

Apprentissage par renforcement | Contrôle continu

Réalisé par : Salem Amenallah¹, Ben Ammar Aziz² & Gaidi Mohamed amine³

Université Paris-Dauphine | Tunis



Table des matières

1	Première partie : Les codes Bandit-Manchot	2
2	Exercices du TP	14
2.1	Iteation de valeurs	14
2.2	Itération des politiques	19

1. amenallah.salem@dauphine.tn
2. aziz.benammar@dauphine.tn
3. mohamedamine.gaidi@dauphine.tn

1 Première partie : Les codes Bandit-Manchot

Exemple bandit manchot :

L'exemple d'environnement d'essai utilisé est un bandit stochastique à 10 manchots dont les valeurs déclenchant l'action sont normalement distribuées avec une moyenne nulle et une variance unitaire. La récompense de chaque bras est elle-même normalement distribuée avec une moyenne égale à la valeur d'action de la variance du bras et de l'unité. Dans ce qui suit, on va implémenter la récompense moyenne et le pourcentage de choix optimal fait sur 2000 expériences de 1000 essais chacune, où chaque expérience correspond à une sélection aléatoire différente de valeurs d'action du manchot.

Nous présentons ici la politique Epsilon-Glouton, qui, à chaque essai, explore au hasard les choix avec probabilité epsilon ou exploite le meilleur choix apparent avec probabilité 1-epsilon. Par exemple, le fait de choisir epsilon = 0,01 est un choix courant, qui explore l'ensemble des choix 1 pour cent du temps. Donc un agent est capable de prendre une initiative parmi un ensemble d'actions à chaque pas du temps. Donc on commence notre code par L'initiative e de choisir à l'aide d'une stratégie basée sur l'historique des actions antérieures et des observations des résultats.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 class Agent(object):
6
7     def __init__(self, manchots, strategie, resultat_initiale=0):
8
9         self.strategie = strategie
10
11        self.z = manchots.z
12
13        self.resultat_initiale = resultat_initiale
14
15        self.estimation = resultat_initiale * np.ones(self.z)
16
17        self.tentatives = np.zeros(self.z)
18
19        self.w = 0
20
21        self.action_finale = None
22
23    def __str__(self):
24
25        return '{}'.format(str(self.strategie))
26
27    def reset(self):
```

Après on réinitialise la mémoire de l'agent à un état initial :

```
35        self.estimation[:] = self.resultat_initiale
36
37        self.tentatives[:] = 0
38
39        self.action_finale = None
40
41        self.w = 0
42
43    def selection(self):
44
45        initiative = self.strategie.selection(self)
46
47        self.action_finale = initiative
48
49        return initiative
50
51    def constatation(self, recompense):
52
53        self.tentatives[self.action_finale] += 1
54
55        Q1 = 1 / self.tentatives[self.action_finale]
56
57        Q2 = self.estimation[self.action_finale]
58
59        self.estimation[self.action_finale] += Q1 * (recompense - Q2)
60
61        self.w += 1
62
63        @property
64        def approximations(self):
65
66            return self.estimation
67
68
69    class BanditMultiManchots(object):
70
71        def __init__(self, z):
72
73            self.z = z
74
75            self.initiative = np.zeros(z)
76
77            self.optimalite = 0
78
79        def reset(self):
80
81            self.initiative = np.zeros(self.z)
82
83            self.optimalite = 0
84
85        def pull(self, initiative):
86            return 0, True
87
88    class B_Manchots(BanditMultiManchots):
```

Les B-Manchots modélisent la récompense d'un bras donné comme distribution normale

avec la moyenne et l'écart type fournis.

```
94     def __init__(self, z, mu=0, sigma=1):
95         super(B_Manchots, self).__init__(z)
96
97         self.mu = mu
98
99         self.sigma = sigma
100
101        self.reset()
102
103    def reset(self):
104        self.initiative = np.random.normal(self.mu, self.sigma, self.z)
105
106        self.optimalite = np.argmax(self.initiative)
107
108    def pull(self, initiative):
109        return (np.random.normal(self.initiative[initiative]),
110               initiative == self.optimalite)
111
112
113 #####
114
115 class Policy(object):
```

Une politique prescrit une mesure à prendre en fonction de la mémoire d'un agent

```
121
122     def __str__(self):
123         return 'stratégie générique'
124
125     def selection(self, agent):
126         return 0
127
128
129 class Glouton(Policy):
```

La stratégie de d'Epsilon-Glouton choisira une initiative aléatoire avec probabilité epsilon et adoptera la meilleure approche apparente avec probabilité 1-epsilon. Si plusieurs actions sont liées pour le meilleur choix, alors une initiative aléatoire de ce sous-ensemble est sélectionnée.

```
139     def __init__(self, epsilon):
140
141         self.epsilon = epsilon
142
143     def __str__(self):
144
145         return '\u03b5-glouton (\u03b5={})'.format(self.epsilon)
146
147     def selection(self, agent):
148
149         if np.random.random() < self.epsilon:
150
151             return np.random.choice(len(agent.approximations))
152
153         else:
154
155             initiative = np.argmax(agent.approximations)
156
157             verification = np.where(agent.approximations == initiative)[0]
158
159             if len(verification) == 0:
160
161                 return initiative
162
163             else:
164
165                 return np.random.choice(verification)
```

```

176     def __init__(self, manchots, strategies, label='Bandit-Manchots'):
177
178         self.manchots = manchots
179
180         self.strategies = strategies
181
182         self.label = label
183
184     def reset(self):
185
186         self.manchots.reset()
187
188         for agent in self.strategies:
189             agent.reset()
190
191     def run(self, num_essais=1000, num_experimentations=2000):
192
193         resultats = np.zeros((num_essais, len(self.strategies)))
194
195         optimalite = np.zeros_like(resultats)
196
197         for _ in range(num_experimentations):
198
199             self.reset()
200
201             for w in range(num_essais):

```

```

    202                 for w in range(num_essais):
203
204                     for i, agent in enumerate(self.strategies):
205
206                         initiative = agent.selection()
207
208                         recompense, solu_optimal = self.manchots.pull(initiative)
209
210                         agent.constatation(recompense)
211
212                         resultats[w, i] += recompense
213
214                         if solu_optimal:
215                             optimalite[w, i] += 1
216
217             resultats = resultats / (num_experimentations + num_essais)
218             optimalite = optimalite / num_experimentations
219
220         return resultats, optimalite

```

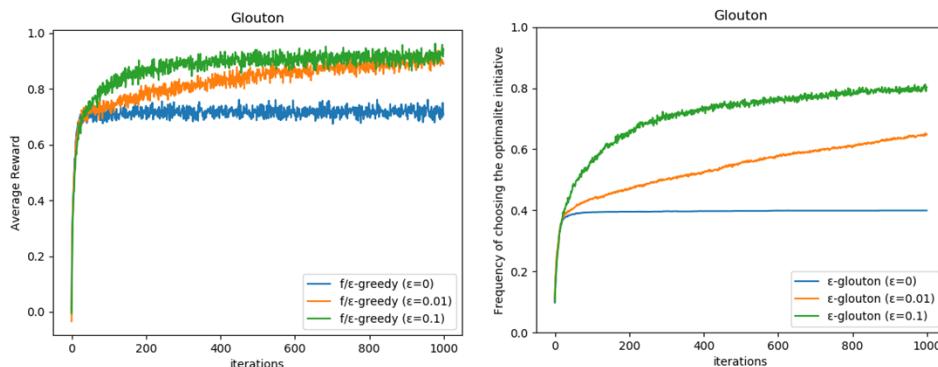
Le tableau de bord :

```

242     manchots = B_Manchots(10, mu=0) # manchots = 10
243     num_essais, num_experimentations = 1000, 2000
244
245     strategies = [
246         Agent(manchots, Glouton(0)),
247         Agent(manchots, Glouton(0.01)),
248         Agent(manchots, Glouton(0.1)),
249     ]
250
251     map = Map(manchots, strategies, 'Glouton')
252     resultats, optimalite = map.run(num_essais, num_experimentations)
253     map.plots(resultats, optimalite)
254

```

Donc on visualise :



Comme on peut le voir, quand l'epsilon est à 0, c'est peu gourmand, donc ça va toujours nous donner les mêmes récompenses moyennes. Et si nous augmentons les epsilons à 0,01, il nous faut un peu de temps pour faire mieux. Mais même s'il y a un peu de bruit, c'est difficile à voir, Sachant que epsilon=0.01 fait bien mieux à la fin.

A propos d'epsilon 0.1, Nous explorons plus, alors nous avons demandé de faire le traitement plus vite mais nous voulons le bien faire à long terme. Parce qu'ici, on gaspille encore nos tests à faire de l'exploration.

Alors lequel est le meilleur choix ? Si nous allons faire cela pendant très longtemps, alors peut-être que nous choisirons un epsilon très petit. Parce qu'en fin de compte, nous allons asymptotiquement vers une valeur plus élevée, mais si nous n'allons pas exécuter cette fonction pendant longtemps, ça serait judicieux qu'on choisisse un epsilon plus grand. Et ainsi, l'exploration se fait rapidement.

Si nos estimations initiales sont basses, l'approche Epsilon-Glouton s'enclenchera sur le premier résultat positif et n'essaiera jamais rien d'autre. Cependant, si nous commençons par une estimation initiale élevée, cela a pour effet d'induire une exploration précoce, car les estimations sont réduites. Essayons de commencer avec une attente préalable de `resultat_initiale=5`. Un agent est capable de prendre une initiative parmi un ensemble d'actions à chaque temps. L'initiative est de choisir à l'aide d'une stratégie basée sur l'historique des actions antérieures et des observations des résultats.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 class Agent(object):
4
5     def __init__(self, manchots, strategie, resultat_initiale=0, alpha=None):
6
7         self.strategie = strategie
8
9         self.z = manchots.z
10
11        self.resultat_initiale = resultat_initiale
12
13        self.alpha = alpha
14
15        self.estimation = resultat_initiale*np.ones(self.z)
16
17        self.tentatives = np.zeros(self.z)
18
19        self.w = 0
20
21        self.action_finale = None
22
23    def __str__(self):
24
25        return '{}'.format(str(self.strategie))
26
27    def reset(self):

```

Donc maintenant on réinitialise la mémoire de l'agent à un état initiale

```

35         self.estimation[:] = self.resultat_initiale
36
37         self.tentatives[:] = 0
38
39         self.action_finale = None
40
41         self.w = 0
42
43     def selection(self):
44
45         initiative = self.strategie.selection(self)
46
47         self.action_finale = initiative
48
49         return initiative
50
51     def constatation(self, recompense):
52
53         self.tentatives[self.action_finale] += 1
54
55
56         if self.alpha is None:
57
58             Q1 = 1 / self.tentatives[self.action_finale]
59
60         else:
61
62             Q1 = self.alpha
63
64             Q2 = self.estimation[self.action_finale]
65
66
67             self.estimation[self.action_finale] += Q1*(recompense - Q2)
68
69             self.w += 1

```

Bandit multi-manchots :

```

75     def approximations(self):
76         return self.estimation
77
78     class BanditMultiManchots(object):
79
80
81         def __init__(self, z):
82             self.z = z
83
84             self.initiative = np.zeros(z)
85
86             self.optimalite = 0
87
88
89         def reset(self):
90
91             self.initiative = np.zeros(self.z)
92
93             self.optimalite = 0
94
95         def pull(self, initiative):
96
97             return 0, True
98
99
100    class B_Manchots(BanditMultiManchots):
101

```

Les B_manchots modélisent le récompense d'un bras donné comme une distribution normale avec la moyenne et l'écart type

```

110     def __init__(self, z, mu=0, sigma=1):
111
112         super(B_Manchots, self).__init__(z)
113
114         self.mu = mu
115
116         self.sigma = sigma
117
118         self.reset()
119
120
121     def reset(self):
122
123         self.initiative = np.random.normal(self.mu, self.sigma, self.z)
124
125         self.optimalite = np.argmax(self.initiative)
126
127
128     def pull(self, initiative):
129
130         return (np.random.normal(self.initiative[initiative]),
131                 initiative == self.optimalite)
132
133
134 ##########
135 class Policy(object):

```

Une politique prescrit une mesure à apprendre en fonction de la mémoire d'un agent

```

143     def __str__(self):
144
145         return 'stratégie générique'
146
147
148
149     def selection(self, agent):
150
151         return 0
152
153
154
155
156 class Glouton(Policy):
157

```

La stratégie d'epsilon-Glouton choisira une initiative aléatoire avec probabilité epsilon et adoptera la meilleure approche apparente avec probabilité 1-epsilon. Si plusieurs action son liées pour le meilleur choix ,alors une initiative aléatoire de ce sous-ensemble est sélectionné

```
def __init__(self, epsilon):
    self.epsilon = epsilon

def __str__(self):
    return '\u0385-glouton (\u0385={})'.format(self.epsilon)

def selection(self, agent):
    if np.random.random() < self.epsilon:
        return np.random.choice(len(agent.approximations))
    else:
        initiative = np.argmax(agent.approximations)
        verification = np.where(agent.approximations == initiative)[0]
        if len(verification) == 0:
            return initiative
        else:
            return np.random.choice(verification)

class Map(object):
    def __init__(self, manchots, strategies, label='Bandit-Manchots'):
        self.manchots = manchots
        self.strategies = strategies
        self.label = label

    def reset(self):
        self.manchots.reset()
        for agent in self.strategies:
            agent.reset()

    def run(self, num_essais=1000, num_experimentations=2000):
        resultats = np.zeros((num_essais, len(self.strategies)))
        optimalite = np.zeros_like(resultats)
```

```
for _ in range(num_experimentations):
    self.reset()

    for w in range(num_essais):
        for i, agent in enumerate(self.strategies):
            initiative = agent.selection()
            recompense, solu_optimal = self.manchots.pull(initiative)
            agent.constatation(recompense)

            resultats[w, i] += recompense
            if solu_optimal:
                optimaleite[w, i] += 1

resultats = resultats / (num_experimentations + num_essais)
optimaleite = optimaleite / num_experimentations
return resultats, optimaleite
```

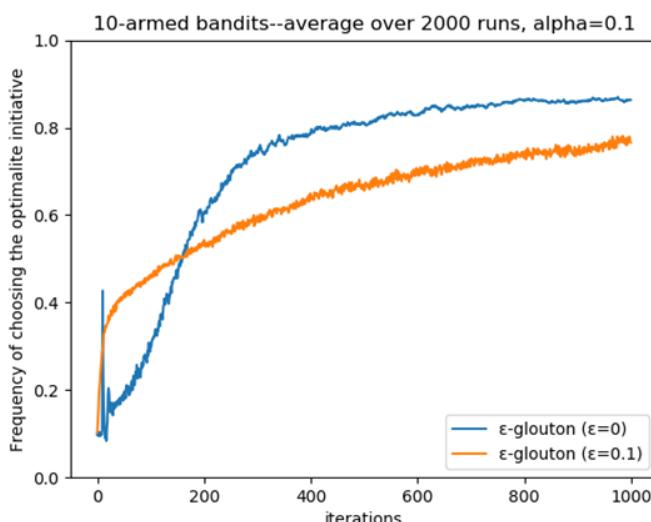


```
def plots(self, resultats, optimaleite):
    plt.title(self.label)
    plt.plot(optimaleite)
    plt.ylim(0, 1)
    plt.ylabel('Frequency of choosing the optimalite initiative')
    plt.xlabel('iterations')
    plt.legend(self.strategies, loc=4)
    plt.show()
```

Le tableau de bord :

```
282 manchots = B_Manchots(10, mu=0).manchots = 10
283 num_essais, num_experimentations = 1000, 2000
284
285
286
287 strategies = [
288     Agent(manchots, Glouton(0), resultat_initiale=5, alpha=0.1),
289     Agent(manchots, Glouton(0.1), alpha=0.1),
290 ]
291
292
293 map = Map(manchots, strategies, '10-armed bandits--average over 2000 runs, alpha=0.1')
294 resultats, optimalite = map.run(num_essais, num_experimentations)
295 mplots(resultats, optimalite)
```

On visualise :



Conclusion : C'est une ruse qui fonctionne généralement bien pour les problèmes stationnaires.

Un agent est capable de prendre une initiative parmi un ensemble d'actions à chaque pas de temps. L'initiative est choisie à l'aide d'une stratégie basée sur l'historique des actions antérieures et des observations des résultats

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 class Agent(object):
6
7     def __init__(self, manchots, strategie, resultat_initiale=0, alpha=0.1):
8
9         self.strategie = strategie
10
11         self.k = manchots.k
12
13         self.resultat_initiale = resultat_initiale
14
15         self.alpha = alpha
16
17         self.estimation = resultat_initiale*np.ones(self.k)
18
19         self.tentatives = np.zeros(self.k)
20
21         self.z = 0
22
23         self.action_finale = None
24
25     def __str__(self):
26
27         return 'f/{}'.format(str(self.strategie))

```

```

31     def reboot(self):
32
33         self.estimation[:] = self.resultat_initiale
34
35         self.tentatives[:] = 0
36
37         self.action_finale = None
38
39         self.z = 0
40
41     def selection(self):
42
43         tentative = self.strategie.selection(self)
44
45         self.action_finale = tentative
46
47         self.z += 1
48
49         return tentative

```

On passe maintenant à Réinitialiser la mémoire de l'agent à un état initial.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 class Agent(object):
6
7     def __init__(self, manchots, strategie, resultat_initiale=0, alpha=0.1):
8
9         self.strategie = strategie
10
11         self.k = manchots.k
12
13         self.resultat_initiale = resultat_initiale
14
15         self.alpha = alpha
16
17         self.estimation = resultat_initiale*np.ones(self.k)
18
19         self.tentatives = np.zeros(self.k)
20
21         self.z = 0
22
23         self.action_finale = None
24
25     def __str__(self):
26
27         return 'f/{}'.format(str(self.strategie))

```

```

31     def reboot(self):
32
33         self.estimation[:] = self.resultat_initiale
34
35         self.tentatives[:] = 0
36
37         self.action_finale = None
38
39         self.z = 0
40
41     def selection(self):
42
43         tentative = self.strategie.selection(self)
44
45         self.action_finale = tentative
46
47         self.z += 1
48
49         return tentative

```

```

53     def constatation(self, recompense):
54
55         self.tentatives[self.action_finale] += 1
56
57         if self.alpha is None:
58
59             Q1 = 1 / self.tentatives[self.action_finale]
60
61         else:
62
63             Q1 = self.alpha
64
65             Q2 = self.estimation[self.action_finale]
66
67             self.estimation[self.action_finale] += Q1*(recompense - Q2)
68
69         self.z += 1
70
71     def approximations(self):
72
73         return self.estimation

```

```

79 class BanditMultiManchots(object):
80
81
82     def __init__(self, k):
83
84         self.k = k
85
86         self.initiative = np.zeros(k)
87
88         self.optimalite = 0
89
90
91     def reboot(self):
92
93         self.initiative = np.zeros(self.k)
94
95         self.optimalite = 0
96
97
98     def tirage(self, tentative):
99
100        return 0, True

```

Les B Manchots modélisent la récompense d'un bras donné comme distribution normale avec la moyenne et le type-fourni fournis.

```

105     class B_Manchots(BanditMultiManchots):
106
107         def __init__(self, k, mu=0, sigma=1):
108
109             super(B_Manchots, self).__init__(k)
110
111             self.mu = mu
112
113             self.sigma = sigma
114
115             self.reboot()
116
117         def reboot(self):
118
119             self.initiative = np.random.normal(self.mu, self.sigma, self.k)
120
121             self.optimalite = np.argmax(self.initiative)
122
123
124         def tirage(self, tentative):
125
126             return (np.random.normal(self.initiative[tentative]),
127
128                         tentative == self.optimalite)

```

Une politique prescrit une mesure à prendre en fonction de la mémoire d'un agent.

```

131     class Policy(object):
132
133         def __str__(self):
134
135             return 'generic strategie'
136
137         def selection(self, agent):
138
139             return 0

```

La stratégie d'Epsilon-Glouton choisira une initiative aléatoire avec probabilité epsilon et adoptera la meilleure approche apparente avec probabilité 1-epsilon

Si plusieurs actions sont liées pour le meilleur choix, alors une initiative aléatoire de ce sous-ensemble est sélectionnée.

```

143     class Glouton(Policy):
144
145         def __init__(self, epsilon):
146
147             self.epsilon = epsilon
148
149         def __str__(self):
150
151             return '\u0335-glouton (\u0335={})'.format(self.epsilon)
152
153         def selection(self, agent):
154
155             if np.random.random() < self.epsilon:
156
157                 return np.random.choice(len(agent.approximations))
158             else:
159                 initiative = np.argmax(agent.approximations)
160
161                 verification = np.where(agent.approximations == initiative)[0]
162
163                 if len(verification) == 0:
164
165                     return initiative
166
167                 else:
168                     return np.random.choice(verification)

```

L'algorithme de la limite supérieure de confiance (UCB). Il applique un facteur d'exploration à la valeur attendue de chaque manchot qui peut influencer une stratégie de sélection gourmande pour explorer plus intelligemment les options moins confiantes.

```

172     class UpperConfidenceBound(Policy):
173
174         def __init__(self, c):
175
176             self.c = c
177
178         def __str__(self):
179
180             return 'UCB (c={})'.format(self.c)
181
182         def selection(self, agent):
183
184             exploration = np.log(agent.z+1) / agent.tentatives
185
186             exploration[np.isnan(exploration)] = 0
187
188             exploration = np.power(exploration, 1/self.c)
189
190             Q2 = agent.approximations + exploration
191
192             tentative = np.argmax(Q2)
193
194             check = np.where(Q2 == tentative)[0]
195
196             if len(check) == 0:
197
198                 return tentative
199             else:
200                 return np.random.choice(check)

201     class Map(object):
202
203         def __init__(self, manchots, strategies, label='Bandit-Manchots'):
204
205             self.manchots = manchots
206
207             self.strategies = strategies
208
209             self.label = label
210
211         def reboot(self):
212
213             self.manchots.reboot()
214
215             for agent in self.strategies:
216                 agent.reboot()

```

```

218     def run(self, num_essais=1000, num_experimentations=2000):
219
220         resultats = np.zeros((num_essais, len(self.strategies)))
221
222         optimalite = np.zeros_like(resultats)
223
224         for _ in range(num_experimentations):
225
226             self.reboot()
227
228             for w in range(num_essais):
229
230                 for i, agent in enumerate(self.strategies):
231
232                     initiative = agent.selection()
233
234                     recompense, solu_optimal = self.manchots.tirage(initiative)
235
236                     agent.constatation(recompense)
237
238                     resultats[w, i] += recompense
239
240                     if solu_optimal:
241                         optimalite[w, i] += 1
242
243         resultats = resultats / (num_experimentations + num_essais)
244         optimalite = optimalite / num_experimentations
245
246     def plots(self, resultats, optimalite):
247
248         plt.title(self.label)
249
250         plt.plot(optimalite)
251
252         plt.ylim(0, 1)
253
254         plt.ylabel('Frequency of choosing the optimalite initiative')
255
256         plt.xlabel('iterations')
257
258         plt.legend(self.strategies, loc=4)
259
260         plt.show()

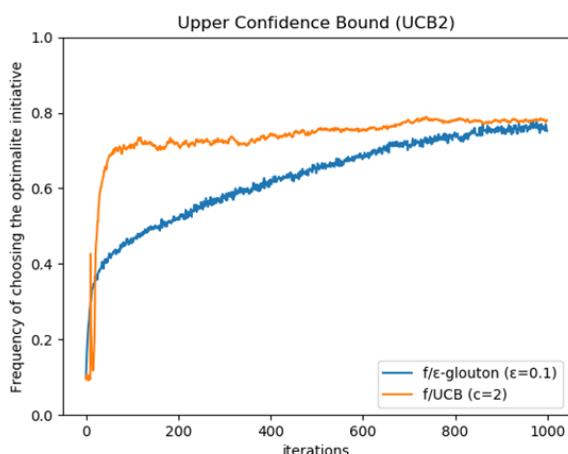
```

```

269     manchots = B_Manchots(10, mu=0) # manchots = 10
270     num_essais, num_experimentations = 1000, 2000
271
272     strategies = [
273         Agent(manchots, Glouton(0.1)),
274         Agent(manchots, UpperConfidenceBound(2))
275     ]
276     map = Map(manchots, strategies, 'Upper Confidence Bound (UCB2)')
277     resultats, optimalite = map.run(num_essais, num_experimentations)
278     map.plots(resultats, optimalite)

```

On obtient ce résultat :



Au lieu d'estimer la récompense attendue de la sélection d'un manchot particulier, nous pouvons nous intéresser uniquement à la préférence relative d'un manchot par rapport à un autre.

Un agent est capable de prendre une action parmi un ensemble d'actions à chaque étape de temps. L'initiative est choisie à l'aide d'une stratégie basée sur l'historique des actions résultat précédentes et des observations de résultats.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 class Agent(object):
6
7     def __init__(self, manchots, strategie, resultat_precedent=0, alpha=0.1):
8
9         self.strategie = strategie
10
11         self.k = manchots.k
12
13         self.resultat_precedent = resultat_precedent
14
15         self.alpha = alpha
16
17         self.estimation = resultat_precedent*np.ones(self.k)
18
19         self.tentatives = np.zeros(self.k)
20
21         self.t = 0
22
23         self.tentative_finale = None

```

Réinitialiser la mémoire de l'agent à un état initial

```

27 def __str__(self):
28     return 'f/{}'.format(str(self.strategie))
29
30 def reboot(self):
31
32     self.estimation[:] = self.resultat_precedent
33
34     self.tentatives[:] = 0
35
36     self.tentative_finale = None
37
38     self.t = 0
39
40
41     def selection(self):
42
43         initiative = self.strategie.selection(self)
44
45         self.tentative_finale = initiative
46
47         return initiative
48
49
50     def constatation(self, recompense):
51
52         self.tentatives[self.tentative_finale] += 1
53
54
55         if self.alpha is None:
56
57             Q1 = 1 / self.tentatives[self.tentative_finale]
58
59         else:
60
61             Q1 = self.alpha
62
63
64             Q2 = self.estimation[self.tentative_finale]
65
66
67             self.estimation[self.tentative_finale] += Q1*(recompense - Q2)
68
69
70             self.t += 1

```

Le Gradient Agent approuve la différence relative entre les actions au lieu de déterminer les estimations des valeurs de récompense.

Il approuve effectivement à privilégier une initiative plutôt qu'une autre.

```

73     def value_estimates(self):
74         return self.estimation
75
76     class GradientAgent(Agent):
77
78         def __init__(self, manchots, strategie, resultat_precedent=0, alpha=0.1, baseline=True):
79
80             super(GradientAgent, self).__init__(manchots, strategie, resultat_precedent)
81
82             self.alpha = alpha
83
84             self.baseline = baseline
85
86             self.moyenne_recompense = 0
87
88         def __str__(self):
89             return 'Q1/\\u0301{}, bl={}'.format(self.alpha, self.baseline)
90
91
92
93     def constatation(self, recompense):
94
95         self.tentatives[self.tentative_finale] += 1
96
97
98         if self.baseline:
99
100             sep = recompense - self.moyenne_recompense
101
102             self.moyenne_recompense += 1/np.sum(self.tentatives) * sep
103
104
105             yp = np.exp(self.value_estimates) / np.sum(np.exp(self.value_estimates))
106
107             S_x = self.value_estimates[self.tentative_finale]
108
109             S_x += self.alpha*(recompense - self.moyenne_recompense)*(1-yp[self.tentative_finale])
110
111             self.estimation -= self.alpha*(recompense - self.moyenne_recompense)*yp
112
113             self.estimation[self.tentative_finale] = S_x
114
115
116         self.t += 1
117
118         def reboot(self):
119
120             super(GradientAgent, self).reboot()
121
122             self.moyenne_recompense = 0
123
124
125     class BanditMultiManchots(object):
126
127         def __init__(self, k):
128
129             self.k = k
130
131             self.action_values = np.zeros(k)
132
133             self.optimalite = 0
134
135
136         def reboot(self):
137
138             self.action_values = np.zeros(self.k)
139
140             self.optimalite = 0
141
142
143         def tirage(self, initiative):
144
145             return 0, True

```

B-Manchots modélise la rétribution d'un manchot donné comme une distribution normale avec une moyenne et un écart-type fournis

```

149     class B_Manchots(BanditMultiManchots):
150
151     def __init__(self, k, mu=0, ecart_type=1):
152
153         super(B_Manchots, self).__init__(k)
154
155         self.mu = mu
156
157         self.ecart_type = ecart_type
158
159         self.reboot()
160
162 ⚡ def reboot(self):
163
164     self.action_values = np.random.normal(self.mu, self.ecart_type, self.k)
165
166     self.optimalite = np.argmax(self.action_values)
167
168
169 ⚡ def tirage(self, initiative):
170
171     return (np.random.normal(self.action_values[initiative]),
172             initiative == self.optimalite)
173
174

```

Une stratégie prescrit une initiative à entreprendre en fonction de la mémoire d'un agent.

```

176     class Strategie(object):
177
178     def __str__(self):
179
180         return 'generic strategie'
182
183     def selection(self, agent):
184
185         return 0

```

La stratégie Epsilon-Greedy choisira une initiative aléatoire avec probabilité epsilon et adoptera la meilleure approche apparente avec obtenu 1-epsilon.

Si plusieurs actions sont à égalité pour le meilleur choix, alors une initiative aléatoire de ce sous-ensemble est choisie.

```

194     class Glouton(Strategie):
195
196     def __init__(self, epsilon):
197
198         self.epsilon = epsilon
199
200 ⚡ def __str__(self):
201
202     return '\u03b5-greedy (\u03b5={})'.format(self.epsilon)
203
204 ⚡ def selection(self, agent):
205
206     if np.random.random() < self.epsilon:
207
208         return np.random.choice(len(agent.value_estimates))
209     else:
210
211         initiative = np.argmax(agent.value_estimates)
212
213         check = np.where(agent.value_estimates == initiative)[0]
214
215         if len(check) == 0:
216
217             return initiative
218         else:
219
220             return np.random.choice(check)

```

La stratégie banditGradient convertit les récompenses estimées du bras en probabilités puis prélève des échantillons au hasard de la distribution résultante.

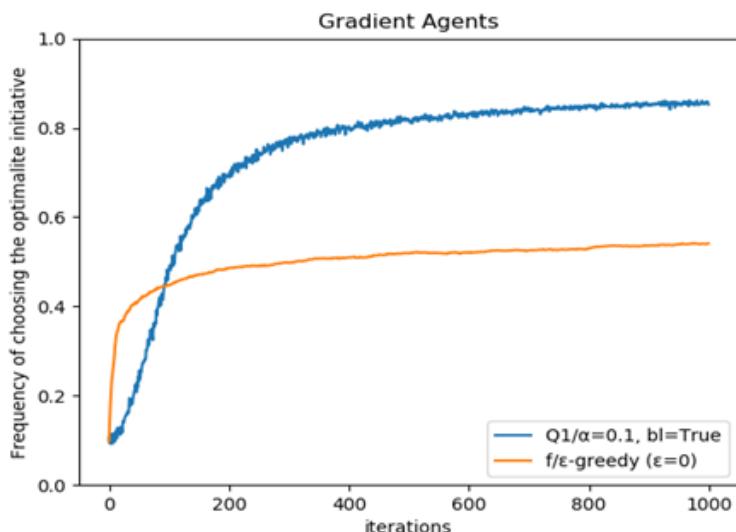
Cette stratégie est principalement utilisée par l'agent de gradient pour l'apprentissage des préférences relatives

```

226 class banditGradient(Strategie):
227
228     def __str__(self):
229         return 'SM'
230
231
232     def selection(self, agent):
233
234         gen = agent.value_estimates
235
236         yp = np.exp(gen) / np.sum(np.exp(gen))
237
238         cdf = np.cumsum(yp)
239
240         s = np.random.random()
241
242         return np.where(s < cdf)[0][0]
243
244
245
246
247 class Map(object):
248
249     def __init__(self, manchots, strategies, label='Multi-Armed Bandit'):
250
251         self.manchots = manchots
252
253         self.strategies = strategies
254
255         self.label = label
256
257     def reboot(self):
258
259         self.manchots.reboot()
260
261         for agent in self.strategies:
262
263             agent.reboot()
264
265
266
267     def run(self, trials=1000, experiments=2000):
268
269         resultats = np.zeros((trials, len(self.strategies)))
270
271         optimalite = np.zeros_like(resultats)
272
273         for _ in range(experiments):
274             self.reboot()
275             for t in range(trials):
276
277                 for i, agent in enumerate(self.strategies):
278
279                     initiative = agent.selection()
280
281                     recompense, is_optimal = self.manchots.tirage(initiative)
282                     agent.constatation(recompense)
283
284                     resultats[t, i] += int(recompense)
285
286                     if is_optimal:
287
288                         optimalite[t, i] += 1
289
290             resultats = resultats / np.float(experiments)
291
292             optimalite = optimalite / np.float(experiments)
293
294     def plots(self, resultats, optimalite):
295
296         plt.title(self.label)
297
298         plt.plot(optimalite)
299
300         plt.ylim(0, 1)
301
302         plt.ylabel('Frequency of choosing the optimalite initiative')
303
304         plt.xlabel('iterations')
305
306         plt.legend(self.strategies, loc=4)
307
308         plt.show()
309
310         manchots = B_Manchots(10, mu=0) # manchots = 10
311         num_essais, num_experimentations = 1000, 2000
312
313         strategie = banditGradient()
314         strategies = [
315             GradientAgent(manchots, strategie, alpha=0.1),
316             Agent(manchots, Glouton(0))
317         ]
318
319         map = Map(manchots, strategies, 'Gradient Agents')
320         resultats, optimalite = env.run(num_essais, num_experimentations)
321
322
323
324
325
326
327
328
329

```

On obtient ce résultat :



Conclusion L'apprentissage des préférences utilise une politique basée sur la fonction Softmax, où les estimations des actions sont converties en une distribution de probabilité à l'aide de la fonction Softmax.

Celle-ci est ensuite échantillonnée pour produire le manchot choisi.

2 Deuxième partie : Exercices du TP

2.1 Iteration de valeurs

```
1 import numpy as np
2
3
4 epsilon = 1e-3
5
6 TETA = 0.9
7
8 TOUTES LES ACTIONS POSSIBLES = ('U', 'D', 'L', 'R')
9
10
11 class Grille: # Environment
12
13     def __init__(self, largeur, hauteur, debut):
14
15         self.largeur = largeur
16
17         self.hauteur = hauteur
18
19         self.u = debut[0]
20
21         self.w = debut[1]
22
```

Les récompenses devraient être un dict de : $(u, w) : \text{rec}$ (rangée, col) : récompense
Les initiatives devraient être un dict de : $(u, w) : A$ (row, col) : liste des initiatives possibles

```
25     def set(self, recompenses, initiatives):
26
27         self.recompenses = recompenses
28
29         self.initiatives = initiatives
30
31     def definirEtat(self, s):
32
33         self.u = s[0]
34
35         self.w = s[1]
36
37     def etatActuel(self):
38
39         return (self.u, self.w)
40
41     def phaseTerminale(self, s):
42
43         return s not in self.initiatives
```

On va maintenant vérifier si le déplacement légal a lieu en premier

```
47     def deplacement(self, initiative):
48
49         if initiative in self.initiatives[(self.u, self.w)]:
50
51             if initiative == 'U':
52
53                 self.u -= 1
54
55             elif initiative == 'D':
56
57                 self.u += 1
58
59             elif initiative == 'R':
60
61                 self.w += 1
62
63             elif initiative == 'L':
64
65                 self.w -= 1
```

On va maintenant chercher à retourner une récompense

```
69     return self.recompenses.get((self.u, self.w), 0)
70
```

On va faire le contraire de ce que **U/D/L/R** devrait faire

```
73     def deplacementDeRetour(self, initiative):
74
75         if initiative == 'U':
76
77             self.u += 1
78
79         elif initiative == 'D':
80
81             self.u -= 1
82
83         elif initiative == 'R':
84
85             self.w -= 1
86
87         elif initiative == 'L':
88
89             self.w += 1
```

On doit mettre une exception si nous arrivons à un endroit où nous ne devrions pas être

```
95     assert(self.etatActuel() in self.tousLesEtats())
96
```

On va maintenant mettre en place la fin de la partie, retourne vrai si le jeu est terminé, sinon faux.

Vrai si nous sommes dans un état où aucune initiative n'est possible.

```
99     def finDePartie(self):
100
101         return (self.u, self.w) not in self.initiatives
102
```

Tous les états ont une position qui a de possibles initiatives prochaines ou bien une position qui rapporte une récompense

```
105     def tousLesEtats(self):
106
107         return set(self.initiatives.keys()) | set(self.recompenses.keys())
108
```

- On va maintenant définir une grille qui décrit la récompense pour être arrivé à chaque état et les initiatives possibles à chaque état.

La grille est définie comme suit :

F signifie qu'on ne peut pas aller là-bas

S signifie la proposition de départ

number désigne la récompense à cette état

```
. . . 1
. F . -1
S . . .
```

```

113     def grilleNormale():
114
115         o = Grille(3, 4, (2, 0))
116
117         recompenses = {(0, 3): 1, (1, 3): -1}
118         initiatives = {
119
120             (0, 0): ('D', 'R'),
121
122             (0, 1): ('L', 'R'),
123
124             (0, 2): ('L', 'D', 'R'),
125
126             (1, 0): ('U', 'D'),
127
128             (1, 2): ('U', 'D', 'R'),
129
130             (2, 0): ('U', 'R'),
131
132             (2, 1): ('L', 'R'),
133
134             (2, 2): ('L', 'R', 'U'),
135
136             (2, 3): ('L', 'U'),
137         }
138         o.set(recompenses, initiatives)
139

```

Dans ce jeu, nous voulons essayer de minimiser le nombre de déplacement, donc nous allons pénaliser chaque déplacement.

```

145     def grilleNegative(coutDuPas=-0.1):
146
147         o = grilleNormale()
148
149         o.recompenses.update({
150
151             (0, 0): coutDuPas,
152
153             (0, 1): coutDuPas,
154
155             (0, 2): coutDuPas,
156
157             (1, 0): coutDuPas,
158
159             (1, 2): coutDuPas,
160
161             (2, 0): coutDuPas,
162
163             (2, 1): coutDuPas,
164
165             (2, 2): coutDuPas,
166
167             (2, 3): coutDuPas,
168
169         })
170

```

```

180     def valeurs(Q, o):
181
182         for u in range(o.largeur):
183
184             print("-----")
185
186             for w in range(o.hauteur):
187
188                 q = Q.get((u,w), 0)
189
190                 if q >= 0:
191
192                     print("% .2f |" % q, end="")
193
194                 else:
195
196                     print("% .2f |" % q, end="")
197
198             print("")

```

(le signe négatif occupe un espace supplémentaire)

```

204     def strategie(W, o):
205
206         for u in range(o.largeur):
207
208             print("-----")
209
210             for w in range(o.hauteur):
211
212                 a = W.get((u,w), ' ')
213
214                 print(" %s |" % a, end="")
215
216             print("")

```

- Dans cette grille vous donne une récompense de -0.1 pour chaque état non-terminal

Nous voulons voir si cela encouragera à trouver un chemin plus court vers le but

```

220 ► if __name__ == '__main__':
221
222     grille = grilleNegative()

```

Afficher récompenses

```
228     print("recompenses:")
229
230     Valeurs(grille.recompenses, grille)
```

état → initiative

Nous choisirons au hasard une initiative et nous la mettrons à jour au fur et à mesure

```
238     Strategie = {}
239
240     for s in grille.initiatives.keys():
241
242         Strategie[s] = np.random.choice(TOUTESLES_ACTIONS_POSSIBLES)
```

Stratégie initiale

```
248     print("Strategie Initiale:")
249
250     strategie(Strategie, grille)
```

On va répéter jusqu'à la convergence :

$$Q[s] = \max[a] \{ \text{sum}[s', \text{rec}] \{ p(s', \text{rec}|s, a) [\text{rec} + \text{teta} * Q[s']] \} \}$$

```
278     while True:
279
280         grandChangement = 0
281
282         for s in etas:
283
284             ancien_q = Q[s]
```

$Q(s)$ n'a de valeur que s'il n'est pas à l'état terminal

```
290     if s in Strategie:
291
292         nv_q = float('-inf')
293
294         for a in TOUTESLES_ACTIONS_POSSIBLES:
295
296             grille.definirEtat(s)
297
298             rec = grille.deplacement(a)
299
300             q = rec + TETA * Q[grille.etatActuel()]
301
302             if q > nv_q:
303
304                 nv_q = q
305
306             Q[s] = nv_q
307
308             grandChangement = max(grandChangement, np.abs(ancien_q - Q[s]))
309
310         if grandChangement < epsilon:
311
312             break
```

On va maintenant chercher à trouver une Stratégie qui conduit à une fonction de valeur optimale et faire le tour de toutes les initiatives possibles pour trouver la meilleure initiative actuelle

```

318     for s in Strategie.keys():
319
320         meilleur_b = None
321
322         meilleur_valeur = float('-inf')
323
324         for a in TOUTES LES ACTIONS POSSIBLES:
325
326             grille.definirEtat(s)
327
328             rec = grille.deplacement(a)
329
330             q = rec + TETA * Q[grille.etatActuel()]
331
332             if q > meilleur_valeur:
333
334                 meilleur_valeur = q
335
336             meilleur_b = a
337
338     Strategie[s] = meilleur_b

```

Notre but ici est de vérifier que nous obtenons la même réponse qu'avec l'itération Strategie

```

344     print("valeurs:")
345
346     Valeurs(Q, grille)
347
348     print("Strategie:")
349
350     strategie(Strategie, grille)

```

FIGURE 1 – Pour TETA=0.9 on obtient les résultats suivants :

```

recompenses:
-----
-0.10|-0.10|-0.10| 1.00|
-----
-0.10| 0.00|-0.10|-1.00|
-----
-0.10|-0.10|-0.10|-0.10|
Strategie Initiale:
-----
 R |  R |  D |  I |  I |
-----
 R |      | R |  I |  I |
-----
 D |  U |  L |  L |  I |
valeurs:
-----
 0.62| 0.80| 1.00| 0.00|
-----
 0.46| 0.00| 0.80| 0.00|
-----
 0.31| 0.46| 0.62| 0.46|
Strategie:
-----
 R |  R |  R |  I |  I |
-----
 U |      | U |  I |  I |
-----
 U |  R |  U |  L |  I |

```

FIGURE 2 – Pour TETA=0.6 on obtient les résultats suivants :

```

recompenses:
-----
-0.10|-0.10|-0.10| 1.00|
-----|-----|-----|-----|
-----|-----|-----|-----|
-----|-----|-----|-----|
Strategie Initiale:
-----
R | R | D | |
-----|-----|-----|-----|
R | | R | |
-----|-----|-----|-----|
D | U | L | L |
-----|-----|-----|-----|
valeurs:
-----
0.62| 0.80| 1.00| 0.00|
-----|-----|-----|-----|
0.46| 0.00| 0.80| 0.00|
-----|-----|-----|-----|
0.31| 0.46| 0.62| 0.46|
-----|-----|-----|-----|
Strategie:
-----
R | R | R | |
-----|-----|-----|-----|
U | | U | |
-----|-----|-----|-----|
U | R | U | L |
-----|-----|-----|-----|

```

FIGURE 3 – Pour TETA=0.3 on obtient les résultats suivants :

2.2 Itération des politiques

On commence par l'environnement :

```

1 import numpy as np
2
3
4 class Grille: # Environnement
5
6     def __init__(self, largeur, hauteur, debut):
7
8         self.largeur = largeur
9
10        self.hauteur = hauteur
11
12        self.u = debut[0]
13
14        self.w = debut[1]

```

Les récompenses devraient être un dict de : $(u, w) : \text{rec}$ (rangée, col) : récompense
les initiatives devraient être un dict de : $(u, w) : A$ (row, col) : liste des initiatives possibles

```

18     def set(self, recompenses, initiatives):
19
20         self.recompenses = recompenses
21
22         self.initiatives = initiatives
23
24
25     def definirLetat(self, s):
26
27         self.u = s[0]
28
29         self.w = s[1]
30
31
32     def etatActuel(self):
33
34         return (self.u, self.w)
35
36
37     def phaseTerminale(self, s):
38
39         return s not in self.initiatives

```

On va vérifier si le déplacement légal a lieu en premier

```

42     def deplacement(self, initiative):
43
44
45         if initiative in self.initiatives[(self.u, self.w)]:
46
47             if initiative == 'U':
48
49                 self.u -= 1
50
51             elif initiative == 'D':
52
53                 self.u += 1
54
55             elif initiative == 'R':
56
57                 self.w += 1
58
59             elif initiative == 'L':
60
61                 self.w -= 1

```

On va retourner une récompense

```

65     return self.recompenses.get((self.u, self.w), 0)
66

```

On va faire le contraire de ce que U/D/L/R

```

69     def deplacementDeRetour(self, initiative):
70
71
72         if initiative == 'U':
73
74             self.u += 1
75
76         elif initiative == 'D':
77
78             self.u -= 1
79
80         elif initiative == 'R':
81
82             self.w -= 1
83
84         elif initiative == 'L':
85
86             self.w += 1

```

On va faire une exception si nous arrivons à un endroit où nous ne devrions pas être

```

92     assert(self.etatActuel() in self.tousLesEtats())
93

```

Retourne vrai si le jeu est terminé, sinon faux
vrai si nous sommes dans un état où aucune initiative n'est possible

```

96     def finDePartie(self):
97
98         return (self.u, self.w) not in self.initiatives
99

```

Tous les états ont une position qui a de possibles initiatives prochaines, ou bien une position qui rapporte une récompense

```

102    def tousLesEtats(self):
103
104        return set(self.initiatives.keys()) | set(self.recompenses.keys())
105

```

Définir une grille qui décrit la récompense pour être arrivé à chaque état et les initiatives possibles à chaque état. La grille est définie comme suit :

F signifie qu'on ne peut pas aller là-bas

s signifie la proposition de départ

number désigne la récompense à cette état

. . . 1 . F . -1 S . . .

```
110 def grilleNormale():
111
112     o = Grille(3, 4, (2, 0))
113     recompenses = {(0, 3): 1, (1, 3): -1}
114     initiatives = {
115
116         (0, 0): ('D', 'R'),
117
118         (0, 1): ('L', 'R'),
119
120         (0, 2): ('L', 'D', 'R'),
121
122         (1, 0): ('U', 'D'),
123
124         (1, 2): ('U', 'D', 'R'),
125
126         (2, 0): ('U', 'R'),
127
128         (2, 1): ('L', 'R'),
129
130         (2, 2): ('L', 'R', 'U'),
131
132         (2, 3): ('L', 'U'),
133     }
134     o.set(recompenses, initiatives)
135
136     return o
```

Dans ce jeu, nous voulons essayer de minimiser le nombre de déplacement, donc nous allons pénaliser chaque déplacement

```
142 def grilleNegative(coutDuPas=-0.5):
143
144     o = grilleNormale()
145
146     o.recompenses.update({
147
148         (0, 0): coutDuPas,
149
150         (0, 1): coutDuPas,
151
152         (0, 2): coutDuPas,
153
154         (1, 0): coutDuPas,
155
156         (1, 2): coutDuPas,
157
158         (2, 0): coutDuPas,
159
160         (2, 1): coutDuPas,
161
162         (2, 2): coutDuPas,
163
164         (2, 3): coutDuPas,
165     })
166
167     return o
```

Seuil de convergence

```

173     epsilon = 1e-3
174
175     def Valeurs(Q, o):
176
177         for u in range(o.largeur):
178
179             print("-----")
180
181             for w in range(o.hauteur):
182
183                 q = Q.get((u,w), 0)
184
185                 if q >= 0:
186
187                     print(" %.2f| " % q, end="")
188
189                 else:
190
191                     print("%.2f| " % q, end="")
192
193             print("")

```

```

199     def strategie(W, o):
200
201         for u in range(o.largeur):
202
203             print("-----")
204
205             for w in range(o.hauteur):
206
207                 a = W.get((u,w), ' ')
208
209                 print(" %s | " % a, end="")
210
211             print("")

```

Tous les $p(s', rec|s, a) = 1$ ou 0
 cette grille nous donne une récompense de -0.1 pour chaque état non-terminal, nous voulons voir si cela encouragera à trouver un chemin plus court vers le but

```

224
225     if __name__ == '__main__':
226
227         grille = grilleNegative()
228
229         # afficher les récompenses
230

```

état → initiative
 Nous choisirons une initiative au hasard et la mettrons à jour au fur et à mesure que nous apprendrons

```

233     Valeurs(grille.recompenses, grille)
234
235     Strategie = {}
236
237     for s in grille.initiatives.keys():
238         Strategie[s] = np.random.choice(TOUTESLES_ACTIONS_POSSIBLES)
239
240     print("Strategie initiale:")
241
242     strategie(Strategie, grille)
243

```

On initialise le $Q(s)$

```

242     strategie(Strategie, grille)
243
244     Q = {}
245
246     etats = grille.tousLesEtats()
247
248     for s in etats:
249
250         # Q[s] = 0
251
252         if s in grille.initiatives:
253
254             Q[s] = np.random.random()
255
256         else:
257
258             Q[s] = 0

```

On répète jusqu'à la convergence

$$Q[s] \leftarrow \max[a][s', rec]'p(s', rec's, a)[rec'teta'Q[s']]$$

```

262     while True:
263         grandChangement = 0
264
265         for s in etats:
266
267             ancien_q = Q[s]
268
269             if s in Strategie:
270
271                 nv_q = float('-inf')
272
273                 for a in TOUTES LES ACTIONS POSSIBLES:
274
275                     grille.definirEtat(s)
276
277                     rec = grille.deplacement(a)
278
279                     q = rec + TETA * Q[grille.etatActuel()]
280
281                     if q > nv_q:
282                         nv_q = q
283
284             Q[s] = nv_q
285
286             grandChangement = max(grandChangement, np.abs(ancien_q - Q[s]))
287
288             if grandChangement < epsilon:
289                 break

```

On cherche à trouver une politique qui mène à une fonction de valeur optimale

```

291     for s in Strategie.keys():
292
293         meilleur_b = None
294
295         meilleure_valeur = float('-inf')
296
297         # boucler toutes les initiatives possibles pour trouver la meilleure initiative en cours
298
299         for a in TOUTES LES ACTIONS POSSIBLES:
300
301             grille.definirEtat(s)
302
303             rec = grille.deplacement(a)
304
305             q = rec + TETA * Q[grille.etatActuel()]
306
307             if q > meilleure_valeur:
308                 meilleure_valeur = q
309
310             meilleur_b = a
311
312     Strategie[s] = meilleur_b

```

Notre but ici est de vérifier que nous obtenons la même réponse qu'avec l'itération de la politique

```

316     print("Valeurs:")
317
318     Valeurs(Q, grille)
319
320     print("Stratégie:")
321
322     strategie(Strategie, grille)

```

```

recompenses:
-----
-0.10|-0.10|-0.10| 1.00|
-----  

-0.10| 0.00|-0.10|-1.00|
-----  

-0.10|-0.10|-0.10|-0.10|
Strategie initiale:  

-----  

U | U | U | |  

-----  

D | | U | |  

-----  

L | L | L | U |  

valeurs:  

-----
0.62| 0.80| 1.00| 0.00|
-----  

0.46| 0.00| 0.80| 0.00|
-----  

0.31| 0.46| 0.62| 0.46|
Strategie:  

-----  

R | R | R | |  

-----  

U | | U | |  

-----  

U | R | U | L |

```

```

recompenses:
-----
-0.10|-0.10|-0.10| 1.00|
-----  

-0.10| 0.00|-0.10|-1.00|
-----  

-0.10|-0.10|-0.10|-0.10|
Strategie initiale:  

-----  

U | U | L | |  

-----  

D | | R | |  

-----  

D | L | L | D |  

valeurs:  

-----
0.20| 0.50| 1.00| 0.00|
-----  

0.02| 0.00| 0.50| 0.00|
-----  

-0.09| 0.02| 0.20| 0.02|
Strategie:  

-----  

R | R | R | |  

-----  

U | | U | |  

-----  

U | R | U | L |

```

FIGURE 4 – Pour TETA=0.9 on obtient les résultats suivants

```

recompenses:
-----
-0.10|-0.10|-0.10| 1.00|
-----  

-0.10| 0.00|-0.10|-1.00|
-----  

-0.10|-0.10|-0.10|-0.10|
Strategie initiale:  

-----  

L | R | R | |  

-----  

L | | D | |  

-----  

R | U | D | L |
valeurs:  

-----
-0.04| 0.20| 1.00| 0.00|
-----  

-0.11| 0.00| 0.20| 0.00|
-----  

-0.13|-0.11|-0.04|-0.11|
Strategie:  

-----  

R | R | R | |  

-----  

U | | U | |  

-----  

U | R | U | L |

```

FIGURE 5 – Pour TETA=0.6 on obtient les résultats suivants

```

import numpy as np
import matplotlib.pyplot as plt

TETA = 0.9
ALPHA = 0.1
TOTES_LES_ACTIONS_POSSIBLES = ('U', 'D', 'L', 'R')
SMALL_ENOUGH = 1e-7
TOTES_LES_ACTIONS_POSSIBLES = ('U', 'D', 'L', 'R')

```

FIGURE 6 – Pour TETA=0.3 on obtient les résultats suivants

```

33     class Grille:
34
35         def __init__(self, largeur, hauteur, debut):
36
37             self.largeur = largeur
38
39             self.hauteur = hauteur
40
41             self.i = debut[0]
42
43             self.j = debut[1]

```

Mettre en place l'environnement :

```

36     def set(self, recompenses, initiatives):
37
38         self.recompenses = recompenses
39
40         self.initiatives = initiatives
41
42
43     def definirEtat(self, s):
44
45         self.i = s[0]
46
47         self.j = s[1]
48
49     def etatActuel(self):
50
51         return (self.i, self.j)
52
53     def phaseTerminale(self, s):
54
55         return s not in self.initiatives

```

```

59     def deplacement(self, initiative):
60
61         if initiative in self.initiatives[(self.i, self.j)]:
62
63             if initiative == 'U':
64
65                 self.i -= 1
66
67             elif initiative == 'D':
68
69                 self.i += 1
70
71             elif initiative == 'R':
72
73                 self.j += 1
74
75             elif initiative == 'L':
76
77                 self.j -= 1

```

On va vérifier d'abord si le placement légal

```

80
81     return self.recompenses.get((self.i, self.j), 0)

```

Retourner une récompense si le cas est échéant

```

85     def deplacementDeRetour(self, initiative):
86
87         if initiative == 'U':
88
89             self.i += 1
90
91         elif initiative == 'D':
92
93             self.i -= 1
94
95         elif initiative == 'R':
96
97             self.j -= 1
98
99         elif initiative == 'L':
100
101             self.j += 1
102
103         assert(self.etatActuel() in self.tousLesEtats())

```

On va mettre en place les contraires de ce que **U/D/L/R**

```
106
107     def finDePartie(self):
108         ...
109
110     return (self.i, self.j) not in self.initiatives
```

Revient vrai si le jeu est terminé, sinon faux Vrai si nous sommes dans un état où aucune initiative n'est possible

```
114     def tousLesEtats(self):
115         ...
116
117     return set(self.initiatives.keys()) | set(self.recompenses.keys())
```

Peut-être buggy, mais moyen simple pour obtenir tous les états Soit une position qui a possible prochaines initiatives ou une position qui donne une récompense

```
123     def grilleNormale():
124
125         g = Grille(3, 4, (2, 0))
126         recompenses = {(0, 3): 1, (1, 3): -1}
127         initiatives = {
128
129             (0, 0): ('D', 'R'),
130
131             (0, 1): ('L', 'R'),
132
133             (0, 2): ('L', 'D', 'R'),
134
135             (1, 0): ('U', 'D'),
136
137             (1, 2): ('U', 'D', 'R'),
138
139             (2, 0): ('U', 'R'),
140
141             (2, 1): ('L', 'R'),
142
143             (2, 2): ('L', 'R', 'U'),
144
145             (2, 3): ('L', 'U'),
146
147         }
148
149         g.set(recompenses, initiatives)
150
151     return g
```

- On va maintenant définir une calandre qui décrit la récompense pour arriver à chaque état et les initiatives possibles dans chaque État

```
155     def grilleNegative(coutDuPas=-0.1):
156
157         g = grilleNormale()
158
159         g.recompenses.update({
160
161             (0, 0): coutDuPas,
162
163             (0, 1): coutDuPas,
164
165             (0, 2): coutDuPas,
166
167             (1, 0): coutDuPas,
168
169             (1, 2): coutDuPas,
170
171             (2, 0): coutDuPas,
172
173             (2, 1): coutDuPas,
174
175             (2, 2): coutDuPas,
176
177             (2, 3): coutDuPas,
178
179         })
180
181     return g
```

Dans ce jeu, nous voulons essayer de minimiser le nombre de coups Nous pénaliserons donc chaque déplacement

```

191     def Valeurs(q, g):
192         for i in range(g.largeur):
193             print("-----")
194             for j in range(g.hauteur):
195                 q = Q.get((i,j), 0)
196                 if q >= 0:
197                     print(" %.2f" % q, end="")
198                 else:
199                     print(" %.2f" % q, end="")
200             print("")
201
215     def strategie(W, g):
216         for i in range(g.largeur):
217             print("-----")
218             for j in range(g.hauteur):
219                 a = W.get((i,j), ' ')
220                 print(" %s | %s, end="")
221             print("")
222
234     def sup_bloc(d):
235         max_key = None
236         max_val = float('-inf')
237         for k, q in d.items():
238             if q > max_val:
239                 max_val = q
240                 max_key = k
241         return max_key, max_val

```

On va envoyer l'argmax (clé) et le max (valeur) d'un dictionnaire puis on va mettre cela dans une fonction puisque nous l'utilisons si souvent

```

256     def initiativeAlea(a, eps=0.1):
257
258         p = np.random.random()
259
260         if p < (1 - eps):
261             return a
262
263         else:
264             return np.random.choice(TOUTES LES ACTIONS POSSIBLES)

```

Nous allons utiliser epsilon-soft pour s'assurer que tous les états sont visités

```

313     grille = grilleNegative(coutDuPas=-0.1)
314
315     print("recompenses:")

```

Si nous utilisons la grille standard, il ya une bonne chance que nous finirons avec -Politiques sous-optimales

- E.g
- R — R — R — —
- R* — — U — —
-
- U — R — U — L —

Au lieu de cela, pénalisons chaque mouvement afin que l'agent trouvera un itinéraire plus court.
grille = grilleNormale()

```

317     Valeurs(grille.recompenses, grille)
318
319     S = {}
320
321     etats = grille.tousLesEtats()
322
323     for s in etats:
324
325         S[s] = {}
326
327         for a in TOUTES LES ACTIONS POSSIBLES:
328             S[s][a] = 0
329
330         # let's also keep track of how many times S[s] has been updated
331
332         comptes_actualises = {}
333
334         update_counts_sa = {}
335
336         for s in etats:
337
338             update_counts_sa[s] = {}
339
340             for a in TOUTES LES ACTIONS POSSIBLES:
341                 update_counts_sa[s][a] = 1.0

```

On va faire une Initialisation de la stratégie, nous tirerons notre stratégie de la plus récente
Initialiser $S(s, a)$

```

345     t = 1.0
346
347     deltas = []
348
349     for it in range(10000):
350
351         if it % 100 == 0:
352             t += 1e-2
353
354         if it % 2000 == 0:
355             print("it:", it)
356

```

On va répéter jusqu'à la convergence

```

361     s = (2, 0)
362
363     grille.definirEtat(s)
364
365     a, _ = sup_bloc(S[s])
366
367     Q = 0
368
369     while not grille.finDePartie():
370         a = initiativeAlea(a, eps=0.5 / t) # epsilon-greedy
371
372         rec = grille.deplacement(a)
373
374         s2 = grille.etatActuel()
375
376         alpha = ALPHA / update_counts_sa[s][a]
377
378         update_counts_sa[s][a] += 0.005
379
380         old_qsa = S[s][a]
381
382         a2, max_q_s2a2 = sup_bloc(S[s2])
383
384         S[s][a] = S[s][a] + alpha * (rec + TETA * max_q_s2a2 - S[s][a])
385
386         Q = max(Q, np.abs(old_qsa - S[s][a]))
387

```

Au lieu de « générer» une épisode, nous allons jouer

- un épisode dans cette boucle
 - puisque nous n'obtenons pas de récompense pour avoir simplement commencé le jeu
 - la valeur de l'état terminal est par définition 0, donc nous n'avons pas
 - le dernier (s, rec) tuple est l'état terminal et la récompense finale
- Initiative aléatoire fonctionne également, mais plus lent puisque nous pouvons heurter les murs
a différence entre le SARSA et le S-Learning est avec S-Learning

```

    comptes_actualises[s] = comptes_actualises.get(s, 0) + 1
    414
    415     print("comptes_actualises:")
    416
    417     total = np.sum(list(comptes_actualises.values()))
    418
    419     for k, q in comptes_actualises.items():
    420         comptes_actualises[k] = float(q) / total
    421
    422     Valeurs(comptes_actualises, grille)
    423
    424     print("valeurs:")
    425
    426     Valeurs(Q, grille)
    427
    428     print("Strategie:")
    429     strategie(Strategie, grille)
    430

```

On veut connaitre quelle est la proportion de temps que nous passons à mettre à jour
chaque partie de S

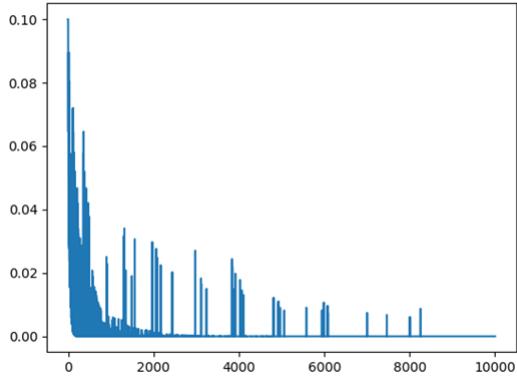


FIGURE 7 – Pour TETA=0.9 ,ALPHA=0.1 on obtient les résultats suivants :

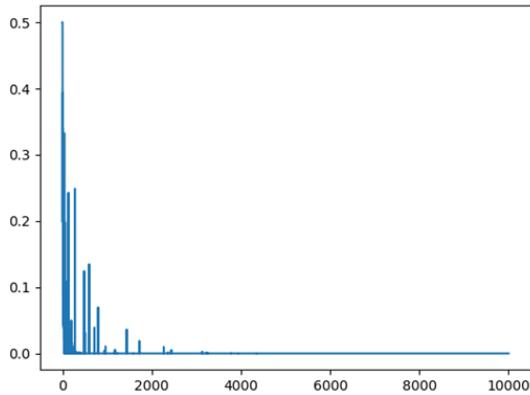


FIGURE 8 – Pour TETA=0.9 ,ALPHA=0.5 on obtient les résultats suivants :

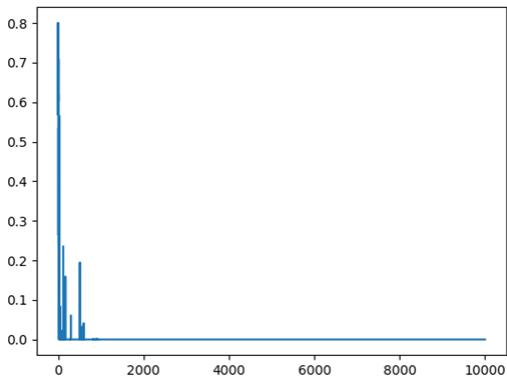


FIGURE 9 – Pour TETA=0.9 ,ALPHA=0.8 on obtient les résultats suivants :

Conclusion :

Comme nous pouvons le constater, la plupart des valeurs de la politique aléatoire uniforme sont négatives. C'est parce que si vous bougez au hasard, il y a de fortes chances que vous vous retrouviez dans l'état de perdant.

il y a deux façons d'entrer dans l'état de perte, Mais il n'y a qu'un seul moyen d'entrer dans l'état de but. La valeur la plus négative est celle de l'État situé juste en dessous de l'État perdant, ce qui est logique. Passons maintenant à la police fixe et au facteur d'actualisation .9. Nous voyons ce que nous nous attendons à voir. Plus on s'éloigne de l'état terminal, plus la valeur diminue. Et à chaque fois, il diminue d'exactement 10 Donc 0,81, c'est 0,9 fois 0,9, .73 est 0,81 fois .9 et ainsi de suite.