rewards)])
        if G >= Gbest: # 5. If G>Gbest then θbest ← θ & Gbest ← G
            Gbest = G
            θbest = policy.θ

        #6. Add a little bit of random noise to θbest
        policy.θ = θbest + noise * np.random.rand(*policy.θ.shape)
        if i_episode % 10 == 0:
            print('Episode {}'.format(i_episode))
            print('Average Score: {:.2f}'.format(np.mean(scores_deque)))
        # 7. Repeat steps 4-6 until Environment solved
        if np.mean(scores_deque)>=env.spec.reward_threshold:
            print('Environnement solved in {:d} episodes!'.format(i_episode))
            print('Average Score: {:.2f}'.format(np.mean(scores_deque)))
            policy.θ = θbest
            break

    return scores
```

Le code est assez limpide, étant annoté avec les étapes du pseudocode correspondant. On peut noter que l’algorithme cherche à maximiser la récompense cumulée actualisée comme suit :

```
discounts = [gamma**i for i in range(len(rewards)+1)]
G = sum([a*b for a,b in zip(discounts, rewards)])
```

Le Hill Climbing est un algorithme itératif simple sans gradient (c'est-à-dire n'utilisant pas les méthodes d'ascension/descente de gradient). On essaie de grimper au sommet en bruitant les poids du réseau neuronal, poids que l’on peut assimiler aux coordonnées de l'endroit où l'on se trouve à un moment donné, le gain potentiel indiquant l'altitude à laquelle on se trouve à ce point :

```
policy.θ = θbest + noise * np.random.rand(*policy.θ.shape)
```

On considère avoir résolu l'environnement dès qu’on dépasse un certain seuil. Pour Cartpole-v0, ce score seuil est de 195, indiqué par env.spec.reward_threshold.

```
scores = hill_climbing(gamma=0.9)
```

Bien que dans cet exemple, nous ayons codé une politique déterministe pour des raisons de simplicité, les méthodes basées sur les politiques peuvent apprendre des politiques stochastiques ou déterministes, et elles peuvent être utilisées pour résoudre des environnements avec des espaces d'action finis ou continus.

Gradient Free Policy Optimization

L'algorithme Hill Climbing n'a pas besoin d'être différentiable ou même continu, mais comme il effectue des étapes aléatoires, cela peut ne pas aboutir au chemin le plus efficace pour gravir la colline. Il existe dans la littérature de nombreuses améliorations de cette approche : mise à l'échelle adaptative du bruit, steepest ascent hill climbing, redémarrages aléatoires, recuit simulé, stratégies d'évolution ou méthode de l'entropie croisée.

Cependant, les solutions habituelles à ce problème considèrent les méthodes de type policy gradient qui estiment les poids d'une politique optimale par l'ascension du gradient.

Méthodes Policy-Gradient - Algorithme REINFORCE

Les méthodes Policy-Gradient sont une sous-classe de méthodes basées sur des politiques/stratégies qui estiment les poids d'une stratégie optimale grâce à l'ascension du gradient.

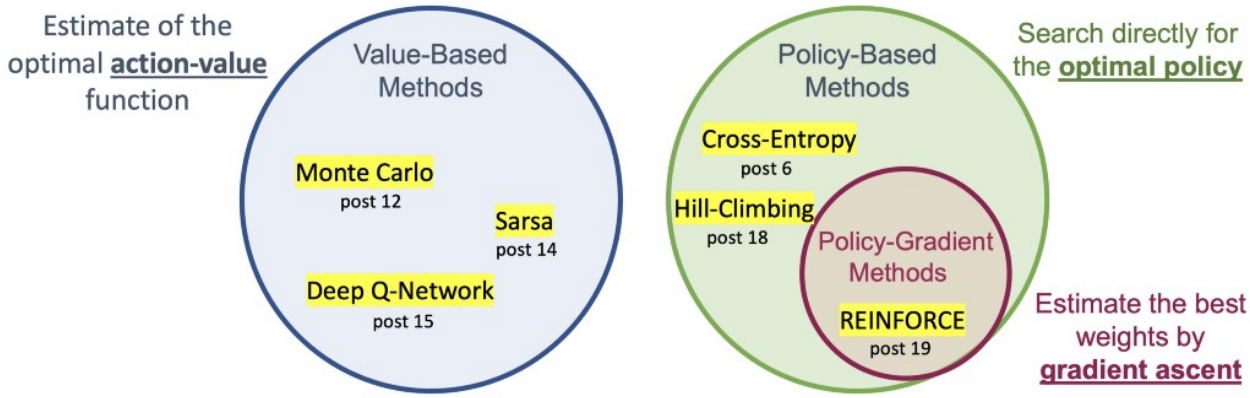


Figure 1: Résumé des approches de l'apprentissage par renforcement. La classification est basée sur le fait que nous voulons modéliser la valeur ou la politique

Intuitivement, l'ascension du gradient commence par une estimation initiale de la valeur des poids de la politique, puis, l'algorithme évalue le gradient à ce point pour évaluer la direction de la plus forte augmentation du gain potentiel attendu, et fait un petit pas dans cette direction.

Nous espérons nous retrouver avec des valeurs des poids qui feront que la nouvelle politique nous permettra d'améliorer le gain potentiel. L'algorithme répète ensuite ce processus d'évaluation du gradient et prend des mesures jusqu'à ce qu'il considère avoir atteint le gain potentiel maximal attendu.

Introduction

Les méthodes basées sur les politiques peuvent apprendre des politiques stochastiques ou déterministes. Avec une politique stochastique, la sortie de notre réseau neuronal est un vecteur d'actions qui représente une distribution de probabilité (plutôt que de renvoyer une seule action déterministe).

La politique que nous suivrons consiste à sélectionner une action à partir de cette distribution de probabilité. Cela signifie que si notre agent se retrouve deux fois dans le même état, nous risquons de ne pas décider la même action à chaque fois. Une telle représentation des actions en tant que probabilités présente de nombreux avantages, par exemple l'avantage d'une représentation lisse: si nous changeons un peu les poids de notre réseau, la sortie du réseau neuronal changera, mais probablement juste un peu.

Dans le cas d'une politique déterministe, avec une sortie discrète, même un petit ajustement des poids peut conduire à un saut vers une action différente. Toutefois, si on passe par une distribution de probabilité, un petit changement de pondération entraînera généralement un petit changement dans la distribution de probabilité. Il s'agit d'une propriété très importante car les méthodes

d’optimisation du gradient consistent à modifier un peu les paramètres d’un modèle pour améliorer les résultats.

Mais comment modifier les paramètres du réseau pour améliorer la politique ? Nous avons résolu un problème très similaire en utilisant la méthode Cross-Entropy: notre réseau a pris des observations comme entrées et a renvoyé la distribution de probabilité des actions. En fait, la méthode de l’entropie croisée est, en quelque sorte, une version préliminaire de la méthode que nous allons présenter.

L’idée clé qui sous-tend la méthode policy-gradient est de renforcer les bonnes actions : augmenter les probabilités d’actions qui conduisent à un gain potentiel plus élevé, et pousser vers le bas les probabilités d’actions qui conduisent à un gain potentiel plus faible, jusqu’à arriver à la politique optimale. La méthode policy-gradient modifie de manière itérative les poids du réseau de politique (avec des mises à jour fluides) pour rendre les paires état-action qui ont entraîné un gain potentiel positif plus probables et rendre les paires état-action qui ont entraîné un gain potentiel négatif moins probables.

Pour introduire cette idée, nous allons commencer par une version de base des méthodes policy-gradient appelée algorithme REINFORCE

http://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf

Cet algorithme est l’algorithme fondamental sur lequel reposent presque tous les algorithmes policy-gradient avancés.

REINFORCE: Définitions mathématiques

Trajectoire

Une trajectoire est une séquence état-action-récompenses. C’est un objet est un peu plus flexible qu’un épisode car il n’y a pas de restrictions sur sa longueur; il peut correspondre à un épisode complet ou seulement à une partie d’un épisode. Nous désignons sa longueur avec un H majuscule, où H représente Horizon, et nous représentons une trajectoire avec τ:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$$

La méthode REINFORCE est construite sur des trajectoires plutôt que sur des épisodes, car la maximisation du retour attendu sur les trajectoires permet à la méthode de rechercher des politiques optimales pour les tâches épisodiques et continues. Bien entendu, dans les cas où une récompense n’est livrée qu’à la fin de l’épisode, il est logique d’utiliser l’épisode complet comme trajectoire, sinon, nous n’avons pas assez d’informations sur les récompenses pour estimer de manière significative le gain potentiel attendu.

Gk est le gain potentiel que nous espérons recueillir à partir du pas de temps k jusqu’à la fin de la trajectoire, et il peut être approximé en ajoutant les récompenses obtenues à partir d’un état dans l’épisode jusqu’à la fin de l’épisode en utilisant gamma γ:

$$G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_t$$

A SUIVRE…..