TIPE: Machine Learning: Apprentissage par renforcement

Stanislas Henrard

2016

Sommaire

Première partie

Introduction générale

La problématique de l'**apprentissage par renforcement** est de faire acquérir à un système artificiel par l'exemple ou par l'expérience, un comportement pour atteindre un objectif.

À l'aide d'un système de récompenses et de pénalités, l'algorithme améliore alors sa stratégie afin de maximiser le cumul de ses récompenses futures.

L'apprentissage par renforcement pose alors la question suivante :

- Sachant qu'il y a un gain à maximiser, comment trouver le bon équilibre entre l'exploration (choisir une action aléatoire pour analyser l'environnement) et l'exploitation (choisir l'action maximisant les gains) ?

Deuxième partie

Introduction à l'apprentissage par renforcement

Chapitre 1

Quelques notions

Définition 1. *On définit un problème de décision Markovien par un quadruplet : ($\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$)*

- *$\mathcal{S}$ est un ensemble fini d'états,*
- *$\mathcal{A}$ est un ensemble fini d'actions. On note $\mathcal{A}(s)$ l'ensemble des actions possibles dans l'état s. $\mathcal{A} = \cup_{s \in \mathcal{S}} \mathcal{A}(s)$;*
- *$\mathcal{P}$ représente la fonction de transition de l'environnement. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$*
- *$\mathcal{R}$ représente la fonction récompense ou fonction de retour. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$*

Dans notre cas, on se place dans un environnement déterministe, c'est-à-dire que la fonction de transition de l'environnement renvoie toujours 1.

Définition 2. *On définit aussi une politique déterministe π comme une application de $\mathcal{S} \to \mathcal{A}$ qui à un état associe une action.*

Définition 3. *On définit enfin le retour R reçu par l'agent à partir de l'instant t par :*

$$R_t = \sum_{k=0} \gamma^k r_{t+k}$$

où $\gamma \in [0, 1]$ est le facteur d'actualisation. Il permet de régler l'importance que l'on donne aux récompenses futures par rapport à celles immédiates : γ près de 1 : les retours futurs sont pris en compte ; γ près de 0 : les retours futurs sont négligés.

Propriété 1. *π^*, la politique optimale, existe pour tout PDM.*

Définition 4. *On définit la valeur d'un état s pour une politique fixée π par :*

$$V^\pi(s) = E[R_t | s_t = s]$$

Définition 5. *On définit la qualité d'une paire état-action (s, a) pour une politique fixée π par :*

$$Q^\pi(s, a) = E[R_t | s_t = s, a_t = a]$$

Il y a une forte relation entre ces deux fonctions :

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} [\mathcal{R}(s, a, s') + \gamma V^\pi(s')] \tag{1.1}$$

Chapitre 2

L'algorithme Q-Learning

2.1 Principe de l'algorithme

Le principe du Q-Learning est d'estimer une fonction Q^* en utilisant une mise-à-jour itérative donnée par :

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.1}$$

- α : taux d'apprentissage qui détermine à quel point la nouvelle information calculée surpassera l'ancienne. Un facteur 0 ne ferait rien apprendre à l'agent tandis qu'un facteur de 1 ne lui ferait considérer que la dernière information qu'il a perçue.
- γ : facteur d'actualisation. On a déjà défini ce paramètre à la définition 3.

En principe, il faut explorer l'environnement pendant un grand nombre d'itérations pour que le Q-Learning puisse converger vers la fonction Q optimale et seulement ensuite on peut utiliser la politique optimale définie par :

$$\pi^*(s) = arg(max_{a \in \mathcal{A}} Q^*(s, a)) \tag{2.2}$$

Algorithme 1 : Algorithme du Q-Learning en pseudo-code

$Q(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
Pour ∞ **faire**
  initialiser l'état initial s_0
  $t \leftarrow 0$
  Répéter
    choisir l'action à émettre
    observer r_t et s_{t+1}
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma max_{a' \in \mathcal{A}(s_{t+1})} Q(s_{t+1},a') - Q(s_t,a_t)]$
  Jusqu'à ce que $s_t \in \mathcal{F}$
Fin Pour

Dans un algorithme Q-Learning, le choix de l'action doit garantir un équilibre entre l'exploration et l'exploitation de l'apprentissage déjà réalisé : faire confiance à l'estimation courante de Q pour choisir la meilleure action à effectuer dans l'état courant (exploitation) ou au contraire, choisir une action *à priori* sous-optimale pour observer les conséquences (exploration). Naturellement, on conçoit intuitivement que l'exploration doit initialement être importante (quand l'apprentissage est encore très partiel, on explore) puis diminuer au profit de l'exploitation quand l'apprentissage a été effectué pendant une période suffisamment longue.

2.2 Stratégies

Plusieurs stratégies sont utilisées classiquement :

- **gloutonne :** elle consiste à toujours choisir l'action comme estimée la meilleure, soit

$$a_{gloutonne} = arg(max_{a \in \mathcal{A}(s_t)}(Q(s_t,a)));$$

- **ϵ-gloutonne :** elle consiste à choisir l'action gloutonne avec une probabilité de ϵ et à choisir une action au hasard avec une probabilité $1-\epsilon$, soit :

$$a_{\epsilon-gloutonne} = \begin{cases} \arg\, max_{a \in \mathcal{A}(s_t)}(Q(s_t,a)) \text{ avec probabilité } \epsilon \\ \text{action prise au hasard dans} \mathcal{A}(s_t) \text{ avec probabilité } 1-\epsilon \end{cases}$$

Un cas particulier est la solution 0-gloutonne qui consiste à choisir une action au hasard ;

Ce sont les deux seules que nous étudieront mais il en existe quelques autres (softmax, boltzmann, etc).

2.3 Convergence du Q-Learning

On a :

Propriété 2. *Pour un PDM fixé, l'algorithme du Q-Learning converge asymptotiquement vers une politique optimale à condition que :*

- *chaque paire (état, action) soit visitée une infinité de fois ;*
- *que l'on ait :* $\sum_t \alpha_t(s) = +\infty$ *et* $\sum_t \alpha_t^2(s) < +\infty, \forall s \in \mathcal{S}$

Troisième partie

L'algorithme d'apprentissage

Chapitre 3

Le labyrinthe

3.1 Le problème

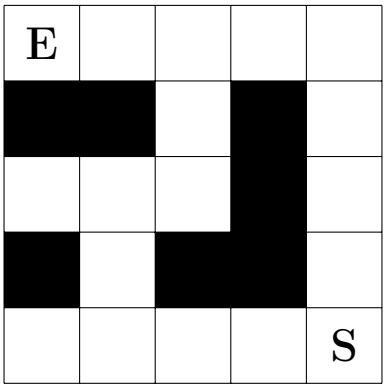

FIGURE 3.1 – Labyrinthe

On prendra comme environnement de départ une grille de taille 5×5. Les cases blanches sont les cases vides où l'algorithme peut se déplacer et les cases noires sont les murs, où il ne peut pas aller. Il ne peut se déplacer qu'à l'intérieur de la grille. L'entrée du labyrinthe sera la coordonnée $(0, 0)$ et la sortie la coordonnée $(4, 4)$. Voir la figure 3.1.

L'algorithme doit alors trouver son chemin dans ce labyrinthe, et essayer d'y arriver avec le moins de déplacements possibles. Ce problème est similaire à celui de recherche de chemin le plus court dans une matrice ou un arbre à la différence qu'ici, l'algorithme ne connait encore rien de son environnement avant de choisir ses déplacements, il ne connait ni le coût des chemins possibles, ni l'endroit où il doit arriver.

3.2 Le tableau des utilités

On utilisera pour mémoriser les valeurs d'utilités un tableau de taille $5 \times 5 \times 4$. On stockera en profondeur les valeurs d'utilités pour chaque direction : O pour haut, 1 pour droite, etc. Chaque case de la grille sera alors divisée en 4 comme indiqué sur la figure : 3.2 :

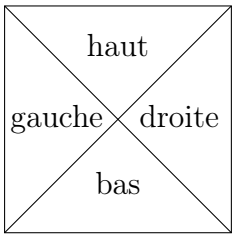

FIGURE 3.2 – Découpage des cases de la grille

3.3 La politique optimale

On remarque que dans ce labyrinthe il y a plusieurs politiques dont une seule est optimale. La figure 3.3 représente une politique quelconque et la politique optimale : la politique quelconque mène l'agent à la sortie en 10 déplacements tandis que la politique optimale en 8 seulement.

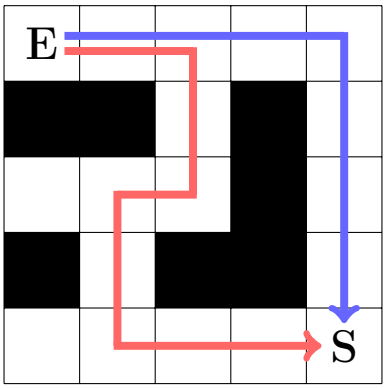

FIGURE 3.3 – Labyrinthe avec politiques

Chapitre 4

L'implémentation

4.1 La structure globale

L'algorithme peut se décomposer en 4 grandes parties :

1. **Les fonctions d'utilités** : qui permettent à l'algorithme de savoir quels mouvements il peut faire, ou bien si la case courante mène à la sortie, etc. Cette partie se situe entre les lignes 49 et 119 sur le code de l'algorithme *Maze Runner*. Il est inutile de détailler cette partie, les commentaires sont suffisants.
2. **Les fonctions d'exploration** : qui vont permettre à l'algorithme de se déplacer dans le labyrinthe jusqu'à ce qu'il trouve la sortie, de la ligne 120 à la ligne 158. Grâce à une boucle for, on peut alors choisir le nombre d'épisodes d'apprentissage (c'est-à-dire le nombre de fois où il aura à atteindre la sortie).
3. **Les fonctions de Q-Learning** : qui vont permettre l'apprentissage de l'agent. On y trouve une fonction permettant de trouver le mouvement d'utilité maximale et une autre choisissant le mouvement à faire et mettant à jour le tableau d'utilité.
4. **Les fonctions d'affichage** : qui vont permettre d'afficher divers graphes.

4.2 Zoom sur la fonction Q-Learning

Listing 4.1 – Fonction Q-Learning

```
1  def Q_deplacement(table,etat,Q,epsilon):
2      "Permet de renvoyer le mouvement choisit par le Q-Learning"
3
4      #on initialise la direction
5      global alpha, gamma
6      direction = 0
7      #on met tous les déplacements a -1 ( imposible )
8      possibles = [-1,-1,-1,-1]
9
10     for i in range(4):
11
12         #si le deplacement est possible,
```

```
13          if deplacement_possible(table,etat,init_deplacement[i]):
14
15              #on rentre sa direction a la place du -1
16              possibles[i] = i
17
18      #strategie epsilon - gloutonne
19      if random()<epsilon:
20
21          #on en prend un au hasard
22          deplacement = choice(deplacements_possibles(table,etat))
23          direction = init_deplacement.index(deplacement)
24
25      else:
26
27          #sinon on prend le meilleur
28          direction = top_deplacement(table,etat,Q,possibles)
29          deplacement = init_deplacement[direction]
30
31      #on initialise la recompense a 0
32      r = 0.00
33
34      #si l'etat est final:
35      if est_final(table,[etat[0]+deplacement[0],etat[1]+deplacement[1]]):
36
37          #la recompense est de +1
38          r = +1.0
39
40      #on utilise la formule de la qualite:
41      Q[etat[0]][etat[1]][direction] +=  alpha*(r + gamma*max(Q[etat[0]
42      + deplacement[0]][etat[1]+deplacement[1]])
43      - Q[etat[0]][etat[1]][direction])
44      return (deplacement,Q)
```

Explication :

On initialise une liste de déplacements possibles. Soit on choisit une action aléatoire avec une probabilité d' 1 - ϵ, soit on choisit le meilleur coup dans la liste des déplacements possibles. Si l'état avec déplacement est final, la récompense sera de +1.0, sinon elle aura pour valeur 0.
On met ensuite à jour le tableau des qualités avec la formule 2.1.

Quatrième partie

Les résultats

Chapitre 5

Vitesse de convergence de l'algorithme

5.1 Stratégie $\epsilon - gloutonne$

J'utilise la stratégie $\epsilon - gloutonne$ avec une décroissance lente d'epsilon ($\times 0,95$) à chaque épisode.
Si on entre : courbe(lab1 (le labyrinthe exploré),150 (le nombre d'épisodes),0.9 (epsilon),50 (le nombre de lancement pour la moyenne)) avec comme valeur d'α : 0.9 et γ : 0.8 on obtient alors la figure 5.1 qui nous montre bien que l'algorithme de Q-Learning converge vers la politique optimale : 8 déplacements.

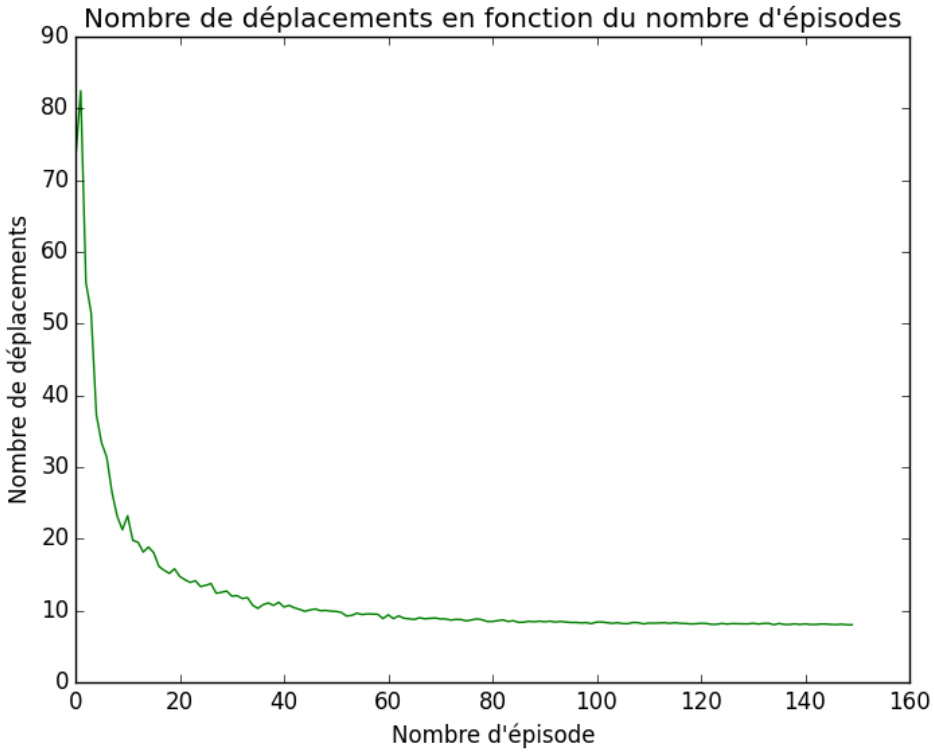

FIGURE 5.1 – Nombre de déplacements en fonction du nombre d'épisodes, stratégie $\epsilon - gloutonne$

5.2 Stratégie gloutonne

En utilisant la stratégie *gloutonne*, la courbe obtenue n'est pas satisfaisante du tout (voir 5.2), on en conclut que l'exploration joue vraiment un rôle important dans la recherche de la politique optimale.

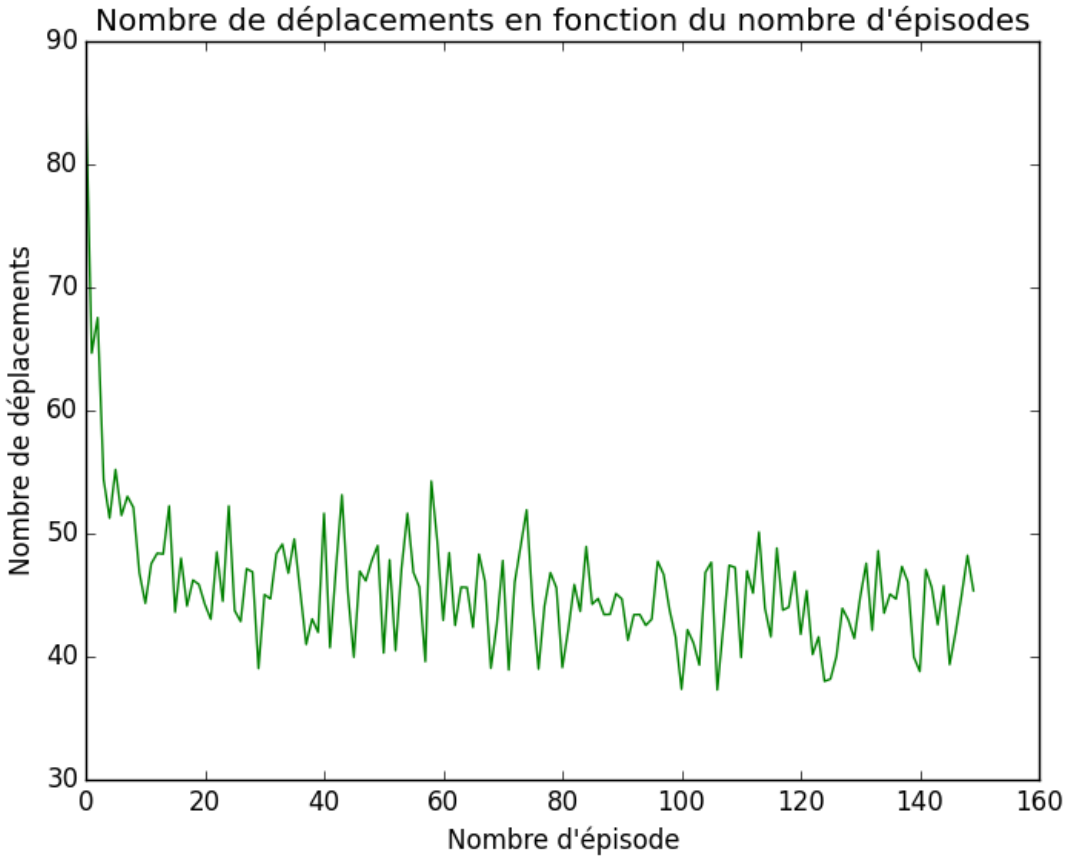

FIGURE 5.2 – Nombre de déplacements en fonction du nombre d'épisodes, stratégie *gloutonne*

5.3 Complexité

La vitesse de convergence nous donne alors la complexité de l'algorithme. On peut voir en figure 5.1 que la politique optimale est obtenue aux alentours de 25, qui est en fait la dimension du labyrinthe ($5 \times 5 = 25$). On en conclut que la **complexité est en** $\mathcal{O}(n)$, avec n la dimension du tableau.

Chapitre 6

Application au jeu de Nim

La méthode du Q-Learning est donc une méthode efficace pour apprendre un certain comportement à un agent. J'ai donc choisi une application où la politique optimale est très simple à trouver : *le jeu de Nim*.
Dans ce jeu mathématique, chaque joueur tire tour à tour 1, 2 ou 3 bâtons placés initialement sur une table de jeu. Le joueur qui tire le dernier bâton est déclaré vainqueur. La politique optimale est simple : pour gagner il faut toujours laisser un nombre multiple de 4 sur la table après tirage. Après quelques phases d'apprentissage, l'algorithme est toujours gagnant si son adversaire ne suit pas la politique optimale, comme on peut le voir sur la figure 6.2 qui représente le ratio de victoires en fonction du nombre d'épisodes d'apprentissage. On remarque aussi que la stratégie $\epsilon-gloutonne$ tend vers un ratio de 0.9 alors que celle *gloutonne* tend seulement vers un ratio de 0.7.

Le tableau des utilités après de nombreux apprentissages :
(avec 100 et -100 comme limites)
Q = [[0, -100, -100, -100], [1, 100, -100, -100], [2, -100, 100, -100], [3, -100, -100, 100], [4, -100, -100, -100], [5, 100, -100, -100], [6, -100, 100, -100], [7, -100, -100, 100], [8, -100, -100, -100], [9, 100, -100, -100], [10, -100, 100, -100], [11, -100, -100, 100], [12, -100, -100, -100], [13, 100, -100, -100], [14, -100, 100, -100], [15, -100, -100, 100], [16, -100, -100, -100], [17, 100, -100, -100], [18, -100, 100, -100], [19, -100, -100, 100], [20, 0, 0, 0]]

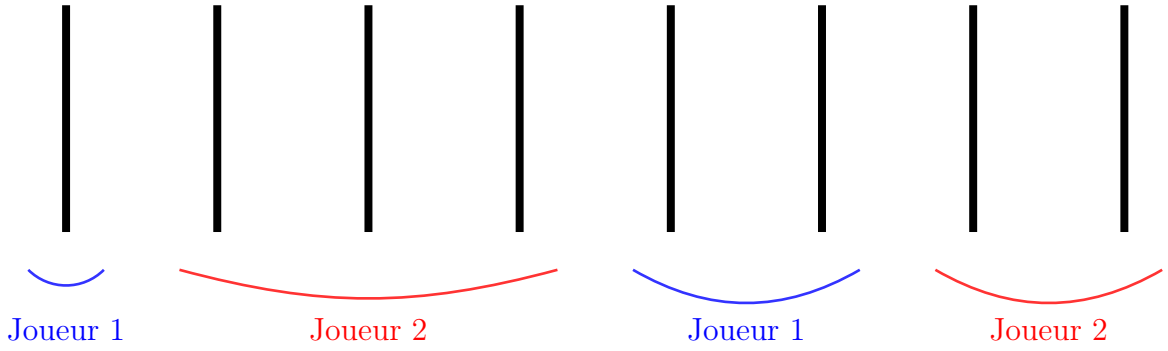

FIGURE 6.1 – Exemple de partie du jeu de Nim

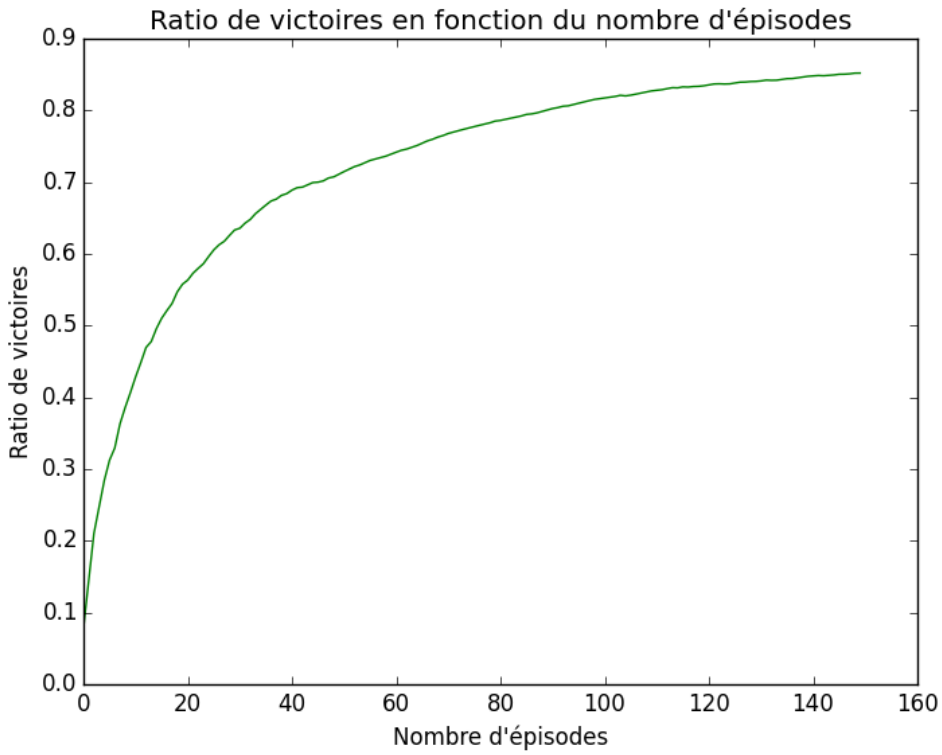

FIGURE 6.2 – Ratio de victoires en fonction du nombre d'épisodes, stratégie $\epsilon - gloutonne$

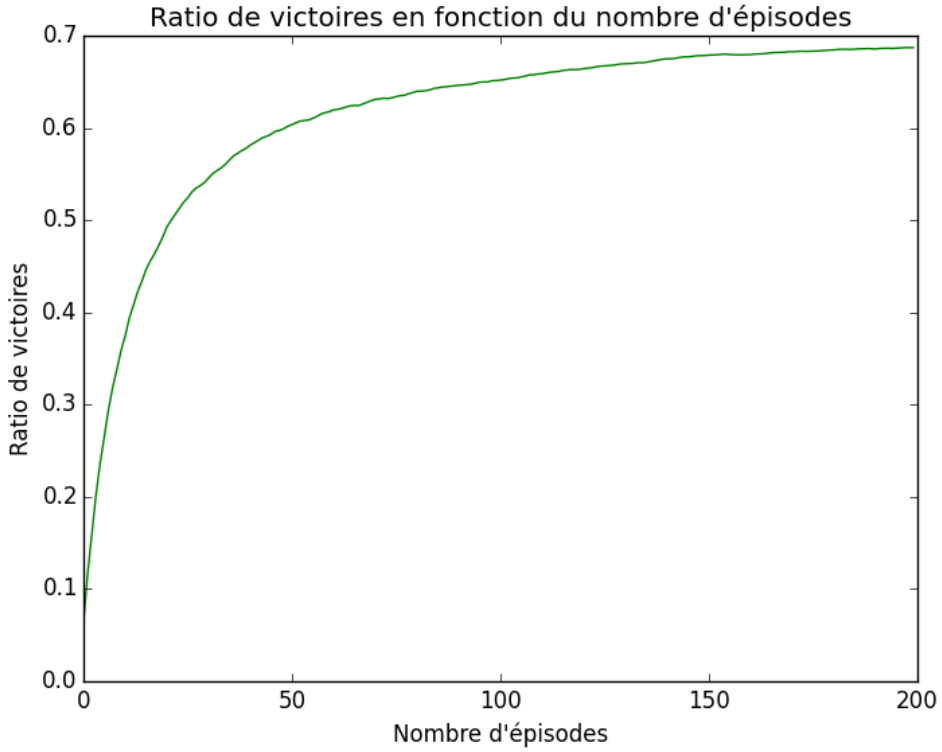

FIGURE 6.3 – Ratio de victoires en fonction du nombre d'épisodes, stratégie *gloutonne*

Cinquième partie

Conclusion générale

Nous pouvons conclure grâce à ces divers résultats, que la meilleure stratégie concernant l'algorithme du Q-Learning est sans aucun doute la stratégie $\epsilon-gloutonne$ avec décroissance progressive de la variable ϵ qui permet à l'agent d'avoir au début une bonne connaissance de son environnement pour ensuite lui permettre d'exploiter au maximum le tableau des utilités qu'il a mis en place. Elle fournit alors une très bonne balance entre exploitation et exploration et permet à l'agent de converger vers la politique optimale le plus rapidement possible.

Sixième partie

Annexe

Listing 6.1 – Maze Runner

```
from random import *
from time import *
from pylab import *
from os import *
from math import *


## QUELQUES LABYRINTHES ET AUTRES VARIABLES:

lab1 = [['E',0,0,0,0], #Labyrinthe qu'on utilisera la plupart du temps
        [-1,-1,0,-1,0],
        [0,0,0,-1,0],
        [-1,0,-1,-1,0],
        [0,0,0,0,'S']]

lab2 = [['E',0,-1,-1,-1,-1,-1], #Labyrinthe de test taille 7x7
        [-1,0,0,0,0,-1,-1],
        [-1,0,-1,-1,0,0,-1],
        [0,0,0,0,0,0,0],
        [0,-1,-1,-1,-1,-1,0],
        [0,0,-1,0,0,0,0],
        [-1,0,0,0,-1,0,'S']]

#Labyrinthe de test taille 10x10
lab3 = [['E',0,-1,0,0,0,-1,-1,-1,-1,-1],
        [-1,0,-1,0,-1,0,-1,0,0,0,0],
        [-1,0,-1,0,-1,0,-1,0,-1,-1,0],
        [-1,0,0,0,-1,0,-1,0,-1,0,0],
        [0,0,-1,0,-1,0,-1,0,-1,0,-1],
        [0,-1,0,0,-1,0,0,0,-1,0,0],
        [0,-1,0,0,0,0,-1,-1,-1,-1,0],
        [0,-1,-1,-1,-1,0,-1,0,0,0,0],
        [0,-1,0,0,0,0,-1,0,-1,-1,0],
        [0,0,-1,0,-1,-1,-1,0,-1,-1,0],
        [-1,0,0,0,0,0,-1,0,0,0,"S"]]

def creation_lab_test(n):
    "Permet de créer un labyrinthe vide pour tester la complexité."

    lab_test = [[0 for i in range(n)] for j in range(n)]
    lab_test[0][0] = 'E'
    lab_test[len(lab_test)-1][len(lab_test)-1] = 'S'
    return lab_test

#déplacements de départ
init_deplacement = [(-1,0),(0,1),(0,-1),(1,0)]
```

```
47 etat_init = [0,0] #état initial
48
49 alpha = 0.1 #vitesse d'apprentissage
50 gamma = 0.1 #facteur d'escompte
51 ## FONCTIONS D'UTILITES
52
53 def est_libre(table,etat):
54     "Renvoie True si la case est vide et False sinon"
55
56     #conditions sur les bords du labyrinthe
57     if 0 <= etat[0] and etat[0] < len(table) and
58                 0 <= etat[1] and etat[1] < len(table[0]):
59
60         #conditions sur le contenu de la case
61         if table[etat[0]][etat[1]] == 0 or
62                         table[etat[0]][etat[1]] == "S" :
63
64             return True
65
66         else:
67
68             return False
69
70     else:
71
72         return False
73
74 def deplacement_possible(table,etat,deplacement):
75     "Renvoie True si le coup est possible."
76
77     #si la case après déplacement est libre,
78     #la case en elle même est libre
79     if est_libre(table,[etat[0]+deplacement[0],
80
81
82         return True
83
84     else:
85
86         return False
87
88 def deplacements_possibles(table,etat):
89     "Renvoie une liste de coups possibles."
90
91     liste = [] #on initialise la liste à la liste vide
92
93     #pour tous les déplacements,
```

```
94      for d in init_deplacement:
95
96          #s'il est possible on l'ajoute
97          if deplacement_possible(table,etat,d):
98
99              liste.append(d)
100
101     return liste
102
103 def est_final(table,etat):
104     "Renvoie True si l'état est final."
105
106     #condition sur la place de la case
107     if etat == [len(table)-1,len(table)-1]:
108
109         return True
110
111     else:
112
113         return False
114
115 def deplacer(table,etat,deplacement):
116     "Déplace l'explorateur du deplacement donné."
117
118     #on remplace la case courante par un 0
119     table[etat[0]][etat[1]] = 0
120     #et celle après déplacement par le joueur
121     table[etat[0]+deplacement[0]][etat[1]+deplacement[1]] = "E"
122     #on met à jour l'état
123     etat = [etat[0]+deplacement[0],etat[1]+deplacement[1]]
124
125     return (table,etat)
126
127 def afficher(table):
128     "Affiche la table courante."
129
130     #on affiche ligne par ligne pour plus de visibilité
131     for i in range(len(table)):
132
133         print(table[i])
134
135     print('\n')
136
137 ## FONCTIONS D'EXPLORATION
138
139 def explorer(table_init,epsilon,n):
140     "Lance l'exploration du labyrinthe."
```

```
141
142     #on initialise la table et l'état
143     table = table_init
144     etat = etat_init
145
146     #si on veut afficher le déplacement, on enlève les # dessous
147     #afficher(table)
148     #sleep(1)
149
150     #on initialise Q
151     Q = init_Q(table,etat)
152
153     #on initialise la table des nombres de déplacements
154     nb_it = []
155
156     #pour le nombre d'épisodes:
157     for i in range(n):
158
159         #on initialise le nombre de mouvement pour l'épisode
160         nb = 0
161
162         #tant que l'état n'est pas final:
163         while etat != [len(table)-1,len(table)-1]:
164
165             #on choisit le déplacement avec la fonction Q
166             (deplacement,Q) = Q_deplacement(table,etat,Q,epsilon)
167
168             #tant que le déplacement n'est pas possible,
169             #on en choisit un autre
170             while not deplacement_possible(table,etat,deplacement):
171
172                 (deplacement,Q) = Q_deplacement(table,etat,Q,epsilon)
173
174             #on déplace l'agent
175             (table,etat) = deplacer(table,etat,deplacement)
176
177             #si on veut afficher la table à partir d'un certain épisode:
178             #if i > 100:
179                 #afficher(table)
180                 #sleep(0.3)
181
182             #on incrémente le nombre de mouvements
183             nb +=1
184
185         #on initialise à nouveau l'état et la table
186         etat = etat_init
187         table[len(table)-1][len(table)-1] = 'S'
```

```
188
189         #on ajoute le nombre de déplacement à la liste
190         nb_it.append(nb)
191
192         #on baisse epsilon
193         epsilon = epsilon
194
195     return(nb_it)
196
197 ## Q-LEARNING
198
199 def init_Q(table,etat):
200     "Fonction d'initialisation du tableau d'utilités."
201
202     #on initialise avec un tableau de même taille que la table
203     #et de profondeur 4
204     Q = [[[0,0,0,0] for i in range(len(table))]
205                                 for j in range(len(table[0]))]
206     return Q
207
208 def top_deplacement(table,etat,Q,possibles):
209     "Choisit le meilleur déplacement à faire."
210
211     #on regarde les utilités des déplacements possibles
212     Qposs = [ Q[etat[0]][etat[1]][i]
213                 if possibles[i] != -1 else -inf for i in range(4)  ]
214     #on prend le maximum
215     maximum = max(Qposs)
216     #on renvoie son index ( sa direction )
217     top = Qposs.index(maximum)
218     return top
219
220 def Q_deplacement(table,etat,Q,epsilon):
221     "Déplacement selon la méthode du Q-learning."
222
223     #on initialise la direction
224     global alpha, gamma
225     direction = 0
226     #on met tous les déplacements à -1 ( "impossible" )
227     possibles = [-1,-1,-1,-1]
228
229     for i in range(4):
230
231         #si le déplacement est possible,
232         if deplacement_possible(table,etat,init_deplacement[i]):
233
234             #on rentre sa direction à la place du -1
```

```
235                 possibles[i] = i
236 
237     #stratégie epsilon - gloutonne
238     if random()<epsilon:
239 
240         #on en prend un au hasard
241         deplacement = choice(deplacements_possibles(table,etat))
242         direction = init_deplacement.index(deplacement)
243 
244     else:
245 
246         #sinon on prend le meilleur
247         direction = top_deplacement(table,etat,Q,possibles)
248         deplacement = init_deplacement[direction]
249 
250     #on initialise la récompense à 0
251     r = 0.00
252 
253     #si l'état est final:
254     if est_final(table,[etat[0]+deplacement[0],
255                                             etat[1]+deplacement[1]]):
256 
257         #la récompense est de +1
258         r = +1.0
259 
260     #on utilise la formule de la qualité:
261     Q[etat[0]][etat[1]][direction] +=  alpha*(r +
262                                 gamma*max(Q[etat[0]+deplacement[0]]
263                                 [etat[1]+deplacement[1]]) -
264                                 Q[etat[0]][etat[1]][direction])
265     return (deplacement,Q)
266 
267 ## AFFICHAGE DES RESULTATS
268 
269 def courbe(table,n,epsilon,m):
270     "Fonction affichant la courbe de la convergence."
271 
272     #on initialise le tableau des nombres de déplacements totaux
273     nb_it_tot = []
274 
275     #et le tableau des sommes et des moyennes pour faire une moyenne
276     somme = [0]*n
277     moyenne = [0]*n
278 
279     #pour la moyenne on fait plusieurs fois:
280     for i in range(m):
281 
```

```
282             #on ajoute le tableau des nombres de déplacements
283             nb_it_tot.append( explorer(table,epsilon,n)   )
284 
285         #on calcule la moyenne:
286         for j in range(n):
287 
288             sommea = 0
289 
290             for i in range(len(nb_it_tot)):
291 
292                 sommea = sommea + nb_it_tot[i][j]
293 
294             somme[j] = sommea
295 
296         for j in range(n):
297 
298             moyenne[j] = somme[j]/m
299 
300         #on met la moyenne sur un graphique
301         a = plot(moyenne,'g')
302         title("Nombre de déplacements en fonction du nombre d'épisodes")
303         a = ylabel("Nombre de déplacements")
304         a = xlabel("Nombre d'épisode")
305         legend()
306 
307         #on affiche
308         show()
```

Listing 6.2 – Jeu de Nim

```
1  from random import *
2  from time import *
3  import os
4  os.chdir("/Users/Stan/Desktop/TIPE")
5 
6  ##ECRITURE ET LECTURE DU FICHIER D'APPRENTISSAGE
7 
8  def existence(nom):
9      "Retourne True s'il existe un fichier de ce nom."
10 
11     return os.path.isfile(nom)
12 
13 def recuperation(nom):
14     "Récupere le tableau d'utilité dans un fichier."
15 
16     fichier = open( nom, 'r')
17     contenu = fichier.readline()
18     contenu = eval(contenu)
```

```
19      fichier.close()
20      return contenu
21
22  def ecriture(texte,nom):
23      "Fonction d'écriture dans le fichier"
24
25      fichier = open ( nom, "w" )
26      fichier.write(texte)
27      fichier.close()
28
29
30  ##FONCTION DE JEU
31
32  def apprentissage(iteration,etat_init):
33      "Fonction permettant à l'algorithme d'apprendre et de jouer."
34
35      nom = 'Sauvegarde.txt'
36
37      trigger = 'appr' #on change le mode en 'apprentissage'
38
39      if not existence(nom) or len(recuperation(nom)) != etat_init +1:
40
41          #s'il n'existe pas de fichier de sauvegarde
42          #ou qu'il n'est pas à la bonne taille
43          print('Creation du fichier de sauvegarde...')
44          input('Appuyez sur entrée pour continuer. \n')
45          ecriture('',nom) #on le crée
46
47          partie = init_partie(etat_init)
48          #on initialise le tableau des utilités
49
50      else: #s'il existe déja un fichier, on demande s'il veut l'utiliser
51
52          print("Voulez vous utiliser le fichier courant ? Y/N \n")
53          rep = input() #on regarde sa réponse
54
55          if rep.lower() == 'n': #si c'est non, on l'écrase
56
57              print('\n Creation du fichier de sauvegarde... ')
58              input('Appuyez sur entrée pour continuer. \n')
59              ecriture('',nom)
60              partie = init_partie(etat_init)
61
62          elif rep.lower() == 'y': #si c'est oui, on le récupère
63
64              print('\n Récupération du fichier de sauvegarde... ')
65              input('Appuyez sur entrée pour continuer.')
```

```
66              partie = recuperation(nom)
67
68      while iteration != 0:
69
70          if iteration % 2 ==0:
71                  #une fois sur deux on fait jouer l'algo en 1er
72
73              jouer(aleatoire,Q_coup,partie,trigger,etat_init)
74
75          else: #ou en 2ème
76
77              jouer(Q_coup,aleatoire,partie,trigger,etat_init)
78
79          iteration = iteration - 1
80
81      ecriture(str(partie),nom)
82      #après l'apprentisage, on remplit le fichier
83      print('Voulez vous jouer contre moi maintenant ? Y/N')
84      rep = input() #on demande si l'utilisateur veut jouer
85
86      while rep.lower() == 'y': # si oui, on change le mode en 'jeu'
87
88          trigger = 'jeu'
89          print("Voulez vous être joueur 1 ou 2 ? \n")
90          rep2 = int(input())
91
92          if rep2 == 1:
93
94              jouer(humain,Q_coup,partie,trigger,etat_init)
95              #et on joue en 1er
96
97          elif rep2 == 2:
98
99              jouer(Q_coup,humain,partie,trigger,etat_init)
100             #ou en 2ème
101
102         print('Voulez vous rejouer ? Y/N \n') #on peut aussi rejouer
103         rep = input()
104
105
106
107 def jouer(joueur1,joueur2,partie,trigger,etat_init):
108     "Fonction de jeu mettant à jour les états durant la partie"
109
110     etat = etat_init #on initialise l'état
111
112     if trigger =='jeu': #si le mode est 'jeu', on affiche tout
```

```
113
114         afficher_etat(etat)
115
116     gagnant = 2 #on initialise le gagnant à deux
117
118     while est_final(etat) == False: #tant que la partie n'es pas finie,
119
120         if trigger =='jeu':
121
122             print('Joueur 1:')
123
124         (partie,coup) = joueur1(partie,etat,trigger)
125         #le joueur 1 choisit son coup
126
127         while not coup in coups_possibles(etat):
128                 #s'il n'est pas possible,
129
130             if trigger =='jeu':
131
132                 print("Erreur, ce coup n'est pas possible !")
133
134             (partie,coup) = joueur1(partie,etat,trigger) #il recommence
135
136         etat = update_etat(etat, coup) #on met à jour l'état
137
138         if trigger == 'jeu':
139
140             print(coup," batons enlevé(s). \n")
141             sleep(0.3) #on attend un peu pour la lisibilité
142             afficher_etat(etat) #on affiche l'état courant
143             sleep(0.3)
144
145         if etat == 0: #si l'état est final, le joueur 1 a gagné
146
147             if trigger =='jeu':
148
149                 gagnant = 1
150
151             break #on sort de la boucle while
152
153         #ON FAIT DE MEME POUR LE JOUEUR 2
154
155         if trigger =='jeu':
156
157             print('Joueur 2:')
158             sleep(0.3)
159
```

```
160             (partie,coup) = joueur2(partie,etat,trigger)
161 
162             while not coup in coups_possibles(etat):
163 
164                 if trigger =='jeu':
165 
166                     print("Erreur, ce coup n'est pas possible !")
167 
168                 (partie,coup) = joueur2(partie,etat,trigger)
169 
170             etat = update_etat(etat, coup)
171 
172             if trigger =='jeu':
173 
174                 print(coup," batons enlevé(s). \n")
175                 sleep(0.3)
176                 afficher_etat(etat)
177                 sleep(0.3)
178 
179         if trigger == 'jeu':
180 
181             print('## Le gagnant est le joueur '+str(gagnant)+' ! ## \n')
182 
183 ##CREATION DU TYPE PARTIE
184 
185 def init_partie(n):
186     "Initialise une partie à n batons."
187 
188     partie = [[i,0,0,0] for i in range(0,n+1)]
189 
190     for i in range(len(partie)):
191 
192         for j in range(1,4):
193 
194             if not j in coups_possibles(i):
195 
196                 partie[i][j] = -100
197 
198     return partie
199 
200 def modifier_utilite(etat,coup,utilite,partie):
201     "Modifie l'utilité d'un état donné pour un coup donné."
202 
203     partie[etat][coup] = utilite
204     return partie
205 
206 ##FONCTIONS D'UTILITES
```

```
207
208
209 def est_final(etat):
210     "Renvoie True si l'état est final"
211
212     if etat == 0 or etat == 1:
213
214         return True
215
216     else:
217
218         return False
219
220 def coups_possibles(etat):
221     "Renvoie une liste des coups possibles."
222
223     if etat == 0:
224
225         return []
226
227     if etat == 1:
228
229         return [1]
230
231     elif etat == 2:
232
233         return[1,2]
234
235     else:
236
237         return [1,2,3]
238
239 def afficher_etat(etat):
240     "Affiche le nb de batons restants."
241
242     print("## Il reste ",etat," batons. ## \n")
243
244 def update_etat(etat, coup):
245     "Renvoie l'état qui suit après un coup donné."
246
247     return (etat - coup)
248
249 def etat_final(etat):
250     "Renvoie gagné ou perdu si l'état est final."
251
252     if etat == 1:
253
```

```
254             return('perdu')
255 
256         elif etat == 0:
257 
258             return('gagné')
259 
260         else:
261 
262             return None
263 
264 ##JOUEURS
265 
266 def humain(partie,etat,trigger):
267     "Permet à un humain de joueur contre l'algorithme."
268 
269     coup = int(input('Prenez 1, 2 ou 3 batons: \n'))
270     return (partie,coup)
271 
272 def aleatoire(partie,etat,trigger):
273     "Adversaire qui joue des coups aléatoires."
274 
275     return (partie,choice(coups_possibles(etat)))
276 
277 ##FONCTION D'APPRENTISSAGE
278 
279 def top_coup(partie,etat):
280     "Retourne le coup d'utilité maximale"
281 
282     return partie[etat].index(max(partie[etat][1::]))
283 
284 def Q_coup(partie,etat,trigger):
285     "Algorithme de remplissage du tableau d'utilités"
286 
287     global alpha,epsilon,gamma
288     coup = 1 #on initialise le coup à 1
289 
290     if random() < epsilon or etat == 21 and trigger == 'appr':
291         #on introduit de l'exploration aléatoire
292 
293         coup = choice(coups_possibles(etat))
294 
295     else: #sinon, on prend le coup d'utilité maximale
296 
297         coup = top_coup(partie,etat)
298 
299         while not coup in coups_possibles(etat):
300             #tant qu'il n'est pas jouable, on en reprend un autre
```

```
301
302             coup = top_coup(partie,etat)
303
304         if trigger == 'appr': #si le mode est 'apprentissage',
305             #on introduit le mouvement
306             #permettant une meilleure exploration
307
308             aleatoire = randrange(100)
309
310             if 80 < aleatoire <= 90:
311
312                 coup = (coup + 1)%3
313
314             elif 90 < aleatoire:
315
316                 coup = (coup - 1)%3
317
318     r = 0.0 #on initialise la récompense à 0.0
319
320     if est_final(update_etat(etat,coup)): #si l'état d'après est final
321
322         if etat_final(update_etat(etat,coup)) == "gagné":
323                 #et qu'il est gagnant
324
325             r = +1.0 #on augmente la récompense
326
327         elif etat_final(update_etat(etat,coup)) == "perdu":
328                 #s'il est perdant,
329
330             r = -1.0 #on la diminue
331
332     if coup in coups_possibles(etat):
333         #permet de ne pas changer l'utilité de 0
334
335         #on utilise la formule de l'utilité
336         partie[etat][coup] -=  alpha*(r + gamma*max(partie
337         [update_etat(etat,coup)][1::]) - partie[etat][coup])
338
339         #si l'utilité est plus grande que 100, on fixe un maximum
340         if partie[etat][coup] >= 100:
341
342             partie[etat][coup] = 100
343
344         #et si elle est plus petite que -100, un minimum
345         elif partie[etat][coup] <= -100:
346
347             partie[etat][coup] = -100
```

```
348
349         else:
350
351             return(partie,coup)
352
353         return (partie,coup)
354
355 ##COURBES D'AFFICHAGES DES RATIOS
356
357
358 ##PARAMETRES EXTERIEURS
359
360 alpha = 1 #facteur d'apprentissage
361 gamma = 0.9 #facteur d'escompte
362 epsilon = 0.1 #facteur d'exploration
```