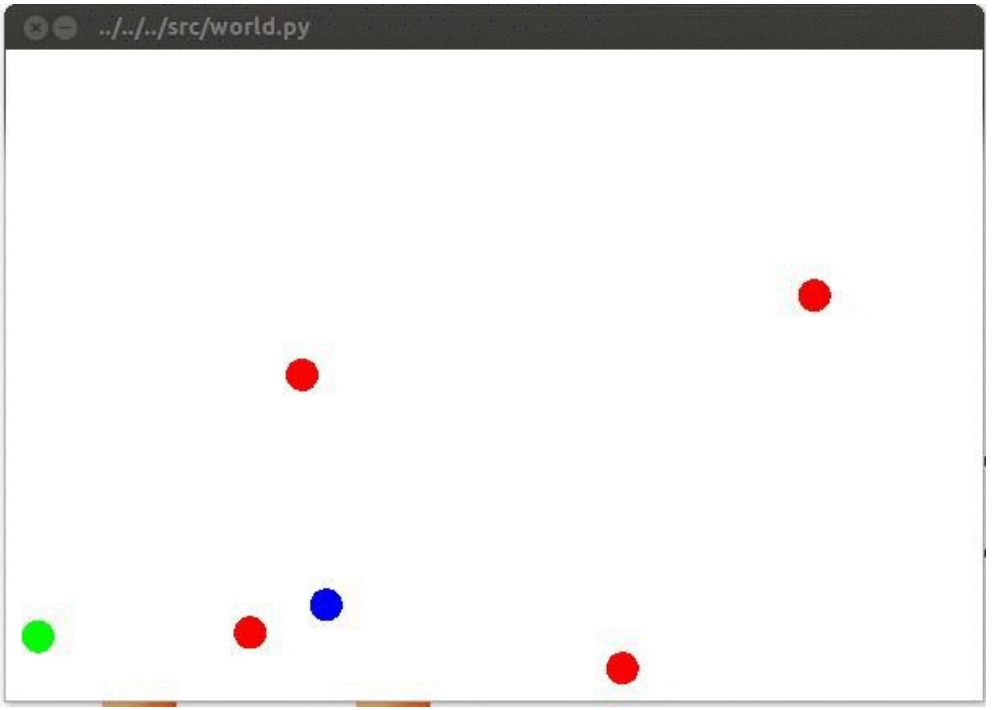
Étude sur l’apprentissage par renforcement dans les jeux

Projet de Bachelor 2018 - hepia

Étudiant : Federico Pfeiffer Professeur : Guido Bologna



h e p i a
Haute école du paysage, d’ingénierie
et d’architecture de Genève

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Plan :

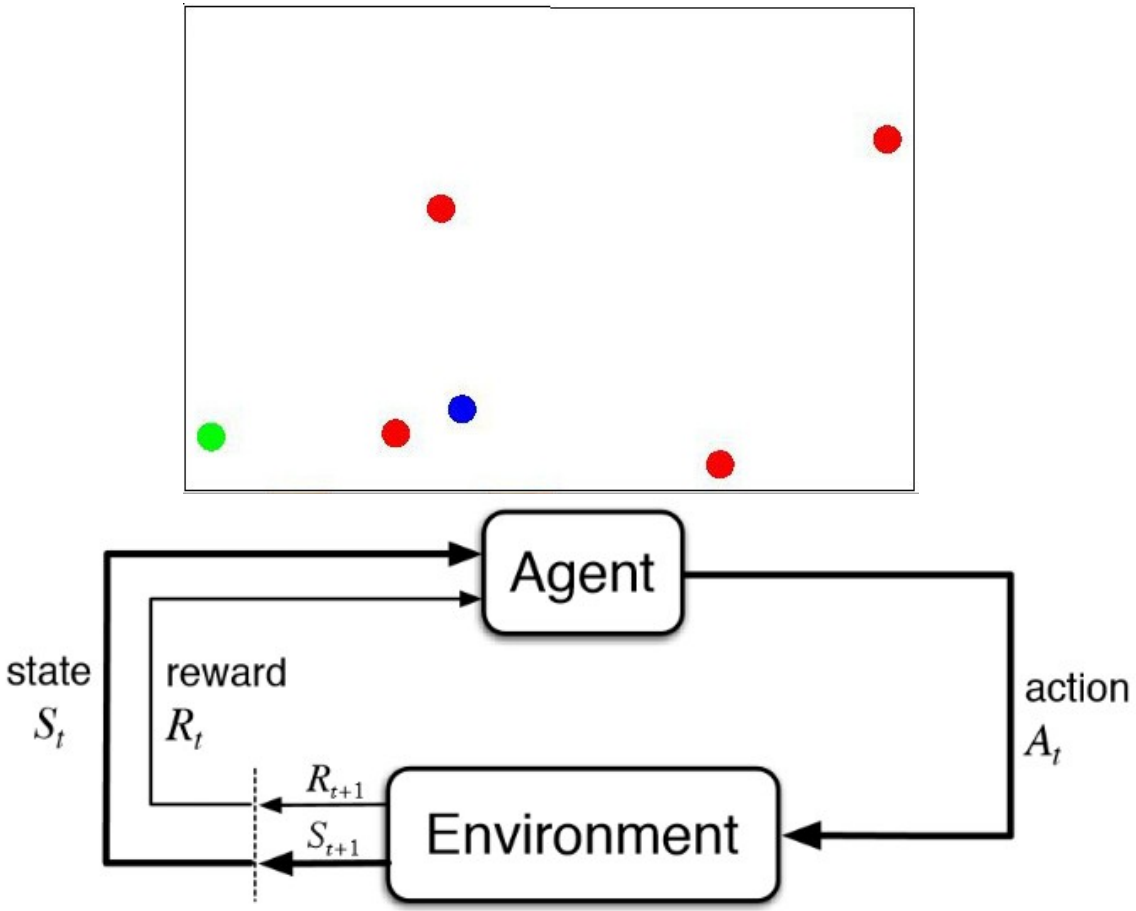
- Cahier des charges
- Apprentissage par renforcement
- Réseaux neuronaux
- Implémentation
- Analyse des résultats
- Conclusion

Cahier des charges

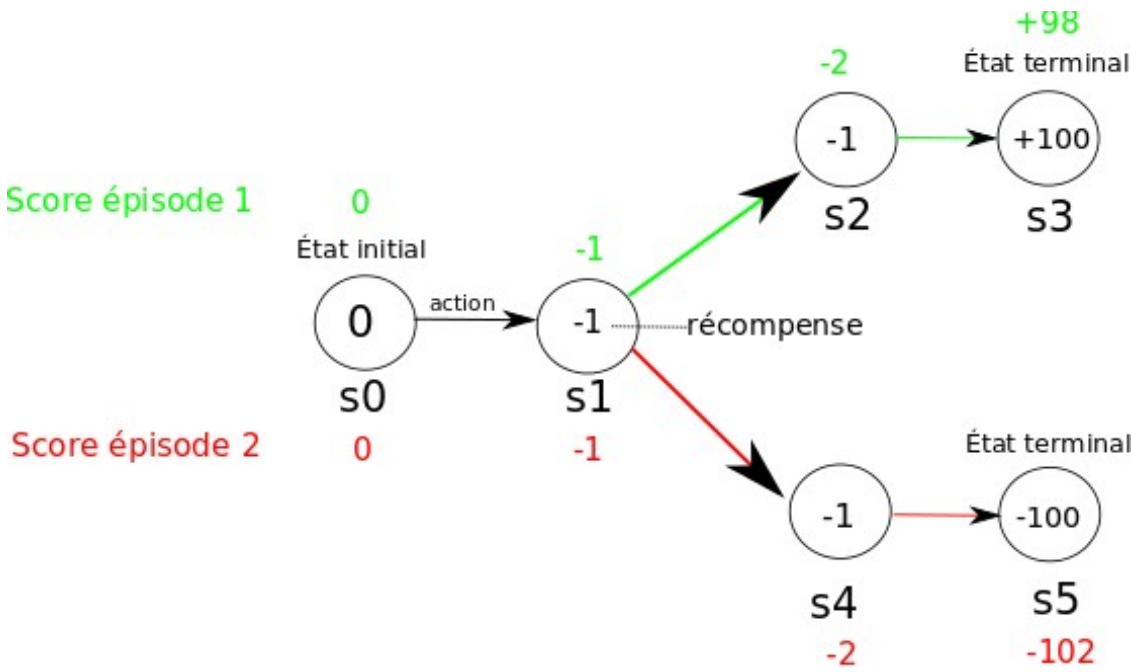
- Comprendre les modèles de réseaux de neurones artificiels standards
- Comprendre les modèles de réseaux de neurones artificiels profonds
- Comprendre les concepts de l'apprentissage par renforcement
- Construire un jeu avec un agent qui apprend par renforcement
- Analyse des résultats
- Rédaction du rapport

Apprentissage par renforcement

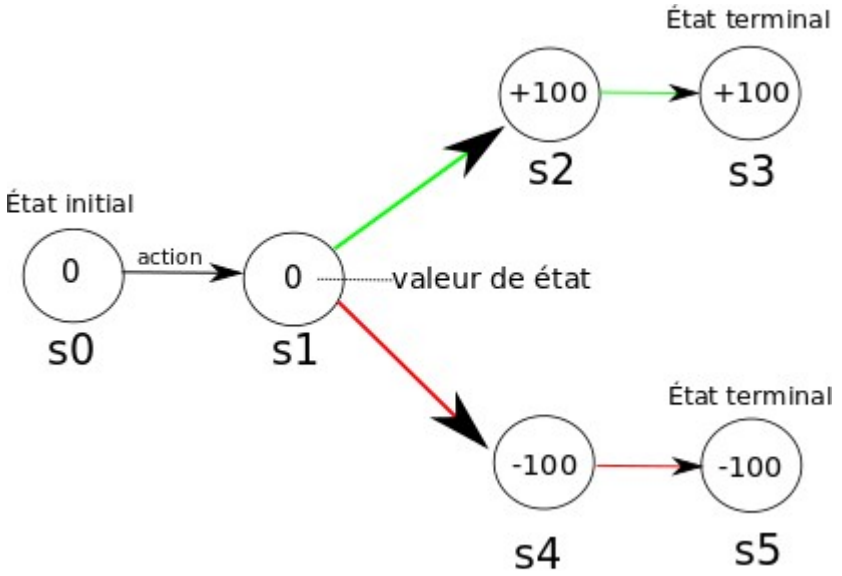
Agent ↔ Environnement



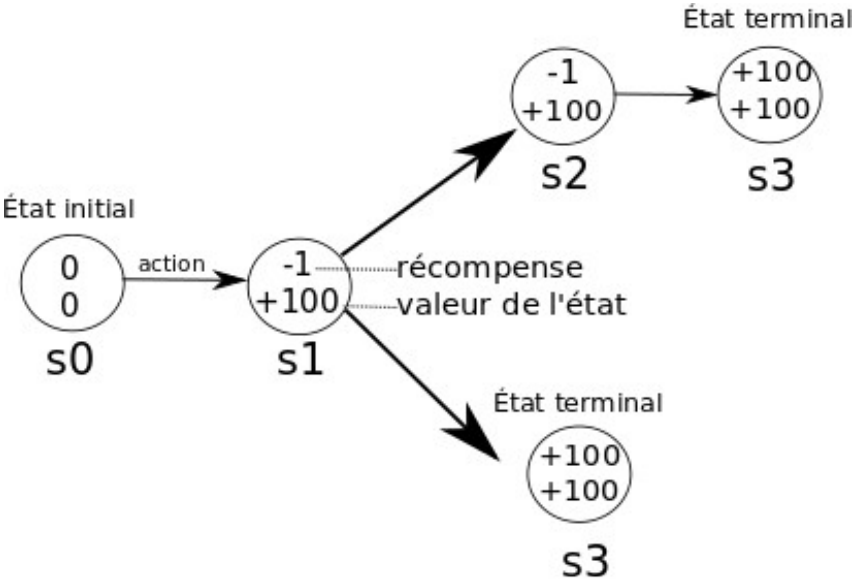
Choix d'une action: valeur d'un état (1)



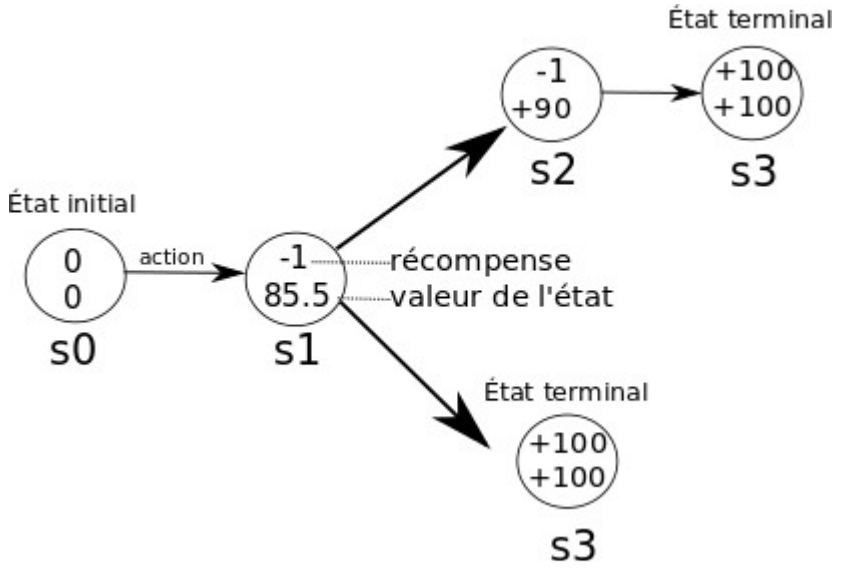
Choix d'une action: valeur d'un état (2)



Choix d'une action: valeur d'un état (3)

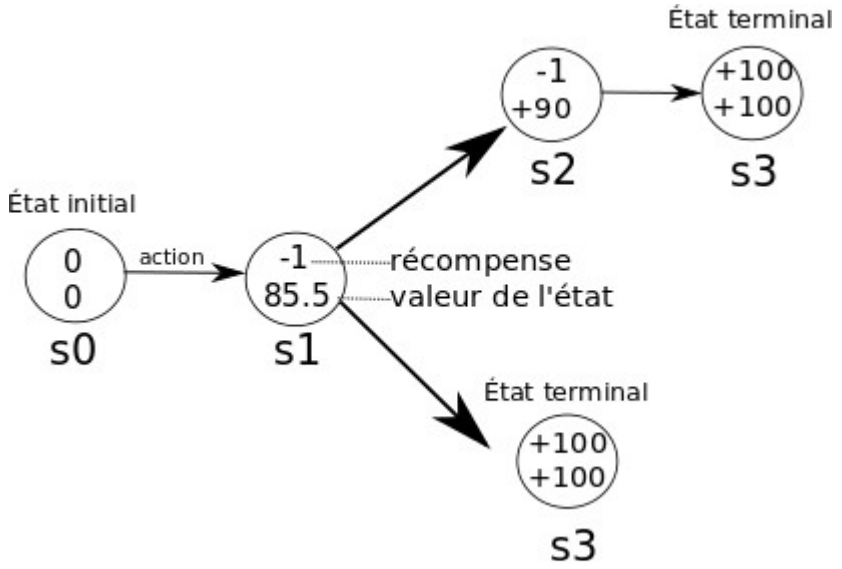


Choix d'une action: valeur d'un état (3)
discount factor (γ)



$$V(s_t) = \gamma V(s_{t+1}) \text{ avec } \gamma \in [0, 1]$$

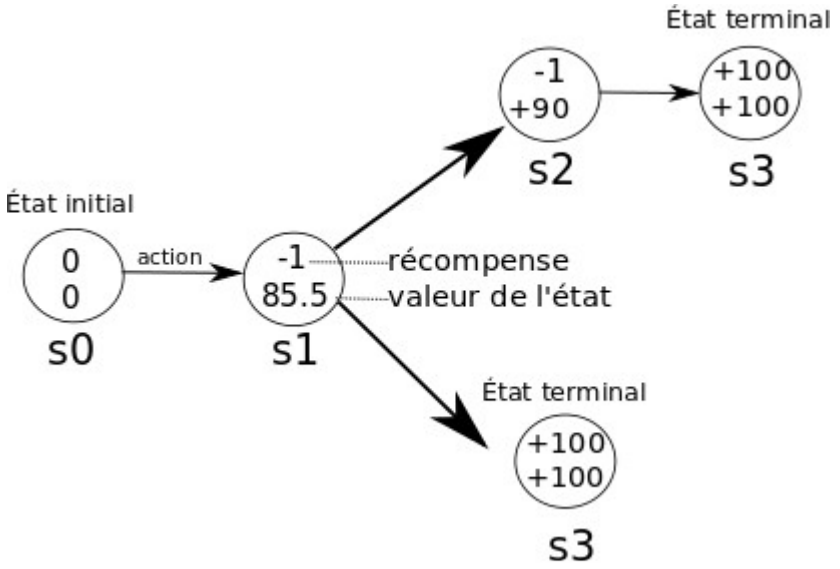
Choix d'une action: « explore vs exploit » proba aléatoire (ε)



$$V(s_t) = \gamma V(s_{t+1}) \text{ avec } \gamma \in [0, 1]$$

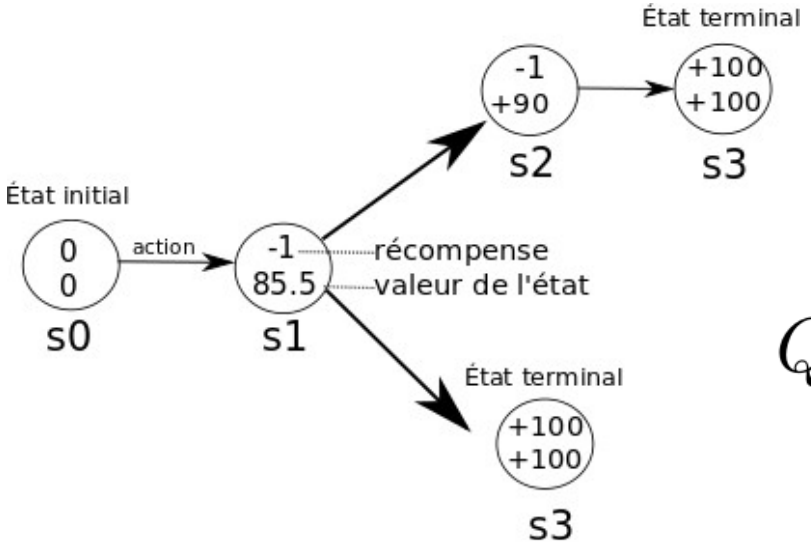
action aléatoire avec p = ε ∈ [0,1]

Valeur d'un état: équation de Bellman (simplifiée)



$$V_\pi(s) = \sum_a \pi(a|s) \left\{ r + \gamma V_\pi(s') \right\}$$

Valeur d'un état: Q-Learning



$$V_\pi(s) = \sum_a \pi(a|s) \left\{ r + \gamma V_\pi(s') \right\}$$

$$Q_\pi(s,a) = \left\{ r + \gamma Q_\pi(s',a') \right\}$$

$$Q(s,a) = Q(s,a) + \alpha(\left\{ r + \gamma Q(s',a') \right\} - Q(s,a))$$

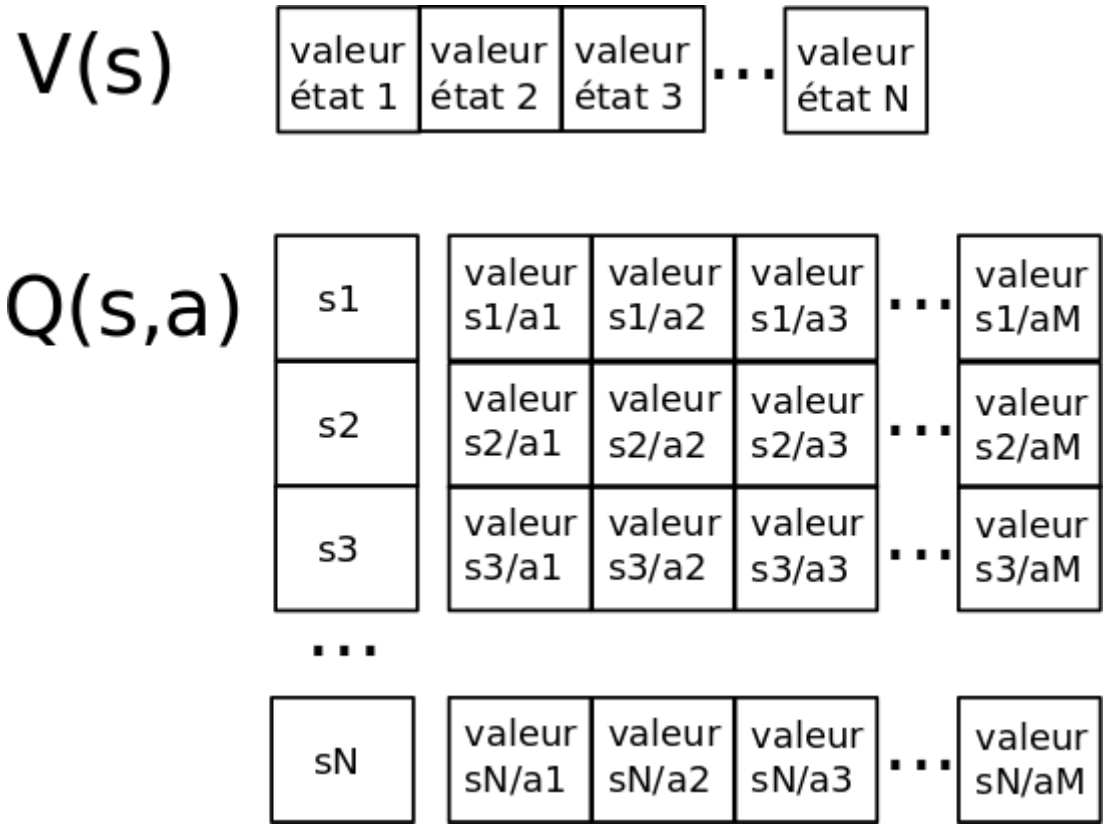
Apprentissage par renforcement

Q-Learning : pseudo-code

```
# phase d'apprentissage
world = new World()
agent = new Agent()
max_episodes = 100000
gamma = 0.9
epsilon = 0.1
alpha = 0.1

for i in range(max_episodes) :
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    reward, next_state = world.do_action(state, action)
    a2 = agent.choose_action(next_state, epsilon)

    if world.game_over() :
      agent.Q[state][action] += alpha*(reward)
    else :
      agent.Q[state][action] += alpha*(reward+gamma*agent.Q[next_state][a2])

    state = next_state


  # phase une fois l'apprentissage effectué
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    state, _ = world.do_action(action)
```

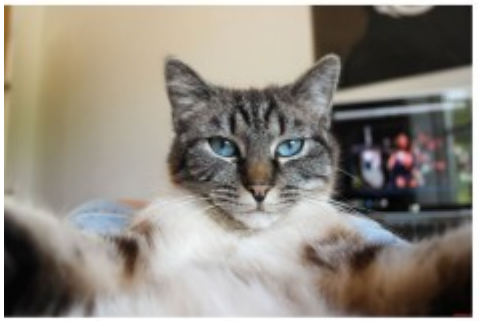
$$Q(s,a) = Q(s,a) + \alpha(\{r + \gamma Q(s',a')\} - Q(s,a))$$

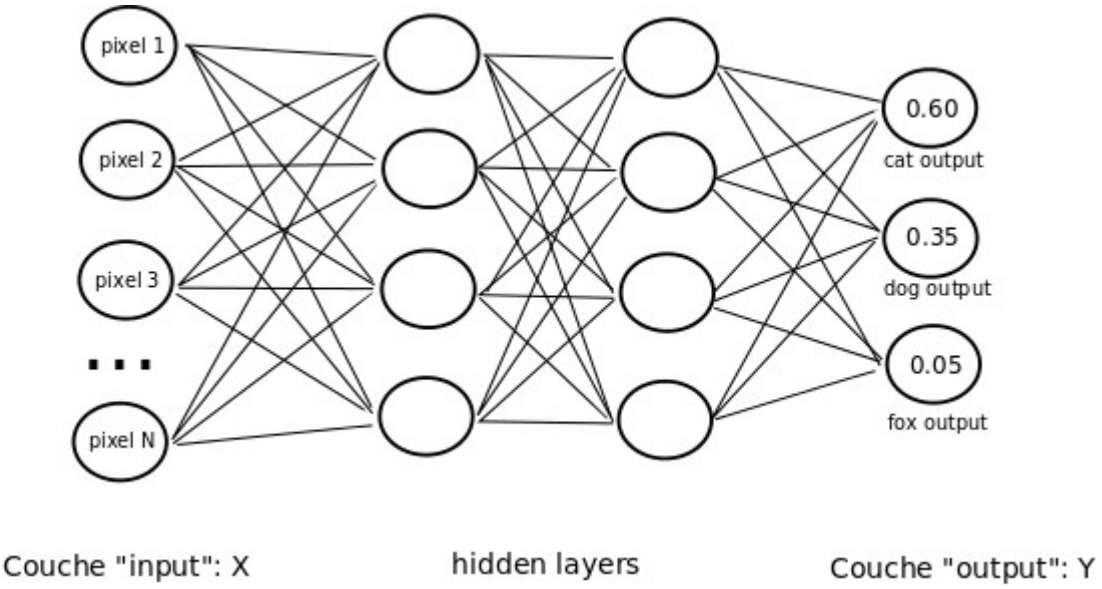
Q-Learning
Nécessité de réseaux neuronaux



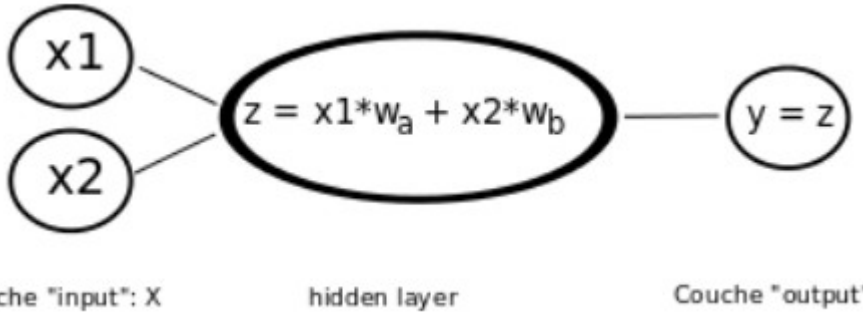
Réseaux Neuronaux

Utilité : estimer la valeur d'un état *Q(s,a)*

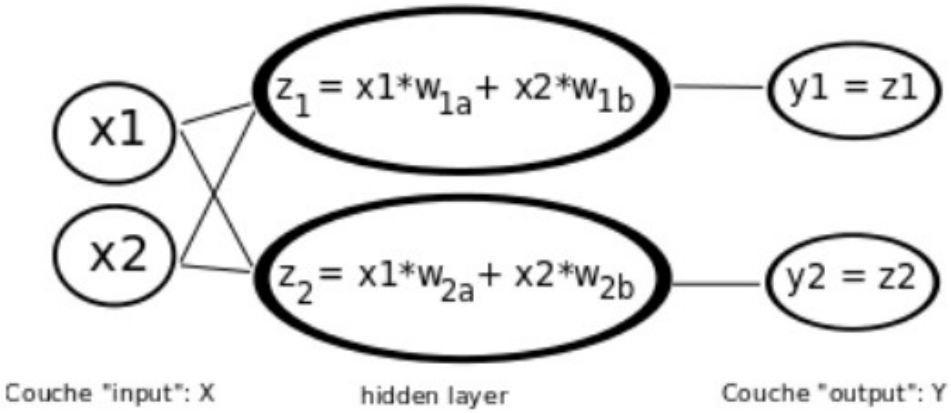




Réseau de neurones « standard »
Prédiction

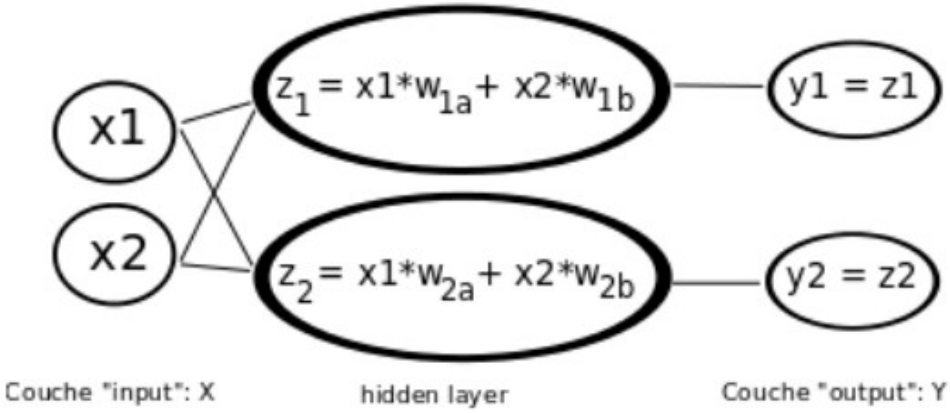


$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$W = (w_a, w_b)$$
$$Z = WX = (w_a, w_b)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (x_1 w_a + x_2 w_b)$$
$$Y = Z = (x_1 w_a + x_2 w_b)$$



$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$W = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix}$$
$$Z = WX = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$
$$Y = Z = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$

Réseau de neurones « standard »
Apprentissage (back-propagation)



$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$W = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix}$$

$$Z = WX = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$

$$Y = Z = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$

But : modifier les poids W en fonction de l'erreur

Y : résultat prédit

T : résultat correct

J(Y,T) : « taille de l'erreur »

$$W = W + \alpha \frac{dJ(Y,T)}{dW}$$

Réseau de neurones profonds

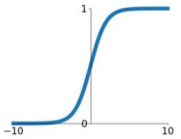
- Rendre les réseaux non-linéaires

$$Z = f(WX + b)$$

Activation Functions

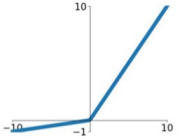
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



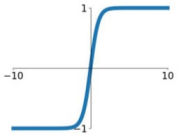
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

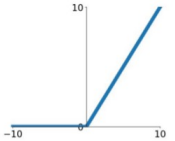


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

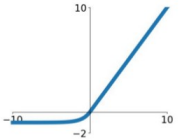
ReLU

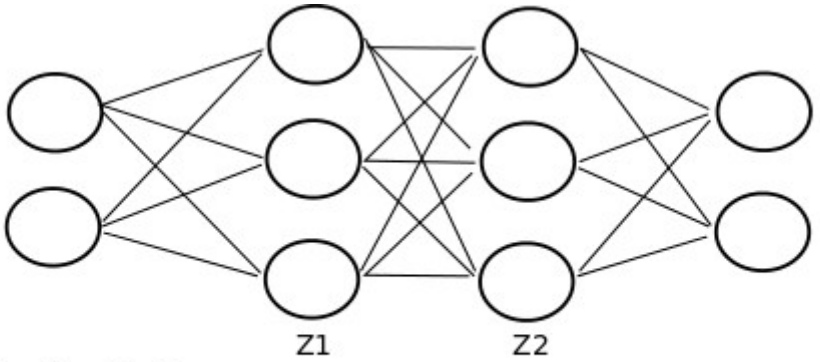
$\max(0, x)$



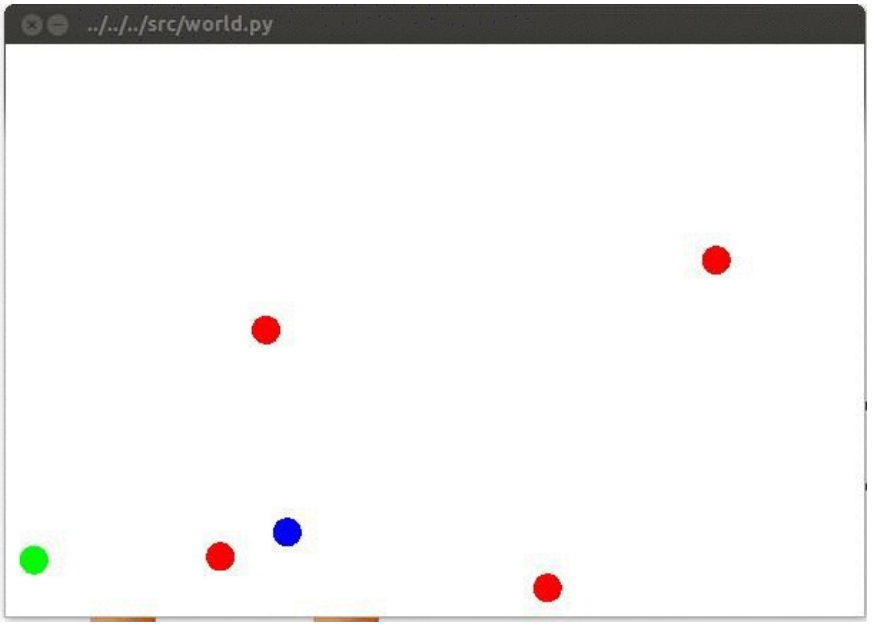
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$





Implémentation :
Rappel environnement

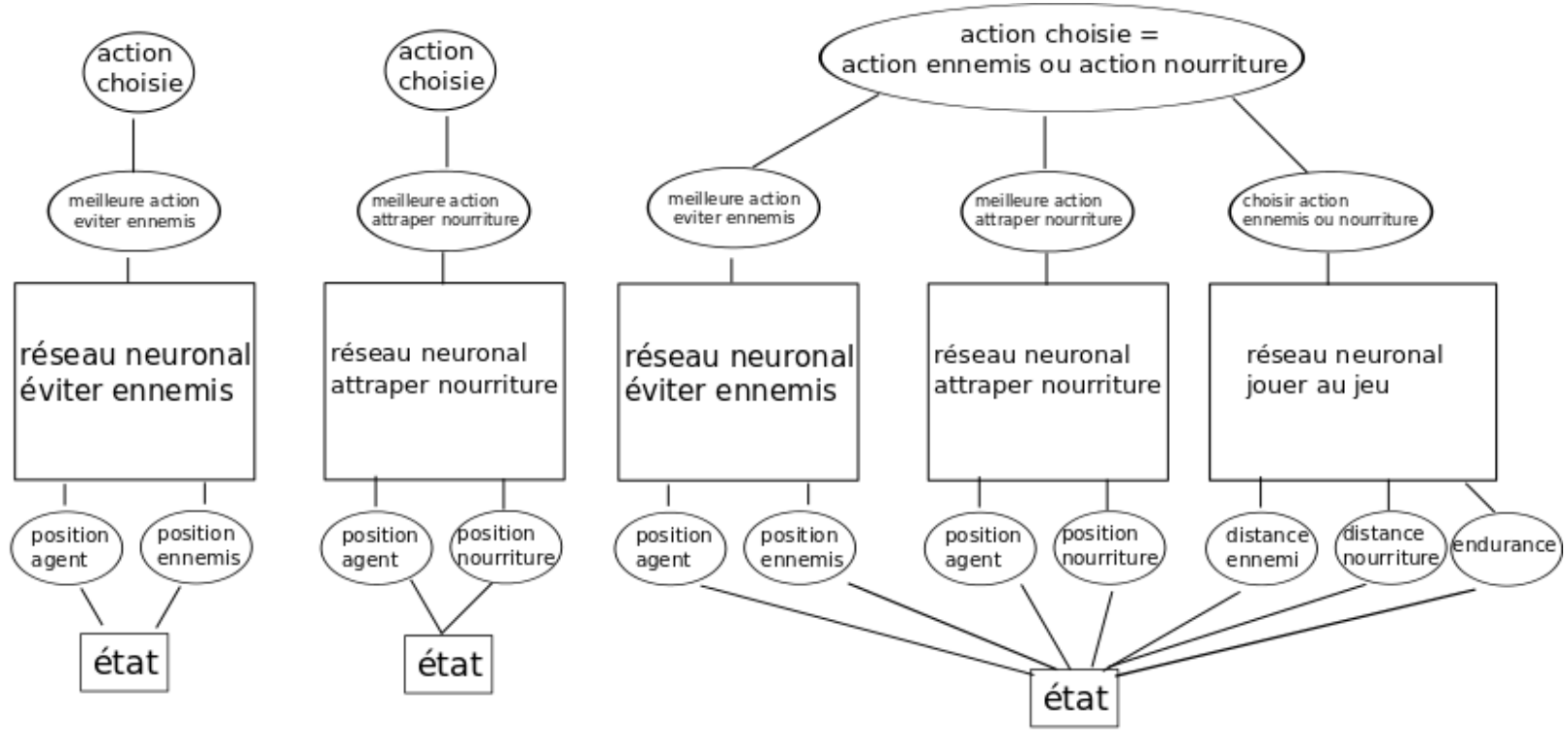


Implémentation (2):
rappel pseudo-code

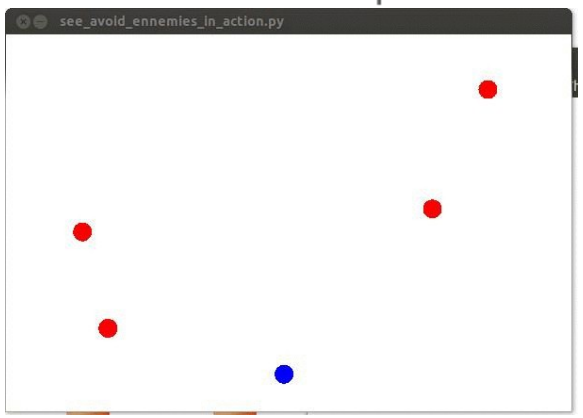
```
# phase d'apprentissage
world = new World()
agent = new Agent()
max_episodes = 100000
gamma = 0.9
epsilon = 0.1
alpha = 0.1

for i in range(max_episodes) :
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    reward, next_state = world.do_action(state, action)
    a2 = agent.choose_action(next_state, epsilon)

    if world.game_over() :
      agent.Q[state][action] += alpha*(reward)
    else :
      agent.Q[state][action] += alpha*(reward+gamma*agent.Q[next_state][a2])

    state = next_state


  # phase une fois l'apprentissage effectué
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    state, _ = world.do_action(action)
```

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

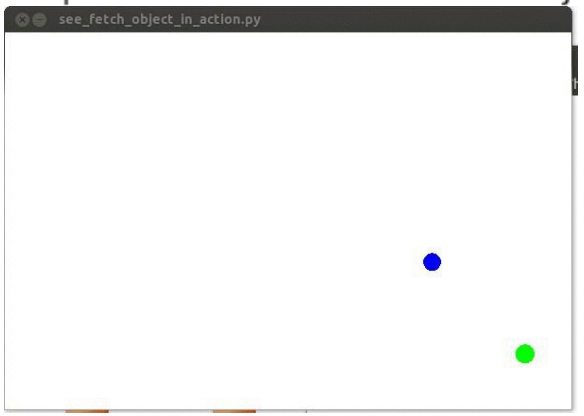
Implémentation (3): phases d'apprentissage



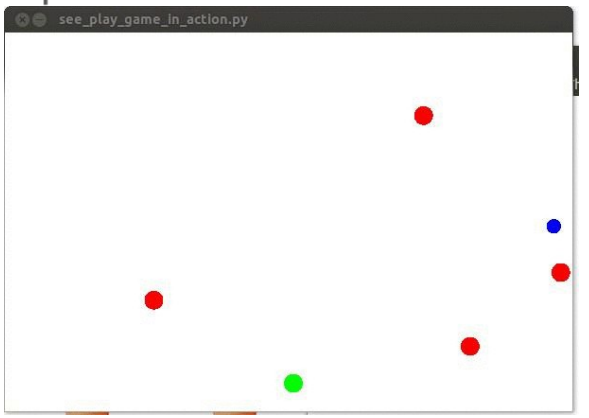
Ennemis uniquement



Nourriture uniquement



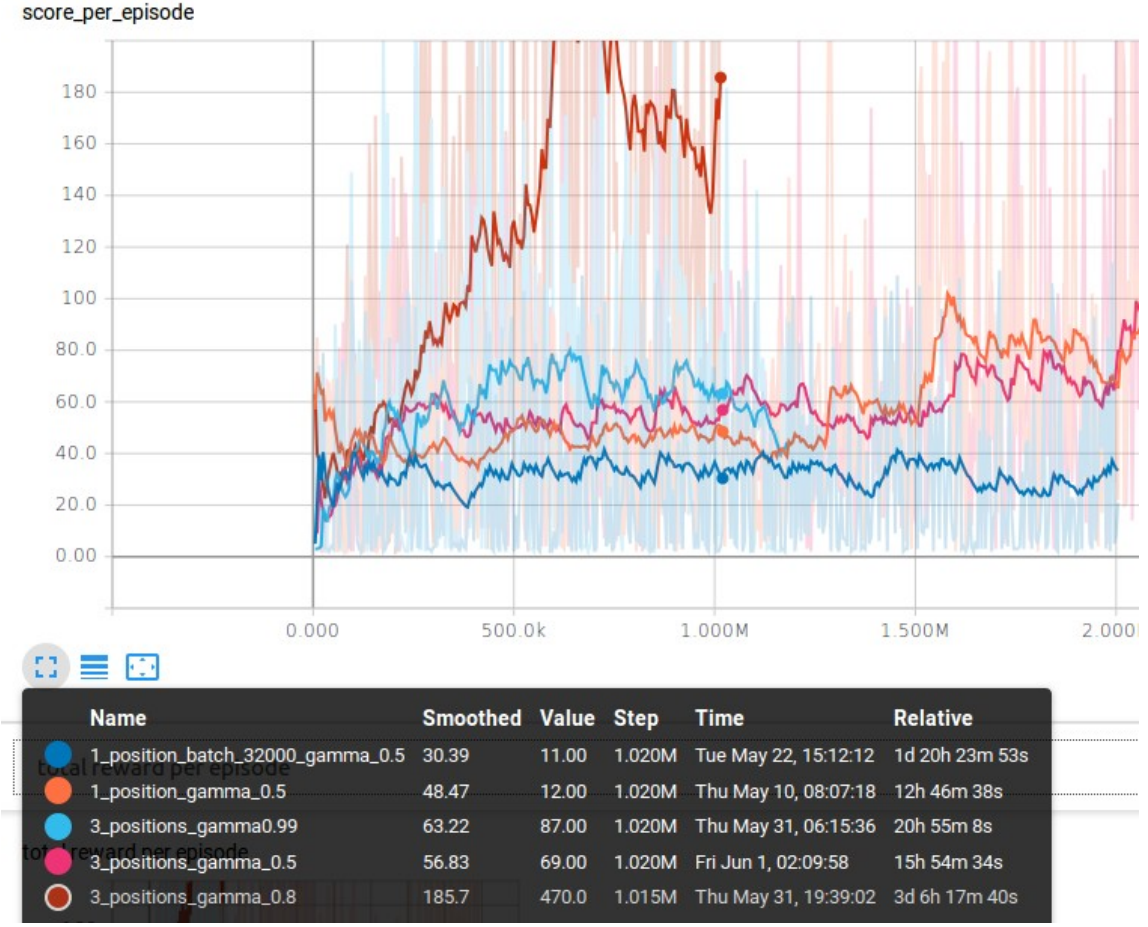
Jeu complet



Résultats

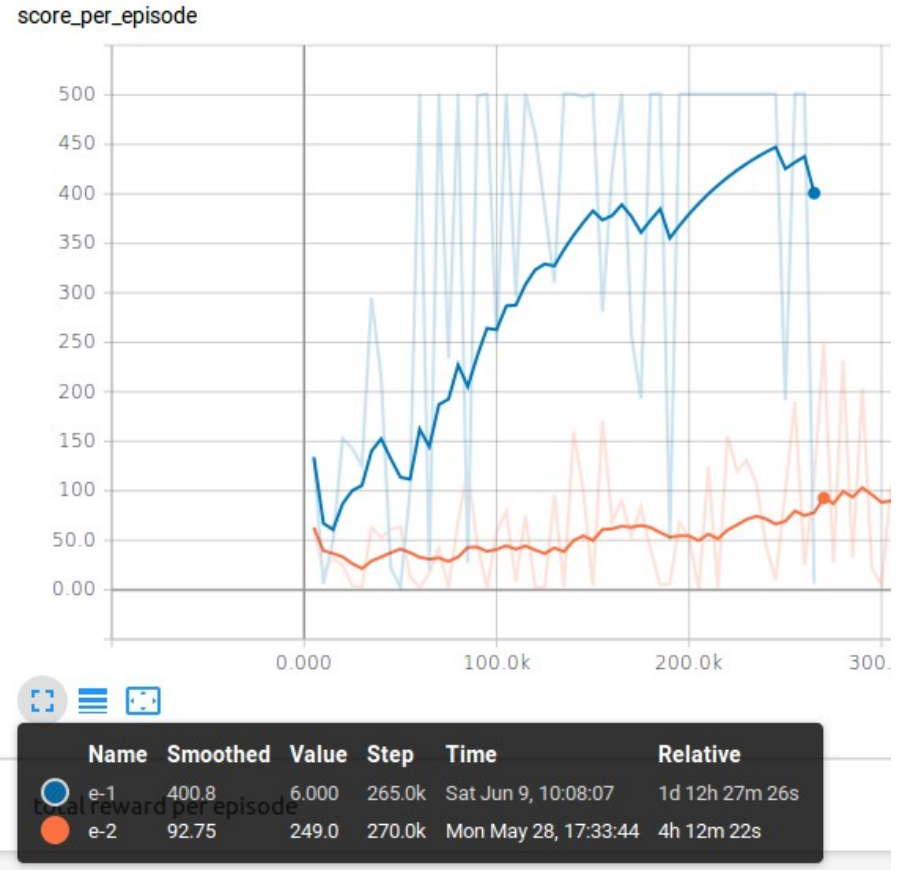
→ Choisir adéquatement la manière de représenter l'environnement

→ Choix du discount factor (γ)



Résultats (2)

→ Learning rate (α) $$Q(s,a) = Q(s,a) + \alpha(\{r + \gamma Q(s',a')\} - Q(s,a))$$



Conclusion :

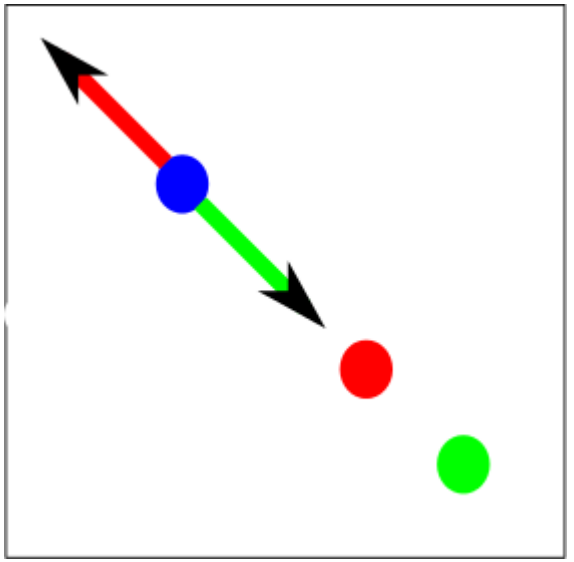
difficultés et perspectives

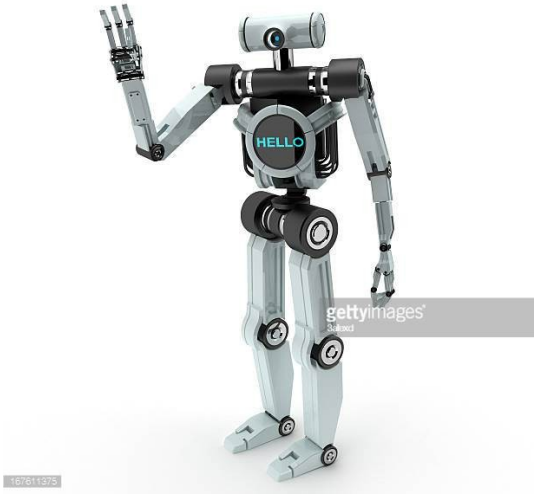
→ Domaine pas enseigné à l'hepia

→ Théorie éloignée de la pratique

→ Domaine évolue sans cesse

→ Architecture pas adaptée au jeu construit

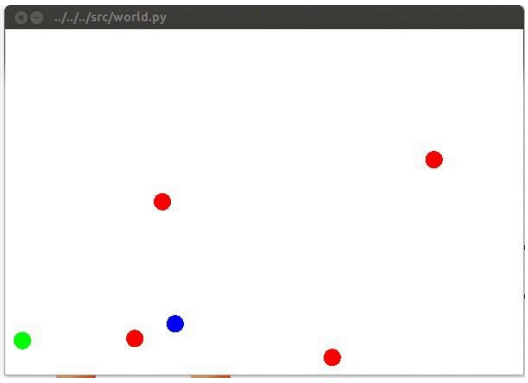




Étude sur l’apprentissage par renforcement dans les jeux

Projet de Bachelor 2018 - hepia

Étudiant : Federico Pfeiffer Professeur : Guido Bologna



h e p i a
Haute école du paysage, d’ingénierie et d’architecture de Genève

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

- travail porté sur apprentissage par renforcement
- branche de l'intelligence artificielle dans laquelle agent (en bleu) apprendre par lui-même à évoluer dans jeu créé pour l'occasion
- Eviter ennemis / attraper nourriture
- dificile résumer en 20 minutes ->parties survolées

Plan :

- Cahier des charges
- Apprentissage par renforcement
- Réseaux neuronaux
- Implémentation
- Analyse des résultats
- Conclusion

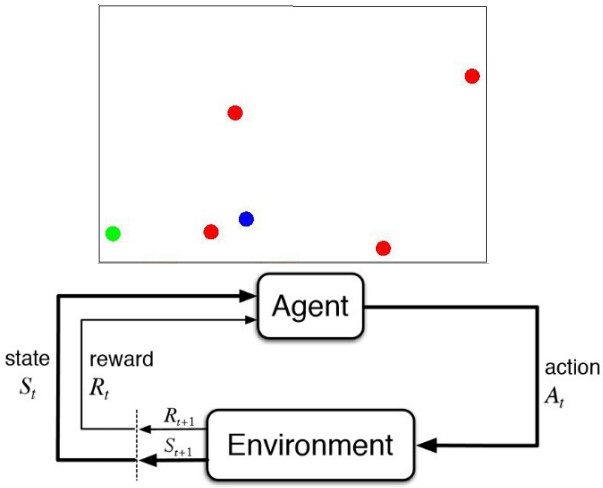
- Présentation déroule manière suivante

Cahier des charges

- Comprendre les modèles de réseaux de neurones artificiels standards
- Comprendre les modèles de réseaux de neurones artificiels profonds
- Comprendre les concepts de l’apprentissage par renforcement
- Construire un jeu avec un agent qui apprend par renforcement
- Analyse des résultats
- Rédaction du rapport

Apprentissage par renforcement

Agent ↔ Environnement



- définition: agent (programme) environnement (le reste). Etat : photo instantanée de environnement.
- agent séparé et autonome.

CLICK

- Fonctionnement apprentissage par renforcement :
  1 agent effectue action
  2 agent a un retour sur action (récompense) (négative ou positive)
  Exemple :
  - foncer ds ennemi : négatif
  - éviter ennemi : positif

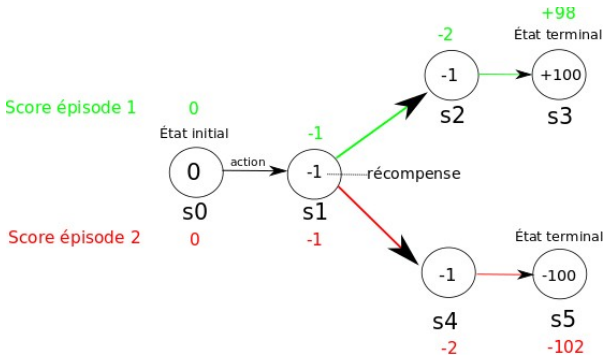
  But agent : avoir le plus de récompenses possibles

CLICK

  3 agent enregistre ce qui vient de se passer
  → prochaine fois qu'il est dans une même situation, action menant à état comportant la meilleur récompense.

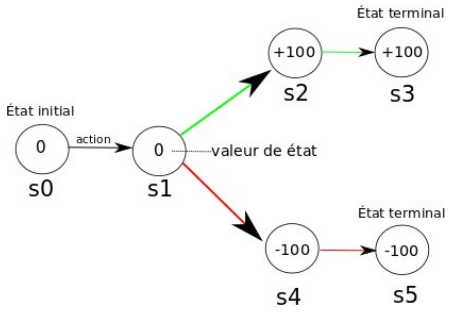
- agent doit explorer par lui-même pour découvrir ce qui lui attends

Choix d'une action: valeur d'un état (1)



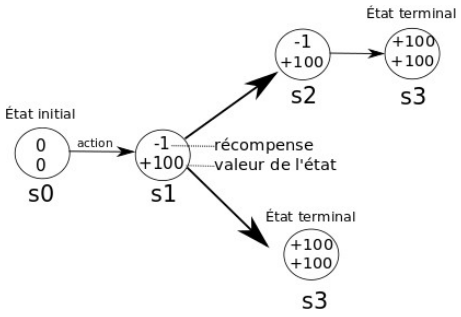
- choisir l'action menant à l'état donnant la meilleure récompense.
  -> attribuer valeurs aux états rencontrés
- décrire exemple
- on peut définir la valeur d'un état en fonction de la récompense reçue
- comment attribuer une valeur à l'état s2 et s4 ?

Choix d'une action: valeur d'un état (2)



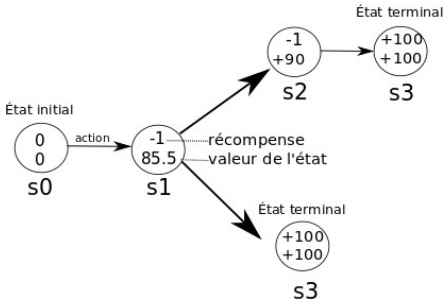
- on peut attribuer la valeur à ce qu'il va nous rapporter.

Choix d'une action: valeur d'un état (3)



- Autre exemple : comment choisir dans ce cas ?

Choix d'une action: valeur d'un état (3)
discount factor (γ)



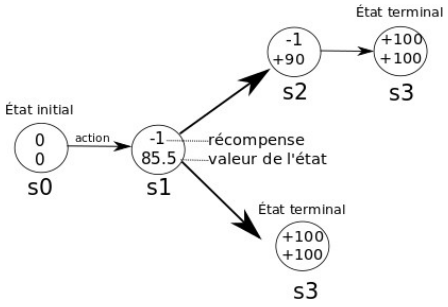
$$V(s_t) = \gamma V(s_{t+1}) \text{ avec } \gamma \in [0, 1]$$

- imaginer que la valeur diminue de manière dégressive
- Là, l'agent sait qu'il a meilleurs temps à aller directement s3.

CLICK

- représenter V(s) comme ça (maths)
- dicsount factor (paramètre important définissable par développeur).

Choix d'une action: « explore vs exploit » proba aléatoire (ε)



$$V(s_t) = \gamma V(s_{t+1}) \text{ avec } \gamma \in [0,1]$$

action aléatoire avec p = ε ∈ [0,1]

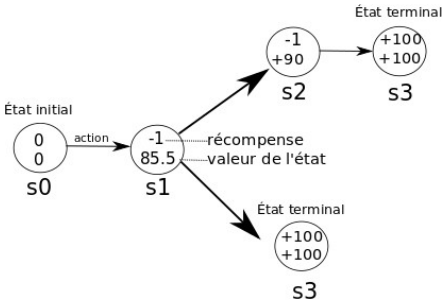
- autre param important : facteur epsilon :
- comment s'assurer que l'on explore suffisament ?
- a quel moment on va arrêter d'explorer et viser que le meileur état ?

→ explorer aléatoirement tous les n actions.

CLICK

- epsilon aussi défini par développeur

Valeur d'un état: équation de Bellman (simplifiée)



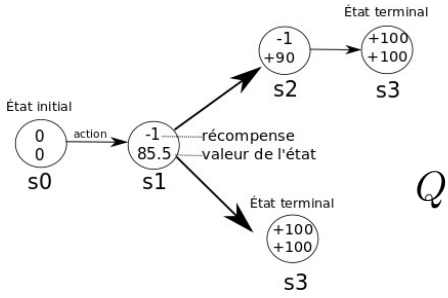
$$V_\pi(s) = \sum_a \pi(a|s) \left\{ r + \gamma V_\pi(s') \right\}$$

- RESUMÈ :
    - on peut connaître la valeur d'un état en fonction de l'état suivant
    - on peut s'assurer d'explorer suffisament à l'aide du facteur espilon
    → L'idée serait d'avoir une fonction mathématique permettant de connaître la valeur de chaque état : équation de bellman

CLICK

- comme ce qu'on a vu, sauf qu'on rajoute la récompense reçue.
- La valeur d'un état dépend aussi de la probabilité qu'on effectue l'action menant à l'état suivant.
- Somme toutes les possibilités d'actions et leurs récompenses.

Valeur d'un état: Q-Learning



$$V_\pi(s) = \sum_a \pi(a|s) \{r + \gamma V_\pi(s')\}$$

$$Q_\pi(s,a) = \{r + \gamma Q_\pi(s',a')\}$$

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

- RESUME: on peut connaître la valeur d'un état si on connaît toutes les récompenses associées à chaque action / états
- Pas pratique : car il faudrait explorer toutes les actions pour chaque état afin d'en connaître la probabilité : on peut modifier l'équation de sorte à ce qu'elle soit plus utile : Q-Learning

CLICK

- valeur d'un état en fonction de l'action effectuée
- ceci fonctionne si on connaît tous les états et actions possibles. C'est pas ce qu'on veut. On veut pouvoir découvrir cette fonction (et la mettre à jour à chaque action que l'on fait.)

CLICK

→ a chaque action, la valeur que l'on connaît de l'état est ajustée d'une faction alpha (learning rate).

- autre paramètre important définissable par Developpeur
- on peut stocker Q(s,a) dans un tableau à 2entrées

Apprentissage par renforcement

Q-Learning : pseudo-code

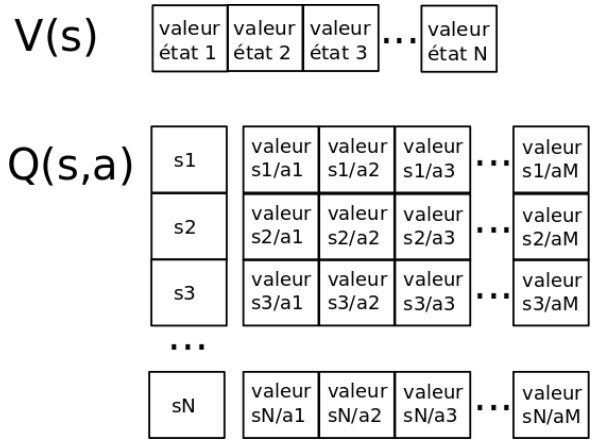
```
# phase d'apprentissage
world = new World()
agent = new Agent()
max_episodes = 100000
gamma = 0.9
epsilon = 0.1
alpha = 0.1

for i in range(max_episodes) :
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    reward, next_state = world.do_action(state, action)
    a2 = agent.choose_action(next_state, epsilon)

    if world.game_over() :
      agent.Q[state][action] += alpha*(reward)
    else :
      agent.Q[state][action] += alpha*(reward+gamma*agent.Q[next_state][a2])

    state = next_state


  # phase une fois l'apprentissage effectué
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    state, _ = world.do_action(action)
```

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

- monde, agent
- Q(s,a) stoké dans tableau à double entrée

Q-Learning

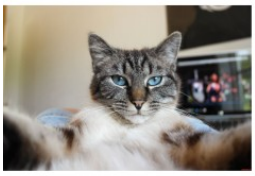
Nécessité de réseaux neuronaux

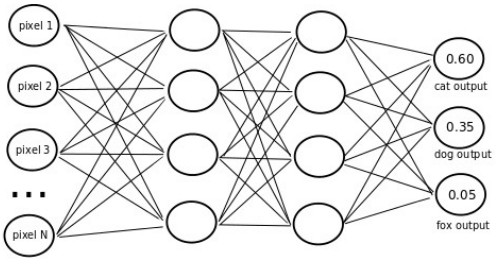


- Au lieu d'enregistrer chaque état, on utilise les réseaux neuronaux pour ESTIMER la valeur d'un état

Réseaux Neuronaux

Utilité : estimer la valeur d'un état *Q(s,a)*

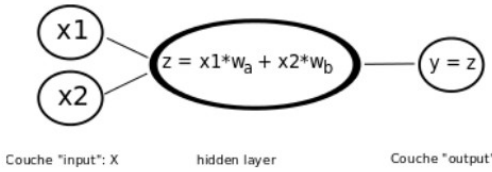




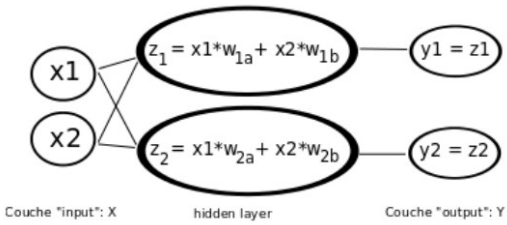
Couche "input": X hidden layers Couche "output": Y

- on utilise les réseaux neuronaux pour estimer la fonction Q(s,a). (prends moins de place en mémoire, mais prends plus de temps à calculer)
- RESUMÈ :
    - entrée → output
    - image chat → prédiction
    - ici : entrée == état : prédiction = meilleure action à effectuer

Réseau de neurones « standard »
Prédiction



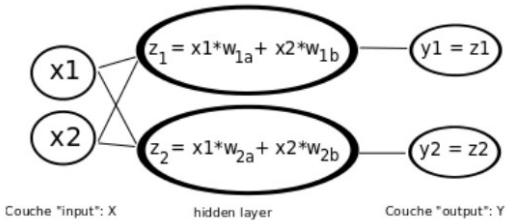
$$
\begin{aligned}
X &= \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
W &= (w_a, w_b) \\
Z &= WX = (w_a, w_b)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (x_1 w_a + x_2 w_b) \\
Y &= Z = (x_1 w_a + x_2 w_b)
\end{aligned}
$$



$$
\begin{aligned}
X &= \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
W &= \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix} \\
Z &= WX = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix} \\
Y &= Z = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}
\end{aligned}
$$

- Niveaux maths, ça fonctionne comme ça.
- On peut noter le caractère Matriciel (pour accélérer les calculs)
- 2 neurones pareil

Réseau de neurones « standard »
Apprentissage (back-propagation)



$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$W = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix}$$

$$Z = WX = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$

$$Y = Z = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}$$

But : modifier les poids W en fonction de l'erreur

Y : résultat prédit

T : résultat correct

J(Y,T) : « taille de l'erreur »

$$W = W + \alpha \frac{dJ(Y,T)}{dW}$$

- Comment modifier les poids de sorte à prédire qqchose de correct?
- Pour enseigner au réseau, on doit fournir un résultat correct.
- On modifie les poids par petits acoups (alpha) (learning rate) : autre paramètre définit par le développeur

Réseau de neurones profonds

- Rendre les réseaux non-linéaires $Z = f(WX + b)$

Activation Functions

Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

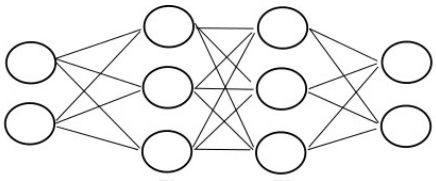
Leaky ReLU
$\max(0.1x, x)$

tanh
$\tanh(x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$

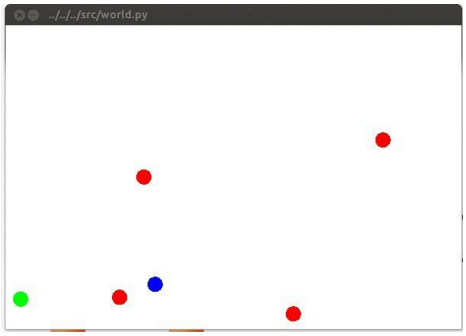
ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



Couche "input": X — hidden layers — Couche "output": Y

- Problème : jusqu'ici on a que des calculs linéaires.
  (donc un réseau jusqu'ici fait des calculs linéaires).
- Ajout d'une fonction d'activation
- Ajout d'un biais (b)
- Back-propagation se fait selon le même principe

Implémentation :
Rappel environnement



- RAPPEL BUT AGENT

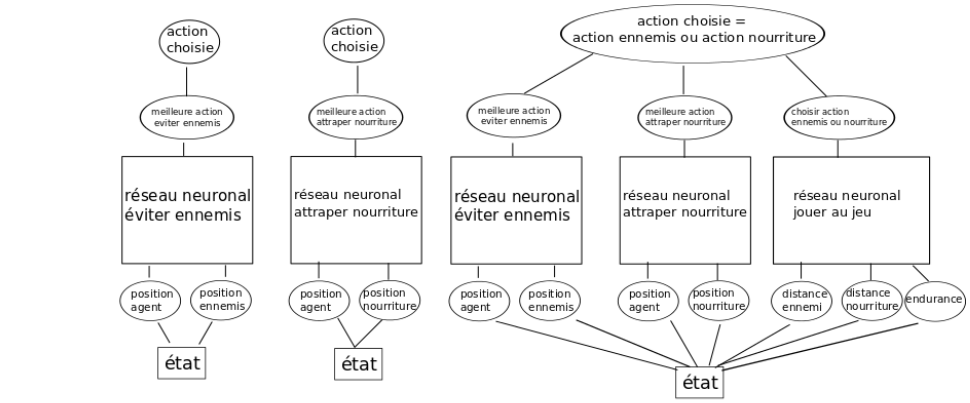
Implémentation (2):
rappel pseudo-code

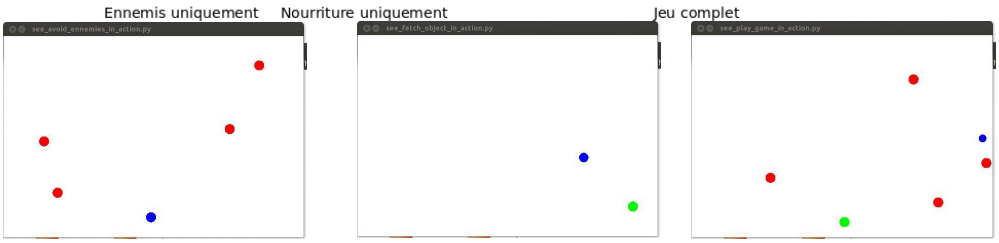
```
# phase d'apprentissage
world = new World()
agent = new Agent()
max_episodes = 100000
gamma = 0.9
epsilon = 0.1
alpha = 0.1

for i in range(max_episodes) :
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    reward, next_state = world.do_action(state, action)
    a2 = agent.choose_action(next_state, epsilon)

    if world.game_over() :
      agent.Q[state][action] += alpha*(reward)
    else :
      agent.Q[state][action] += alpha*(reward+gamma*agent.Q[next_state][a2])

    state = next_state


  # phase une fois l'apprentissage effectué
  state = world.reset()
  while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    state, _ = world.do_action(action)
```

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

- RAPPEL fonctionnement apprentissage

Implémentation (3): phases d'apprentissage

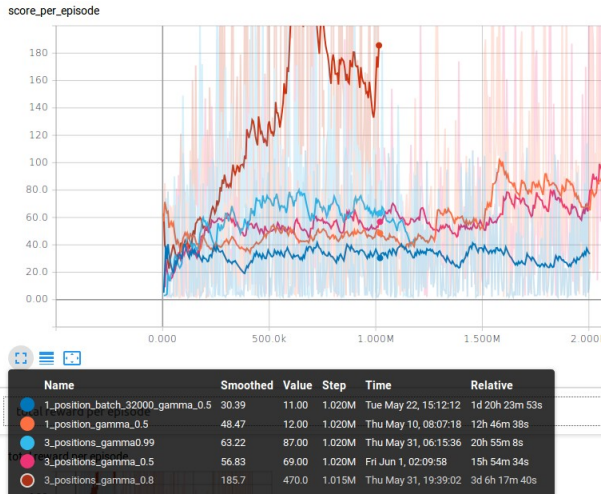




- 3 phases d'apprentissage :
(une fois que l'on a un résultat satisfaisant, enregistrer le réseau pour qu'il puisse être utilisé après)
    1 : éviter ennemis.
CLICK
    2 : attrapper nourriture
CLICK
    3 : jongler entre les deux (EXPLIQUER)
CLICK
- pas top, expliquera à la fin

Résultats

→ Choisir adéquatement la manière de représenter l'environnement
→ Choix du discount factor (γ)



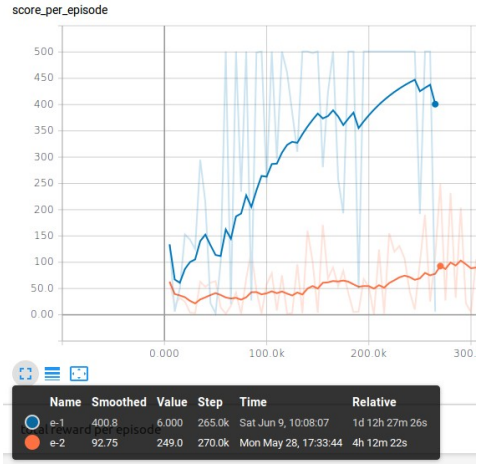
Name	Smoothed	Value	Step	Time	Relative
1_position_batch_32000_gamma_0.5	30.39	11.00	1.020M	Tue May 22, 15:12:12	1d 20h 23m 53s
1_position_gamma_0.5	48.47	12.00	1.020M	Thu May 10, 08:07:18	12h 46m 38s
3_positions_gamma0.99	63.22	87.00	1.020M	Thu May 31, 06:15:36	20h 55m 8s
3_positions_gamma_0.5	56.83	69.00	1.020M	Fri Jun 1, 02:09:58	15h 54m 34s
3_positions_gamma_0.8	185.7	470.0	1.015M	Thu May 31, 19:39:02	3d 6h 17m 40s

- mettre en avant les paramètres les plus importants ayant contribué à l'amélioration du résultat
- au début : 14 jours d'apprentissages.
- choix de la manière de représenter l'environnement :
- eviter ennemis : 3 dernières positions vs simples positions.
- eviter ennemis : discount factor de 0.8

→ au lieu de prendre 14 jours d'apprentissage, on a 4/5 jours 8)

Résultats (2)

→ Learning rate (α) $Q(s,a) = Q(s,a) + \alpha(\{r + \gamma Q(s',a')\} - Q(s,a))$



- choix du learning rate (alpha 0.1 au lieu de 0.01)
- 4/5 jours à 24/48h d'apprentissage

Conclusion :

difficultés et perspectives

→ Domaine pas enseigné à l'hepia

→ Théorie éloignée de la pratique

→ Domaine évolue sans cesse

→ Architecture pas adaptée au jeu construit