

Deep Reinforcement Learning

P. carré

philippe.carre@univ-poitiers.fr

Histoire

2013 : Small company DeepMind

“Playing Atari with Deep Reinforcement Learning” Arxiv.

How a computer learned to play Atari video games by observing just the screen pixels and receiving a reward when the game score increased.

1. The games and the goals in every game are very different.
2. The games are designed to be challenging for humans.

New



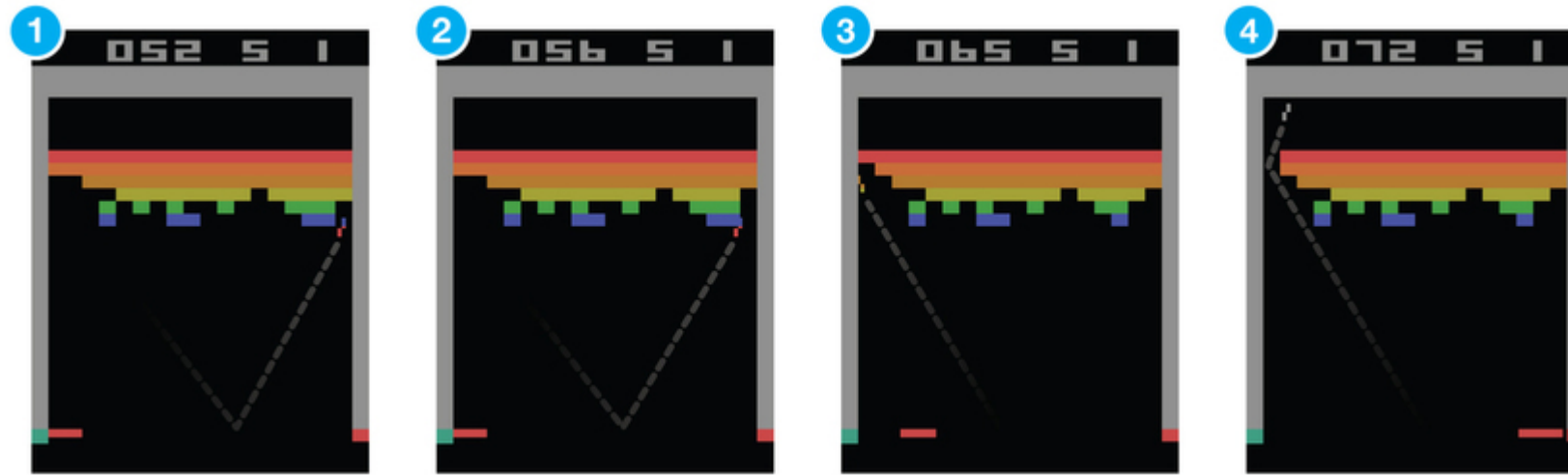
AI that can survive in a variety of environments,
instead of being confined such as playing chess.

Bought by Google

2015 : “Human-level control through deep reinforcement learning” cover of Nature

The same model to 49 different games , best results than human player

Example



Atari Breakout game. DeepMind.

Build a neural network to play this game

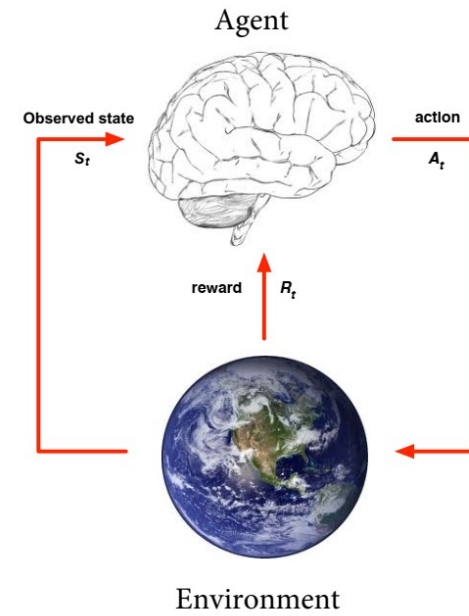
Input: screen images

Output: three actions: left, right or fire

Best strategy ?

A good strategy for an agent would be to always choose an action, that maximizes the discounted future reward.

Actuellement



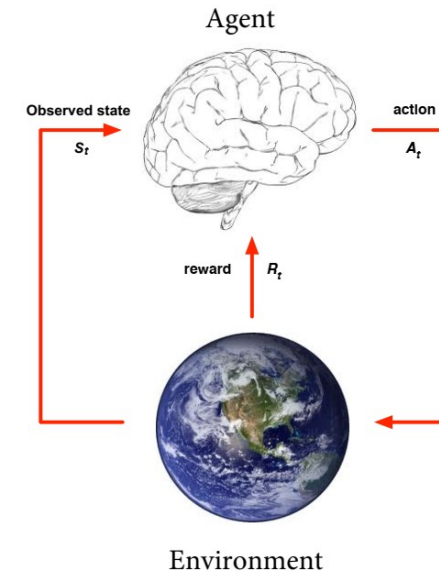
Apprentissage par renforcement

Principe

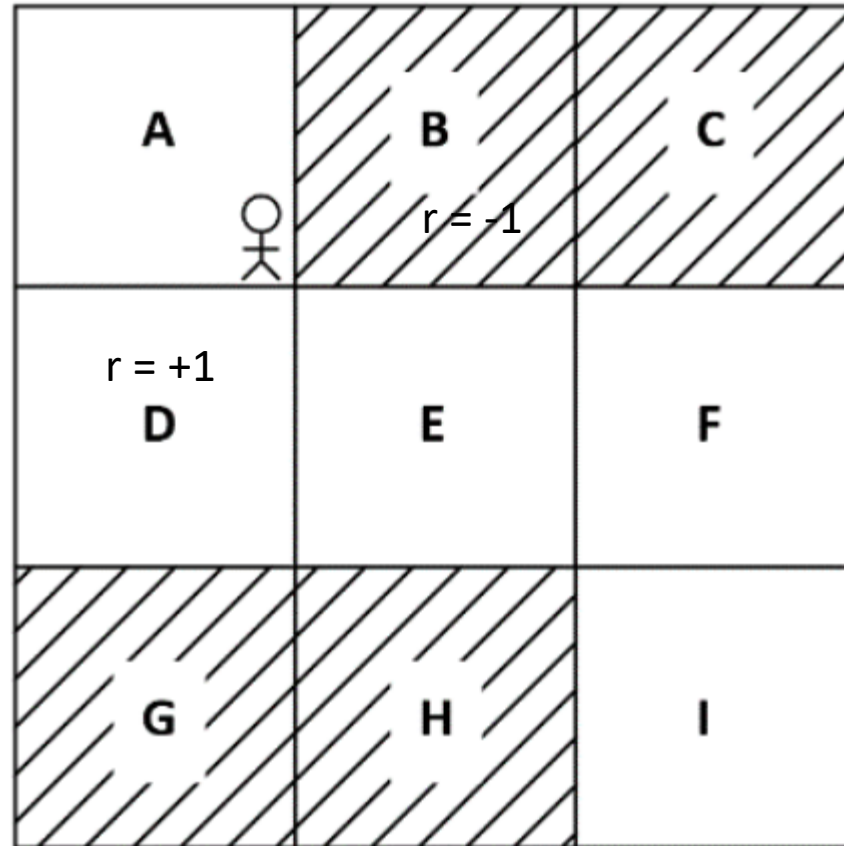
- Interaction entre un agent et son environnement
 - Optimisation de la récompense à long terme
 - Processus de décision Markovien
-
- L'agent observe l'univers :
 - Capteur, notion d'état
 - L'agent effectue une action :
 - Modification de l'état de l'univers
 - Au départ, ces modifications sont inconnues de l'agent

- Après chaque action, l'agent reçoit une récompense immédiate
- La récompense peut être positive, négative ou nulle
 - Au départ, ces récompenses sont en général inconnues de l'agent

Le but de l'agent est de maximiser sa récompense sur le long terme



Un exemple



Markov es-tu là ?

Une formalisation mathématique des problèmes de prise de décision séquentielle dont l'objectif est d'atteindre un but : lorsque l'agent effectue une action, cela influe non seulement la récompense immédiate, mais aussi celle engendrée par les futurs états.

Un processus décisionnel de Markov est un quadruplet $M = (\mathcal{S}, \mathcal{A}, P, \mathcal{R})$, où :

— \mathcal{S} est l'ensemble des états

— \mathcal{A} est l'ensemble des actions

— P est la matrice de probabilité de transition d'état

Probabilité que l'univers passe dans l'état s' quand l'action a est effectuée dans l'état s

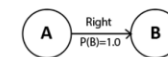
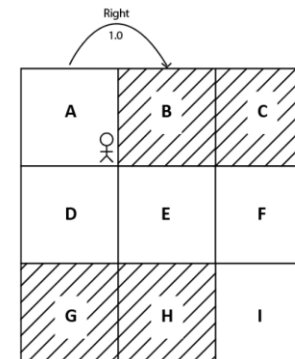
$$p(s'|s, a) = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$$

— \mathcal{R} est la fonction de récompense

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

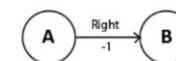
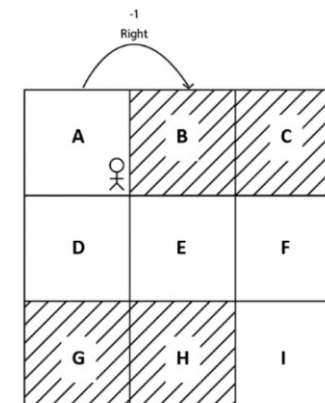
Récompense que l'on peut espérer à court terme

Hypothèse markovienne : la récompense et la fonction de transition ne dépendent que de l'état (et action) en cours, pas de l'historique

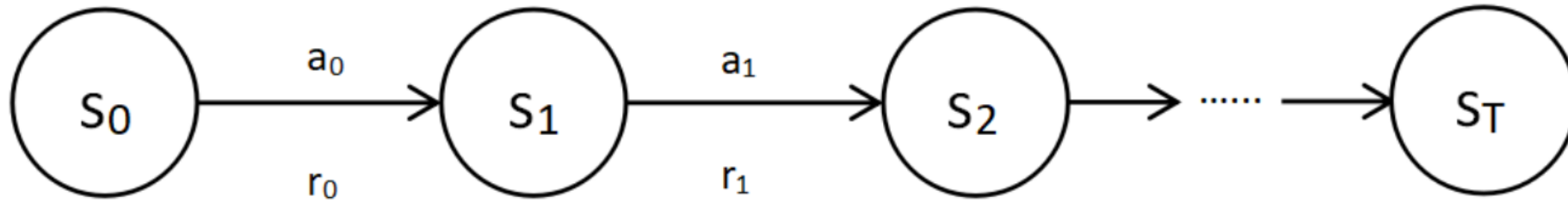


Récompense obtenue (réel)

$$r(s_t, a_t, s_{t+1})$$



Les gains cumulés



Une fois l'action A_t prise par l'agent, l'agent reçoit une valeur numérique, la récompense R_t qui peut être une variable aléatoire réelle. L'agent se retrouve alors dans un nouvel état S_{t+1} .

Tout au long de ce processus, l'agent décrit alors une trajectoire tr :
 $tr: S_0; A_0; R_0; S_1; A_1; R_1; S_2; A_2; R_2; S_3; A_3; R_3; \dots$

Le but de l'agent est de maximiser la quantité de récompenses obtenues au cours d'une trajectoire

L'agent cherche à maximiser l'espérance des récompenses à moyen terme, la fonction valeur qui correspond à l'agrégation des récompenses

Ce que l'on cherche à maximiser : récompense sur la trajectoire

$$G_t = r_t + r_{t+1} + r_{t+2} + \dots + r_T$$

Etat final

Les gains cumulés

Dans certains cas, il n'y a pas d'état terminal

$$G_t = \infty$$

Permet de pondérer les récompenses à long terme vs court terme
Convergence

$$0 \leq \gamma < 1$$

Récompense future avec rabais

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots = \sum_{\tau=t}^{+\infty} \gamma^{\tau-t} r_{\tau}$$

Si $\gamma = 0$ alors l'agent ne se concentrera que sur l'objectif de maximiser la récompense immédiate R_{t+1} et donc se restreindra à seulement apprendre à choisir de manière optimale A_t .

Si γ est proche de 1, alors la valeur du retour prend plus en compte les futures récompenses, l'agent devient donc plus prévoyant.

récompense sur la trajectoire à partir de l'itération t

Immédiatement après

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots = r_t + \gamma (r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots) = r_t + \gamma G_{t+1}$$

Notion de récurrence

Dans le futur

Des éléments clefs

A chaque instant, l'environnement envoie à l'agent un réel appelé la récompense immédiate. Le seul objectif de l'agent est de maximiser l'ensemble des récompenses obtenues lors d'une séquence.
Les récompenses permettent d'évaluer une politique.

La Politique : la façon de se comporter pour un agent face à une situation donnée à un instant donné.

Une application qui associe à une situation dans laquelle se trouve l'agent à un instant donné l'action qu'il doit faire.

$$\Pi(s) = a$$

Objectif : déterminer l'action permettant d'obtenir sur le long terme d'obtenir la plus grande valeur de récompenses cumulées.

Déterminer la valeur de la récompense immédiate est simple grâce à la fonction récompense.
Déterminer la valeur de la récompense cumulée pour un état est beaucoup plus complexe.

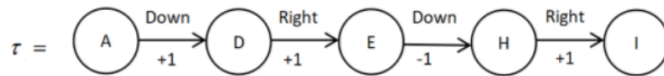
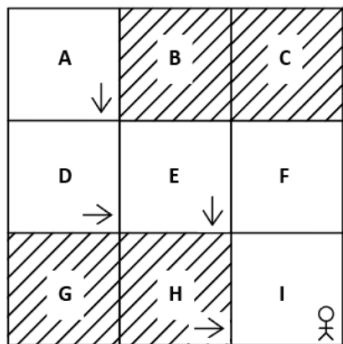
Fonction valeur d'état $v_{\pi} : \mathcal{S} \longrightarrow \mathbb{R}$

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid S_t = s] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s\right]$$

$E_{\pi}[\cdot]$ dénote l'espérance mathématique sur l'ensemble des réalisations du MDP en suivant la politique

La fonction valeur d'un état s par rapport à une politique π , notée $V^{\pi}(s)$ est l'espérance de G_t à partir de l'état s à l'étape temporelle t suivant la politique π .

Cette fonction indique quantitativement en quoi il est intéressant pour l'agent d'être dans cet état (étant donné une politique) dans le but de maximiser la somme des récompenses immédiates obtenues le long d'une trajectoire



$$V(A) = 2 ; V(D) = 1 ; V(E) = 0 ; V(H) = 1$$

Comment estimer

$$V^{\pi}(s) = E^{\pi}\left[\sum_{t=0}^{\infty} r_t \mid s_0 = s\right]$$

Monte-Carlo


Estimer

$$V^\pi(s) = E^\pi \left[\sum_{t=0}^{\infty} r_t \mid s_0 = s \right]$$

Une façon simple de réaliser cette estimation consiste à simuler des trajectoires à partir de chacun des états s jusqu'à l'état terminal T

Une estimation de la fonction de valeur V en s après $k + 1$ trajectoires

la somme cumulée obtenue le long de la trajectoire k en suivant la politique π

$$V_{k+1}(s) = \frac{G_1(s) + G_2(s) + \dots + G_k(s) + G_{k+1}(s)}{k + 1}$$


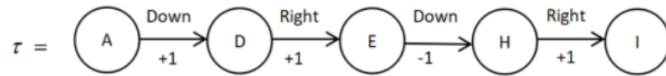
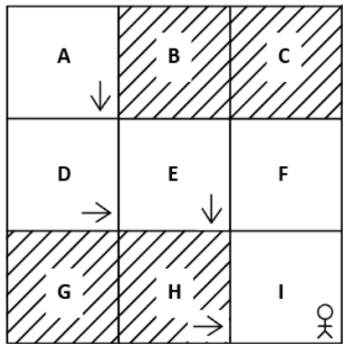
Cette fonction indique quantitativement en quoi il est intéressant pour l'agent d'être dans cet état

$v_\pi : \mathcal{S} \longrightarrow \mathbb{R}$ Dans cette expression il manque l'action, on ne sait pas ce que l'agent doit faire

La fonction valeur d'état/action

La fonction valeur d'état-action : la fonction quantifiant la qualité d'une action a prise à partir d'un état s par rapport à la politique π à l'étape temporelle t .

$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [G_t \mid S_t = s, A_t = a] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s, A_t = a \right]$$



$$q_{\pi} : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$$

Suivant cette politique $Q(A, \text{down}) = 2$

Probabilité que la politique nous fasse choisir a à partir de l'état s

Lien entre les deux valeurs



$$v_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(A_t = a \mid S_t = s) q_{\pi}(s, a) \quad \forall s \in \mathcal{S}$$

Politique optimale

Rechercher parmi une famille de politiques celle qui optimise un critère de performance donné pour le processus décisionnel markovien considéré



Cela revient toujours à évaluer une politique sur la base d'une mesure du *cumul espéré* des récompenses instantanées le long d'une trajectoire

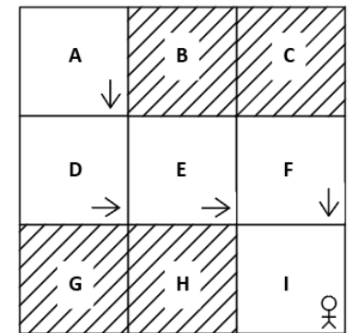
– le critère γ -pondéré : $E[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots + \gamma^t r_t + \dots \mid s_0]$

– le critère total : $E[r_0 + r_1 + r_2 + \dots + r_t + \dots \mid s_0]$

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

Optimal Policy

| State | Action |
|-------|--------|
| A | Down |
| D | Right |
| E | Right |
| F | Down |



Meilleure politique : apprentissage par renforcement

- Apprentissage supervisé, la meilleure réponse possible est fournie à l'apprenant (pas besoin de la rechercher)
- L'apprentissage par renforcement : lorsqu'il reçoit un premier signal d'évaluation, l'apprenant ne sait toujours pas si la réponse qu'il a donnée est la meilleure possible ; il doit essayer d'autres réponses pour déterminer s'il peut recevoir une meilleure évaluation.

Meilleure politique : Equation de Bellman

Rappel

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots = r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots) = r_t + \gamma G_{t+1}$$

Simplification : processus déterministe

Court-terme

A partir de l'état suivant

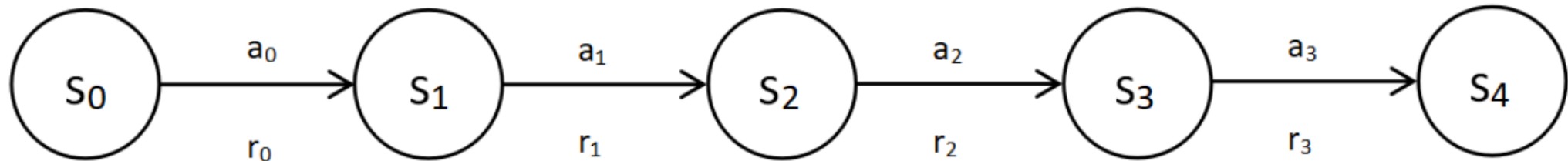


$$Q^\pi(s, a) = R(s, a, s') + \gamma Q^\pi(s', a')$$

De même pour $V(s)$

$$V(s) = R(s, a, s') + \gamma V(s')$$

Pour une politique



$$Q(s_0, a_0) = r_0 + \gamma Q(s_1, a_1)$$

Meilleure politique : Equation de Bellman

Dans un environnement stochastique

Probabilité que l'univers passe dans l'état s' quand l'action a est effectuée dans l'état s

$$Q^\pi(s, a) = \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma Q^\pi(s', a')]$$

Somme sur toutes les possibilités


Notion d'optimalité

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

Equation d'optimalité de Bellman

Permet de déterminer la meilleure stratégie

On choisit l'action qui maximise Q


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

Problème : on ne connaît pas les probabilités de transition et on ne connaît pas les récompenses apriori.

Equation de Bellman : Temporal Difference learning

Comment estimer $Q(s,a)$?

$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [G_t \mid S_t = s, A_t = a]$$



Somme sur les trajectoires

Une observation (s,a,s',a') : $Q^{\pi}(s, a) = R(s, a, s') + \gamma Q^{\pi}(s', a')$

Notion de moyenne incrémentale : on construit une estimation pas à pas

1. Initialize a Q function $Q(s, a)$ with random values
2. For each episode:
 1. Initialize state s
 2. Extract a policy from $Q(s, a)$ and select an action a to perform in state s
 3. For each step in the episode:
 1. Perform the action a and move to the next state s' and observe the reward r
 2. In state s' , select the action a' using the epsilon-greedy policy
 3. Update the Q value to
 $Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$
 4. Update $s = s'$ and $a = a'$ (update the next state s' -action a' pair to the current state s -action a pair)
 5. If s is not a terminal state, repeat steps 1 to 5

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

Diagram illustrating the incremental update of the Q function:

- Estimation de $Q(s,a)$** : Points to the $Q(s, a)$ term on the right side of the equation.
- Valeur actuelle**: Points to the $Q(s, a)$ term on the left side of the equation.
- Coef d'apprentissage**: Points to the α term.
- $s,a \rightarrow s',r \rightarrow a'$** : Points to the $r + \gamma Q(s', a')$ term, representing the new information from the transition.

Q-Learning : recherche de la politique optimale

Le Q-learning consiste à déterminer une fonction (une table) $Q(s, a)$ qui prend deux paramètres :

- s : L'état du système
- a : l'action que l'on veut effectuer

Et cette fonction renvoie une valeur, qui est la récompense potentielle que l'on obtiendra à long terme en choisissant cette action.

$Q(s,a)$ "the best possible score at the end of game after performing action a in state s "

Equation de Bellman



$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$$

On peut exprimer la Q-value d'un état s et d'une action a suivant la Q-value de l'état suivant s'

$$Q^\pi(s, a) = R(s, a, s') + \gamma Q^\pi(s', a') \longrightarrow Q(s, a) = r_n + \gamma \max_{a'} Q(s_{n+1}, a')$$

Q-learning : algorithme

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]$$

Ce que l'on obtient à court terme Ce que l'on obtient « juste » après (stratégie gloutonne) Ce que l'on pense obtenir

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmax_a' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

Après construction de la table : politique

Exploration – renforcement : Epsilon-Greedy

L'agent est dans un certain état s , on choisit une action a selon :

La meilleure avec une probabilité $1 - \epsilon$

Au hasard avec une probabilité ϵ

compromis entre exploration et exploitation

$$\epsilon = N_{parties} / (N_{parties} + iteration)$$

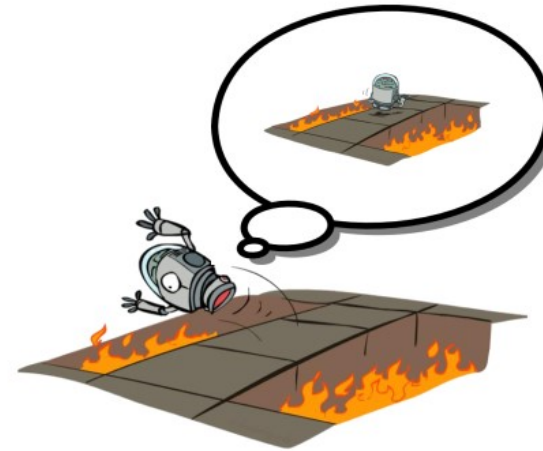
Bilan

Politique $\pi(s) \leftarrow a \in \mathcal{A}(s)$ qui maximise $Q(s, a)$

A Faire

- Régler les états
- Régler les actions
- Et surtout régler les récompenses !

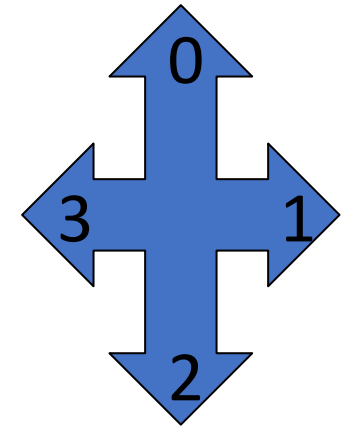
On doit pouvoir jouer : Exploration



Projet Partie 1

Réalisation d'un algorithme de Q-learning

| | | | |
|-----------------------|-----------------------|-----------------------|--------------------------------------|
| Start | | | |
| Dragon Go to Start | | Dragon Go to Start | |
| | | Reward = 0 | Dragon Reward = -1 Go to Start |
| | Dragon Go to Start | | Jail Reward = 1 Go to Start |



Pour un environnement réaliser l'apprentissage par Renforcement

- Faire jouer de nombreuses parties (par exemple 10000)
- Pour chaque partie, limiter le nombre de cout (100)
- Construire l'estimation de la politique optimale

L'agent est dans un certain état s , on choisit une action a selon :

Q 16 lignes, 4 colonnes

Au hasard avec une probabilité ϵ

La meilleure avec une probabilité $1-\epsilon$

$$\epsilon = N_{parties} / (N_{parties} + iteration)$$

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha [r_n + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]$$

Ce que l'on obtient à court terme

Ce que l'on obtient « juste » après (stratégie gloutonne)

Ce que l'on pense obtenir

 **A Faire**

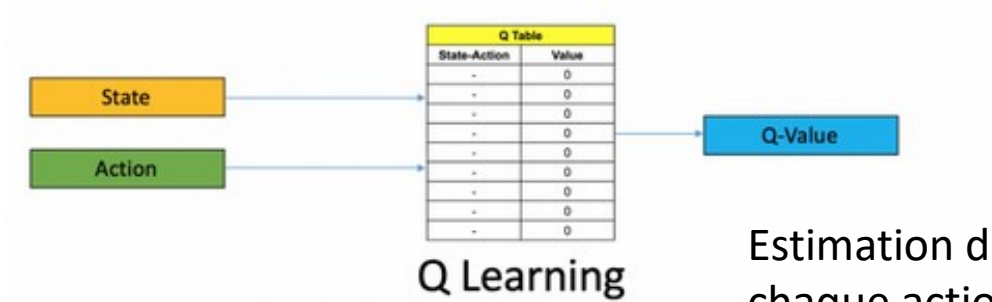
Jouer sur le Reward pour optimiser le déplacement

Deep Q Network ?

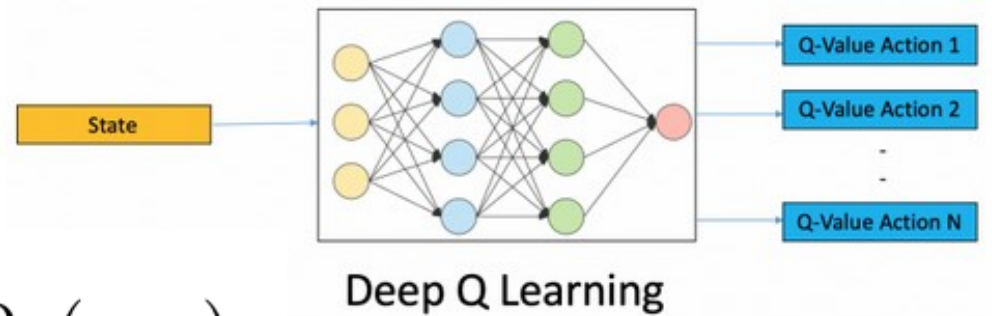
Q-learning

| State | Action | Value |
|-------|--------|-------|
| A | up | 17 |
| A | down | 10 |
| B | up | 11 |
| B | down | 20 |

Lorsque le nombre d'état
devient trop grand

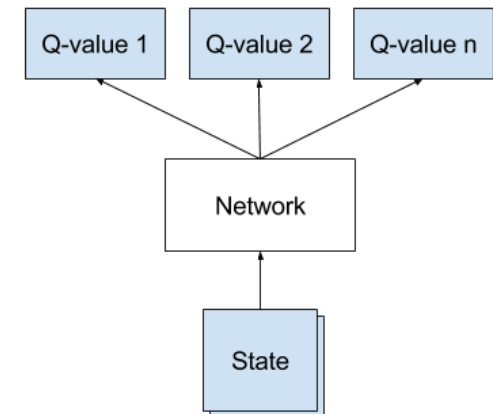


Estimation de Q pour
chaque action possible



$$Q_{\theta}(s, a)$$

Paramètres du RNN

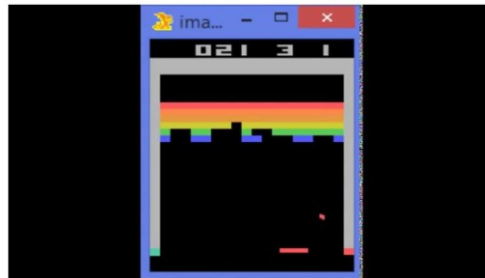
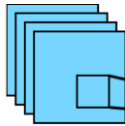


Deep QL : contexte



Nombre d'états ... infini

Input:
4 images = frame
courante + 3
précédentes



2) Output: $Q(s, a_i)$

$Q(s, a_1)$

$Q(s, a_2)$

$Q(s, a_3)$

.

.

.

.

.

.

.

.

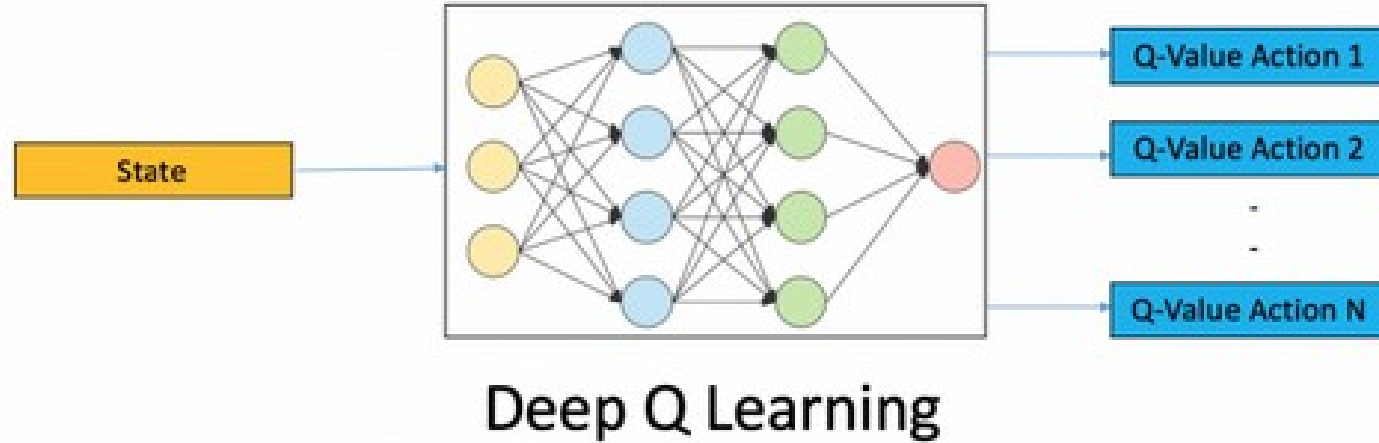
.

.

$Q(s, a_{18})$



Structure neuronale : comment apprendre ?



Problème de régression
Estimer la valeur $Q(s,a)$

classiquement

$$\text{MSE} = \frac{1}{K} \sum_{i=1}^K (y_i - \hat{y}_i)^2$$

Valeur prédite (pointing to \hat{y}_i)
Valeur cible (pointing to y_i)

Valeur cible : $Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a')$

Valeur prédite : sortie du réseau $Q_\theta(s, a)$

Fonction loss

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

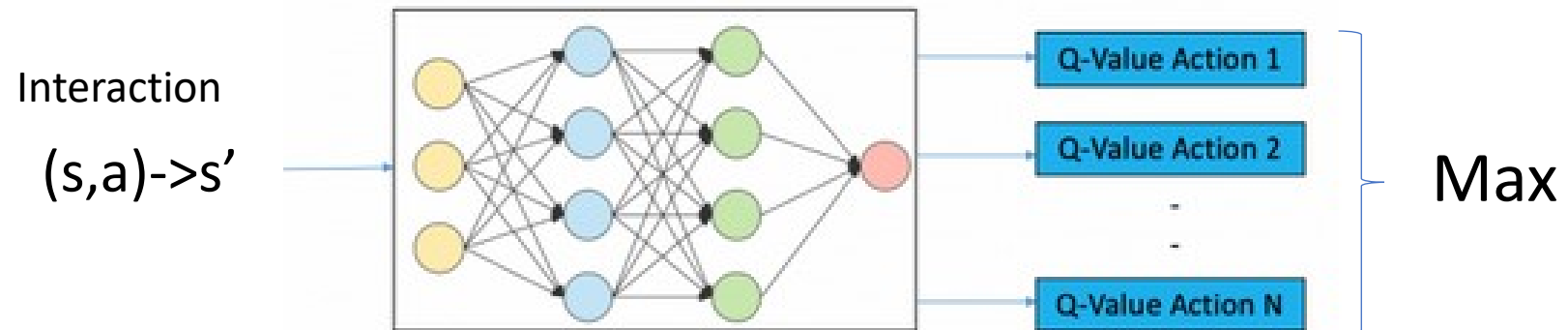
Comment calculer ?

Structure neuronale : fonction loss

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

↓ Utilisation du réseau pour déterminer $Q(s', a')$

$$L(\theta) = r + \gamma \max_{a'} Q_{\theta}(s', a') - Q_{\theta}(s, a)$$



$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \max_{a'} Q_{\theta}(s'_i, a') - Q_{\theta}(s_i, a_i))^2$$

Apprentissage

$$\longrightarrow \theta = \theta - \alpha \nabla_{\theta} L(\theta)$$

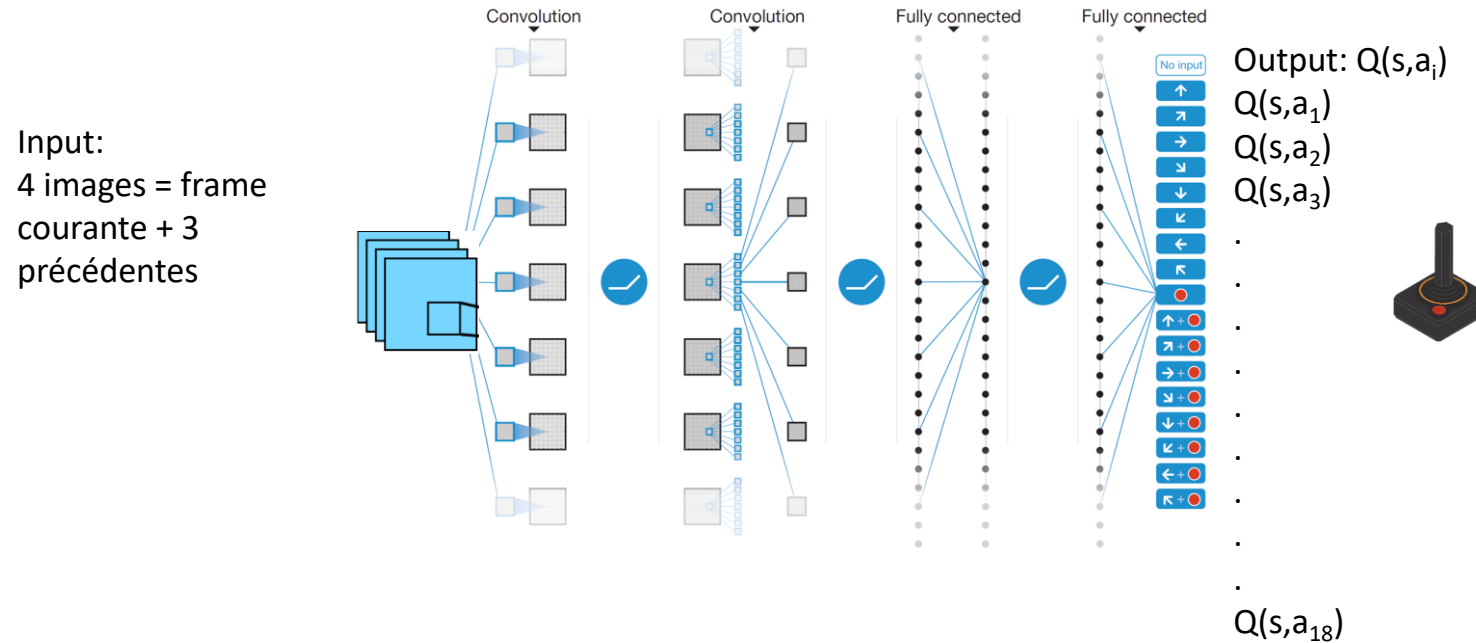
Deep

Input network: 84×84 grayscale game screens.

Outputs network: Q-values for each possible action (18 in Atari).

Q-values can be any real values

| Layer | Input | Filter size | Stride | Num filters | Activation | Output |
|-------|----------|-------------|--------|-------------|------------|----------|
| conv1 | 84x84x4 | 8x8 | 4 | 32 | ReLU | 20x20x32 |
| conv2 | 20x20x32 | 4x4 | 2 | 64 | ReLU | 9x9x64 |
| conv3 | 9x9x64 | 3x3 | 1 | 64 | ReLU | 7x7x64 |
| fc4 | 7x7x64 | | | 512 | ReLU | 512 |
| fc5 | 512 | | | 18 | Linear | 18 |

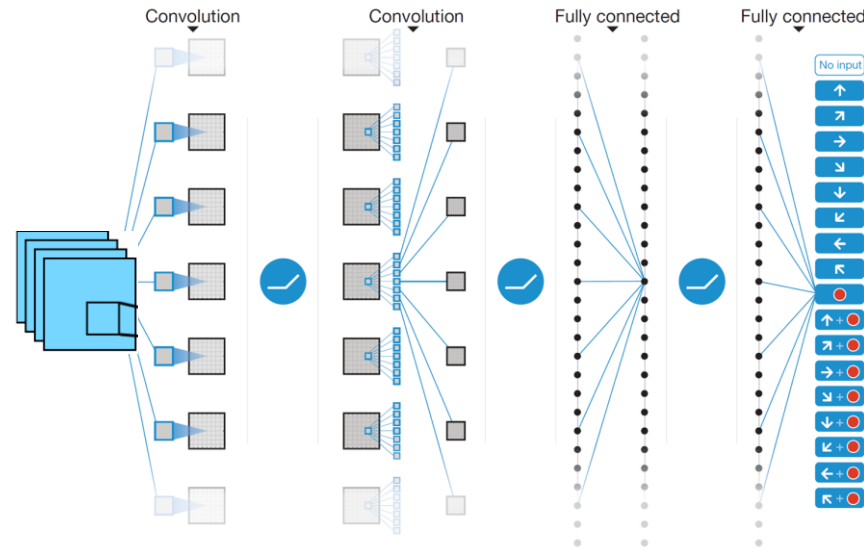


Deep Q Network

Cost

$$L = \frac{1}{2} [\underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Input:
4 images = frame
courante + 3
précédentes



Output: $Q(s, a_i)$

$Q(s, a_1)$

$Q(s, a_2)$

$Q(s, a_3)$

...

...

...

...

...

...

$Q(s, a_{18})$



1. Do a feedforward pass for the current state s to get predicted Q-values for all actions. **Choose an action using the Epsilon-Greedy Exploration Strategy**
2. Do a feedforward pass for the next state s' and calculate maximum over all network outputs $\max_{a'} Q(s', a')$.
3. Set Q-value target for action a to $r + \gamma \max_{a'} Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

Ce que l'on pense obtenir

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha [r_n + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]$$

Ce que l'obtient à court terme

Ce que l'on obtient

« juste » après (stratégie gloutonne)

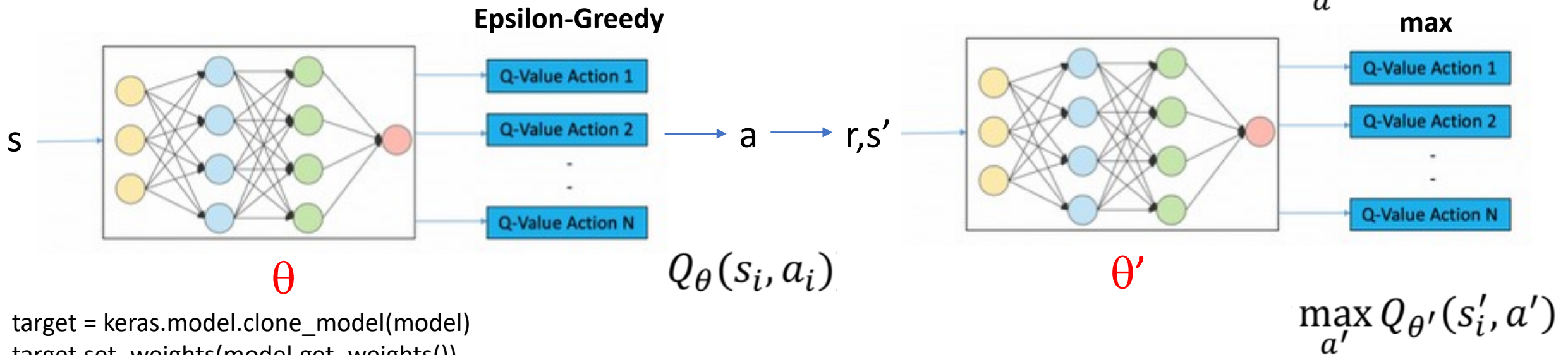
Evolution 1 : target NN

$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \overset{\text{target}}{\max_{a'} Q_{\theta}(s'_i, a')} - \overset{\text{prédite}}{Q_{\theta}(s_i, a_i)})^2$$

Problème : même structure

Entraine des phénomènes de divergence

➔ Solution : figer la valeur « target » pendant un certain nombre d'itérations



```
target = keras.model.clone_model(model)
target.set_weights(model.get_weights())
```

...

```
Q_nplus1 = target.predict(statesn+1)
```

...

```
If episode % nbre == 0:
```

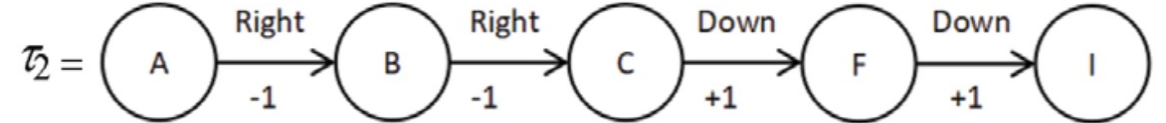
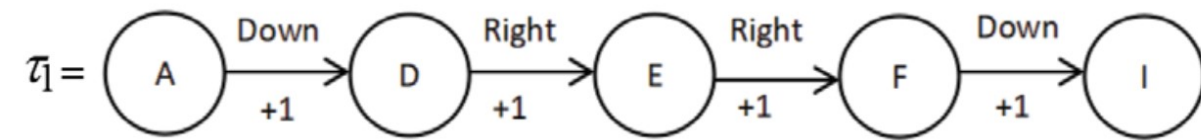
```
    target.set_weights(model.get_weights())
```

$$\text{RMQ : } \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_{\theta}(s'_i, a') & \text{if } s' \text{ is not terminal} \end{cases}$$

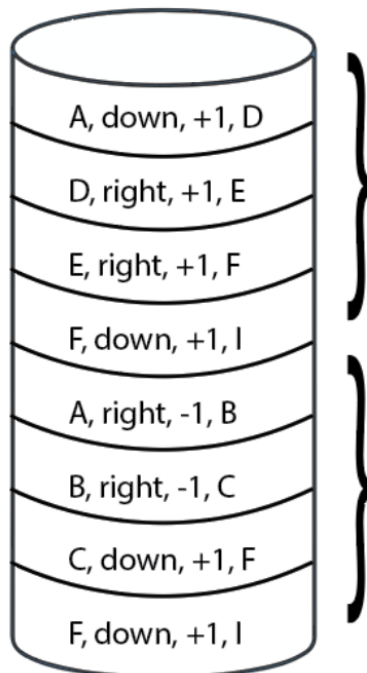
Evolution 2 : Replay Buffer

Une trajectoire : une suite de (s,a,r,s')

Construire un ensemble d'apprentissage



Enregistrement
des transitions



τ_1
Episode 1

τ_2
Episode 2



Choix au hasard de transition

Fonctionnement en FIFO

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

With probability ε select a random action a_t

otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

where ϕ preprocess last 4 image frames to represent the state.

AlphaGo

$$\frac{\partial \log p_{\sigma}(a | s)}{\partial \sigma}$$

$$\frac{\partial \log p_{\rho}(a_t | s_t)}{\partial \rho} z_t$$

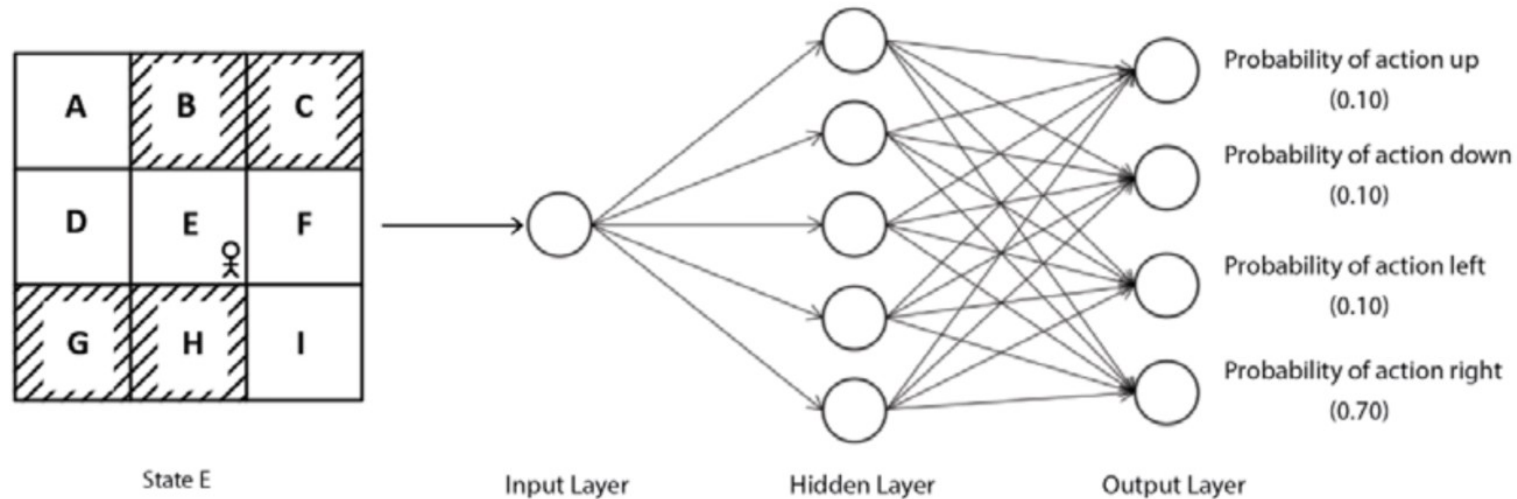
Policy Gradient

Pourquoi évaluer une valeur Q pour ensuite en déduire une politique ?

Estimer directement la meilleure politique

Repose en général sur une vision stochastique (notion de probabilité de meilleure action)

Principe général



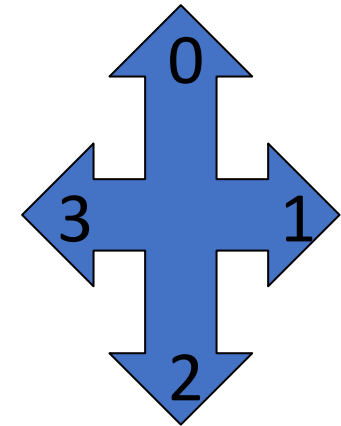
Quelques éléments d'algorithmes

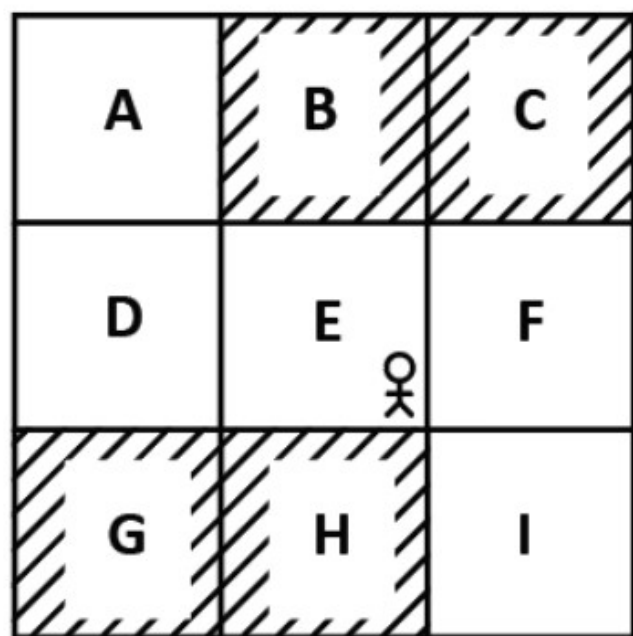
1. Initialize the main network parameter θ with random values
2. Initialize the target network parameter θ' by copying the main network parameter θ
3. Initialize the replay buffer \mathcal{D}
4. For N number of episodes, perform *step 5*
5. For each step in the episode, that is, for $t = 0, \dots, T-1$:
 1. Observe the state s and select an action using the epsilon-greedy policy, that is, with probability epsilon, select random action a and with probability $1-\text{epsilon}$, select the action $a = \arg \max_a Q_\theta(s, a)$
 2. Perform the selected action and move to the next state s' and obtain the reward r
 3. Store the transition information in the replay buffer \mathcal{D}
 4. Randomly sample a minibatch of K transitions from the replay buffer \mathcal{D}
 5. Compute the target value, that is, $y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a')$
 6. Compute the loss, $L(\theta) = \frac{1}{K} \sum_{i=1}^K (y_i - Q_\theta(s_i, a_i))^2$
 7. Compute the gradients of the loss and update the main network parameter θ using gradient descent: $\theta = \theta - \alpha \nabla_\theta L(\theta)$
 8. Freeze the target network parameter θ' for several time steps and then update it by just copying the main network parameter θ

Projet Partie 2

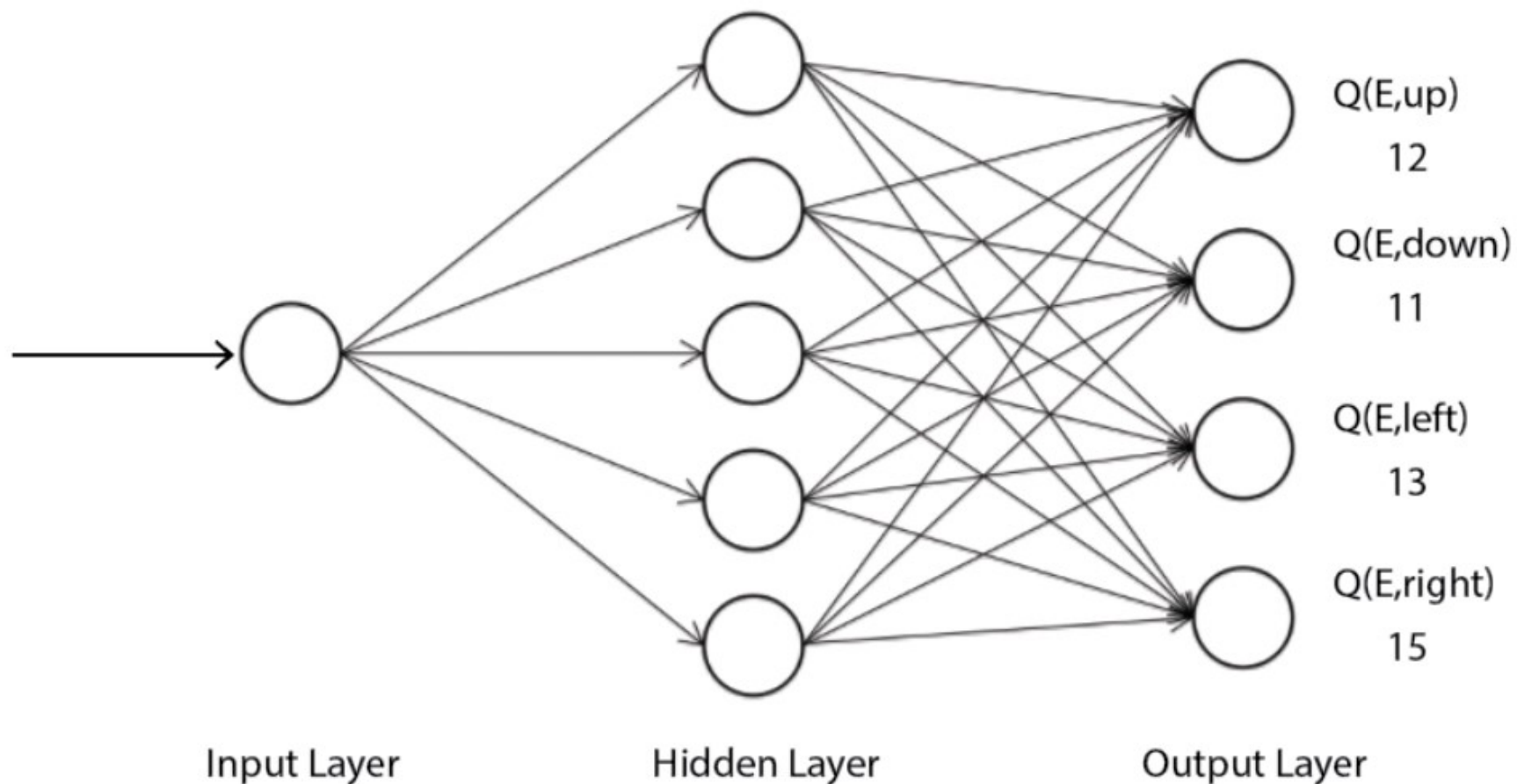
Réalisation d'un algorithme de Deep Q-learning

| | | | |
|-----------------------|-----------------------|-----------------------|--------------------------------------|
| Start | | | |
| Dragon Go to Start | | Dragon Go to Start | |
| | | Reward = 0 | Dragon Reward = -1 Go to Start |
| | Dragon Go to Start | | Jail Reward = 1 Go to Start |





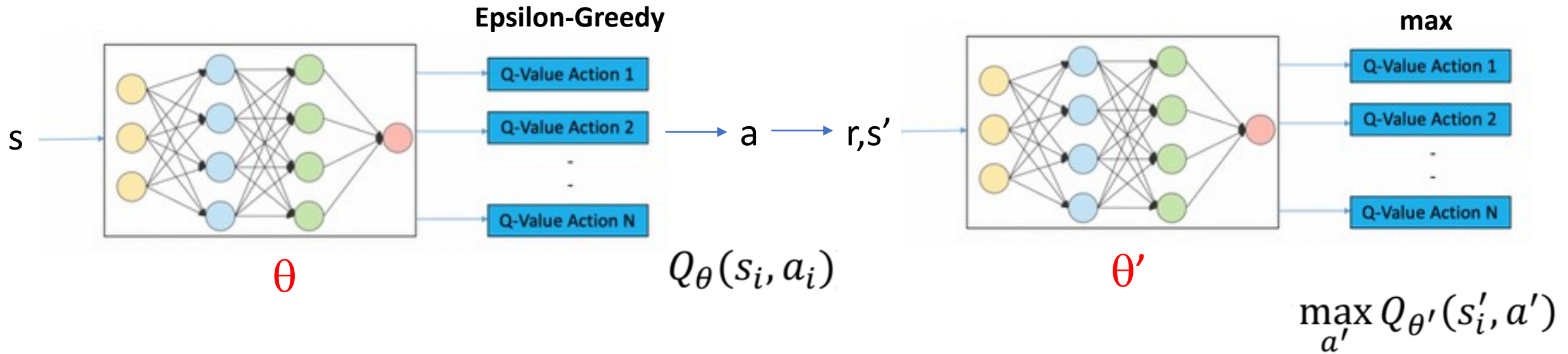
State E



TD Machine 2

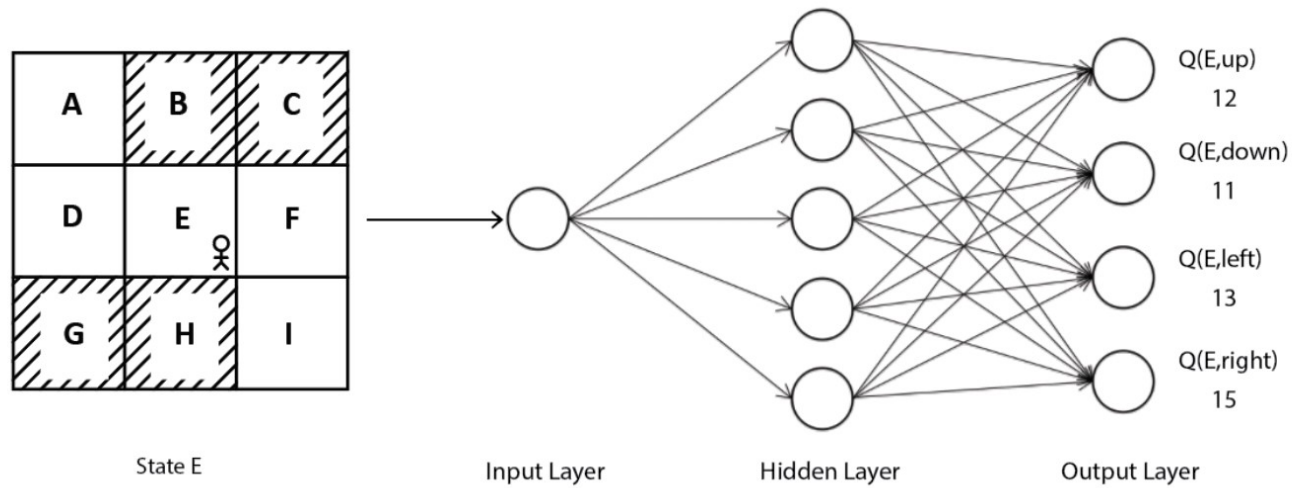
Relecture Deep Q-learning

$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \overset{\text{target}}{\max_{a'} Q_{\theta'}(s'_i, a')} - \overset{\text{prédite}}{Q_{\theta}(s_i, a_i)})^2$$



$$\text{RMQ: } \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_{\theta}(s'_i, a') & \text{if } s' \text{ is not terminal} \end{cases}$$

Relecture du projet Deep Q-learning



Créer en entrée un vecteur de taille $N \times N$ avec comme nombre de sorties le nombre d'action possible

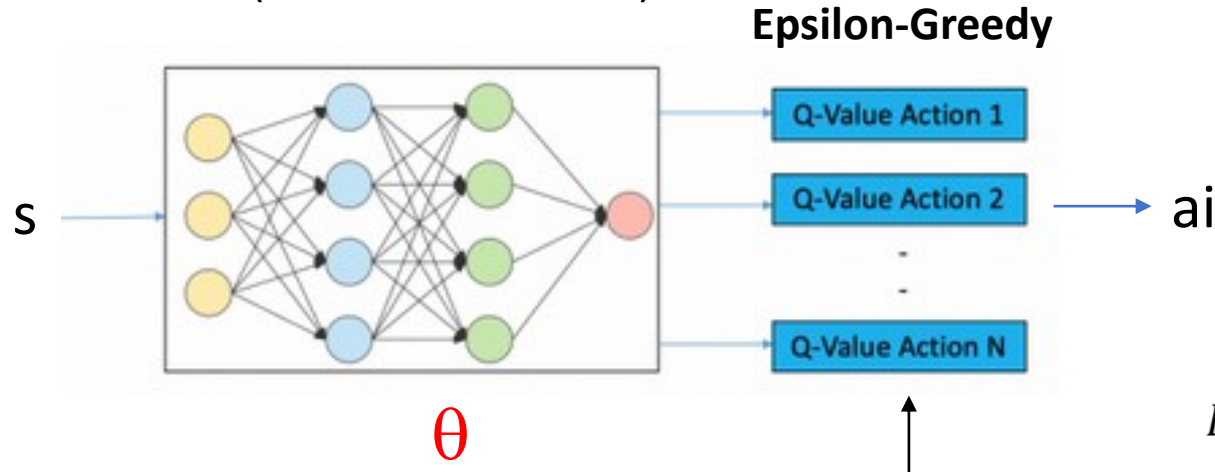
```
vec_etat = np.zeros(16)
vec_etat[int(N*position[0] + position[1])] = 1
```

Une structure simple avec 16 entrées et 4 sorties, la sortie est sans activation

```
model = keras.models.Sequential([
    keras.layers.Dense(4, activation='relu', input_shape = [16]),
    keras.layers.Dense(4, activation='relu'),
    keras.layers.Dense(4)
])
```

A chaque itération

On est dans un certain état s (vecteur de taille 16)



On va optimiser en fonction de

$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a') - Q_{\theta}(s_i, a_i))^2$$

target prédite

L'agent est dans un certain état s , on choisit une action a selon :

- Au hasard avec une probabilité ϵ
- La meilleure avec une probabilité $1 - \epsilon$

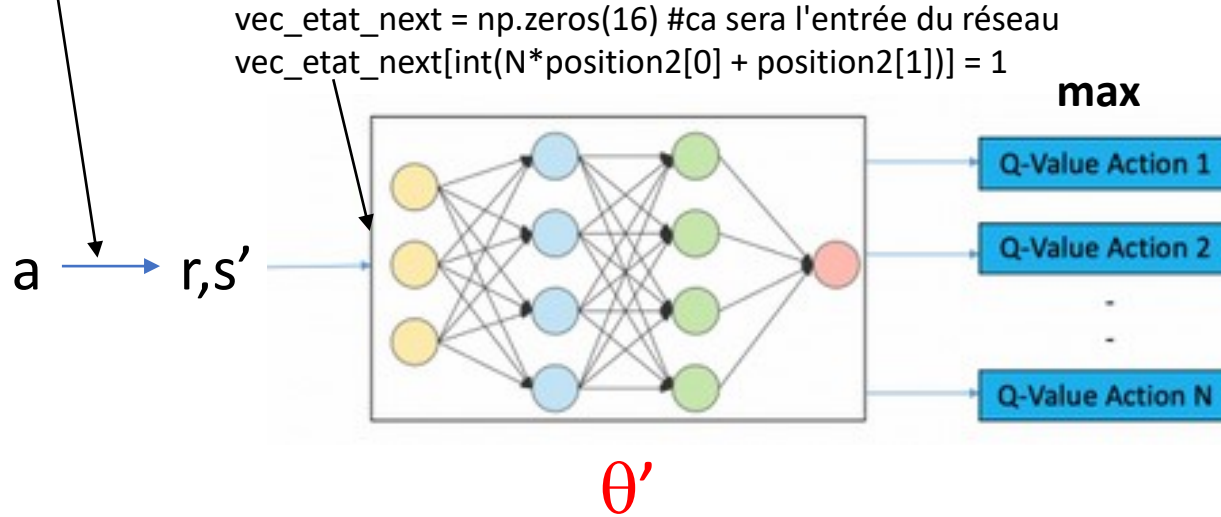
Taille (1,16)

```
Sortie_Q = model.predict(np.array([vec_etat])) #En entrée le vecteur symbolisant l'état  
action = np.argmax(Sortie_Q[0]) #On sélectionne l'action associée avec la sortie max
```

Taille (1,4)

A chaque itération

```
position2,state2,Reward,fin = application_action(action,position,Space)
```



En préambule création d'un second modèle

```
model_stable =  
keras.models.clone_model(model)
```

```
model_stable.set_weights(model.get_weights  
( ))
```

$$\max_{a'} Q_{\theta'}(s'_i, a')$$

```
next_Q = model_stable.predict(np.array([vec_etat_next]))  
next_Q_max = np.max(next_Q[0])
```

$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \underbrace{\max_{a'} Q_{\theta'}(s'_i, a')}_{\text{target}} - \underbrace{Q_{\theta}(s_i, a_i)}_{\text{prédite}})^2$$

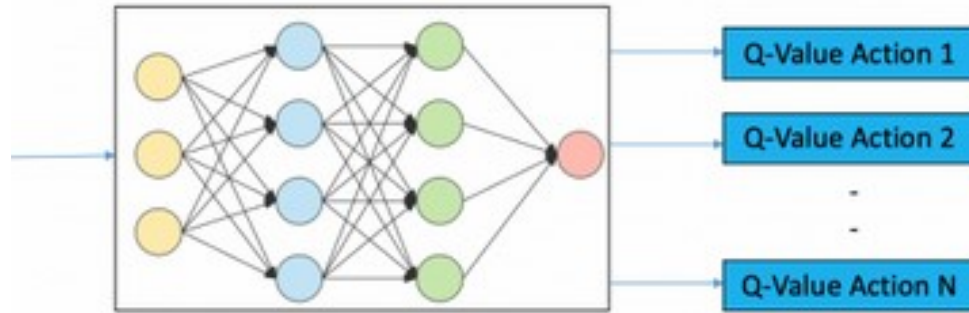


target = Reward + gamma * next_Q_max * (1-fin)

A chaque itération

Reste une opération

Optimiser les paramètres du réseau θ



En fonction de

$$(r_i + \gamma \max_{a'} \overset{\text{target}}{Q_{\theta'}(s'_i, a')} - \overset{\text{prédite}}{Q_{\theta}(s_i, a_i)})^2$$

target = Reward + gamma * next_Q_max * (1-fin)

Nous ne pouvons pas utiliser *model.fit* car il faut pouvoir réaliser la descente de gradient avec une fonction *loss* particulière

Interlude

Comment programmer une descente de gradient avec TF/Keras

On veut réaliser l'apprentissage d'une structure perceptron simple

Définition de la méthode d'optimisation et de la fonction de cout

```
X_train, X_test, Y_train, Y_test = train_test_split(data, labels_true, test_size=0.2, random_state = 1, stratify = labels_true)
```

Version simple on-line avec présentation aléatoire des individus à chaque epoch

```
model = keras.models.Sequential([
    keras.layers.Dense(4, activation='sigmoid', input_shape = [2]),
    keras.layers.Dense(4, activation='sigmoid'),
    keras.layers.Dense(1, activation='sigmoid')
])
```

```
optimizer = keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = keras.losses.mean_squared_error
```

```
list_idx = np.arange(0, len(X_train))
for epoch in range(n_epochs):
    som_loss = 0
    random.shuffle(list_idx)
    for idx in list_idx:
        x = np.array([X_train[idx, 0:2]])
        y = Y_train[idx]
```

Pour un individu x

calcul de la sortie prédite

calcul de l'erreur

calcul de la rétropropagation du gradient

modification des poids

Somme cumulée de l'erreur

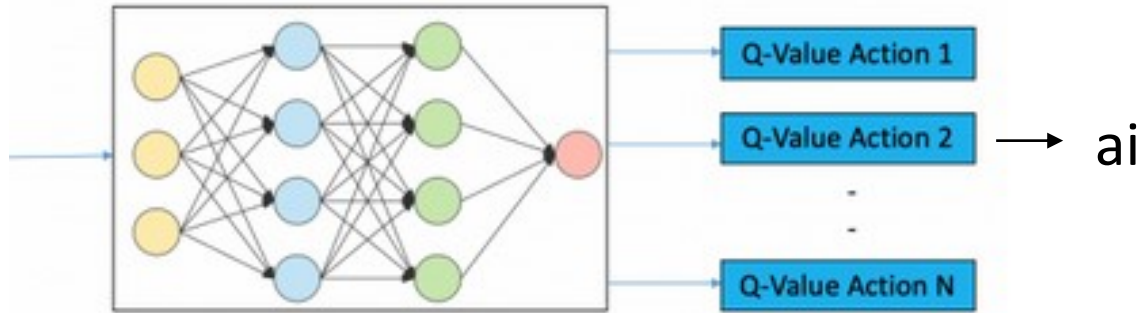
```
with tf.GradientTape() as tape:
    y_pred = model(x)
    loss = loss_fn(y, y_pred)
gradients = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(gradients, model.trainable_variables))
som_loss += loss
```

↓
Associe à chaque variable son « erreur »

A chaque itération

Optimiser les paramètres du réseau θ

En fonction de



$$(r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a') - Q_{\theta}(s_i, a_i))^2$$

target prédite

```
x = np.array([vec_etat])
```

```
with tf.GradientTape() as tape:
```

```
    predict = model(x) #Ce que l'on pense obtenir
```

pb predict contient les valeurs pour toutes les actions
seule l'action "action" nous intéresse

```
    mask = tf.one_hot(action,nbre_action)
```

```
    val_predict = tf.reduce_sum(predict*mask,axis = 1)
```

```
    loss = loss_fn(target,val_predict)
```

$$(r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a') - Q_{\theta}(s_i, a_i))^2$$

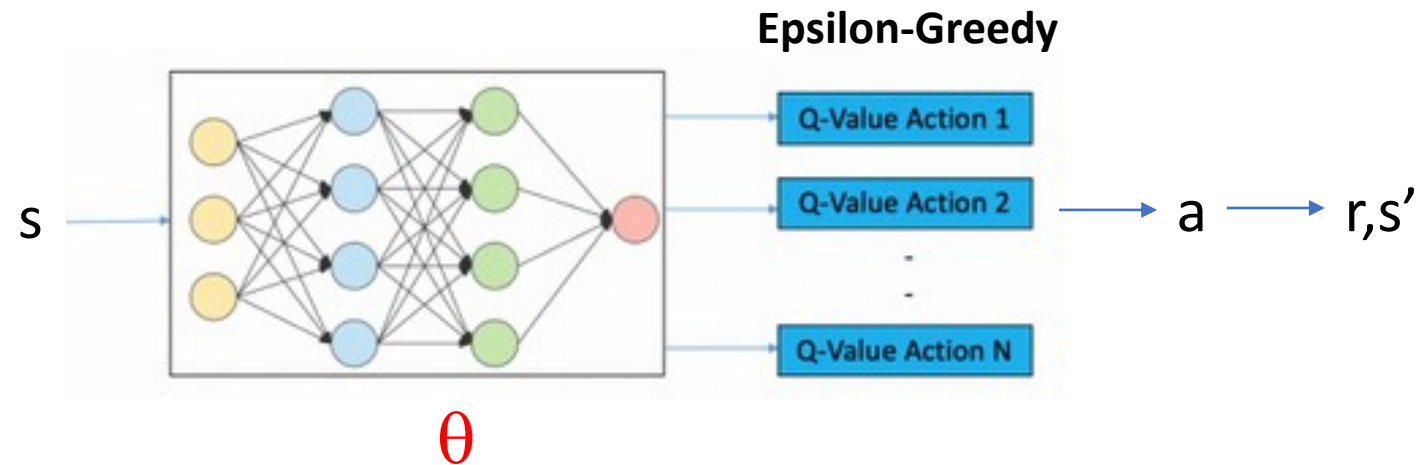
target prédite

```
    gradients = tape.gradient(loss,model.trainable_variables)
```

```
    optimizer.apply_gradients(zip(gradients,model.trainable_variables))
```

Optimisation des paramètres

A chaque itération



On va se positionner à l'état s' et on poursuit

Ne pas oublier : figer la valeur « target » pendant un certain nombre d'itérations (par exemple 10)

```
if t % 10 == 0:  
    model_stable.set_weights(model.get_weights())
```

Après apprentissage on peut jouer

Tant que non(fin)

```
vec_etat = np.zeros(16)
```

```
vec_etat[int(N*position[0] + position[1])] = 1
```

```
Sortie_Q = model.predict(np.array([vec_etat]))
```

```
action = np.argmax(Sortie_Q[0])
```

```
position2,state2,Reward,fin =
```

```
application_action(action,position,Space)
```

```
position = position2
```