Assignment 3 : Multi-Armed Bandits

INFO-F-409 - Learning Dynamics

Florentin Hennecker (ULB 000382078)

1 N-Armed Bandit

1.1 Exercise 1

Let us compare the performance of several algorithms trying to maximise the reward they receive from an N-armed bandit which has 4 arms, each of which producing a reward sampled from a normal distribution of which the parameters are shown in table 2.

Arm	μ	σ
1	2.3	0.9
2	2.1	0.6
3	1.5	0.4
4	1.3	2

Table 1 – Parameters of the 4-armed bandit

The algorithms to compare are ϵ-greedy with parameters 0, 0.1 and 0.2, Softmax with temperatures 1 and 0.1, and the random action selection policy. Figure 1 shows how each of these algorithms manage to maximise their reward after playing for many iterations.

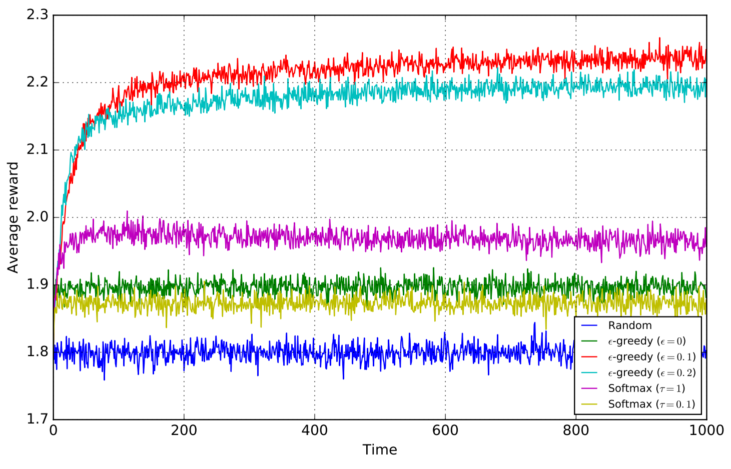
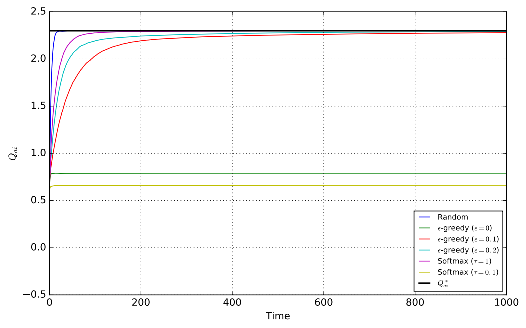


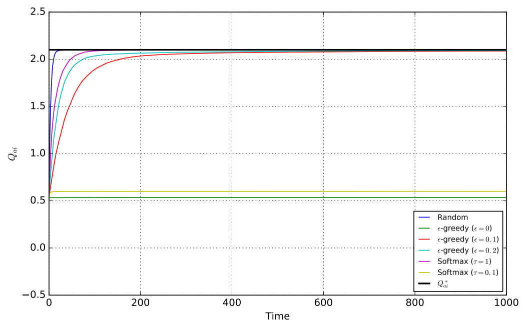
Figure 1 – Evolution of the average reward over 10000 runs during training

As we can see, only two algorithms managed to converge to the best arm : ϵ-greedy 0.1 and 0.2. Let us first explain the behaviour of ϵ-greedy 0. It selects its first action at random, but for all the following iterations, it will choose it again as its Q-value will be non-zero and there is no room to explore. If all the arms had negative Q^*_{ai}, ϵ-greedy would however pick the less negative one after it has explored all of the actions.

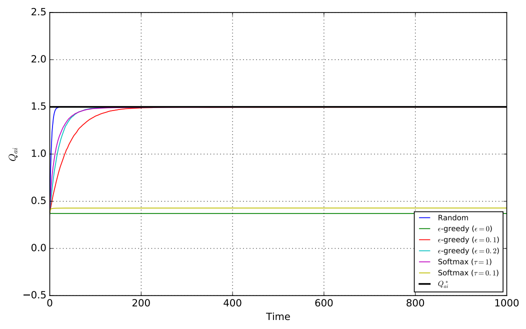
Softmax outperforms the random selection policy but does not do very well. This is likely caused by the fact that there is a lot of noise in the rewards and that the discrete distribution regulating the choice of action is fairly homogeneous. This hypothesis can be confirmed by looking at figures 2 and 3.



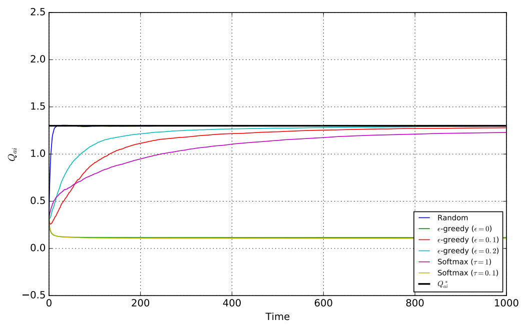
(a) Arm 1



(b) Arm 2



(c) Arm 3



(d) Arm 4

FIGURE 2 – Evolution of Q_{ai} for each arm

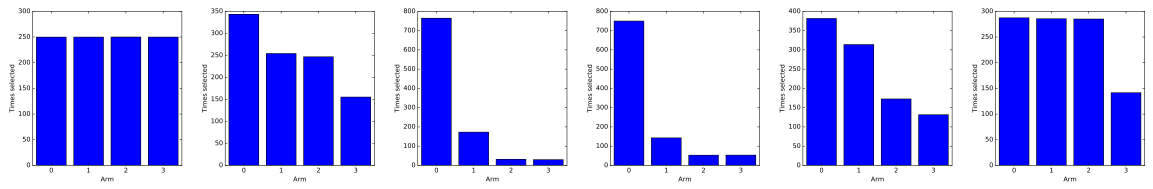
The first thing to note on figure 2 is that the random selection policy estimates $Q^*_{ai}*$ quite precisely very fast for each arm, which is normal. One could say that random exploration is very good, but it only is at the very start and for small problems with small action-state spaces.

Another interesting trend is that the best arm gets learnt much faster than the worst one. This is due to the fact that the learning algorithms should pick it more often, therefore allowing us to understand quicker its reward distribution.

As for the speed at which each algorithm converges, it makes sense to see that the higher ϵ is, the quicker it will converge because it follows a more random behaviour. Softmax 1 has an interesting behaviour. It learns very quickly about the quality of the good arms but much slower about the bad ones.

Softmax 0.1 seems to have trouble learning this problem. In fact, it doesn't have enough exploration power and will behave just like ϵ-greedy 0. Both will pick an action at the start and will stick with it until then end. The graphs of figure 2 do not convey the idea that both algorithms will actually learn the value of one arm very quickly, but leave the others at 0 ; this is due to the fact that each graph shows an average of many runs.

We have the same issue on figure 3 : it looks like all actions are being taken roughly the same amount of time during a run, but both algorithms actually only pick one (or almost one for Softmax 0.1) action for the whole run and what we see is the fact that the action picked is picked at random at the start.



(a) Random (b) ϵ-greedy 0 (c) ϵ-greedy 0.1 (d) ϵ-greedy 0.2 (e) Softmax 1 (f) Softmax 0.1

FIGURE 3 – Actions taken by each algorithm

Concerning ϵ-greedy 0.1 and 0.2, we clearly see that both algorithms have learnt that the first arm is the best one. Also, with a higher ϵ, the non-optimal actions get chosen a bit more. Softmax 1 seems to have learnt that the best arm is the first one too but has a too high temperature to always choose the best arm.

1.2 Exercise 2

If we do the exact same exercise but double the standard deviation of the reward for each arm, we see on figure 4 that it is a harder problem to learn.

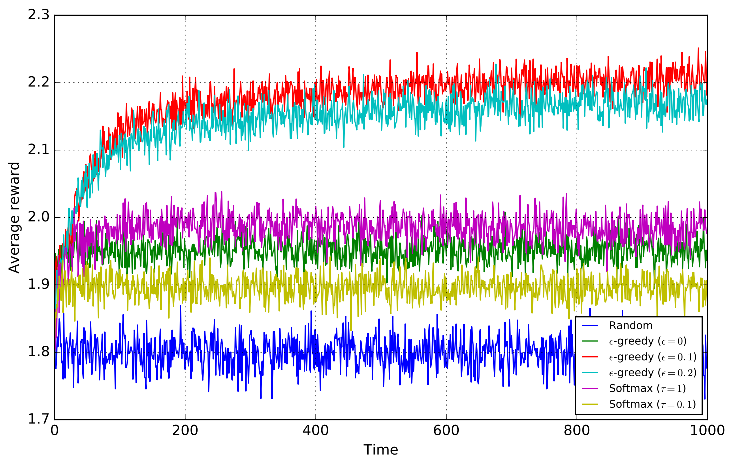
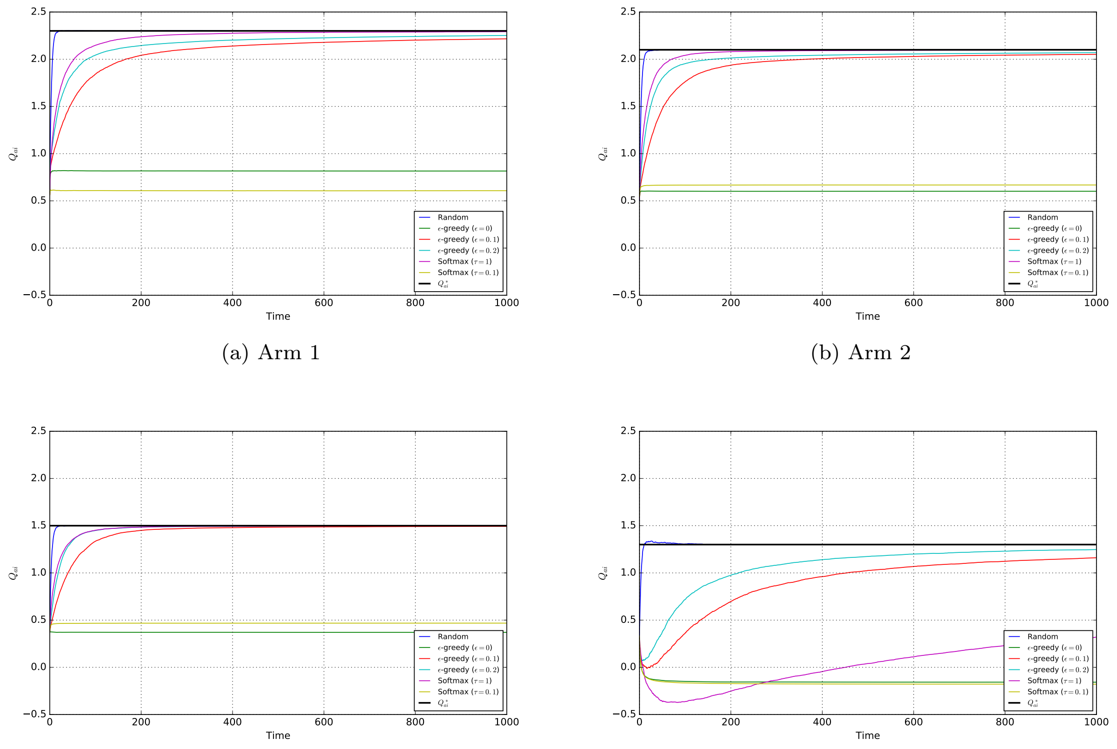


FIGURE 4 – Evolution of the average reward over 10000 runs during training

When we compare this graph to the one shown in figure 1, we notice that the variance of each learning curve is higher, but most importantly, we see that the two algorithms that manage to train correctly learn slower. The average reward climbs above 2.2 before 200 iterations in the first case, and it takes a lot more time in the second case. However, after 1000 iterations, the difference is small ; and it is for all the algorithms.

The fact that this problem is harder to learn is visible on figure 5 too : all algorithms take more time to converge to the true values. The last (and worst) arm is notably different. All algorithms (except the random one) underestimate fairly wildly its value.

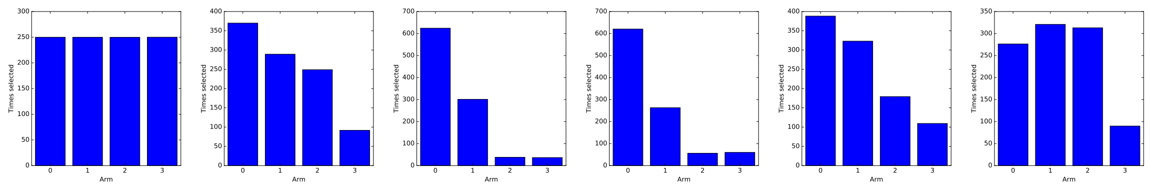
The graphs of figure 6, although they are a bit more homogeneous for ϵ-greedy 0.1 and 0.2, are barely different than the ones of figure 3.



(a) Arm 1 (b) Arm 2

(c) Arm 3 (d) Arm 4

FIGURE 5 – Evolution of Q_{ai} for each arm



(a) Random (b) ϵ-greedy 0 (c) ϵ-greedy 0.1 (d) ϵ-greedy 0.2 (e) Softmax 1 (f) Softmax 0.1

FIGURE 6 – Actions taken by each algorithm

1.3 Exercise 3

Let us now compare ϵ-greedy and Softmax with varying time parameters on the problem posed in the first exercise. We will implement and compare :
— ϵ-greedy with $\epsilon = 1/\sqrt{t}$
— Softmax with $\tau = 4 * \frac{1000-t}{1000}$

ϵ-greedy performs better than its fixed ϵ variants on every aspect : it converges quicker, and reaches a better average reward at every iteration. However, on figure 8a, we see that even if it gets close to the actual Q^*_{ai} faster than most of the other algorithms, it levels off before the actual value and then converges very slowly even though this action gets selected very often (see figure 9g).

Softmax is a very interesting one. During the first few hundred iterations, it is left exploring all actions. But once the temperature goes down, its average reward climbs as it starts choosing the best arm more often and it manages to shoot past all other algorithms at the end, being the only one reaching the mean value of the best arm.

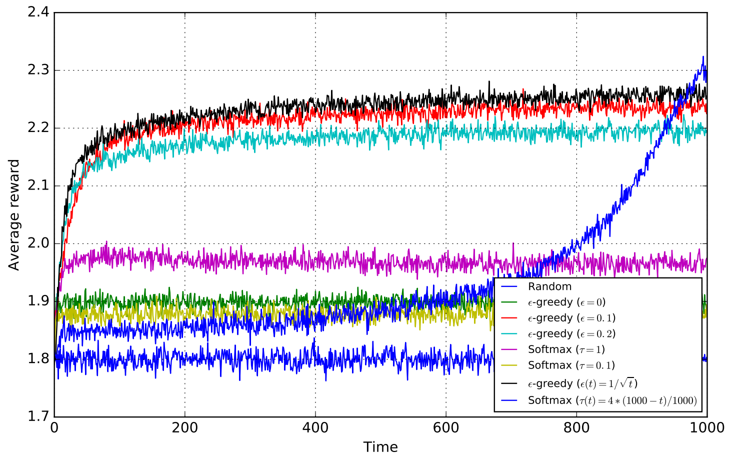
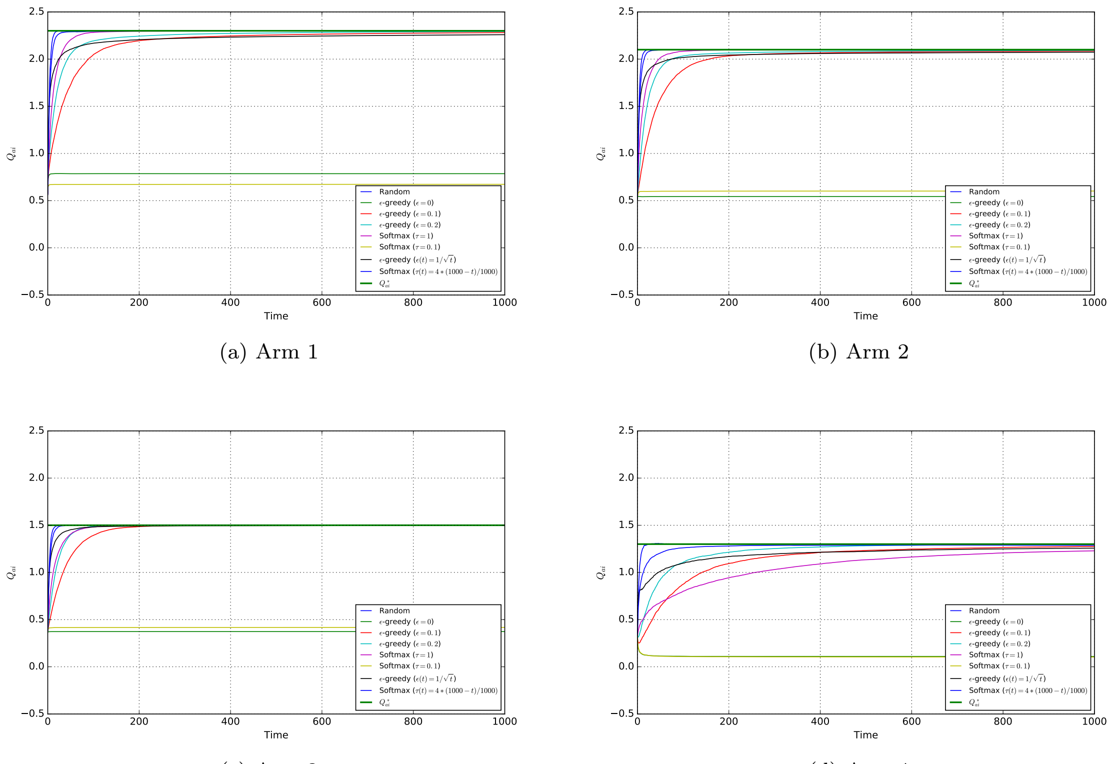


FIGURE 7 – Evolution of the average reward over 10000 runs during training



(a) Arm 1

(b) Arm 2

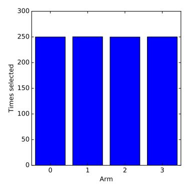
(c) Arm 3

(d) Arm 4

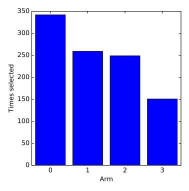
FIGURE 8 – Evolution of Q_{ai} for each arm

Once again, the graphs in figure 9 do not show the full picture. The graph for Softmax with a varying τ doesn't show that once it has learned a good estimation of all Q^*_{ai} and the temperature is decreased, it will always choose the optimal arm.

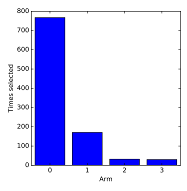
Looking at figure 8, it is clear that Softmax with a varying τ is the quickest (except for the random policy) to learn about the Q values. We could decrease the temperature much earlier to reach optimal rewards.



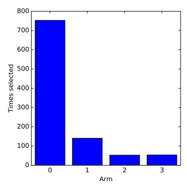
(a) Random



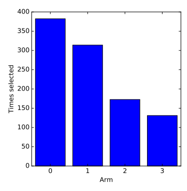
(b) ϵ-greedy 0



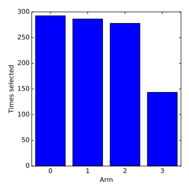
(c) ϵ-greedy 0.1



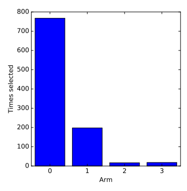
(d) ϵ-greedy 0.2



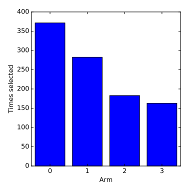
(e) Softmax 1



(f) Softmax 0.1



(g) $\epsilon(t)$



(h) $\tau(t)$

FIGURE 9 – Actions taken by each algorithm

2 Stochastic Reward Game

Two agents play a game where both try to maximise the reward they get from joint actions. The game is the stochastic climbing game :

	a_1	a_2	a_3
b_1	$\mathcal{N}(11, \sigma_0^2)$	$\mathcal{N}(-30, \sigma^2)$	$\mathcal{N}(0, \sigma^2)$
b_2	$\mathcal{N}(-30, \sigma^2)$	$\mathcal{N}(7, \sigma_1^2)$	$\mathcal{N}(6, \sigma^2)$
b_3	$\mathcal{N}(0, \sigma^2)$	$\mathcal{N}(0, \sigma^2)$	$\mathcal{N}(5, \sigma^2)$

TABLE 2 – The stochastic climbing game

2.1 Plotting

We use the standard method described in [1] which consists of doing Boltzmann exploration using expected values considering the probabilities that the other player will play each action.

The heuristic chosen is optimistic Boltzmann. All graphs are smoothed with an exponential average :

$$\widehat{x}_{t+1} = (1-\alpha)\widehat{x}_t + \alpha * x$$

with $\alpha = 0.02$. The temperature chosen for both algorithms is $\tau = 1$

As we can see on figure 10, the first type converges to a reward of 7. This is the expected equilibrium as shown in figures 3, 4 and 5 in [1]. However, we also see that the heuristic manages to overcome the 'fear' of getting a large negative reward by being optimistic and converges to 11.

When the highest reward is made more uncertain (figure 11), the optimistic reward seems to converge to about 7 as well. However, if we make the reward for $<a_2, b_2>$ more uncertain (figure 12), both algorithms will end up choosing this action and will receive a reward with very high variance (as can be seen from the much less stable plot)

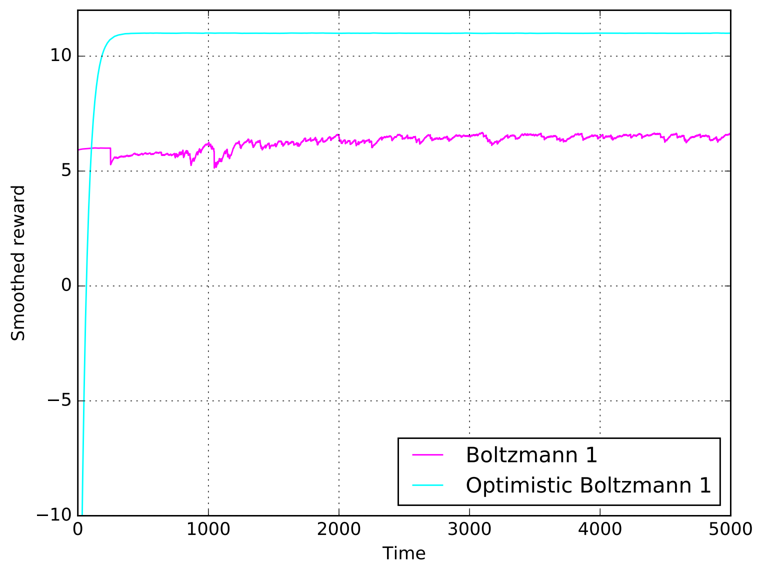


FIGURE 10 – Evolution of the average reward during training with $\sigma_0 = \sigma_1 = \sigma = 0.2$

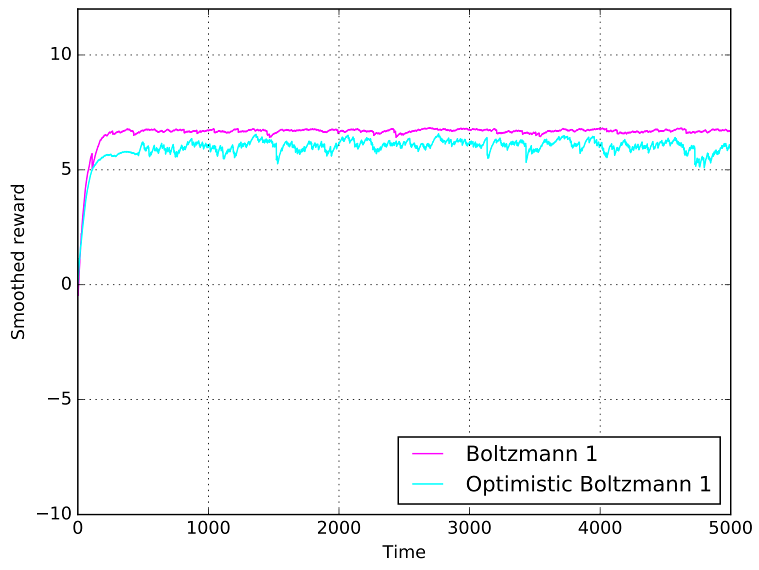


FIGURE 11 – Evolution of the average reward during training with $\sigma_0 = 4$ and $\sigma_1 = \sigma = 0.1$

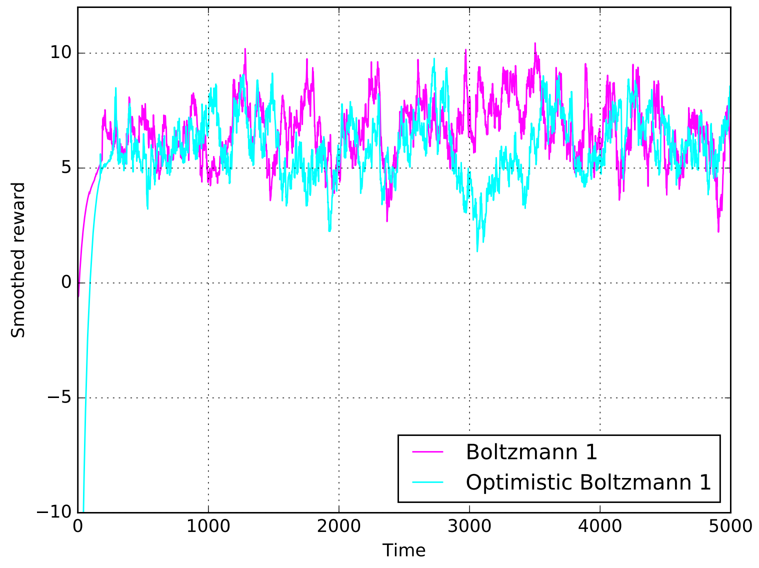


FIGURE 12 – Evolution of the average reward during training with $\sigma_1 = 4$ and $\sigma_0 = \sigma = 0.1$

2.2 Discussion

How will the learning process change if we make the agents independent learners ? In this case, both agents only learn about their actions (so they will only have three Q values instead of nine). As explained in [1], but in the case of a simpler game, cooperation still emerges even though both agents have less information. The only difference is that the joint action learners converge slightly quicker to cooperation.

How will the learning process change if we make the agents always select the action that according to them will yield the highest reward (assuming the other agent plays the best response ?) This would be equivalent to removing entirely the exploration steps. As soon as the reward is positive, the agent will keep playing that action. let us assume that a is the greedy player and b is playing the best response. If a chooses action 0, and b chooses action 0, or$a$ chooses action 1 and b chooses action 1 ; or if a chooses action 2 and b chooses action 1 or 2 ; a will receive a positive reward every time and so will keep playing that action since all the other actions are valued at 0.

Références

[1] C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, (s 746) :752, 1998.

A N-Armed Bandit code

```
import numpy as np
import matplotlib.pyplot as plt
import sys

def reward(bandits, choice):
    return bandits[choice, 1] * np.random.randn() + bandits[choice, 0]

def q_values(reward_sum, times_played):
    results = np.copy(reward_sum)
    results[times_played != 0] /= times_played[times_played != 0]
    return results

def random_selection(q_values, args={}):
    return np.random.randint(0, len(q_values))

def e_greedy(q_values, args={'epsilon':0.1, 't':None}):
    epsilon = args['epsilon']
    try: epsilon = 1/np.sqrt(args['t'])
    except: pass

    if np.random.rand() < epsilon:
        return np.random.randint(0, len(q_values))
    return np.argmax(q_values)

def softmax(q_values, args={'tau':0.1, 't':None}):
    tau = args['tau']
    try: tau = 4. * 1.*(1000.-args['t']) / 1000.
    except: pass

    distribution = np.exp(q_values/tau) / np.sum(np.exp(q_values/tau))
    return np.random.choice(np.arange(len(q_values)), p=distribution)

def get_avg_reward(bandits, time_steps, iterations, methods):
    results = np.zeros((time_steps, len(methods)))
    Qai = np.zeros((time_steps, len(methods), len(bandits)))
    times_selected = np.zeros((len(methods), len(bandits)))
    for m, (func, args) in enumerate(methods):

        for i in range(iterations):

            reward_sum = np.zeros(len(bandits))
            times_played = np.zeros(len(bandits))

            a = random_selection(q_values(reward_sum, times_played))
            for t in range(time_steps):
                if 't' in args:
                    args = {'epsilon':0, 'tau':0, 't':t}
                r = reward(bandits, a)
                times_played[a] += 1
                reward_sum[a] += r
                Q = q_values(reward_sum, times_played)
                Qai[t, m] += Q

                # getting next action with generic action selection method
                a = func(Q, args=args)
                results[t, m] += r

            times_selected[m] += times_played

            print '\r' + func.__name__, i,
            sys.stdout.flush()

        times_selected[m,:] /= iterations
        results[:, m] /= iterations
        Qai[:, m, :] /= iterations
        print '\r' + func.__name__, 'done'
    return results, Qai, times_selected

def run(exercise):
    bandits = np.array([
        [2.3, 0.9],
        [2.1, 0.6],
        [1.5, 0.4],
        [1.3, 2.0]
    ])
    if exercise == 2 : bandits[:,1] *= 2
    print bandits

    algos = [
        (random_selection, {}),
        (e_greedy, {'epsilon':0}),
        (e_greedy, {'epsilon':0.1}),
        (e_greedy, {'epsilon':0.2}),
        (softmax, {'tau':1}),
        (softmax, {'tau':0.1}),
    ]
    if exercise == 3:
        algos += [
            (e_greedy, {'t':None}),
            (softmax, {'t':None}),
        ]


    results, Qai, times_selected = get_avg_reward(bandits, 1000, 10000, algos)


    plt.figure(figsize=(10, 6))
    plt.plot(results)
    legend = ['Random',
        '$\epsilon$-greedy ($\epsilon=0$)', '$\epsilon$-greedy ($\epsilon=0.1$)',
        '$\epsilon$-greedy ($\epsilon=0.2$)',
        'Softmax ($\\tau=1$)', 'Softmax ($\\tau=0.1$)']
    if exercise == 3:
        legend += ['$\epsilon$-greedy ($\epsilon(t)=1/\sqrt{t}$)',
            'Softmax ($\\tau(t)=4*(1000-t)/1000)$']
    plt.legend(legend, loc=4, prop={'size':9})
    plt.grid(True)
    plt.xlabel('Time'); plt.ylabel('Average reward')
    plt.savefig('fig/ex1-%d.pdf'%exercise)
    plt.clf()
    #  plt.show()

    legend.append('$Q^*_{ai}$')
    for arm in range(len(bandits)):
        plt.plot(Qai[:,:,arm])
        plt.grid(True)
        plt.xlabel('Time'); plt.ylabel('$Q_{ai}$')
        plt.ylim([-0.5, 2.5])
        plt.plot([0, len(Qai)], [bandits[arm,0], bandits[arm,0]], linewidth=2)
        plt.legend(legend, loc=4, prop={'size':9})
        plt.savefig('fig/ex1-%d-q%d.pdf'%(exercise, arm))
        plt.clf()
        #  plt.show()

    for algo in range(len(algos)):
        plt.figure(figsize=(5,5))
        plt.bar(range(len(bandits)), times_selected[algo,:],
                tick_label=range(len(bandits)), align='center')
        plt.xlabel('Arm'); plt.ylabel('Times selected')
        plt.savefig('fig/ex1-%d-a%d.pdf'%(exercise, algo))
        plt.clf()
        #  plt.show()

if __name__ == "__main__":
    run(1)
    run(2)
    run(3)
```

B Stochastic Reward Game code

```
import numpy as np
import matplotlib.pyplot as plt

def reward(grid, a, b):
    return grid[b,a,0] + np.random.randn() * grid[b,a,1]

def boltzmann(total_rewards, total_plays, tau, player):
    if player not in ['row', 'col']: raise ValueError('Incorrect player type')

    Q = q_values(total_rewards, total_plays)
    P = np.ones(len(Q[0])) * 1./len(Q[0])
    if np.sum(total_plays) != 0:
        P = np.sum(total_plays, 0 if player == 'row' else 1) / np.sum(total_plays),
    if player == 'row': P = np.reshape(P, (1,-1))
    else:               P = np.reshape(P, (-1,1))
    EV = np.sum(Q * P, 1 if player == 'row' else 0)

    distribution = np.exp(EV/tau)/np.sum(np.exp(EV/tau))
    return np.random.choice(range(EV.size), p=distribution)

def optimistic_boltzmann(total_rewards, total_plays, tau, player):
    Q = q_values(total_rewards, total_plays)
    maxQ = np.max(Q, 1 if player == 'row' else 0)
    distribution = np.exp(maxQ/tau)/np.sum(np.exp(maxQ/tau))
    return np.random.choice(range(maxQ.size), p=distribution)

def q_values(total_rewards, total_plays):
    results = np.copy(total_rewards)
    results[total_plays != 0] /= total_plays[total_plays != 0]
    return results

def run(sigmas, func, tau=0.1):
    sigma, sigma0, sigma1 = np.square(sigmas)
    grid = np.array([
        [[11, sigma0], [-30, sigma], [0, sigma]],
        [[-30, sigma], [7, sigma1], [6, sigma]],
        [[0, sigma], [0, sigma], [5, sigma]]
    ])

    total_rewards = np.zeros((len(grid), len(grid)))
    total_plays = np.zeros((len(grid), len(grid)))
    n_steps = 5000
    rewards = np.zeros(n_steps)
    avg_rewards = np.zeros(n_steps)
    alpha = 0.02
    probas = np.zeros((n_steps,3))

    for t in range(n_steps):
        row_choice = func(total_rewards, total_plays, tau, 'row')
        col_choice = func(total_rewards, total_plays, tau, 'col')
        r = reward(grid, col_choice, row_choice)
        total_plays[row_choice, col_choice] += 1
        total_rewards[row_choice, col_choice] += r
        rewards[t] = r
        if t == 0:
            avg_rewards[0] = r
        else:
            avg_rewards[t] = (1-alpha) * avg_rewards[t-1] + alpha * r

    return avg_rewards

if __name__ == '__main__':
    def beautify(funcname):
        return funcname.replace('_', ' ').title()
    params = [[0.2, 0.2, 0.2], [0.1, 4, 0.1], [0.1, 0.1, 4]]
    taus = [1]
    funcs = [boltzmann, optimistic_boltzmann]
    for index, param in enumerate(params):
        results, legends = [], []
        plt.figure(index)
        for func in funcs:
            for tau in taus:
                results.append(run(param, func, tau=tau))
                legends.append(beautify(func.__name__)+' '+str(tau))
        for res, color in zip(results, ['magenta', 'cyan']):
            plt.plot(res, color=color)
        plt.legend(legends, loc=4)
        plt.ylim([-10, 12]);
        plt.ylabel('Smoothed reward'); plt.xlabel('Time')
        plt.grid(True)
        plt.savefig('fig/ex2-%d.pdf'%index)
        plt.show()
```