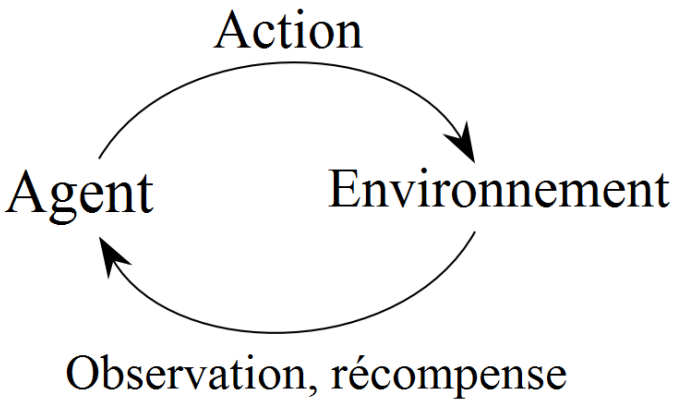
Rapport de projet intégrateur L3 MAPI^3

Stabilisation du pendule inversé par apprentissage automatique

Mai 2017

L'apprentissage par renforcement

Cette apprentissage automatique a comme objectif de créer un algorithme "intelligent", au travers de processus d'apprentissage et d'évolution, en observant le résultat de certaines actions. La machine ne sait pas quelles actions exécuter mais elle doit découvrir quelles actions donnent le plus de récompenses en les essayant. L'agent va donc déterminer automatiquement le comportement idéal afin de maximiser ses performances. Pour cela, un retour des résultats des actions est nécessaire pour apprendre comment les machines doivent agir c'est ce qu'on appelle le signal de renforcement. On a donc un agent situé dans un environnement inconnu qui, à chaque étape, prend une action et reçoit une observation et une récompense de l'environnement. L'algorithme va chercher à maximiser la récompense totale de l'agent à travers un processus d'apprentissage.



Finalement, si suffisamment d'états sont observés, une politique de décision optimale sera générée et nous aurons un algorithme qui agira parfaitement dans son environnement.

Dans le cas de la stabilisation de notre pendule inversé, nous nous trouvons dans le cas où nous ne pouvons pas savoir a priori quelle serait la bonne action à effectuer à un instant t donné. Nous sommes donc dans le cadre d'un apprentissage par renforcement. Nous présenterons dans la partie suivante la théorie sous-jacente à ce type d'apprentissage.

Processus de décision markovien

Afin de comprendre ce qu'est un processus de décision markovien, supposons que l'on ait un système évoluant dans le temps, à chaque instant le système est dans un état donné et il existe une certaine probabilité pour que le système évolue vers tel ou tel autre état à l'instant suivant en effectuant une transition. Supposons maintenant que l'on aimerait contrôler ce système afin de l'amener dans un état désiré, en évitant de lui faire traverser des états néfastes. On va donc mesurer l'efficacité de notre contrôle grâce à des gains ou à des pénalités reçus après chaque action. Ainsi, le raisonnement à base du processus de décision markovien peut se ramener au discours suivant : dans un état donné et en faisant une certaine action, nous avons une certaine probabilité de se retrouver dans certain nouvel état avec un certain gain.

De manière plus formelle, un processus de décision markovien est un modèle général pour un environnement stochastique dans lequel un agent peut prendre des décisions et reçoit des récompenses. Il est nécessaire de faire une supposition markovienne de premier ordre sur la

distribution des états visités pour modéliser un problème avec un processus de décision markovien, c'est à dire que l'état n+1 ne va déprendre que de l'état n et des actions possibles à l'état n (il n'y a pas d'implication du passé).

Un processus de decision markovien est défini par 4 éléments :

- Un ensemble d'états S.
- Un ensemble d'actions possibles lorsqu'on se trouve à l'état "s", représenté par une fonction qui prend en entrée un état possible qui appartient à notre ensemble d'état et qui retourne les actions possibles que l'agent peut effectuer.
- Un modèle de transition $P(s'/s, a)$ où "a" à l'ensemble des actions possibles à l'état "s". C'est la probabilité de se trouver à l'état "s'" si on est à l'état "s" et qu'on fait l'action "a".
- Une fonction de récompense $R(s)$ qui représente l'utilité pour l'agent d'être dans l'état "s'".

On ne pourra utiliser ce type de processus de décision que si notre but peut se formaliser sous la forme d'une fonction de récompense basée seulement sur l'état courant. C'est à dire qu'il va falloir qu'on arrive à décrire notre but sous la forme d'une fonction de récompense. Typiquement le cas simple est d'affecter une récompense élevée à un état qui correspond au plus près à notre but. Dans le cas de notre pendule, notre but étant de le maintenir en équilibre haut, chaque action menant le pendule à un état où il est toujours en l'air (respectant un certain angle et une certaine distance de notre point de départ) recevra une récompense. C'est donc sur ce modèle mathématique que s'appuie l'apprentissage par renforcement, cependant c'est un modèle assez général qui offre plusieurs possibilités d'implémentation.

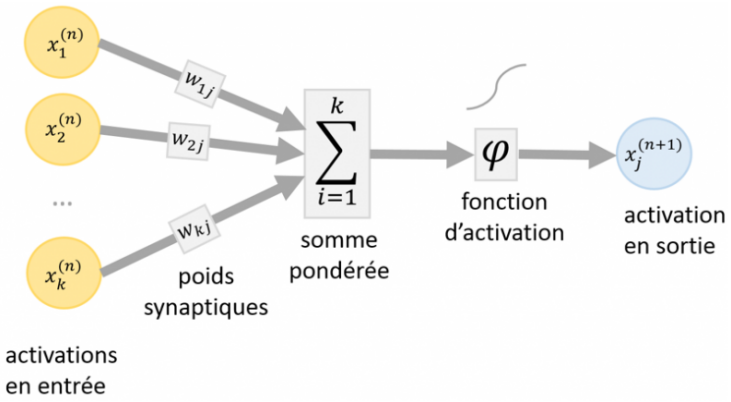
Implémentation de l'apprentissage par renforcement

D'un point de vue algorithmique, il existe deux approches différentes :

1. **La programmation dynamique** qui s'appuie sur un principe simple, appelé le principe d'optimalité de Bellman : toute solution optimale s'appuie elle-même sur des sous-problèmes résolus localement de façon optimale. Cela consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. Cela s'inspire de la méthode diviser pour régner sauf que la programmation dynamique résout chaque sous-problème une seule fois et mémorise la réponse dans un tableau, évitant ainsi le recalcul. Dans le cas de l'apprentissage par renforcement, l'algorithme le plus populaire est le Q-learning. Comme nous l'avons vu précédemment, le but de l'agent est de maximiser sa récompense totale par l'apprentissage de l'action optimale pour chaque état. Pour cela, l'algorithme va calculer une fonction de valeur action-état $Q : A \times S \to \mathbb{R}$.
   Le cœur de l'algorithme est une simple mise à jour de cette fonction de façon itérative à la suite de chaque transition d'un état s_n à un état s_{n+1} par une action a_n. Cette mise à jour se fait sur la base de la récompense associée, l'algorithme va simplement chercher à maximiser la valeur de la fonction à chaque étape pour avoir les suites d'actions les plus récompensantes . Ainsi la politique optimale pourra être construite en sélectionnant l'action correspondant à la valeur maximale de Q pour chaque état.
2. **Les réseaux de neurones :** Leur conception est à l'origine très schématiquement inspirée du fonctionnement des neurones biologiques, et qui par la suite s'est rapprochée des méthodes statistiques. Ils vont casser la linéarité des algorithmes classiques et permettre

d'effectuer de l'optimisation de politique. Contrairement au Q-learning, l'optimisation de la politique vise à optimiser la fonction qui associe directement une action à une observation. C'est cette approche que nous allons utiliser dans la suite pour la stabilisation de notre pendule.

L'architecture d'un réseau de neurones consiste en général à organiser les neurones en couches successives avec des interconnexions limitées aux couches adjacentes. Chaque couche i est composée de N_i neurones, prenant leurs entrées sur les N_{i-1} neurones de la couche précédente. À chaque synapse (représentant la connection entre les neurones de couches différentes) est associé un poids synaptique, de sorte que les N_{i-1} valeurs sorant des neurones de la couche $i-1$ soient multipliés par ce poids, puis additionnés par les neurones de niveau i. On a ensuite une fonction d'activation en sortie de chaque neurone qui permet ou non de transmettre l'information aux neurones suivant. Une couche de neurones est donc représentée ainsi :



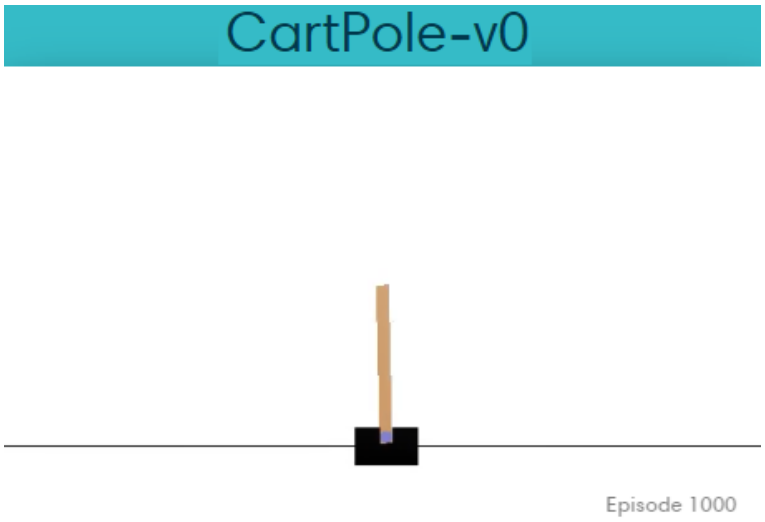
On classifie les couches en trois catégories : une couche d'entrée, des couches cachées et une couche de sortie. Dans le cadre de notre pendule inversé, la couche d'entrée correspond à l'état de notre système pendule-chariot à un instant t, cela correspond à un vecteur de dimension 4 contenant la position x, l'angle θ, la vitesse x' du chariot et la vitesse angulaire θ' du pendule. La couche de sortie est en l'occurrence un seul neurone constituée de la fonction d'activation φ qui va permettre d'introduire une non-linéarité dans le fonctionnement de notre réseau. Il existe différentes fonctions d'activation avec différentes caractéristiques, mais celle qui se rapproche le plus de l'activation biologique des neurones est la fonction sigmoïde. Elle permet, en fonction des niveaux d'activation des neurones de la couche cachée pondérés par les poids synaptiques, de donner une probabilité d'action. L'entraînement du réseau va consister à trouver des poids synaptiques tels que la couche de sortie permette d'effectuer le bon mouvement pour un état donné. On peut se demander avant tout si de tels poids existent toujours quel que soit l'objectif assigné au réseau de neurones. Par chance, un résultat mathématique connu sous le nom de théorème d'approximation universel garantit que c'est effectivement possible, même pour un réseau ne comportant qu'une seule couche cachée, à condition toutefois que φ soit non linéaire et qu'un nombre suffisant de neurones soient mis en jeu en fonction de la marge d'erreur tolérée.

Pour mieux comprendre le fonctionnement d'un réseau de neurone pour l'apprentissage par renforcement, nous allons expliciter celui-ci dans le cas simple de notre pendule tout en détaillant son implémentation.

Mise en pratique avec le modèle du pendule : programmation

- **L'environnement** Dans un premier temps on va importer l'environnement qui va modéliser notre pendule. Grâce à la bibliothèque OpenAI gym, nous avons directement un environnement CartPole-v0 représentant de façon simplifiée notre problème du maintient en équilibre du pendule en position haute.

```
import gym
env = gym.make('CartPole-v0').env
```



Le pendule est modélisé comme étant attaché par une liaison parfaite avec un chariot se déplaçant sur une piste sans frottement. Le système est contrôlé en appliquant une force unitaire de +1 ou -1 au chariot. Le pendule commence debout et le but est de l'empêcher de tomber. Une récompense de +1 est fournie après chaque action qui conserve le pendule en l'air. On considère que l'épisode est terminé lorsque le pendule est à plus de 15 degrés par rapport à la verticale, ou si le chariot se déplace plus de 2,4 unités du centre. Cet environnement va prendre en entrée soit la valeur 0 qui correspond à appliquer la force -1 sur le chariot, soit la valeur 1 qui correspond à appliquer la force +1. Il retourne un vecteur de dimension 4 $(x, \dot{x}, \theta, \dot{\theta})$ correspondant à l'état du système après chaque action.

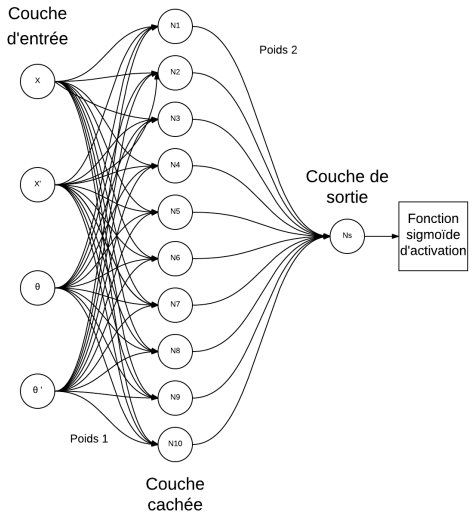
- **Contruction de notre réseau de neuronnes** Dans un premier temps, on commence par affecter la valeur des différents paramètres de notre algorithme.

```
nbr_neurones = 10
batch_size = 5
dimension = 4
```

La paramètre "batch size" va correspondre au nombre d'épisodes, composant notre échantillon d'entraînement, à effectuer avant de modifier notre politique. C'est à dire que le réseau apprendra à partir de cet échantillon regroupant les différents états/actions/récompenses effectués.
Nous allons maintenant coder notre réseau de neurones. Pour se faire nous allons utiliser la bibliothèque TensorFlow qui facilite grandement la création de celui-ci ainsi que sa manipulation.

```
observations = tf.placeholder(tf.float32, [None, dimension])
poids1 = tf.Variable(tf.random_normal([4,nbr_neurones]))
layer1 = tf.nn.relu(tf.matmul(observations,poids1))
poids2 = tf.Variable(tf.random_normal([nbr_neurones,1]))
output = tf.nn.sigmoid(tf.matmul(layer1,poids2))
```

Dans cette exemple nous avons créé un réseau de neurones assez simple : on a une couche d'entrée, qui correspond aux observations de l'état du système, qui est relié par des synapses pondérées par les poids 1 à une couche cachée de 10 neurones, elle-même reliée à l'unique neurone de sortie par des synapses pondérées par les poids 2. Nos poids sont initialisés aléatoirement suivant une distribution gaussienne. À la sortie de la première couche de neurones, on utilise la fonction d'activation "relu" de TensorFlow qui correspond à une rectification linéaire $f(x) = max(0, x)$, ce qui permet de créer une activation discrète des neurones qui se rapproche un peu plus des neurones biologiques. Chaque neurone de la couche cachée reçoit la somme pondérée des observations et retourne la valeur de cette fonction "relu" appliquée à cette somme. Puis le neurone de la couche de sortie reçoit la somme pondérée de ces différentes valeurs d'activation et délivre la probabilité p d'action par la fonction d'activation sigmoïde. Arbitrairement on établit que p correspond à la probabilité d'avoir un 1 et donc d'effectuer une action $+1$ sur le chariot. Comme c'est un choix binaire, on a une probabilité $1 - p$ d'avoir 0 et d'appliquer une force -1. Intuitivement, les neurones dans la couche cachée avec les poids 1 vont pouvoir détecter différents états de notre système et les poids 2 vont décider si avec cet état le chariot doit aller vers la gauche ou vers la droite. Bien entendu dans un premier temps, les poids étant aléatoirement initialisés, le chariot aura des mouvements incohérents. La stabilisation du chariot se ramène donc à trouver les bons poids qui permettent de guider correctement le chariot à chaque étape. Voici la représentation schématique de notre réseau :



Théorie de l'apprentissage On a vu précédemmment que la politique est la fonction qui pour un état "s" du système va nous renvoyer une certaine action. Dans le cas de notre réseau de neurone, cette fonction est représentée par tout le réseau, l'action étant donnée in fine par la fonction activatrice du neurone de sortie. Mais pour changer cette politique, on ne peut qu'agir sur les différents poids synaptiques. À chaque phase d'apprentissage, on va avoir une estimation stochastique de la somme des récompenses créée par la politique donnée qu'on va pouvoir utiliser enfin d'optimiser ses poids.

Reprenons le processus de décision markovien, nous avons notre politique stochastique π qui pour un état "s", nous donne une probabilité conditionnelle d'action "a" : $\pi(a/s) = a$
On aura donc schématiquement le long d'un épisode :
Un première état donné par une distribution μ aléatoire :

$$s_0 = \mu(s_0)$$

puis une première action dépendant de cet état :

$$a_0 = \pi(a_0/s_0)$$

entrainant une transition P ainsi qu'une récompense "r" :

$$s_1, r_0 = P(s_1/s_0, a_0)$$

Et ainsi de suite jusqu'à l'état terminal s_T lorsque l'épisode se finit. Après avoir récolter les informations de différents épisodes, notre objectif va être de maximiser les récompenses sous la forme d'une fonction « avantage ». La plus courante est $R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$ où $\gamma \in [0, 1]$ est le facteur d'actualisation. Il représente l'importance qu'on donne aux récompenses futures à notre action effectuée à un instant t. On somme toutes les récompenses ultérieures à l'action mais avec une décroissance exponentielle.
Pour maximiser cette fonction, il convient de trouver la politique π la plus adaptée. Cependant, la politique est une distribution paramétrée par les différents poids des synapses. Par simplification d'écriture nous noterons l'ensemble de ces paramètres sous le terme θ, le problème se ramène donc à trouver θ tel que la politique soit la plus performante, c'est à dire que la fonction "avantage" soit la plus grande possible. On effectue donc un certain nombre de simulations d'épisodes qui vont former notre échantillon à partir duquel on va calculer l'espérance de notre fonction "avantage". Les différentes récompenses sont distribuées suivant la politique qui représente la densité de probabilité $p(x/\theta)$ d'avoir une action a donnée pour un état s.
Comment devrait-on décaler la distribution à travers les paramètres θ pour augmenter l'espérance de la fonction de cet échantillon, c'est à dire comment modifions-nous les paramètres du réseau afin que les actions échantillonnées obtiennent des récompenses plus élevées à l'avenir ?
On va calculer le gradient de l'espérance de la fonction des récompenses pour déterminer le θ qui maximise celle-ci.
D'après la définition de l'espérance nous avons :

$$\begin{aligned}
\nabla_\theta \mathbb{E}_\theta[f(X)] &= \nabla_\theta \sum_x p(x/\theta).f(x) \\
&= \sum_x \nabla_\theta p(x/\theta).f(x) \\
&= \sum_x p(x/\theta).\frac{\nabla_\theta p(x/\theta)}{p(x/\theta)}.f(x) \\
&= \sum_x p(x/\theta).\nabla_\theta \log(p(x/\theta)).f(x) \\
&= \mathbb{E}_\theta[f(X).\nabla_\theta \log(p(x/\theta))]
\end{aligned}$$

On est donc ramené à calculer $\nabla_\theta \log(p(x/\theta))$ qui correspond au vecteur qui nous donne la direction dans l'espace des paramètres qui conduirait à augmenter la probabilité assignée

à un x. Nous devons donc prendre cette direction et multiplier celle-ci par $f(x)$. Cela fait en sorte que les échantillons qui ont une valeur de cette fonction plus élevée "tirent" sur la densité de probabilité plus fortement que les échantillons qui ont un score plus faible, donc si nous devions effectuer une mise à jour en fonction de plusieurs échantillons de x, la densité de probabilité se déplacerait dans la direction des scores plus élevés de la fonction "avantage", ce qui rend les échantillons hautement récompensant plus susceptibles de se reproduire.

C'est exactement le même principe que l'on va mettre en place pour optimiser notre réseau de neurones sauf que nous allons chercher à minimiser le négatif de la log-vraisemblance $\nabla_\theta \log(p(x/\theta))$ de notre échantillon. Ce principe coïncide avec le principe du maximum de vraisemblance sauf qu'il est plus aisé de trouver les paramètres θ qui minimisent cette fonction grâce aux méthodes de descente de gradient.

On implémente donc en premier la fonction "avantage" :

```
def fonction_avantage (recompenses) :
 return np.array([val * (gamma ** i) for i, val in enumerate(recompenses)])
```

Puis notre politique :

```
variables = [poids1, poids2]
input_y = tf.placeholder(tf.float32, [None,1])
avantages = tf.placeholder(tf.float32)
log_vraisemenblance = - tf.log(input_y * (input_y - output) + (1 - input_y) * (output))
loss = tf.reduce_mean(log_vraisemenblance * avantages)
grad = tf.gradients(loss, variables)
```

Après chaque observation, le réseau nous donne la probabilité p d'avoir 1. On tire donc U une uniforme sur $[0, 1]$, si U est inférieure à p alors notre action sera 1, sinon elle sera 0. La vraisemblance $p(x = 1/\theta)$ est donc égale à p et $p(x = 0/\theta)$ est égale à $1 - p$. On code donc notre log-vraisemblance par tf.log(input y * (input y - output) + (1 - input y) * (output)) où "output" correspond à p et input y correspond à l'action que l'on donne à l'environnement CartPole-v0. On calcule ensuite le gradient de la moyenne empirique des log-vraisemblances négatives des différentes actions prises pondérées par leur fonction avantage et grâce à nos fonctions TensorFlow ci-dessous :

```
p1_grad = tf.placeholder(tf.float32)
p2_grad = tf.placeholder(tf.float32)
P = [p1_grad, p2_grad]
optimisation = tf.train.AdamOptimizer(learning_rate=10^-1)
update_grad = optimisation.apply_gradients(zip(P, [poids1, poids2]))
```

L'optimisation va se faire par descente de gradient grâce l'algorithme d'Adam et une fois les poids optimaux calculés ils seront appliqués au réseau.

Le fonctionnement général est ainsi posé, il suffit maintenant de lancer la boucle d'entraînement. On commence par définir nos tableaux qui vont accumuler les différentes observations, actions et récompenses de chaque épisode.

```
reward_sum = 0
observations_cumulees = np.empty(0).reshape(0,4)
action_y = np.empty(0).reshape(0,1)
rewards = np.empty(0).reshape(0,1)
```

On initialise nos poids aléatoirement, et on démarre notre environnement qui nous donne une première observation.

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
observation = env.reset()
```

Il reste plus qu'à lancer le réseau de neurones pour qu'il procède à l'apprentissage par renforcement.

```
gradients = np.array([np.zeros(var.get_shape()) for var in variables])
#On crée un tableau de la taille de nos poids.
episodes_max = 10000
num_episode = 0

while num_episode < episodes_max:
                x = np.reshape(observation, [1, 4])

                #On redimensionne les observations pour que les fonctions de TensorFlow
                #puisse les utiliser.

                #On rentre les observations dans notre réseau de neurones.
                #et celui-ci nous retourne la probabilité d'action.

                proba = sess.run(output, feed_dict={observations: x})

                #On détermine notre action y en fonction de cette probabilité.

                y = 0 if proba > np.random.rand() else 1

                #On stocke l'état du système et l'action associée.

                observations_cumulees = np.vstack([observations_cumulees, x])
                action_y = np.vstack([action_y, y])
```

La fonction "env.step()" va permettre d'appliquer à l'environnement notre action. Elle retourne 4 valeurs :
- l'état du système
- la récompense associée à l'action
- "done" qui indique si l'épisode est terminé si True
- des informations supplémentaires dont on se servira pas

```
                observation, reward, done, info = env.step(y)

                #On accumule les récompenses de l'épisode.

                reward_sum += reward
                rewards = np.vstack([rewards, reward])
```

Si l'épisode est terminé on va calculer la fonction "avantage" de chaque action de celui-ci, puis on calcule le gradient associé. On remet ensuite les accumulateurs d'observations, d'actions et de récompenses à zéros.

```
if done == True :

        A = fonction_avantage(rewards)
        A -= A.mean()
        A /= A.std()
        #On centre et on réduit A
        gradients += np.array(sess.run(grad,feed_dict={
        observations: observations_cumulees,input_y: action_y,avantages: A}))

        observations_cumulees = np.empty(0).reshape(0,4)
        action_y= np.empty(0).reshape(0,1)
        rewards = np.empty(0).reshape(0,1)
```

Si la taille de notre échantillon d'entraînement est atteint, on applique la descente de gradient et on met à jour nos poids synaptiques :

```
if num_episode % batch_size == 0:

                sess.run(update_grad, feed_dict={p1_grad: gradients[0],


                # On remet nos gradients de l'échantillon à zéro.

                gradients *= 0

                num_episode += 1

                #On relance l'environnement.

                observation = env.reset()
```

Ainsi le programme va apprendre à stabiliser le pendule au fur et à mesure des différents entraînements et de l'optimisation de ces poids. Les poids vont finir par converger et la récompense maximale pour cette configuration de réseau sera en moyenne atteinte pour chaque épisode. Ce réseau de neurones pourrait être améliorer en augmentant sa complexité : plus on augmente le nombre de neurones et le nombre de couches, plus l'algorithme va pouvoir déceler dans une certaine mesure les subtilités du modèle sous-jascent à la stabilité en position haute de notre pendule.